Operational semantics of proto

Mirko Viroli, Jacob Beal, Kyle Usbeck

PII:	S0167-6423(12)00214-6
DOI:	10.1016/j.scico.2012.12.003
Reference:	SCICO 1485

To appear in: *Science of Computer Programming*

Received date:	7 August 2011
Revised date:	28 April 2012
Accepted date:	10 October 2012

Please cite this article as: M. Viroli, J. Beal, K. Usbeck, Operational semantics of proto, *Science of Computer Programming* (2012), doi:10.1016/j.scico.2012.12.003

Operational Semantics of Proto

Mirko Viroli[a], Jacob Beal[b], Kyle Usbeck[b]

[a]*Alma Mater Studiorum – Università di Bologna, Via Venezia 52, 47521 Cesena (FC), Italy*
[b]*BBN Technologies, 10 Moulton Street, Cambridge, MA 02138, USA*

Abstract

The Proto spatial computing language [1] simplifies the creation of scalable, robust, distributed programs by abstracting a network of locally communicating devices as a continuous geometric manifold. However, Proto's successful application in a number of domains is challenging its coherence across different platforms and distributions. We thus present a complete operational semantics for the Proto language, as executed asynchronously on a network of fast message-passing devices. This semantics covers all of the operations of the three space-time operator families unique to Proto—restriction, feedback, and neighborhood—as well as the current pointwise operations that it shares with most other languages. This formalization will provide a reference to aid implementers in preserving language coherence across platforms, domains, and distributions. The formalization process has also advanced the Proto language in several ways, which we explain in detail.

Keywords: spatial computing, distributed algorithms, amorphous medium, operational semantics, Proto

1. Introduction and Motivation

As the scale and variety of deployed and emerging distributed systems continues to increase, the challenges of creating scalable, robust aggregate be-

☆Partial support for this work was provided by the EU FP7 project "SAPERE - Self-aware Pervasive Service Ecosystems" under contract No. 256873 and by DARPA DSO under contract W91CRB-11-C-0052; the views and conclusions contained in this document are those of the authors and not those of the sponsors or the US or EU government.

Email addresses: `mirko.viroli@unibo.it` (Mirko Viroli), `jakebeal@bbn.com` (Jacob Beal), `kusbeck@bbn.com` (Kyle Usbeck)

havior are increasing. One large class of these systems are *spatial computers*—potentially large collections of devices distributed to fill some space, such that the difficulty of moving information between devices is strongly dependent on the distance separating them. Examples of spatial computers include sensor networks, mobile ad-hoc networks (MANETs), colonies of engineered bacteria, robotic swarms, and pervasive computing systems. Aggregate-level programming and control of such systems is an important and unresolved challenge. Many different approaches have been proposed, most of which are surveyed in [2]. They include, among the others, distributed logic programming [3, 4], viral tuple-passing [5, 6], chemical-like reactions [7, 8, 9], abstract graph algorithms [10], spatial data streaming [11], and topological surgery [12], none of which has yet proven successful enough to be adopted into widespread use.

One promising approach to the programming and control of such systems is to view the network of devices as a discrete approximation of the space through which they are distributed. The Proto spatial computing language [1] embraces this approach: a program is specified in terms of geometric computations and information flow on a continuous manifold[1]. These aggregate-level programs are automatically transformed into a set of local interactions between discrete devices, which approximate the desired global behavior. This approach has advantages for scalability, since more devices are simply a better approximation of the continuous model; for portability, since changing to a different platform just means changing how the continuous model is approximated; and for robustness, since small changes are just changes in approximation quality and large changes appear as changes in manifold structure, which geometric computations inherently adapt to [13]. Moreover, Proto has already been successfully applied to problems in such diverse areas as sensor networks [14], swarm and modular robotics [15], and synthetic biology [16, 17].

The breadth, however, is becoming a challenge to the coherence of Proto, as different platforms have very different demands and execution characteristics, and the number of platforms to which Proto is being applied is steadily increasing. Although the continuous abstraction of Proto is the same across all of these platforms, the details of how this is approximated in actual eval-

[1]A *manifold* is a topological space that locally resembles Euclidean space, but globally can be more complex.

uation are necessarily different. It would thus be easy for different versions of Proto to start diverging in implementation and effective semantics, particularly since the current reference implementation of the Proto tool-chain is distributed as free and open source software [18].

We thus believe that is it imperative to establish a formal semantics for Proto, and in this paper we do so, building on the core Proto semantics developed in [19], properly extended with a special "memory trees" representation of device state, enabling us to cover the entirety of Proto, including variable definitions, function calls, and actuation. Accordingly, we begin with a brief review of Proto, then present an operational semantics for the complete current form of the Proto language. As Proto is an inherently distributed language, the semantics of program execution also depend on the nature of communication between devices; we thus include the most frequently used Proto network model, of asynchronous execution on a network of fast message-passing devices. The main technical difficulties in the design of the proposed semantics include: *(i)* splitting the formalisation in two orthogonal parts dealing with device internals and with platform issues; *(ii)* designing a mechanism of evaluation contexts allowing us to provide the semantic of each language construct into a single transition rule; *(iii)* conceiving the "memory trees" concept, by which we keep track of the state of variables in a computing device, and support interactions only with neighbor devices having compatible trees; and *(iv)* designing each transition rule so as to couple evaluation of expressions with a proper traversal of the memory tree.

The formalization we develop may thus serve as a reference for implementers, helping to preserve the coherence of the language across the increasing number of platforms, application domains, and distributions. We also note several advancements in the Proto language that this formalization process has produced, including *(i)* a new indexing semantics for function calls extending the set of allowed Proto programs (beyond those where total inlining was possible) and reducing the binary size of programs, *(ii)* reducing computation delay in expressions combining neighbouring and feedbacks operations, and *(iii)* detecting a number of cases of ill-structured Proto programs.

The remainder of this paper is organized as follows:

§ Section 2 describes the development of the domain-specific language, Proto, and gives a brief tutorial for its use.

§ Section 3 formalizes the semantics of the Proto language for individual devices.

§ Section 4 formalizes a complementary platform semantics that describes the evaluation of Proto on a network of devices.

§ Section 5 discusses the important findings of this work, and its implications for the Proto language.

2. The Proto Language

In this section, we give a brief review of the Proto language, then explain the purpose and benefits of programming a spatial computer using the Proto domain-specific language. We shall also occasionally refer to elements of MIT Proto [18], a suite of tools that includes a compiler and a virtual machine (called ProtoKernel [20]), that comprise the current reference implementation of Proto. In this paper, we distinguish among these by referring to Proto as the language, and specifically addressing the compiler and virtual machine (VM) respectively.

2.1. Proto as a Domain-Specific Language

Proto is a Domain-Specific Language (DSL) for representing the description of aggregate device behavior in a spatial computer. According to Mernik *et al.*, DSLs are,

> "languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application" [21].

To this definition, Proto allows programmers to write complex, distributed programs for potentially large numbers of devices in a simple and concise manner.

The key insight in creating the Proto DSL syntax is the identification of a set of composable constructs inherent in the spatial computing domain. Proto creates scalable distributed applications by representing 1) groups of locally communicating devices as continuous manifolds and 2) composable operations as manipulations of those geometric manifolds. Thus, interpretation of the Proto DSL yields space-time manipulations on continuous manifolds approximated by the discrete network of devices.

There are four classes of Proto operations, separated by their interaction with space and time (see Figure 1). Pointwise operations (e.g., **`+`**, **`sqrt`**, **`tup`**, and **`mux`**) involve neither space nor time. Restriction operations (e.g., **`if`**) restrict program execution to a sub-space. Feedback operations (e.g., **`rep`**) evolve device state in continuous time. Neighborhood operations (e.g., **`nbr`**, **`int-hood`**, **`min-hood`**) express information flow through the spatial computer by summarizing computations over neighbors—the set of devices within communication range.

	No Space	**Space**
No Time	Pointwise	Restriction
Time	Feedback	Neighborhood

Figure 1: The four classes of Proto operations, separated by their interaction with space and time.

The Proto DSL manipulates the spatial computer in both space and time via composition of these different classes of operators.

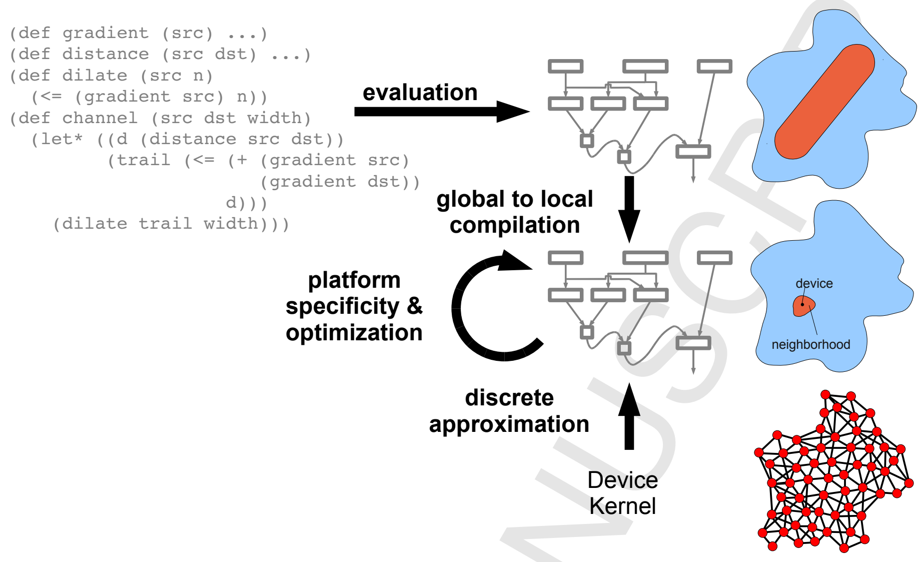
Proto can also be viewed as an application generator [22], a tool derived from a formal language definition. In the case of Proto, the formal language definition is the DSL describing the global system behavior and the derived tool is the "local" program for the devices in the spatial computer network. Figure 2 illustrates the process of compiling the global system behavior into a local program which approximates the continuous behavior on a discrete network of devices.

2.1.1. Evolution of Proto as a DSL

In this section, we map the history of the Proto DSL development onto the framework for deciding when and how to develop a DSL given in [21], with four stages[2]:

1. Decision,
2. Analysis,
3. Design, and
4. Implementation.

[2][21] includes a fifth stage, *Deployment*, which is out of the scope of both Mernik *et al.*'s study, and our discussion in this paper.



```
(def gradient (src) ...)
(def distance (src dst) ...)
(def dilate (src n)
  (<= (gradient src) n))
(def channel (src dst width)
  (let* ((d (distance src dst))
         (trail (<= (+ (gradient src)
                       (gradient dst))
                    d)))
    (dilate trail width)))
```

Figure 2: The Proto compiler acts as an application generator by compiling the global spatial computer behavior into programs that are executed on the discrete network of devices that approximate the continuous geometric manifold.

Decision. The decision to create a DSL occurred as a direct result of *task automation* [21]. In distributed system programming, a repetitive pattern (described in [23]) often occurs. This pattern has the following generic steps:

- Gather data from neighbors,
- Do some computation separately on each neighbor's data, and
- Return a function combining the results from the neighbors.

Computations incorporating such patterns are often further organized into rounds, where in each round all neighbor data is gathered in parallel over some instance of time, then all computations run to completion using the current neighbor data before the next round begins.

The fact that this pattern is independent of neighborhood size, network structure, and data content makes it a perfect candidate for abstracting into its own DSL as an "enabler of reuse," as explained in [21].

Analysis. An *informal* domain analysis [21] concluded in the key insight explained in [24]: that the repetitive pattern, now called a *neighborhood* computation, works in continuous space. Utilizing a continuous space abstraction in Proto allowed for, among other things, the possibility to more easily prove (in continuous semantics) that specific local device actions result in a certain global (a.k.a. "emergent") behavior. To this day, neighborhood operations follow the original pattern, although there have been many improvements to the original concept (i.e., restriction, continuous time).

Design. Proto is designed as a *piggyback* [21] on a LISP-style syntax, although not a LISP itself, with S-expressions as the primary notation. *Language exploitation* was used primarily as a method of providing familiarity to end-users, with the added benefit that the LISP style offers an elegant way of composing individual operations into a global behavior. Since the original choice to use LISP-style notation, it has been shown that some features of LISP are impossible in the Proto DSL (e.g., arbitrary first-class functions [25]; see also discussion below in Section 5), however the syntax still provides a clean abstraction for the many newly-added language features.

Implementation. During the specification of the Proto language, the Proto compiler and virtual machine were also being developed. The implementation of the compiler was primarily driven by the requirement to create local programs from a global behavior description: termed the *global-to-local* transformation. Thus, the implementation pattern from [21] was a *Compiler/application generator*.

Simultaneously, the implementation of the original ProtoKernel VM was driven by the requirement to execute the generated applications on small, cheap embedded devices, such as Mica2 Motes. As the Proto VM was ported to more device types, each with differing feature-sets, a plug-in architecture was adopted as the mechanism for specifying device-specific behavior (i.e., sensors and actuators). Now, with Proto being used in a wider variety of domains and across multiple platforms, the specification of Proto's core operational semantics is vital to ensuring consistency between implementations.

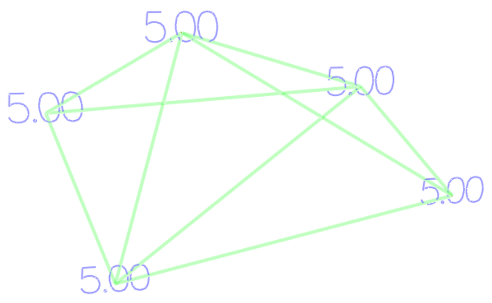
2.2. Using Proto

Proto is a LISP-like, purely functional language. Proto's prefix operator statements are associated using parenthesis. Program 1 shows a simple (pointwise) addition expression executing on a small network.

Although this looks syntactically identical to a LISP addition, at a global level, Program 1 is manipulating not numbers but *fields*, functions that map points in space to data values. There are two main types of values for such

Program 1 A simple pointwise operator in Proto.

```
(+ 2 3)
```



fields (as will be explained in detail later in Section 3.2.1): **Local** valued fields, such as scalars and tuples, and **Field** valued fields, where each device maps to a set of values from neighboring devices. In the case of Program 1, the program is adding a local field that maps all devices to the scalar value 2 to another local field that maps all devices to the scalar value 3.

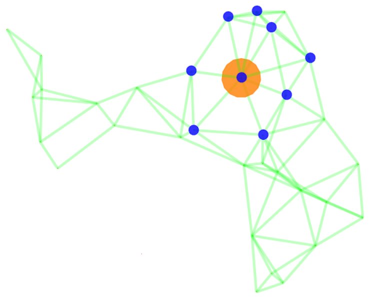
Proto operators work over some combination of space and time. Program 2 shows the **`any-hood`** and **`nbr`** operators working over space and time. The `nbr` operator collects information from each neighboring device—so from some distance away, and from some time in the past since the information does not travel instantaneously. In this case, it returns a map of neighbors to the value of the neighbor's "sensor number one" (a Boolean-valued test sensor used for debugging in the MIT Proto simulator). The **`any-hood`** operator summarizes these results, returning **`true`** if *any* of the neighbors have **`(sense 1)`** enabled. The **`blue`** operator simply enables a blue LED on the device if the input is **`true`**.

The **`rep`** command (of the feedback operator class) evolves a variable over time. In Program 3, the variable **`timer`** is initialized to 0 and evolves by adding **`(dt)`**, the change in time since the program was last evaluated. Therefore, the program is essentially updating the time on each device, in continuous time (though implemented by discrete rounds of evaluation).

By combining the manipulation of space and time, Proto can transport information over multiple hops of the spatial computer network. In Program 4, the **`distance-to`** function is first defined and then evaluated. Its definition is a combination of space and time calculations.

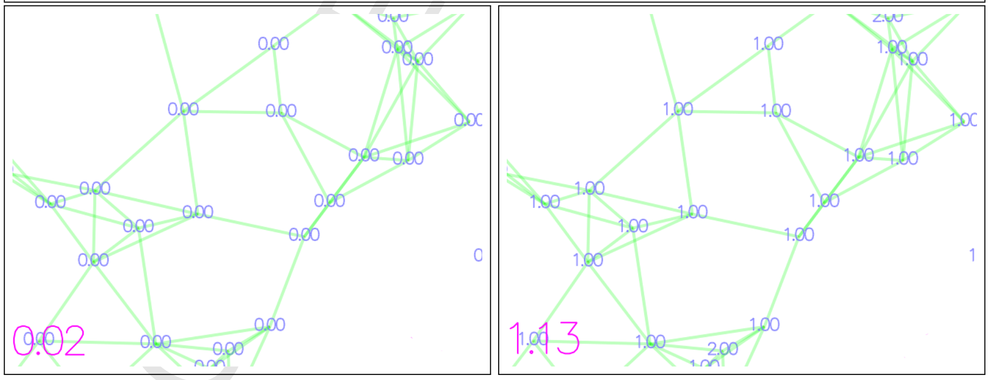
Program 2 An example of operators that work in *space* and *time*.

```
(blue (any-hood (nbr (sense 1))))
```



Program 3 An example of an operator that works in *time*. Large numbers in the lower left show elapsed seconds of simulated time.

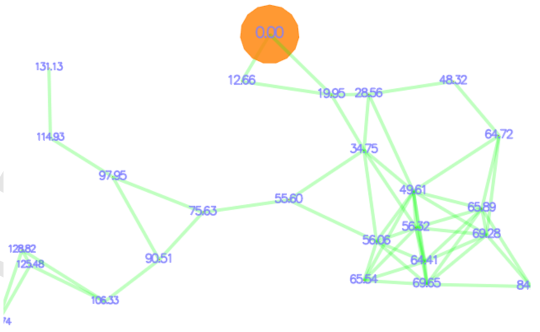
```
(rep timer 0 (+ timer (dt)))
```



In the definition of **distance-to** in Program 4, the variable, **d**, is initialized to infinity and evolves by finding the minimum distance to the source, **src**. The minimum distance is defined as the lowest (dictated by **min-hood**) sum of the distance to each neighbor (**nbr-range**) plus that neighbor's distance to the source, **(nbr d)**. The **mux** operator is a "multiplexer" operator, similar to a branch, which evaluates both branches, then selects one value to return. In this program, **mux** is used to select between the general case and the base case where the distance to the source device is zero. The evaluation performs a relaxation computation, applying the triangle inequality over neighbors until each devices already returns its distance to the nearest device whose **(sense 1)** is enabled.

Program 4 Combining space and time yields long-range information flow.

```
(def distance-to (src)
   (rep d inf (mux src 0
      (min-hood
         (+ (nbr d)
            (nbr-range)))))
(distance-to (sense 1))
```

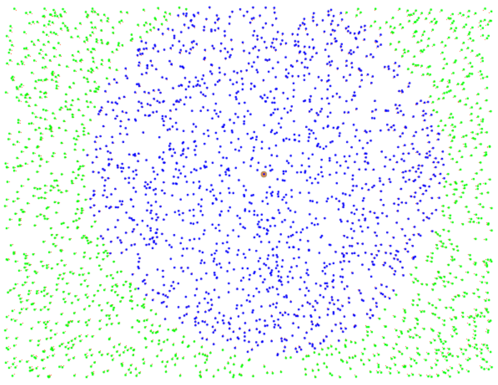


The notion of *restriction* dictates the space in which a program executes. In other words, restriction is the method by which Proto controls where an expression is evaluated. The syntax for restriction in Proto uses the **if** construct, which takes three arguments: a condition, a true-branch, and a false-branch. If the condition evaluates to **true**, then only the true-branch is evaluated; if it is **false** then only the false-branch is evaluated. Program 5 uses restriction to enable different color LEDs based on the distance away from a source device.

In Program 5, the restriction condition splits the space into the set of devices whose distance to a **(sense 1)** device is less than 50 (true branch), and those whose distance is greater-than or equal to 50 (false branch). The devices in the true branch enable their blue LED, whereas those in the false branch enable their green LED.

Program 5 Restriction determines the space in which a program executes.

```
(if (< (distance-to
          (sense 1)) 50)
   (blue 1)
   (green 1))
```



Proto can also control device movement by computing mass-flow over vector fields, as described in [15]. Program 6 shows an example of Proto device movement using the **mov** operator.

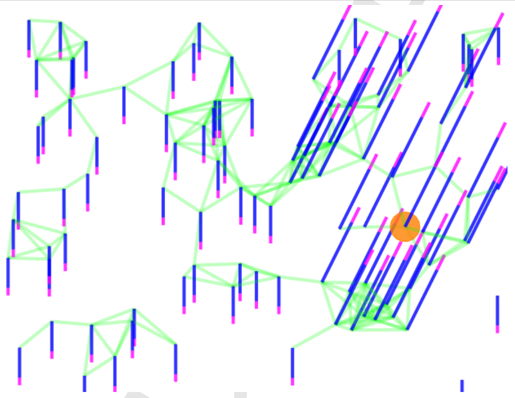
The 2D motion example in Program 6 is similar to the last program, but instead of switching a LED the **if** statement selects between two possible vectors. Devices within 50 of a `(sense 1)` device get a vector created with **tup** with a first element of 10 and second element of 20. Devices further away have a different vector, with first element 0 and second element -10. The composite vector field, displayed in the simulator as a blue line with a magenta tip to indicate direction, is then fed to the **mov** operator, which will cause each device to move with the speed and direction dictated by its vector.

3. Language Semantics

We now provide the operational semantics for the Proto language, when evaluated on an asynchronous network of discrete devices with fast communication. The fast communication assumption is that the time for a message to be transmitted and propagate through the carrier medium from device to device is much faster than a round of communication (e.g. radio packet transmission over short range). This assumption is just to allow a simpler semantics that treats sending and receiving of messages as an atomic event—this assumption can be weakened to allow slow, long duration, or lossy communication without impacting the correctness of the semantics (though particular distributed algorithms may have more stringent requirements).

Program 6 Support of device movement as mass-flow over vector fields.

```
(mov (if (< (distance-to
              (sense 1)) 50)
         (tup 10 20)
         (tup 0 -10)))
```



Our specification is divided in two parts, described in this section and Section 4. The first part, presented in this section, concerns internal device behavior: we present notation and terminology (Section 3.1), then the data types that will be used and relevant language syntax (Section 3.2) and a syntax over the state of a device during evaluation (Section 3.3). We then use these to provide an operational semantics that describes the execution of Proto programs with respect to a single device and its local state (Section 3.4). This semantics corresponds to a correct implementation of ProtoKernel [20] on an individual device, ignoring optional features such as transmission frequency backoff. The second part provides a complementary semantics for the platform on which Proto is executed, describing how devices interact with their environment, how messages are exchanged between devices, and how topology changes should be handled.

Our formalization necessarily combines techniques typical of functional languages (since Proto is closely related to LISP) and process algebras (since it is used to program message exchanges in a distributed system, like in π-calculus [26]). Similar combined approaches have previously been used elsewhere; see for example [27], [28]. The formalization we present in this paper is an extension of the core Proto semantics previously formalized in [19]: the current work introduces "memory trees" as a fundamentally new representation of state, enabling us to cover the entirety of Proto, including variable definitions, function calls, and actuation.

Symbol	Meaning
$\mathtt{x}^y$	a meta-variable of type x named y
$\mathtt{e}$	a tokenized text expression
$\mathtt{v}$	a variable name
$\mathcal{B}$	Script indicates a Proto data type; e.g., $\mathcal{B}$ is any Boolean
a	a device
$\overline{\mathtt{x}}$	an ordered set of x
$\lvert\overline{\mathtt{x}}\rvert$	size of set x
$\mathtt{x}_i$	ith element of ordered set $\overline{\mathtt{x}}$
$\bullet$	the empty set
$\mathtt{s} = \langle\Sigma\rangle e$	execution state of a device
Σ	store of memory at a device
$\mid$	composition operator for (multi)sets
$\{\overline{\mathtt{x}}\}$	a set or sequence of elements matching pattern x
$\square$	A "hole" indicating a current focus of evaluation
τ	A memory tree
T	A memory context tree (a memory tree with one hole)
$\mathtt{E}$	An expression context tree (an expression with one hole)
$\mathtt{X}[\![\mathtt{x}]\!]$	A memory or expression tree made by replacing the hole of X with x.
$\perp$	No result
σ	Export
$\triangleright$	Neighbor Alignment
$\rightarrow$	Device state transition
$\rightarrow^*$	Closure of device state transitions
$a \mapsto \tau$	Imported Message τ from device a
$\otimes$	Extract element from set
$a \overset{\mathcal{V}}{\longmapsto} a'$	Network link between a and a', with metric information $\mathcal{V}$
$a \twoheadrightarrow [\tau]a'$	Pending message τ from a to a'
$\rightarrowtail$	Platform state transition

Table 1: Table of symbols used in describing semantics

3.1. Notation and Terminology

We adapt our syntactic notation and terminology from standard and widely accepted frameworks like FJ [29]. Our transition rules use natural semantics: the precondition is generally over a closure $\rightarrow^*$ of transition rules. We have preferred this form for two reasons: first, it produces a more compact semantics and second, it allows for more flexibility in the ordering of operations in the implementation. This choice of semantics does raise two issues, however: first, there is the problem of distinguishing between non-termination and errors. At present, however, Proto does not have any runtime error-handling facilities, and so the two are also typically indistinguishable in practice, both leading to program crash through bounds violations. We thus defer this challenge for future work when the problem of distributed error-handling will be addressed. Second, it will be important to prove that all paths are equivalent. The complexity of the semantics puts proving this and other relevant properties beyond the scope of this paper, and we reserve this effort as well for future work.

In formulating transition rules, we refer to *metavariables* as variables used in the formalization, as schema of formulas or terms—to distinguish them from the notion of variables used within the Proto language. We adopt a notation where a metavariable $\mathtt{x}$ is shown in typewriter font, a literal expression $\mathbf{\mathtt{x}}$ is shown in bold typewriter font, and a data type $\mathcal{X}$ is shown in script. We use superscript to distinguish instances of metavariables, e.g., $\mathtt{x}$, $\mathtt{x}'$, and $\mathtt{x}^a$ are three different metavariables of category $\mathtt{x}$.

Given any metavariable $\mathtt{x}$, metavariable $\overline{\mathtt{x}}$ is used to denote a sequence of zero or more elements of kind $\mathtt{x}$. We let $\mathtt{x}_1, \ldots, \mathtt{x}_n$ be the elements of a sequence, $|\overline{\mathtt{x}}|$ be the length of the sequence, and $\mathit{tail}_i(\overline{\mathtt{x}})$ be the sequence $\mathtt{x}_{i+1}, \mathtt{x}_{i+2}, \ldots, \mathtt{x}_n$ Finally, $\bullet$ shall represent the empty sequence, and we define both $\bullet_i$ and $\mathit{tail}_i(\bullet)$ to be $\bullet$.

For unordered collections, we compose elements with operator "|". This operator is assumed to be commutative, associative, and to absorb $\bullet$, thus forming a multiset composition operator.

Some further ad-hoc notation is added to simplify the management of sequences and multisets. Given a structure $\mathtt{k}$, we write $\{\mathtt{k}\}$ for a collection of elements of kind $\mathtt{k}$. We take the type of collection (multiset or sequence) to be implicit in the kind of $\mathtt{k}$, since we will not define any meta-variable kinds that are collected in both sets and sequences. If there are sequence metavariables in $\mathtt{k}$, then these sequences are aligned with one another. For example, let $\overline{a}$ be the sequence of neighbors a_0, a_1, a_2, and $\overline{\mathcal{S}}$ the sequence of scalars $1, 2, 3$. Then $\{\overline{a} \mapsto \overline{\mathcal{S}}\}$ is the field $\{a_0 \mapsto 1 \mid a_1 \mapsto 2 \mid a_2 \mapsto 3\}$ that maps each neighbor to the corresponding value. We will use this notation for manipulating subsets of values,

We also add an auxiliary composition operator denoted "$\otimes$", which will be used for extracting elements from sets. This acts as a variant of operator "|" guaranteeing that the multisets on the left and right side have no elements in common. Formally, we define "$\otimes$" as a partial binary function that yields no result if the two multisets given as arguments have non-empty intersection, and yielding their composition by "|" otherwise. We will generally use this for updating the contents of a set. For example, if $\mathcal{F}$ is a field, then matching the expression $\{a \mapsto \mathcal{L}\} \otimes \mathcal{F}'$ against $\mathcal{F}$ extracts one arbitrary element from the field, leaving the remainder in $\mathcal{F}'$. For another example, $\{\overline{a} \mapsto \overline{\mathcal{S}}\} \otimes \mathcal{F}'$ extracts all elements with type **Scalar**.

We shall define particular metavariables as we go; for the convenience of the reader, we have collected a summary of symbols used in this paper in Table 1.

3.2. Proto Expressions

We now begin by presenting the content of Proto expressions: first the data types that are used in Proto computations, followed by the syntax used by the programmer to express Proto programs.

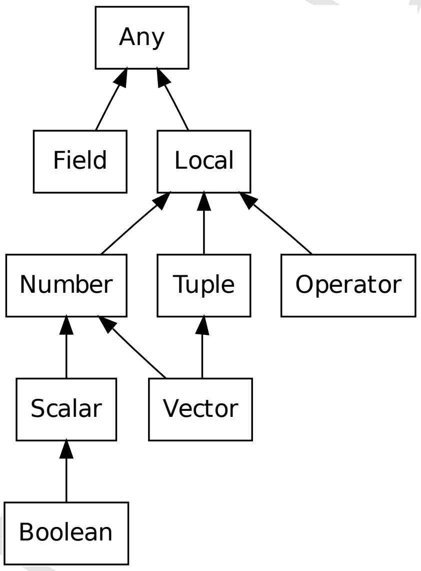


Figure 3: Proto data types; arrows indicate parent relationships.

3.2.1. Types in Proto

The data types that are used in Proto computations (Figure 3) are split into two main categories: **Local** values, which span numbers, tuples, and operators, and **Field** values, which map each member of a set of neighbor devices to a **Local** value.

The full set of current Proto types, ranged over by meta-variable `t`, are:

- **Any** ($\mathcal{A}$) is any value of any type.

- **Field** ($\mathcal{F}$) is a map from a subset of the device's neighborhood to **Local** values. A field may also be described as a set $\{\overline{a} \mapsto \overline{\mathcal{L}}\}$, mapping each included neighbor a to a **Local** value.
- **Local** ($\mathcal{L}$) is any non-**Field** value: a **Number**, **Tuple**, or **Operator**.
- **Operator** ($\mathcal{O}$) is a primitive or user-defined function. An operator may also be described as a pair ($\overline{\texttt{arg}}, \texttt{e}$) of its arguments and the expression for its body, using the syntax defined in the Section 3.2.2. Note that the use of **Operator** data types is highly restricted at present, due to the challenges of distributed first class functions discussed in [30].
- **Tuple** ($\mathcal{T}$) is an ordered set of $k \geq 0$ **Local** values. Note that all zero-length tuples are equivalent "null" tuples.
- **Number** ($\mathcal{N}$) is a **Scalar** or **Vector**.
- **Scalar** ($\mathcal{S}$) is any real number, plus the special values infinity, negative infinity, and not-a-number, which behave in the usual manner defined by IEEE floating point standards [**IEEE 754-2008**].
- **Vector** ($\mathcal{V}$) is a **Tuple** of **Scalar** values. Note that a vector of one element is semantically equivalent to a scalar.
- **Boolean** ($\mathcal{B}$) is a **Scalar** interpreted as a logical value. Any non-zero value is **`true`**, canonically represented by 1; **`false`** is represented by 0.

The **Field** and **Tuple** types may be further refined by specifying the types of their contents. We will do this via functional notation. Thus, for example, type $\mathcal{F}(\mathcal{S})$ would be a **Field** that maps from neighbors to **Scalar** values, and type $\mathcal{T}(\mathcal{B}, \mathcal{L}, \mathcal{S})$ would be a **Tuple** whose first element is a **Boolean**, whose second element is a **Local**, and whose third element is a **Scalar**. When a type is excluded, we annotate it as $\mathcal{X} - \mathcal{Y}$; for example, $\mathcal{A} - \mathcal{O}$ means any type except for **Operators**.

In the current implementation of Proto, the compiler attempts to resolve all types at compile-time, and throws compile errors whenever it is unable to resolve a type to a "concrete" value—a **Scalar**, **Boolean**, **Operator**, or a **Field** or **Tuple** of concrete values. In our semantics, we will assume that such type checking has already been performed by the compiler, such that we need not check dynamically for type mismatches.

We will also assume two simple type coercion rules. First, **Scalar** values can be coerced to **Boolean** values by mapping everything non-zero to 1. We shall thus use the literals **`true`** and 1 and **`false`** and 0 interchangeably. Second, **Local** values can be coerced to **Field** values by an insertion of the **`local`** operator, described below. We will assume that the first is done automatically whenever a **Scalar** is supplied where a **Boolean** is needed, and that the second is inserted by the compiler.

Note that although this type system represents the current state of the art in Proto, this type system is clearly not yet complete as it lacks many of the standard features of modern languages. Notable missing portions include text or symbols, structures or objects, and exception handling. As Proto has been primarily a research language, such missing features have not yet been an obstacle to its use.

As Proto continues to mature and begins to be put into real-world deployments, however, such missing features are expected to be added to later versions of Proto. Since these "standard language" missing features are largely in the pointwise class of computations on individual devices, adding them is expected to have little impact on the semantics of Proto. For example, the first two mentioned, text and objects, are expected to be simple extensions of the pointwise operators—strutures are syntactic sugar on tuples, while objects will require some semantic extensions to support methods, inheritance, etc; exception handling will require more careful thought, given the distributed and asynchronous model of execution in Proto.

3.2.2. Proto Syntax

Proto syntax is based on LISP S-expressions, as a simple way of expressing functional programs. For this discussion of semantics, we will omit those portions of Proto syntax that are necessarily resolved at compile-time: macro handling, language extensions (e.g., platform-specific sensors and actuators like the **`(sense 1)`** debug sensor used above), and code for controlling the behavior of the compiler.

We shall define metavariable `e` as any Proto expression, and metavariable `v` as any variable name. Figure 4 shows the syntax of Proto expressions: the essence of any Proto program is a single functional expression `e`. We shall refer to the expression containing the entire program as $\mathtt{e}^p$.[3] Any expression

[3]Programmers actually write a sequence of expressions $\overline{\mathtt{e}}$, in normal LISP style, which

`e` is either a local literal value in $\mathcal{L}$ (e.g., the number 13), a variable reference `v` (e.g., to a function argument `x`), the application of an operator `op` to a possibly empty sequence of expressions $\overline{\texttt{e}}$ (e.g., **`(+ x 13)`**), a stateful or conditional computation (e.g., an **`if`** or **`letfed`** statement), or one of several forms of definition. An operator `op` may be any of a large variety of built-in primitives, or may be defined by the user in-place or elsewhere. The arguments `arg` of an operator are variables `v`, which may optionally be restricted to a type `t` using the form `v` **`|`** `t`.

e	::=		Expressions
		$\mathcal{L}$	Literal values
	|	v	Variable name
	|	(op $\overline{\texttt{e}}$)	Operator calls
	|	(**rep** v e e)	Stateful computation
	|	(**letfed** ($\overline{\texttt{(v e e)}}$) $\overline{\texttt{e}}$)	Generalized stateful computation
	|	(**if** e e e)	Conditional evaluation
	|	(**let** ($\overline{\texttt{(v e)}}$) $\overline{\texttt{e}}$)	Local variable
	|	(**def** v e)	Global variable
	|	(**def** v ($\overline{\texttt{arg}}$) $\overline{\texttt{e}}$)	Function
	|	(**fun** ($\overline{\texttt{arg}}$) $\overline{\texttt{e}}$)	Anonymous function
op	::=	pointwise | neighbor | summary | io	Primitive operators
	|	v	User-defined operator
	|	(**fun** ($\overline{\texttt{arg}}$) $\overline{\texttt{e}}$)	Anonymous function
pointwise	::=	**all** | **tup** | **not** | **elt** | **+** | **-** | ***** | **/**	Pointwise operators
	|	**<** | **>** | **=** | **<=** | **>=** | **min** | **max** | **abs**	
	|	**floor** | **ceil** | **round** | **pow** | **sqrt** | **log**	
	|	**mod** | **rnd** | **sin** | **cos** | **tan**	
	|	**asin** | **acos** | **atan2** | **sinh** | **cosh** | **tanh**	
	|	**len** | **vdot** | **mux**	
neighbor	::=	**local** | **nbr** | metric	Neighborhood
metric	::=	**nbr-range** | **nbr-bearing** | **nbr-vec**	Space-time Metrics
	|	**nbr-lag** | **nbr-delay**	
summary	::=	**fold-hood***	Neighbor Summary
io	::=	**hood-radius** | **infinitesimal** | **density**	Universal sensors
	|	**dt** | **mid** | **speed** | **bearing**	
	|	**mov** | **flex** | **set-dt** | **probe**	Universal actuators
arg	::=		Arguments for functions
		v	Variable name
	|	v "|" t	Typed variable

Figure 4: Syntax of Proto in BNF, modified to use our symbol conventions: literals are denoted by bold rather than quotes (except for the "`|`" in the typed variable expressions), and sequences are shown with an overbar.

the compiler implicitly wraps together into a single **`all`** expression: (**`all`** $\overline{\texttt{e}}$)

A few notes regarding syntax: first, note that the programmer is only allowed to use **Local** values as literals, and not **Field** values. This is because the set of neighbors is only known at run-time. Since a **Field** type is a map from neighbors to values, this means that all **Field** values must be produced at run-time, e.g., by means of the **`nbr`** and **`local`** constructs. Accordingly, during evaluation we will use an extended syntax that is identical except that values can be of any type ($\mathcal{A}$) rather than just locals.

Second, note that a number of widely used Proto language constructs are omitted from this semantics-focused syntax, because they are defined using macros rather than primitives. Among the notable missing are **`and`** and **`or`** (defined using branches), **`let*`** (defined as a chain of **`let`** expressions), and summaries like **`min-hood`** and **`any-hood`** (defined continuously, but on a discrete network is executed equivalently to **`fold-hood*`**, a reduce operator over neighbor values). As there is no semantic content to these constructs, we have omitted them for simplicity; most of their definitions may be found by reading the macros in the `core.proto` in the MIT Proto distribution. We have made one exception, regarding the **`letfed`** operator used the create and update state variables: **`rep`** is a macro, defined as a **`letfed`** of a single variable, which returns only that variable. However, the **`rep`** construct is useful in explaining how Proto handles state, so we have retained it for pedagogical reasons.

Finally, note that this is not a minimal semantics: all Proto primitives are included here, despite the fact that some could be implemented with macros or functions instead. For example, **`def`** of a function could be defined as a macro that combines **`def`** of a variable with an anonymous function. There are a number of other such examples, particularly mathematical operations like **`tan`**, **`sqrt`**, **`abs`**, etc.

3.3. Device State

The semantics of evaluation also depends on two types of state: the state of the program on a given device (e.g., memory, sensors and actuators, values shared with neighbors), and the state of the evaluation itself (e.g., the order in which expressions are traversed).

3.3.1. Memory Trees

Just as expressions are tree-structured in Proto, so too are the memory constructs that are used to track the side-effect state created by evaluating an expression. Each round, an evaluation will compute a tree of side-effect state, which will then be used in the computation of the next evaluation and also partially shared with neighbors. The reason for using tree structures in this way is to ensure that state from different devices, which may have taken different branches during evaluation, can still be correctly aligned with one another. For example, in an expression:

```
(+ (some-complicated-function) (distance-to (sense 1)))
```

the **`distance-to`** computation needs to be able to share state between neighboring devices, even if each neighbor has created a different number of side-effect states while evaluating the complicated function that precedes it.

We thus introduce the following syntax for memory trees:

$$\tau \ ::= \ \mathcal{B} \ | \ \mathtt{v} := \mathcal{A} \ | \ nbr(\mathcal{L}) \ | \ \overline{\tau} \ | \ (\tau)$$

A memory tree τ is a tree, where the leaves contain information needed for implementing the space-time operators of Proto: **Boolean** values indicating which branch of an **`if`** was taken, variable assignments $\mathtt{v} := \mathcal{A}$ that store state, or $nbr(\mathcal{L})$ expressions containing **Local** values being exported by the **`nbr`** function. In our notation, we will consider τ to admit only non-empty trees, while $\overline{\tau}$ also admits the tree $\bullet$. Pointwise operators (the purely functional portions of Proto) will generate empty sub-trees (the empty sequence tree $\bullet$). Because these are represented by the empty sequence, they are absorbed into adjacent tree elements by the following evident relation:

$$\overline{\tau}, \bullet, \overline{\tau}' \equiv \overline{\tau}, \overline{\tau}'$$

To illustrate how memory trees are used, let us consider a few cases:

- A purely functional expression such as **`(+ (+ 1 2) (+ 3 4))`** generates the empty tree $\bullet$.
- Neighborhood operations like **`nbr`** add export leaves into the tree. For example, the expression

  ```
  (+ 1 (fold-hood* min inf (+ (nbr 1) (* (nbr-range) (nbr 3)))))
  ```

  creates the tree $nbr(1), nbr(3)$.

- Branching with the **if** construct creates a sub-tree that starts with a **Boolean** value recording the results of the test (the first expression in the **if**). The rest of the sub-tree contains state for the evaluated sub-expression. For example, the expression:

  ```
  (if (= 8 8)
      (+ (fold-hood* min inf (nbr 1))
         (if (= 6 7)
             (fold-hood* min inf (nbr 2))
             (fold-hood* min inf (* (nbr 3) (nbr 4)))))
      (fold-hood* min inf (nbr 5)))
  ```

  creates the tree ($\texttt{true}, nbr(1), (\texttt{false}, nbr(3), nbr(4))$).

- Stateful computation constructs such as **rep** generate a tree reporting variable assignment. These expressions embed a branch that tests whether there is already a state in memory. On the first evaluation, where there is not, the second (initialization) expression is evaluated; on subsequent evaluations, the third (update) expression is evaluated instead. For instance, consider this expression **(rep x 0 (+ x 1))** implementing a simple counter. When first evaluated, it produces the memory tree ($\texttt{false}, \mathrm{x} := 0$), initializing `x` to zero. On the second evaluation, `x` is already in memory, so it instead produces ($\texttt{true}, \mathrm{x} := 1$), then ($\texttt{true}, \mathrm{x} := 2$) on the third evaluation, and so on.

- Branching constructs reuse a previous sub-tree only if the test (the first expression) has the same value as in the previous evaluation. Otherwise, the branch is discarded and replaced. For example, expression:

  ```
  (rep x 0 (if (= x 0) (rep y 0 (+ 1 y)) 0))
  ```

  produces this sequence of memory trees:

  $$\begin{array}{l}(\texttt{false}, \mathrm{x} := 0)\\ (\texttt{true}, \mathrm{x} := 0, (\texttt{true}, (\texttt{false}, \mathrm{y} := 0)))\\ (\texttt{true}, \mathrm{x} := 1, (\texttt{true}, (\texttt{true}, \mathrm{y} := 1)))\\ (\texttt{true}, \mathrm{x} := 0, (\texttt{false}))\\ (\texttt{true}, \mathrm{x} := 0, (\texttt{true}, (\texttt{false}, \mathrm{y} := 0)))\\ (\texttt{true}, \mathrm{x} := 1, (\texttt{true}, (\texttt{true}, \mathrm{y} := 1)))\\ (\texttt{true}, \mathrm{x} := 0, (\texttt{false}))\\ \ldots\end{array}$$

At any given point during evaluation, only a certain sub-tree of the overall tree is under consideration. To keep track of it, we introduce the concept of a memory context tree T (using the notion of evaluation context from [31]), which is a memory tree with precisely one hole $\Box$ contained in some leaf T where:

$$T \quad ::= \quad \Box \ \mid \ \overline{\tau}, T, \overline{\tau} \ \mid \ (T) \quad \text{Memory context tree}$$

Such a hole can be filled with a tree τ by notation $T[\![\tau]\!]$, which simply gives the tree obtained by substituting the hole with τ. We can thus represent consideration of a sub-tree by $T : \tau$, where τ is the sub-tree and T is the memory tree with a hole in place of τ, and then reassemble the memory tree when we are done with $T[\![\tau]\!]$.

Walking and manipulation of trees can be annotated by filling the hole with another memory context tree: $T[\![T']\!]$. For instance, by $T[\![(\overline{\tau}, \Box, \overline{\tau}')]\!]$ we indicate a memory context tree in which we can match the sub-tree in which the hole occurs as $(\overline{\tau}, \Box, \overline{\tau}')$. Accordingly, T represents a memory context tree in which the hole has been transferred one step upwards, to the parent node of where it had been in $T[\![(\overline{\tau}, \Box, \overline{\tau}')]\!]$. To illustrate this, consider the memory tree:

$$(\texttt{true}, \mathtt{x} := 0, (\texttt{true}, (\texttt{false}, \mathtt{y} := 0, \Box)))$$

Matching the expression $T[\![(\overline{\tau}, \Box, \overline{\tau}')]\!]$ against this memory tree, we find a match if we set:

$$\begin{aligned} T &= (\texttt{true}, \mathtt{x} := 0, (\texttt{true}, \Box)) \\ \overline{\tau} &= \texttt{false}, \mathtt{y} := 0 \\ \overline{\tau}' &= \bullet \end{aligned}$$

In order to facilitate management of trees across neighbors, we introduce two auxiliary operators: export and alignment.

Export. The export operator σ is used to filter a memory tree down to only the state that is to be shared with neighbors. This is done simply by erasing all occurrences of variable assignments, since only the branching structure and export values are shared:

$$\begin{aligned} \sigma \mathtt{v} := \mathcal{A} &\equiv \bullet \\ \sigma(\overline{\tau}) &\equiv (\sigma\overline{\tau}) \\ \sigma\overline{\tau} &\equiv \sigma\tau_1, \sigma\, tail_1(\overline{\tau}) \\ \sigma\tau &\equiv \tau \qquad\qquad \text{otherwise} \end{aligned}$$

Note that σ applies to the entire variable assignment, $\mathtt{v} := \mathcal{A}$, in the first rule, which turns variable assignments into empty lists, thereby erasing them. The rest of the rules distribute σ through the tree. For example, if we consider a tree $\tau = (\mathbf{true}, \mathtt{x} := 0, (\mathbf{false}, nbr(4)))$, its export is:

$$\sigma\tau = (\mathbf{true}, (\mathbf{false}, nbr(4)))$$

Neighbor Alignment. We define a partial substitution operator $T \triangleright \tau$, which is used to find exported values from neighbors that match the current memory context tree.

$$\begin{array}{rcll}
\square \triangleright nbr(\mathcal{L}) & \equiv & \square & \\
\square \triangleright \bullet & \equiv & \square & \\
(\mathcal{B}, \overline{\tau}, T, \overline{\tau}') \triangleright (\mathcal{B}, \overline{\tau}^a, \tau^b, \overline{\tau}^c) & \equiv & (\mathcal{B}, \overline{\tau}^a, T \triangleright \tau^b, \overline{\tau}^c) & \text{if } |\overline{\tau}| = |\overline{\tau}^a| \text{ and } |\overline{\tau}'| = |\overline{\tau}^c| \\
(\overline{\tau}, T, \overline{\tau}') \triangleright (\overline{\tau}^a, \tau^b, \overline{\tau}^c) & \equiv & (\overline{\tau}^a, T \triangleright \tau^b, \overline{\tau}^c) & \text{if } \tau_1, \tau_1^a \notin \mathcal{B} \text{ and } |\overline{\tau}| = |\overline{\tau}^a| \\
 & & & \qquad \text{and } |\overline{\tau}'| = |\overline{\tau}^c| \\
T \triangleright \tau & \equiv & \perp & \text{otherwise}
\end{array}$$

The first rule matches a hole with a state export: this will be used to create fields using values exported by neighbors. The second rule matches against the empty set, meaning that it is only testing for equivalent branching structure. This will be used to create fields of space-time metrics or local values. The next two rules walk the tree while checking that the relevant structure matches, and the last defines the failure case. The net effect is that if the hole in the memory context tree T matches with a state export in the target tree τ, then the result is a memory context tree obtained from τ by placing a hole exactly where it occurred in T. If the structure of the trees cannot be aligned, then the function yields no result (indicated as $\perp$). Although it is well defined for all trees, we will only apply it only to trees that have been filtered for export using σ. As long as the neighbors are evaluating the same expression (which is generally true in Proto), alignment will never product $\perp$, and thus we will assume that in the future. Some examples of its use:

$$\begin{array}{rcl}
(\mathbf{true}, \square, nbr(2)) \triangleright (\mathbf{true}, nbr(3), nbr(5)) & = & (\mathbf{true}, \square, nbr(5)) \\
(\mathbf{true}, \square) \triangleright (\mathbf{false}, nbr(3)) & = & \perp \\
(nbr(1), (\mathbf{true}, nbr(9)), \square) \triangleright (nbr(2), (\mathbf{false}), nbr(4)) & = & (nbr(2), (\mathbf{false}), \square) \\
(nbr(1), (\mathbf{true}, \square), nbr(7)) \triangleright (nbr(2), (\mathbf{false}), nbr(4)) & = & \perp
\end{array}$$

3.3.2. Configuration Syntax

Having established these preliminaries regarding memory trees, we now describe the full state of a Proto program on a device. This consists of the program expression itself, a store of memory, state sent to and from neighbors, metrics regarding neighbors, and sensor and actuator values.

The syntax we use to describe device state is as follows:

$$\begin{array}{rcll}
\mathtt{s} & ::= & \langle\Sigma\rangle\mathtt{e} & \text{Device state}\\
\Sigma & ::= & \bullet \mid T:\tau \mid \mathit{io}(\overline{\mathtt{v} := \mathcal{L}}) \mid \mathit{imp}(I) \mid \mathit{metric}(\mathcal{F}(\mathcal{V})) \mid (\Sigma \text{ "|" } \Sigma) & \text{Store}\\
I & ::= & \bullet \mid a \mapsto \tau \mid (I \text{ "|" } I) & \text{Import}
\end{array}$$

The state of a device is the expression `e` under evaluation and a store Σ keeping track of side-effects and interaction with the environment. The latter is a composition, by operator "|", of four elements:

- The memory $T : \tau$, is a memory tree, split into the sub-tree τ currently under consideration and the remainder as its memory context tree T. This is initialized to $\Box : \bullet$, and at the end of each evaluation becomes $\Box : \tau$, where τ is the full memory tree computed during that evaluation. This memory persists until the next evaluation, when $\Box : \tau$ will be the initial state and a new τ' computed, and so on.

- The sensor and actuator state $\mathit{io}(\overline{\mathtt{v} := \mathcal{L}})$ holds the sequence of values being received from sensors or sent to actuators. Each element in the sequence is a variable assignment associating the name of the sensor or actuator with its current input/output state.

- The imported messages $\mathit{imp}(I)$ are a set of associations $a \mapsto \tau$. There is one for each neighbor a, associating it with the most recent memory tree τ received from that neighbor.

- The space-time metrics $\mathit{metric}(\mathcal{F}(\mathcal{V}))$ are a collection of best-effort vectors describing the space-time displacement to each neighbor. These are used for resolving neighborhood metrics operations such as **`nbr-range`** and **`nbr-delay`**.

3.3.3. Evaluation Context

The final ingredient needed for our operational semantics is a representation of the current state of the evaluation itself. We identify the sub-expression of the program under consideration in the same fashion as we defined the sub-tree under consideration in the memory tree: an evaluation context `E` is an expression with one hole □ in it, representing the next place where a sub-expression needs to be evaluated. Likewise, we let notation `E`⟦`e`⟧ mean evaluation context `E` after substituting □ with expression `e`. Thus, the overall program expression $\mathtt{e}^p$ can always be represented as `E`⟦`e`⟧, where `e` is the current sub-expression under consideration.

Order of evaluation is set by the syntactic definition of evaluation contexts:

$$\mathtt{E} \;::=\; \square \mid (\mathtt{op}\ \overline{\mathcal{A}}\ \mathtt{E}\ \overline{\mathtt{e}}) \mid (\mathbf{if}\ \mathtt{E}\ \mathtt{e}\ \mathtt{e}) \mid (\mathbf{let}\ (\overline{(\mathtt{v}\ \mathcal{A})}\ (\mathtt{v}\ \mathtt{E})\ \overline{(\mathtt{v}\ \mathtt{e})})\ \overline{\mathtt{e}}) \mid (\mathbf{def}\ \mathtt{v}\ \mathtt{E})$$

This syntax sets a sequential order of evaluation for all ordinary operators `op`. In **`if`** expressions, only the test is unconditionally evaluated—there is a rule in the operational semantics that determines which branch expression actually gets evaluated. For local variable definition with **`let`**, the variable definitions are handled in sequential order, then left for rules in the operational semantics to store the assignments into the state; a similar thing is done for global variables created with **`def`**. All other expressions, such as **`rep`** and **`letfed`**, have their evaluation handled directly through rules in the operational semantics.

To see how evaluation context is used, consider the expression `(+ (- 5 1) (* (/ 6 2) 4))`. There are two possible matches of this expression with pattern `E`⟦`e`⟧: one that sets:

```
E = (+ □ (* (/ 6 2) 4))
e = (- 5 1)
```

and another that sets:

```
E = □
e = (+ (- 5 1) (* (/ 6 2) 4))
```

Of these two, however, only the first has operational semantics that can apply to evaluate it. This will be generally the case: there may be up to as many matches as there are levels of expression nesting, but all will chain to applying an evaluation rule on the innermost matching evaluation context.

In this case, the next evaluation step is thus to compute sub-expression `(- 5 1)`. Once that has been computed, the expression will be `(+ 4 (* (/`

`6 2) 4))` and the next effective match for $\mathtt{E}[\![\mathtt{e}]\!]$ with will be:

$$\begin{array}{rcl}\mathtt{E} & = & \mathtt{(+\ 4\ (*\ \Box\ 4))} \\ \mathtt{e} & = & \mathtt{(/\ 6\ 2)}\end{array}$$

3.4. Operational Semantics

Operational semantics is given by a transition system of the kind $(\mathbb{S}, \rightarrow)$, where $\mathbb{S}$ is the collection of possible device states $\mathtt{s}$ (as defined in Section 3.3.2) and $\rightarrow \subseteq \mathbb{S} \times \mathbb{S}$ is the transition relation. As is often done, we write $\mathtt{s} \rightarrow \mathtt{s}'$ as a shorthand for $(\mathtt{s}, \mathtt{s}') \in \rightarrow$, meaning that the device of interest moves from state $\mathtt{s}$ to state $\mathtt{s}'$ by an internal computation step. We will use $\mathtt{s} \rightarrow^* \mathtt{s}'$ to indicate the transitive closure of this relationship, meaning that there is some sequence of transitions leading from state $\mathtt{s}$ to state $\mathtt{s}'$.

We will describe the operational semantics by standard inference rules for formulas $s \rightarrow s'$, written using notation:

$$\frac{\textit{preconditions}}{\mathtt{s} \rightarrow \mathtt{s}'} \quad \text{[[RULE-NAME]]}$$

meaning that the transition $\mathtt{s} \rightarrow \mathtt{s}'$ applies whenever the preconditions can be satisfied. Intuitively, these preconditions can usually be thought of as sub-computations that implement the transition. Each rule covers a fragment of the transition relation, specifying the whole as their union.

In the model we present here, we consider a single execution round for a device, namely:

- As the initial state we consider $s = \langle \Sigma \rangle \mathtt{e}^p$. As defined above, $\mathtt{e}^p$ is the entire program and Σ is a store containing a memory tree (starting as $\Box : \overline{\tau}$ each evaluation), sensor and actuator state, imported state from neighbors, and space-time metrics information about neighbors.

- The computation then is evaluated on a step-by-step basis, with the transition rules of the operational semantics acting on the program expression $\mathtt{e}^p$ and possibly affecting/reading the store Σ, until the expression is completely evaluated to a value in $\mathcal{A}$—the result of the computation round on that device.

- At that point, the memory will be in state $\overline{\tau}, \Box : \bullet$, where $\overline{\tau}$ is the new memory tree, and $\sigma\overline{\tau}$ is the export data that will be shared with the device's neighbors.

We assume that it is the role of the platform to properly manage all parts of the store except for the memory tree, as well as restoring the initial expression at each computation round—this will be formalized in the next section. The operational semantics presented in this section will use information from the whole store, but modify only the memory tree and the current expression.

Rules of the operational semantics are provided in Figure 5 (meta-operations), Figure 6 (pointwise computation), Figure 8 (functions and variables), Figure 9 (branching), Figure 10 (stateful computation), and Figure 11 (neighborhood computation). These semantics and their interactions are explained in detail in the accompanying text.

$$\frac{\langle\Sigma'\rangle\mathtt{e} \to^* \langle\Sigma''\rangle\mathtt{e}'}{\langle\Sigma \mid \Sigma'\rangle\mathtt{e} \to \langle\Sigma \mid \Sigma''\rangle\mathtt{e}'} \qquad \text{[META-STORE]}$$

$$\frac{\langle\Sigma\rangle\mathtt{e} \to^* \langle\Sigma'\rangle\mathcal{A}}{\langle\Sigma\rangle\mathtt{E}[\![\mathtt{e}]\!] \to \langle\Sigma'\rangle\mathtt{E}[\![\mathcal{A}]\!]} \qquad \text{[CTX-SUB]}$$

$$\frac{\overline{\tau}' \neq \bullet}{\langle\Box : \tau, \overline{\tau}'\rangle\mathtt{e} \to \langle\Box, \overline{\tau}' : \tau\rangle\mathtt{e}} \qquad \text{[CTX-INIT]}$$

$$\frac{\overline{\tau}' \neq \bullet}{\langle T[\![\overline{\tau}, \Box, \overline{\tau}']\!] : \bullet\rangle\mathtt{e} \to \langle T[\![\overline{\tau}, \Box, tail_1(\overline{\tau}')]\!] : \tau_1'\rangle\mathtt{e}} \qquad \text{[CTX-ADVANCE]}$$

Figure 5: Operational Semantics of Proto language meta-operations.

3.4.1. Meta-Semantics

The meta-semantic rules presented here have two purposes: simplifying the notation of further rules, and keeping the memory context tree and evaluation context in alignment.

We start with rule [META-STORE], which is the counterpart of parallel composition rules, such as those in π-calculus. It says that when considering whether a system state moves from $\mathtt{s}$ to $\mathtt{s}'$, we can consider any sub-part Σ' of the overall store $\Sigma \mid \Sigma'$: if Σ' is moved to Σ'' by our semantics, then the overall store moves to $\Sigma \mid \Sigma''$. Intuitively, what this means is that the rest of our semantics only needs to mention the portions of the store relevant to the transition at hand.

Similarly, the rule [CTX-SUB] uses the evaluation context notation to allow us to specify only what happens with a sub-expression. It says that whenever there is a sub-expression e that can be evaluated in the current evaluation context, then the store changes from Σ to Σ' as prescribed by the evaluation of the sub-expression and result $\mathcal{A}$ of the evaluation can be substituted for e, as $\mathtt{E}[\![\mathcal{A}]\!]$. This means that we need only specify how top-level expressions are evaluated, and the rest will follow from [CTX-SUB].

The [CTX-INIT] and [CTX-ADVANCE] rules work together to ensure that the evaluation context and the memory context trees remain aligned. At any time during evaluation, the memory $T : \tau$ should have a non-sequence sub-tree selected as τ, the current sub-tree under consideration. This will be the next element of state to be updated, once the evaluation reaches its corresponding sub-expression.

The [CTX-INIT] rule sets this up by taking an initial state $\langle \Box : \tau, \overline{\tau}' \rangle$ and shifting the hole to the first element τ, producing the memory state $\Box, \overline{\tau}' : \tau$ (in the case where there is only one or zero elements in memory, no setup is required). After a piece of memory is updated, it will be replaced into the memory context tree, moving the hole forward and leaving no memory sub-tree under consideration—a state of the form $T[\![\overline{\tau}, \Box, \overline{\tau}']\!] : \bullet$. When this happens, rule [CTX-ADVANCE] shifts the next element of memory, τ'_1 (if it exists), to be the memory sub-tree under consideration.

In this way, the [CTX-*] meta-rules act as "glue" facilitating the computation and simplifying the notation of the semantic rules to follow. We will further illustrate the operation of these rules in the next section, once we have introduced some rules that actually compute.

$$\frac{\mathit{math}(\mathtt{pointwise}, \overline{\mathcal{A}}) = \mathcal{A}'}{\langle \Sigma \rangle (\mathtt{pointwise}\ \overline{\mathcal{A}}) \rightarrow \langle \Sigma \rangle \mathcal{A}'} \quad \text{[MATH]}$$

$$\frac{-}{\langle \mathit{io}(\mathtt{v} := \mathcal{L}) \rangle (\mathtt{v}) \rightarrow \langle \mathit{io}(\mathtt{v} := \mathcal{L}) \rangle \mathcal{L}} \quad \text{[SENSE]}$$

$$\frac{\mathcal{L}'' = (\text{if } |\overline{\mathcal{L}}'| = 1 \text{ then } \mathcal{L}'_1 \text{ else } \mathcal{T}(\overline{\mathcal{L}'}))}{\langle \{\mathit{io}(\mathtt{v} := \overline{\mathcal{L}})\} \otimes \Sigma \rangle (\mathtt{v}\ \overline{\mathcal{L}'}) \rightarrow \langle \mathit{io}(\mathtt{v} := \mathcal{L}'') \otimes \Sigma \rangle \mathcal{L}'_1} \quad \text{[ACT]}$$

Figure 6: Operational Semantics of Proto language pointwise operations.

$math(\texttt{pointwise}, \overline{\mathcal{A}})$	$\mathcal{A}'$	$math(\texttt{pointwise}, \overline{\mathcal{A}})$	$\mathcal{A}'$
all, $\overline{\mathcal{A}}$	$\mathcal{A}_{\lvert\overline{\mathcal{A}}\rvert}$	**tup**, $\overline{\mathcal{A} - \mathcal{O}}$	$\mathcal{T}(\overline{\mathcal{A} - \mathcal{O}})$
not, $\mathcal{B}$	$\neg\mathcal{B}$	**elt**, $\mathcal{T}(\overline{\mathcal{A} - \mathcal{O}}), \mathcal{S}$	$\mathcal{A}_{\mathcal{S}+1}$
+, $\overline{\mathcal{N}}$	$\mathcal{N}_1 + \mathcal{N}_2 + \cdots + \mathcal{N}_{\lvert\overline{\mathcal{N}}\rvert}$	**-**, $\mathcal{N}, \overline{\mathcal{N}}$	$\mathcal{N}_1 - \mathcal{N}_2 - \cdots - \mathcal{N}_{\lvert\overline{\mathcal{N}}\rvert}$
*****, $\overline{\mathcal{S}}, \mathcal{N}$	$\mathcal{S}_1 * \mathcal{S}_2 * \cdots * \mathcal{S}_{\lvert\overline{\mathcal{S}}\rvert} * \mathcal{N}$	**/**, $\mathcal{N}, \overline{\mathcal{S}}$	$\mathcal{N}/\mathcal{S}_1/\mathcal{S}_2/\ldots/\mathcal{S}_{\lvert\overline{\mathcal{S}}\rvert}$
<, $\mathcal{N}, \mathcal{N}'$	$\mathcal{N} < \mathcal{N}'$	**>**, $\mathcal{N}, \mathcal{N}'$	$\mathcal{N} > \mathcal{N}'$
=, $\mathcal{N}, \mathcal{N}'$	$\mathcal{N} = \mathcal{N}'$	**<=**, $\mathcal{N}, \mathcal{N}'$	$\mathcal{N} <= \mathcal{N}'$
>=, $\mathcal{N}, \mathcal{N}'$	$\mathcal{N} >= \mathcal{N}'$	**min**, $\overline{\mathcal{N}}$	$\mathcal{N}_i$ s.t. $\forall_{\mathcal{N}_j \in \overline{\mathcal{N}}} \mathcal{N}_i <= \mathcal{N}_j$
max, $\overline{\mathcal{N}}$	$\mathcal{N}_i$ s.t. $\forall_{\mathcal{N}_j \in \overline{\mathcal{N}}} \mathcal{N}_i >= \mathcal{N}_j$	**abs**, $\mathcal{S}$	$\lvert\mathcal{S}\rvert$
floor, $\mathcal{S}$	$\lfloor\mathcal{S}\rfloor$	**ceil**, $\mathcal{S}$	$\lceil\mathcal{S}\rceil$
round, $\mathcal{S}$	$\lfloor\mathcal{S} + 0.5\rfloor$	**pow**, $\mathcal{S}, \mathcal{S}'$	$\mathcal{S}^{\mathcal{S}'}$
sqrt, $\mathcal{S}$	$\sqrt{\mathcal{S}}$	**log**, $\mathcal{S}$	$\ln\mathcal{S}$
mod, $\mathcal{S}, \mathcal{S}'$	$\mathcal{S} \bmod \mathcal{S}'$	**rnd**, $\mathcal{S}, \mathcal{S}'$	Random number in $[\mathcal{S}, \mathcal{S}']$
sin, $\mathcal{S}$	$\sin\mathcal{S}$	**cos**, $\mathcal{S}$	$\cos\mathcal{S}$
tan, $\mathcal{S}$	$\tan\mathcal{S}$	**asin**, $\mathcal{S}$	$\sin^{-1}\mathcal{S}$
acos, $\mathcal{S}$	$\cos^{-1}\mathcal{S}$	**atan2**, $\mathcal{S}, \mathcal{S}'$	$\tan^{-1}(\mathcal{S}/\mathcal{S}')$
sinh, $\mathcal{S}$	$\sinh\mathcal{S}$	**cosh**, $\mathcal{S}$	$\cosh\mathcal{S}$
tanh, $\mathcal{S}$	$\tanh\mathcal{S}$	**len**, $\mathcal{T}$	$\lvert\mathcal{T}\rvert$
vdot, $\mathcal{V}, \mathcal{V}'$	$\mathcal{V} \cdot \mathcal{V}'$	**mux**, $\mathcal{B}, [\mathcal{A} - \mathcal{O}], [\mathcal{A} - \mathcal{O}]'$	(if $\mathcal{B}$ then $[\mathcal{A} - \mathcal{O}]$ else $[\mathcal{A} - \mathcal{O}]'$)

Figure 7: Values computed by Proto pointwise operators.

3.4.2. Pointwise Computation

All ordinary pointwise computation is handled through the single rule [MATH]. Since this rule is not enabled until all of the arguments $\overline{\mathcal{A}}$ have been evaluated, [MATH] does not interact with the store at all. Therefore the store is indicated by the most generic form Σ and is unaffected by the transition.

The result expression itself is computed from the arguments according to the appropriate element of relation *math*, as defined in Figure 7. Almost all of the current Proto pointwise operators implement simple mathematical operations, such as addition and trigonometry. A few non-standard operators worth noting:

- **all** takes an arbitrary number of arguments and returns the last.
- **mux** is a multiplexer operation, which first evaluates the three arguments, then returns the value of its second argument if the first is **true**, and the value of its third argument if the first is **false**.

- When comparators like $>$ are applied to vectors, they compare them lexicographically and pad with zeros to make them the same length. Thus, for example, the vector $(1, 2, 3)$ is greater than the vector $(0, 5)$.

Now let us consider how the set of rules defined so far apply together to evaluate a pointwise expression, such as `(+ (- 5 1) (* (/ 6 2) 4))`. We begin with an empty memory and the raw expression:

$$\langle\Box : \bullet\rangle(\texttt{+}\ (\texttt{-}\ 5\ 1)\ (\texttt{*}\ (\texttt{/}\ 6\ 2)\ 4))$$

The [CTX-SUB] rule can apply, matching `e` to $(\texttt{-}\ 5\ 1)$ and applying rule [MATH] to evaluate recursively inside of the preconditions:

$$\dfrac{\dfrac{\ldots}{\langle\Box : \bullet\rangle(\texttt{-}\ 5\ 1) \rightarrow \langle\Box : \bullet\rangle 4}\ [\text{MATH}]}{\langle\Box : \bullet\rangle(\texttt{+}\ (\texttt{-}\ 5\ 1)\ (\texttt{*}\ (\texttt{/}\ 6\ 2)\ 4)) \rightarrow \langle\Box : \bullet\rangle(\texttt{+}\ 4\ (\texttt{*}\ (\texttt{/}\ 6\ 2)\ 4))}\ [\text{CTX-SUB}]$$

Now that the first term has been evaluated, the [CTX-SUB] rule can apply again: the innermost match for it sets `e` to $(\texttt{/}\ 6\ 2)$, again applying rule [MATH] to evaluate in the preconditions:

$$\dfrac{\dfrac{\ldots}{\langle\Box : \bullet\rangle(\texttt{/}\ 6\ 2) \rightarrow \langle\Box : \bullet\rangle 3}\ [\text{MATH}]}{\langle\Box : \bullet\rangle(\texttt{+}\ 4\ (\texttt{*}\ (\texttt{/}\ 6\ 2)\ 4)) \rightarrow \langle\Box : \bullet\rangle(\texttt{+}\ 4\ (\texttt{*}\ 3\ 4))}\ [\text{CTX-SUB}]$$

Similarly, one more stage of such evaluation brings us to $\langle\Box : \bullet\rangle(\texttt{+}\ 4\ 12)$, which moves to $\langle\Box : \bullet\rangle 16$ by another evaluation directly invoking the [MATH] rule. The final result of this evaluation is the **Scalar** value 16. This is, in fact, the definitive and only result, although there is generally some variation in how [CTX-*] rules may be applied to reach this result.

All of our other semantic rules will operate in a similar manner to [MATH] (except that they will interact with the store in various ways), ensuring a deterministic progression of the evaluation. This is important because we need to make sure that the memory tree is aligned from round to round and from device to device.

For example, sensors and actuators are handled almost identically to pointwise operators, except that they read and write values from sensor and actuator state, respectively. Rule [SENSE] reads sensor `v` by finding its value assignment $io(\texttt{v} := \mathcal{L})$ from the store, returning this value and leaving the store intact. Rule [ACT] uses the form $\{io(\texttt{v} := \overline{\mathcal{L}})\} \otimes \Sigma$ to remove whatever prior information on actuator `v` might exist in the store. It then assigns `v` to the values it has been passed (bundled into a **Tuple** if there is more than one). The value of the first argument to the actuator is passed through as a return value—a useful feature as it allows debugging actuators to be "wrapped" around expressions without having to create a **`let`** variable. The set of "universal" sensors and actuators required to be best-effort supported on every Proto platform are defined below in Table 2.

$$\frac{-}{\langle T[\![\mathtt{v} := \mathcal{A}, \Box]\!] : \tau\rangle\mathtt{v} \rightarrow \langle T[\![\mathtt{v} := \mathcal{A}, \Box]\!] : \tau\rangle\mathcal{A}} \quad \text{[VAR-FOUND]}$$

$$\frac{\tau' \neq \mathtt{v} := \mathcal{A} \qquad \langle T[\![\overline{\tau}, \Box, \tau', \overline{\tau}'']\!] : \bullet\rangle\mathtt{v} \rightarrow^* \langle T[\![\overline{\tau}, \Box, \tau', \overline{\tau}'']\!] : \bullet\rangle\mathcal{A}}{\langle T[\![\overline{\tau}, \tau', \Box, \overline{\tau}'']\!] : \tau'''\rangle\mathtt{v} \rightarrow \langle T[\![\overline{\tau}, \tau', \Box, \overline{\tau}'']\!] : \tau'''\rangle\mathcal{A}} \quad \text{[VAR-LOOK-BACK]}$$

$$\frac{\langle T : \bullet\rangle\mathtt{v} \rightarrow^* \langle T : \bullet\rangle\mathcal{L}}{\langle T[\![(\Box, \overline{\tau})]\!] : \tau'\rangle\mathtt{v} \rightarrow \langle T[\![(\Box, \overline{\tau})]\!] : \tau'\rangle\mathcal{L}} \quad \text{[VAR-LOOK-UP]}$$

$$\frac{\langle T : \bullet\rangle\mathtt{v} \rightarrow^* \langle T : \bullet\rangle\mathcal{F} \qquad \mathtt{I} = \{\overline{a} \mapsto \sigma T[\![(\Box, \overline{\tau})]\!] \triangleright \overline{\tau}''\} \otimes \mathtt{I}' \qquad \mathcal{F}' = \{\overline{a} \mapsto \overline{\mathcal{L}}\} \otimes \mathcal{F}}{\langle \mathtt{I} \mid T[\![(\Box, \overline{\tau})]\!] : \tau'\rangle\mathtt{v} \rightarrow \langle \mathtt{I} \mid T[\![(\Box, \overline{\tau})]\!] : \tau'\rangle\mathcal{F}'} \quad \text{[VAR-LOOK-UP-FIELD]}$$

$$\frac{-}{\langle T[\![\overline{\tau}, \Box, \overline{\tau}']\!] : \tau''\rangle(\mathbf{def}\ \mathtt{v}\ \mathcal{A}) \rightarrow \langle T[\![\overline{\tau}, \mathtt{v} := \mathcal{A}, \Box, \overline{\tau}']\!] : \bullet\rangle\mathcal{A}} \quad \text{[DEF-VAR]}$$

$$\frac{\begin{array}{c}\tau' = \{\overline{\mathtt{v}} := \overline{\mathcal{A}}\} \qquad (\text{if } \tau = \bullet \text{ then } \overline{\tau}'' = \bullet \text{ else } \tau = (\overline{\tau}'')) \\ \langle T[\![(\tau', \Box, tail_{|\overline{\mathtt{v}}|}(\overline{\tau}''))]\!] : \bullet\rangle(\mathbf{all}\ \overline{\mathtt{e}}) \rightarrow^* \langle T[\![(\overline{\tau}''', \Box)]\!] : \bullet\rangle\mathcal{A}\end{array}}{\langle T : \tau\rangle(\mathbf{let}\ (\{(\overline{\mathtt{v}}\ \overline{\mathcal{A}})\})\ \overline{\mathtt{e}}) \rightarrow \langle T[\![(\overline{\tau}'''), \Box]\!] : \bullet\rangle\mathcal{A}} \quad \text{[LET]}$$

$$\frac{\mathcal{O} = (\overline{\mathtt{arg}}, (\mathbf{all}\ \overline{\mathtt{e}}))}{\langle T[\![\overline{\tau}, \Box, \overline{\tau}']\!] : \tau''\rangle(\mathbf{def}\ \mathtt{v}\ (\overline{\mathtt{arg}})\ \overline{\mathtt{e}}) \rightarrow \langle T[\![\overline{\tau}, \mathtt{v} := \mathcal{O}, \Box, \overline{\tau}']\!] : \bullet\rangle\mathcal{O}} \quad \text{[DEF-FUN]}$$

$$\frac{\mathcal{O} = (\overline{\mathtt{arg}}, (\mathbf{all}\ \overline{\mathtt{e}}))}{\langle \Sigma\rangle(\mathbf{fun}\ (\overline{\mathtt{arg}})\ \overline{\mathtt{e}}) \rightarrow \langle \Sigma\rangle\mathcal{O}} \quad \text{[ANON-FUN]}$$

$$\frac{\langle \Sigma\rangle\mathtt{e} \rightarrow^* \langle \Sigma\rangle\mathcal{O} \qquad \mathcal{O} = (\overline{\mathtt{arg}}, \mathtt{e}')}{\langle \Sigma\rangle(\mathtt{e}\ \overline{\mathcal{A}}) \rightarrow \langle \Sigma\rangle\mathtt{e}'[\overline{\mathcal{A}}/\overline{\mathtt{arg}}]} \quad \text{[APPLY-FUN]}$$

Figure 8: Operational Semantics of Proto language function and variable operations.

3.4.3. Functions and Variables

Next we consider functions and variables, both defining them and using them later. The general pattern is the same for both: definitions add a variable to the memory tree, usages search backward and up through the memory tree to find the nearest matching variable definition.[4]

Let us begin by considering variable lookup. The rules [VAR-*] implement recursive and lexically scoped retrieval of variable assignments within the memory tree, without affecting its contents. Rule [VAR-FOUND] is the terminal case: given a variable `v`, the leaf immediately before the hole is the assignment $\mathtt{v} := \mathcal{A}$, so the result is $\mathcal{A}$ (recall that τ matches an ordered set as well as a single element). In any other case, the rules [VAR-LOOK-BACK], [VAR-LOOK-UP], and [VAR-LOOK-UP-FIELD] recursively walk the hole through the tree until [VAR-FOUND] can apply. Lexical scoping is implemented by making [VAR-LOOK-BACK] dominate, such that the search does not descend into sub-trees and only raises the hole to a super-tree when it cannot move backward. The [VAR-LOOK-UP] and [VAR-LOOK-UP-FIELD] rules only differ in that [VAR-LOOK-UP-FIELD] performs filtering on **Fields**, such that a reference ultimately retrieves only those neighbors that align with the location where the variable is referenced (i.e., were produced by the same expression at the neighbor).

Given this, rule [DEF-VAR] is the simplest case of creating a binding. It simply takes a variable name `v` and a value $\mathcal{A}$ and inserts a variable assignment $\mathtt{v} := \mathcal{A}$ into the store (replacing the prior definition, if such existed). The value $\mathcal{A}$ is returned.

Rule [LET] does the same for a batch of assignments, $\{(\overline{\mathtt{v}}\ \overline{\mathcal{A}})\}$, but with the additional feature that the bindings are placed into a sub-tree, effectively limiting their scope to the body of the **`let`** expression. Note that in the preconditions we make expressions **`all`** $\overline{\mathtt{e}}$ evolve under a sub-tree where first element contains all n assignments, the second one is the hole, and the remainder is formed by those subtrees for the body that remained of τ after skipping n elements.

Function definition via the [DEF-FUN] rule is the same as variable definition, except that the value $\mathcal{O}$ is created by binding together the unevaluated expressions for the arguments $\overline{\mathtt{arg}}$ and the body of the function $\overline{\mathtt{e}}$. The body is ensured to be a single expression by wrapping it with an **`all`**. Just as for [DEF-VAR], the [DEF-FUN] rule adds a variable assignment binding $\mathtt{v} := \mathcal{O}$ into the memory tree, and returns the value $\mathcal{O}$. Anonymous function defini-

[4]In actual implementation, the number of lookups made at run-time can be greatly reduced by resolving as much as possible at compile-time.

tion is the same, except that no binding is produced. As far as comparison with functional languages is concerned, note that in these semantics definition of functions always occur in an empty environment, that is, there are no lexically visible variables the function can access, except for its formal arguments. Recent extension in [30] allows closures in Proto, and is a target for future extension of these semantics.

Finally, rule [APPLY-FUN] provides semantics for using functions. It searches for the proper definition of function `e`, then returns the retrieved body $\mathtt{e}'$ after substituting formal parameters $\overline{\mathtt{arg}}$ with their actual values $\overline{\mathcal{A}}$. Note that functions as first-class objects pose some significant challenges in distributed computation (for example, assuring that an equivalent tree was generated by an equivalent function–see discussion in Section 5), so Proto in its present form only allows functions to be passed within scopes where it can be assured that every device is running the same function. Thus, **`mux`** and tuple manipulation exclude functions, as will the [IF] rule defined in the next section. Likewise, we assume that the compiler also checks to see that all variables referenced by the function will have values accessible to it in each of its applications.

For an example of using these rules, let us consider the expression

$$\langle \Box : \bullet \rangle (\mathbf{let}\ ((\mathtt{x}\ 1))\ (\mathtt{+}\ \mathtt{x}\ \mathtt{x}))$$

that matches rule [LET], which binds `x` to 1, and then proceeds to evaluate the body of the **`let`**:

$$\frac{\langle (x := 1, \Box) : \bullet \rangle (\mathbf{all}\ (\mathtt{+}\ \mathtt{x}\ \mathtt{x})) \to^{*} \ldots}{\langle \Box : \bullet \rangle (\mathbf{let}\ ((\mathtt{x}\ 1))\ (\mathtt{+}\ \mathtt{x}\ \mathtt{x})) \to \ldots}\ \text{[LET]}$$

To solve the top part of the rule, [CTX-SUB] matches and begins evaluation of the sub-expression `x`, which is resolved by [VAR-FOUND].

$$\frac{\dfrac{\ldots}{\langle (x := 1, \Box) : \bullet \rangle \mathtt{x} \to \langle (x := 1, \Box) : \bullet \rangle 1}\ \text{[VAR-FOUND]}}{\langle (x := 1, \Box) : \bullet \rangle (\mathbf{all}\ (\mathtt{+}\ \mathtt{x}\ \mathtt{x})) \to \langle (x := 1, \Box) : \bullet \rangle (\mathbf{all}\ (\mathtt{+}\ 1\ \mathtt{x}))}\ \text{[CTX-SUB]}$$

Another round of [CTX-SUB] and [VAR-FOUND] proceeds similarly:

$$\frac{\dfrac{\ldots}{\langle (x := 1, \Box) : \bullet \rangle \mathtt{x} \to \langle (x := 1, \Box) : \bullet \rangle 1}\ \text{[VAR-FOUND]}}{\langle (x := 1, \Box) : \bullet \rangle (\mathbf{all}\ (\mathtt{+}\ 1\ \mathtt{x})) \to \langle (x := 1, \Box) : \bullet \rangle (\mathbf{all}\ (\mathtt{+}\ 1\ 1))}\ \text{[CTX-SUB]}$$

At that point simple application of rule [MATH] leads to $\langle (x := 1, \Box) : \bullet \rangle 2$. Hence, the application of [LET] rule returns the result and places the hole back on the level where it previously was:

$$\frac{\langle (x := 1, \Box) : \bullet \rangle (\texttt{all}\ (\texttt{+}\ \texttt{x}\ \texttt{x})) \rightarrow^* \langle (x := 1, \Box) : \bullet \rangle 2}{\langle \Box : \bullet \rangle (\texttt{let}\ ((\texttt{x}\ 1))\ (\texttt{+}\ \texttt{x}\ \texttt{x})) \rightarrow \langle (x := 1), \Box : \bullet \rangle 2}\ [\text{LET}]$$

$$\frac{\overline{\tau}'' = (\text{if } \tau = (\mathcal{B}, \overline{\tau}') \text{ and } \mathcal{B} = \mathcal{B}' \text{ then } \overline{\tau}' \text{ else } \bullet) \qquad \texttt{e}'' = (\text{if } \mathcal{B}' \text{ then } \texttt{e} \text{ else } \texttt{e}') \\ \langle T[\![(\mathcal{B}', \Box, tail_1(\overline{\tau}''))]\!] : \tau_1'' \rangle \texttt{e}'' \rightarrow^* \langle T[\![(\mathcal{B}', \overline{\tau}''', \Box)]\!] : \bullet \rangle \mathcal{L} - \mathcal{O}}{\langle T : \tau \rangle (\texttt{if}\ \mathcal{B}'\ \texttt{e}\ \texttt{e}') \rightarrow \langle T[\![(\mathcal{B}', \overline{\tau}'''), \Box]\!] : \bullet \rangle \mathcal{L} - \mathcal{O}}\ [\text{IF}]$$

Figure 9: Operational Semantics of Proto language branching operation.

3.4.4. Branching

We now begin to consider operations that manipulate space and time. In the manifold abstraction of Proto, branching is modeled as a restriction of the space where a computation runs. What this means for our semantics is that rule [IF] needs to not share memory between two different branches, either from execution to execution or from device to device.

Thus, rule [IF] produces a memory tree of the kind $(\mathcal{B}, \overline{\tau}')$, where $\mathcal{B}$ is the truth value of first argument, and $\overline{\tau}'$ is the sub-tree generated by the evaluation of second or third argument. First, [IF] checks whether the sub-tree in the previous round has the same truth value as the current one. If this is the case, the existing sub-tree $\overline{\tau}'$ is considered for next computation; otherwise we use $\bullet$, discarding the old tree and building a new one from scratch. Then, we evaluate the second or third argument, depending on the value of $\mathcal{B}$, to get a **Local** that is not a function, creating replacement sub-tree $\overline{\tau}'''$. Finally, the hole is raised back to the level where it began.

Let us illustrate [IF] with an example, evaluating the expression `(if true (+ 1 (if false 2 3)) (if true 4 5))` against the memory $\Box : (\texttt{false}, (\texttt{true}))$. Since the test is already a literal value, we can proceed directly to the first application of [IF], which recurses into an evaluation of the true expression:

$$\frac{\langle (\texttt{true}, \Box) : \bullet \rangle (\texttt{+}\ 1\ (\texttt{if}\ \texttt{false}\ 2\ 3)) \rightarrow^* \ldots}{\langle \Box : (\texttt{false}, (\texttt{true})) \rangle (\texttt{if}\ \texttt{true}\ (\texttt{+}\ 1\ (\texttt{if}\ \texttt{false}\ 2\ 3))\ (\texttt{if}\ \texttt{true}\ 4\ 5)) \rightarrow \ldots}\ [\text{IF}]$$

Notice that the $(\texttt{true})$ portion of the state has been discarded because the **true** branch is taken, while the prior state is associated with the **false** branch.

The top is resolved by application of rules [CTX-SUB] and [IF] as follows

$$\dfrac{\dfrac{\ldots}{\langle(\texttt{true},\Box):\bullet\rangle(\texttt{if false } 2\ 3) \to \langle(\texttt{true},\Box):\bullet\rangle 3}\,[\text{IF}]}{\langle(\texttt{true},\Box):\bullet\rangle(\texttt{+ } 1\ (\texttt{if false } 2\ 3)) \to \langle(\texttt{true},\Box):\bullet\rangle(\texttt{+ } 1\ 3)}\,[\text{CTX-SUB}]$$

which is then followed by a direct application of [MATH] which leads to $\langle(\texttt{true},\Box):\bullet\rangle 4$. Hence, the top-level application of [IF] leads to final state $\langle(\texttt{true},(\texttt{false}),\Box):\bullet\rangle 4$.

$$\dfrac{\langle T[\![(\Box)]\!]:\bullet\rangle \mathtt{e} \to^* \langle T[\![(\overline{\tau},\Box)]\!]\rangle \mathcal{L}}{\langle T:\bullet\rangle(\texttt{rep v e e}') \to \langle T[\![(\texttt{false},\mathtt{v}:=\mathcal{L},(\overline{\tau})),\Box]\!]:\bullet\rangle\mathcal{L}} \quad [\text{REP-INIT}]$$

$$\dfrac{\overline{\tau}' = (\text{if } \mathcal{B} = \texttt{true} \text{ then } \overline{\tau} \text{ else } \bullet) \qquad \langle T[\![(\mathcal{B},\mathtt{v}:=\mathcal{L},\Box)]\!]:\overline{\tau}'\rangle \mathtt{e}' \to^* \langle T[\![(\mathcal{B},\mathtt{v}:=\mathcal{L},\overline{\tau}'',\Box)]\!]:\bullet\rangle\mathcal{L}'}{\langle T:(\mathcal{B},\mathtt{v}:=\mathcal{L},(\overline{\tau}))\rangle(\texttt{rep v e e}') \to \langle T[\![(\texttt{true},\mathtt{v}:=\mathcal{L}',(\overline{\tau}'')),\Box]\!]:\bullet\rangle\mathcal{L}'} \quad [\text{REP-UPDATE}]$$

$$\dfrac{\begin{array}{c}\forall_{\mathtt{v}_i}\langle T[\![(\Box)]\!]:\bullet\rangle \mathtt{e}_i \to^* \langle T[\![(\tau_i',\Box)]\!]:\bullet\rangle\mathcal{L}_i \qquad \overline{\tau} = \{\overline{\mathtt{v}} := \overline{\mathcal{L}}\} \\ \langle T[\![(\texttt{false},\overline{\tau},\{(\overline{\tau}')\},\Box)]\!]:\bullet\rangle(\texttt{all } \overline{\mathtt{e}}'') \to^* \langle T'[\![(\overline{\tau}'',\Box)]\!]:\bullet\rangle\mathcal{L}'\end{array}}{\langle T:\bullet\rangle(\texttt{letfed } (\{(\overline{\mathtt{v}}\ \overline{\mathtt{e}}\ \overline{\mathtt{e}}')\})\ \overline{\mathtt{e}}'') \to \langle T'[\![(\overline{\tau}''),\Box]\!]:\bullet\rangle\mathcal{L}'} \quad [\text{LETFED-INIT}]$$

$$\dfrac{\begin{array}{c}\overline{\tau} = \{\overline{\mathtt{v}} := \overline{\mathcal{L}}\} \qquad |\overline{\tau}'| = |\overline{v}| \qquad \overline{\tau}^p = (\text{if } \mathcal{B} = \texttt{true} \text{ then } \overline{\tau} \text{ else } \bullet) \\ \forall_{\mathtt{v}_i}\langle T[\![(\overline{\tau}^p,\Box,\tau_i')]\!]:\bullet\rangle \mathtt{e}_i' \to^* \langle T[\![(\overline{\tau}^p,\tau_i^u,\Box)]\!]:\bullet\rangle\mathcal{L}_i' \qquad \overline{\tau}^v = \{\overline{\mathtt{v}} := \overline{\mathcal{L}}'\} \\ \langle T[\![(\texttt{true},\overline{\tau}^v,\{(\overline{\tau}^u)\},\Box,\overline{\tau}''):\bullet]\!]\rangle(\texttt{all } \overline{\mathtt{e}}'') \to^* \langle T'[\![(\overline{\tau}^n,\Box)]\!]:\bullet\rangle\mathcal{L}''\end{array}}{\langle T:(\mathcal{B},\overline{\tau},\{(\overline{\tau}')\},\overline{\tau}'')\rangle(\texttt{letfed } (\{(\overline{\mathtt{v}}\ \overline{\mathtt{e}}\ \overline{\mathtt{e}}')\})\ \overline{\mathtt{e}}'') \to \langle T'[\![(\overline{\tau}^n),\Box]\!]:\bullet\rangle\mathcal{L}''} \quad [\text{LETFED-UPDATE}]$$

Figure 10: Operational Semantics of Proto language state operations.

3.4.5. Stateful Computation

State in Proto is established with state evolution functions, which assign a variable to some initial value, then update this value in each following round of evaluation.

We begin with the [REP-*] rules, which establish a single state variable and return its value. The **rep** function is actually a macro of **letfed**, with form:

$$(\texttt{rep v e e}') \equiv (\texttt{letfed } ((\mathtt{v\ e\ e}'))\ \mathtt{v})$$

We present it with its own rules here, however, to show a simpler case of [LETFED] before generalizing to the full complexity.

The [REP-*] rule acts like a combination of [IF] and [LET], creating a sub-tree of the form $(\mathcal{B}, \mathtt{v} := \mathcal{L}, (\overline{\tau}))$, where $\mathtt{v} := \mathcal{L}$ is the current state, $\mathcal{B}$ is false for the first round of execution and true from then on, and $\overline{\tau}$ is the tree computed out of evaluation of the second or third argument. Rule [REP-INIT] handles the first round, when there is no prior state. In this case, we evaluate the second argument $\mathtt{e}$ and produce the output tree $(\mathtt{\textbf{false}}, \mathtt{v} := \mathcal{L}, (\overline{\tau}))$, moving the hole forward as we do so. Rule [REP-UPDATE] handles subsequent rounds. The third argument $\mathtt{e}'$ is evaluated, starting from an empty sub-tree if $\mathcal{B}$ is false. Note that this evaluation is made using a memory tree that includes the previous variable assignment $\mathtt{v} := \mathcal{L}$, such that the prior value can be used in the computation. The output is similar to [REP-INIT], though here we set the new truth value to **`true`**.

We can now view the [LETFED-*] rules as a generalization of the [REP-*] rules. Rather than binding one state variable, a sequence of state variables are bound, followed by a sequence of sub-trees containing the state produced by computing them (this is the reason that [REP-*] put the $\overline{\tau}$ into its own sub-tree). Note that each state variable computation has access to all of the prior values of all of the other state variables. Finally, rather than return a state variable, the [LETFED-*] rules return an arbitrary computation that has access to the newly values of the state variables.

3.4.6. Neighborhood Computation

Finally, we consider the *raison d'etre* for Proto: computation over neighborhood values. We begin with the three methods of obtaining **Field** values: [NBR], [METRIC], and [LOCAL].

Rule [NBR] is used for gathering a value from neighbors (and sharing the device's own in return). Let I be the set of imports for this device (the latest messages that neighbours sent containing the exports of their memory trees, including the last round's export from this device). Out of these, we isolate those whose tree matches with $(\sigma T \triangleright \overline{\tau}')[\![nbr(\overline{\mathcal{L}}')]\!]$, namely, those in which there is a corresponding **`nbr`** construct which resides in the same sub-tree. We let $\overline{a}$ be the corresponding neighbours and $\overline{\mathcal{L}}'$ the values they computed for this **`nbr`** construct: these are used to create the **Field** value that is returned as output. A device is considered part of its own neighborhood, so value $a' \mapsto \mathcal{L}$ is also inserted into the **Field**, letting a' be the device where

$$\frac{\mathtt{I} = \{\overline{a} \mapsto (\sigma T \triangleright \overline{\tau}')[\![nbr(\overline{\mathcal{L}}')]\!]\} \otimes \mathtt{I}' \qquad a' = \mathit{self}}{\langle T : \tau \otimes \mathit{imp}(\mathtt{I})\rangle(\mathbf{nbr}\ \mathcal{L}) \rightarrow \langle T[\![nbr(\mathcal{L}), \Box]\!] \otimes \mathit{imp}(\mathtt{I})\rangle\{a' \mapsto \mathcal{L} \mid \overline{a} \mapsto \overline{\mathcal{L}}'\}} \quad \text{[NBR]}$$

$$\frac{\mathtt{I} = \{\overline{a} \mapsto \sigma T \triangleright \overline{\tau}\} \otimes \mathtt{I}' \qquad \{\overline{\mathcal{L}} = m(\mathtt{metric}, \overline{\mathcal{V}})\}}{\langle \mathit{imp}(\mathtt{I}) \otimes \Sigma \mid \mathit{metric}(\{\overline{a} \mapsto \overline{\mathcal{V}}\})\rangle(\mathtt{metric}) \rightarrow \langle \mathit{imp}(\mathtt{I}) \otimes \Sigma \mid \mathit{metric}(\{\overline{a} \mapsto \overline{\mathcal{V}}\})\rangle\{\overline{a} \mapsto \overline{\mathcal{L}}\}} \quad \text{[METRIC]}$$

$$\frac{\mathtt{I} = \{\overline{a} \mapsto \sigma T \triangleright \overline{\tau}\} \otimes \mathtt{I}'}{\langle \mathit{imp}(\mathtt{I}) \otimes \Sigma\rangle(\mathbf{nbr}\ \mathcal{L}) \rightarrow \langle \mathit{imp}(\mathtt{I}) \otimes \Sigma\rangle\{\overline{a} \mapsto \mathcal{L}\}} \quad \text{[LOCAL]}$$

$$\frac{\mathcal{F}_j = \{a_i \mapsto \mathcal{L}_{i,j}\} \otimes \mathcal{F}' \qquad \forall_{a_i} \langle\Sigma\rangle(\mathtt{pointwise}\ \overline{\mathcal{L}}_i) \rightarrow^* \langle\Sigma\rangle\mathcal{L}'_i}{\langle\Sigma\rangle(\mathtt{pointwise}\ \overline{\mathcal{F}}) \rightarrow \langle\Sigma\rangle\{\overline{a} \mapsto \overline{\mathcal{L}}'\}} \quad \text{[FIELD-MATH]}$$

$$\frac{\mathcal{O} = (\overline{\mathtt{arg}}, \mathtt{e}) \qquad \langle\Sigma\rangle\mathtt{e}[\mathcal{L}, \mathcal{L}'/\overline{\mathtt{arg}}] \rightarrow^* \langle\Sigma\rangle\mathcal{L}''}{\langle\Sigma\rangle(\mathbf{fold\text{-}hood*}\ \mathcal{O}\ \mathcal{L}\ \{a \mapsto \mathcal{L}'\} \otimes \mathcal{F}) \rightarrow \langle\Sigma\rangle(\mathbf{fold\text{-}hood*}\ \mathcal{O}\ \mathcal{L}''\ \mathcal{F})} \quad \text{[FHOOD-R]}$$

$$\frac{-}{\langle\Sigma\rangle(\mathbf{fold\text{-}hood*}\ \mathcal{O}\ \mathcal{L}\ \bullet) \rightarrow \langle\Sigma\rangle\mathcal{L}} \quad \text{[FHOOD-F]}$$

Figure 11: Operational Semantics of Proto language neighborhood operations.

the evaluation is taking place (denoted here as the symbolic name *self*, which is some member of the set of devices a). Additionally, the local value $\mathcal{L}$ passed to **nbr** is stored in the memory, which will be made part of the export generated at the end of the computation round.

Rule [METRIC] is similar, though once the set of compatible neighbours $\overline{a}$ is identified, we extract estimated metric values from the $\mathit{metric}(\mathcal{F}(\mathcal{V}))$ term and use this to produce the output by means of the corresponding function in Figure 12.

Rule [LOCAL] is used for mixing together **Field** values obtained from [NBR] and [METRIC] with **Local** values.[5] It computes the set of compatible neighbors, just as for [METRIC], but maps each to the **Local** value $\mathcal{L}$ that is supplied.

Next, rule [FIELD-MATH] applies pointwise functions over **Field** values, by arranging for the [MATH] transition to apply to the set of values from

[5]Typically, **local** is inserted automatically by the compiler, rather than directly by a programmer.

$m(\texttt{metric}, \mathcal{V})$	$\mathcal{L}$
`nbr-lag`, $\mathcal{V}$	$\mathcal{V}_1$
`nbr-delay`, $\mathcal{V}$	$\mathcal{V}_2$
`nbr-vec`, $\mathcal{V}$	$tail_3(\mathcal{V})$
`nbr-range`, $\mathcal{V}$	$\sqrt{\sum_{i\geq 3} \mathcal{V}_i^2}$
`nbr-bearing`, $\mathcal{V}$	(if $\lvert\mathcal{V}\rvert = 4$ then $tan^{-1}(\mathcal{V}_4/\mathcal{V}_3)$ else $\perp$)

Figure 12: Values computed by Proto space-time metric operators.

each neighbor.

Finally, rules [FHOOD-*] handle the **`fold-hood*`** construct, which transforms **Field** values back into **Local** values. For example, the expression **`(fold-hood* min Inf`** $\mathcal{F}(\mathcal{N})$**`)`** returns the minimum value in the field $\mathcal{F}(\mathcal{N})$. The [FHOOD-R] rule deals with the case that the field (third argument) is not empty: we extract an item $a \mapsto \mathcal{L}'$ from it, and substitute the partial summary value $\mathcal{L}$ with the result of applying folding function $\mathcal{O}$ to $\mathcal{L}$ and $\mathcal{L}'$. When the field is empty, the [FHOOD-F] simply returns the second argument $\mathcal{L}$.

4. Platform Semantics

After describing the inner details of computations inside devices, we now turn to platform details, giving semantics for the overall system behavior: how messages are exchanged, how devices get initialized each time, and how the network of devices can change its structure over time.

4.1. Syntax

The syntactical aspects of a network configuration are described as follows.

N	$::=$		Network
		$\bullet$	Empty network
	$\vert$	$(N$ "`|`" $N)$	Composition
	$\vert$	$a :: s$	Device
	$\vert$	$a \overset{\mathcal{V}}{\mapsto} a'$	Link
	$\vert$	$a \twoheadrightarrow [\tau]a'$	Pending message from a to a'

Due to the use of operator "—" again, a network N is a multiset of three kinds of element: *(i)* $a :: s$ means the network includes device a in state s; *(ii)* $a \xmapsto{\mathcal{V}} a'$ means device a is connected with device a' and their estimated space-time displacement is $\mathcal{V}$ (first two elements are time from and to the device; remaining elements are relative spatial coordinates); and *(iii)* $a \twoheadrightarrow [\tau]a'$ means a message from a has been produced to reach a' with content τ—namely, a memory tree.

$$\frac{\langle\Sigma\rangle\mathsf{e} \rightarrow \langle\Sigma'\rangle\mathsf{e}'}{N \mid a :: \langle\Sigma\rangle\mathsf{e} \rightarrowtail N \mid a :: \langle\Sigma'\rangle\mathsf{e}'} \quad \text{[DEVICE]}$$

$$\frac{\mathsf{e}^p \text{is the program expression}}{\{a \twoheadrightarrow [\overline{\tau}]\overline{a}'\} \otimes \{a \xmapsto{\overline{\mathcal{V}}} \overline{a}''\} \otimes a :: \langle T : \bullet \mid \Sigma\rangle\mathcal{A} \otimes N \rightarrowtail \{a \xmapsto{\overline{\mathcal{V}}} \overline{a}''\} \otimes a :: \langle \Box : T[\![\bullet]\!] \mid \Sigma\rangle\mathsf{e}^p \mid \{a \twoheadrightarrow [\sigma T[\![\bullet]\!]]\overline{a}''\} \otimes N} \quad \text{[RELOAD]}$$

$$\frac{-}{N \otimes (a \twoheadrightarrow [\tau]a') \otimes a' :: \langle imp(\{a \mapsto \overline{\tau}'\} \otimes I) \otimes \Sigma\rangle\mathsf{e}^p \rightarrowtail N \otimes a' :: \langle imp(\{a \mapsto \tau\} \otimes I) \otimes \Sigma\rangle\mathsf{e}^p} \quad \text{[RECEIVE]}$$

$$\frac{\overline{\dashv}'' \text{is the new set of neighbors,} \overline{\mathcal{V}}' \text{is the new distances}}{\{a \xmapsto{\overline{\mathcal{V}}} \overline{a}'\} \otimes N \otimes a :: \langle metric(\{\overline{a}' \mapsto \overline{\mathcal{V}}\} \otimes \mathcal{F}) \otimes \Sigma\rangle\mathsf{e}^p \rightarrowtail \{a \xmapsto{\overline{\mathcal{V}}'} \overline{a}''\} \otimes N \otimes a :: \langle metric(\{\overline{a}'' \mapsto \overline{\mathcal{V}}'\} \otimes \mathcal{F}) \otimes \Sigma\rangle\mathsf{e}^p} \quad \text{[MOVE]}$$

$$\frac{-}{\{\bullet \xmapsto{\overline{\mathcal{V}}} a\} \otimes \{a \xmapsto{\overline{\mathcal{V}}'} \bullet\} \otimes N \mid a :: \langle\Sigma\rangle\mathsf{e} \rightarrowtail N} \quad \text{[DROP]}$$

$$\frac{\overline{\mathcal{L}}' \text{is the new values}}{N \mid a :: \langle io(\{\overline{\mathtt{v}} := \overline{\mathcal{L}}\}) \otimes \Sigma\rangle\mathsf{e}^p \rightarrowtail N \mid a :: \langle io(\{\overline{\mathtt{v}} := \overline{\mathcal{L}}'\}) \otimes \Sigma\rangle\mathsf{e}^p} \quad \text{[SIGNAL]}$$

Figure 13: Operational Semantics of Proto platform

4.2. Operational Semantics

Similar to devices, the platform operational semantics is given by a transition system of the kind $(\mathbb{N}, \rightarrowtail)$, where $\mathbb{N}$ is the set of states N of the network, and $\rightarrowtail \subseteq \mathbb{N} \times \mathbb{N}$ is the transition relation. This transition system models all the possible events that can occur in the network and that the platform has somehow to manage, in order to keep the network in a consistent state.

The main assumptions of our model are as follows:

- Message exchange is asynchronous, with each device sending sometime between computation rounds. Order is preserved for each sender, but messages may be lost.

- Execution is asynchronous, parallel, and atomic. As soon as a device computation round is over, the execution of a new round is enabled to start.

Rules of the operational semantics are provided in Figure 13. Rule [DEVICE] is a standard model of asynchronous parallel computation in a device: at any time, any device may perform a single computation step, according to the language semantics in the previous section.

Rule [RELOAD] can fire when a device a completes a computation round (program evaluated to result $\mathcal{A}$). In this case, the Proto platform *(i)* erases any undelivered messages $\overline{\tau}$ previously sent by a, *(ii)* retrieves information about neighbours $\overline{a}$, *(iii)* restores the device program to original state e^p, *(iv)* and finally sends one message containing an export of the current memory tree $\sigma\tau'$ to all neighbours. Note that such messages overwrite any other message previously sent by a and not yet received by its recipient. As this rule is fired, the device could start another computation round, though most current Proto implementations wait a predefined time Δt.

With rule [RECEIVE], device a' consumes a message sent from a with content τ: the effect is that the pending message is consumed and replaces all the imports corresponding to a currently existing in the store of a'—there could be 0 or 1. Note that the expression must be e^p: [RECEIVE] is not allowed to occur in the middle of a computation. This will be the same for most of the rest of the transitions.

Rule [MOVE] handles topological changes (e.g., a device moves or fails). This is modeled as a change in the outgoing connections of a device a, moving from devices $\overline{a}'$ with space-time displacements $\overline{\mathcal{V}}$ to devices $\overline{a}''$ with distances $\overline{\mathcal{V}}'$ (changes involving more devices are modelled as a sequence of [MOVE] transitions). As this change occurs, the platform changes the store of a, properly updating structure $metric(\mathcal{F}(\mathcal{V}))$.

Similarly, rule [DROP] handles a device leaving the network (e.g., due to a failure), removing its representation $a :: s$. Note that this rule is enabled only if there are no links to/from this device—some previous transitions of [MOVE] must execute to clean the neighbourhood first.

Finally, rule [SIGNAL] processes the actuators and changes the values sensed by device a, replacing the $io(\overline{\mathtt{v} := \mathcal{L}})$ term in the store of a. Table 2 shows the set of universal sensor and actuator relations; all others are platform specific.

v	Type	Sense/Act	Behavior
hood-radius	$\mathcal{S}$	sensor	Estimated farthest possible neighbor
density	$\mathcal{S}$	sensor	Estimated devices/volume
infinitesimal	$\mathcal{S}$	sensor	Estimated volume/device
dt	$\mathcal{S}$	sensor	Elapsed time since last evaluation
mid	$\mathcal{S}$	sensor	Unique identifier for device
speed	$\mathcal{S}$	sensor	Estimated magnitude of speed
bearing	$\mathcal{S}$	sensor	Estimated compass bearing (if applicable)
mov	$\mathcal{V}$	act	Attempt to move with velocity $\mathcal{V}$
flex	$\mathcal{S}$	act	Attempt to set local curvature of device network to $\mathcal{S}$
set-dt	$\mathcal{S}$	act	Attempt to change time between rounds to $\mathcal{S}$
probe	$\mathcal{V}(\mathcal{L},\mathcal{S})$	act	Place $\mathcal{L}$ in debugging slot $\mathcal{S}$

Table 2: Universal sensors and actuators for Proto.

4.3. Example

To shed light on the most important rules of the platform semantics that we have defined above, we now show some details of an example execution. Considering the `distance-to` example, we let $\mathsf{e}^p =$ `(distance-to (sense 1))`. For a network, we assume a simple linear topology of three devices a_0, a_1, a_2, where each a_i is linked to just a_{i-1} and a_{i+1}. Distance between each pair of neighbours is 10, and sensor 1 is active for a_0 only. As a result, we shall define the following stores, to represent the sensors (Σ_0^S,Σ_1^S, and Σ_2^S for the three nodes) and memory trees ($\Sigma_\perp^T$ initially, Σ_∞^T after first firing, and Σ_n^T for a node firing distance value n)

$$\Sigma_0^S = io(\texttt{\textbf{s1}} := \texttt{\textbf{true}}) \qquad \Sigma_1^S = \Sigma_2^S = io(\texttt{\textbf{s1}} := \texttt{\textbf{false}})$$
$$\Sigma_\perp^T = \Box : \bullet \quad \Sigma_\infty^T = \Box : (\texttt{\textbf{false}}, \texttt{x} := \texttt{inf}) \quad \Sigma_n^T = \Box : (\texttt{\textbf{true}}, \texttt{x} := n, (\texttt{\textbf{nbr}}(n))$$

Each node a_i also has metric values Σ_i^m always aligned with topology, e.g., $\Sigma_1^m = metric(a_0 \mapsto v(a_1, a_0)) \otimes metric(a_2 \mapsto v(a_1, a_2))$ where $v(a_i, a_j)$ is the vector from node a_i to node a_j. The initial network state is hence formed by a term:

$$\begin{array}{rcl} N_0 & = & a_0 :: \langle \Sigma_0^S \otimes \Sigma_0^m \otimes \Sigma_\perp^T \rangle \mathsf{e}^p \\ & \mid & a_1 :: \langle \Sigma_1^S \otimes \Sigma_1^m \otimes \Sigma_\perp^T \rangle \mathsf{e}^p \\ & \mid & a_2 :: \langle \Sigma_2^S \otimes \Sigma_2^m \otimes \Sigma_\perp^T \rangle \mathsf{e}^p \mid N_L \end{array}$$

where N_L is the specification of all topology links N_L (definition omitted for simplicity since it is essentially the same information stored in metric values).

As this system bootstraps, devices start computing by rule [DEVICE]. The first computation round for any device simply sets variable `d` to `inf`. When this computation is over, rule [RELOAD] fires which retrieves (and sends) the export tree to all neighbours and restores initial expression e^p. Assuming the first computation round is executed for all three of a_0, a_1 and a_2, we reach the new state:

$$\begin{array}{rcl} N_1 & = & a_0 :: \langle \Sigma^S_0 \otimes \Sigma^m_0 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \mid a_0 \twoheadrightarrow [\Sigma^T_\infty] a_1 \\ & \mid & a_1 :: \langle \Sigma^S_1 \otimes \Sigma^m_1 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \mid a_1 \twoheadrightarrow [\Sigma^T_\infty] a_0 \mid a_1 \twoheadrightarrow [\Sigma^T_\infty] a_2 \\ & \mid & a_2 :: \langle \Sigma^S_2 \otimes \Sigma^m_2 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \mid a_2 \twoheadrightarrow [\Sigma^T_\infty] a_1 \mid N_L \end{array}$$

where many pending messages have been created. By a [RECEIVE] transition all the pending messages for a node are consumed and become imports. As this is executed for all the tree nodes, we reach state:

$$\begin{array}{rcl} N_2 & = & a_0 :: \langle \mathit{imp}(a_1 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_0 \otimes \Sigma^m_0 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \\ & \mid & a_1 :: \langle \mathit{imp}(a_0 \mapsto \Sigma^T_\infty) \otimes \mathit{imp}(a_2 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_1 \otimes \Sigma^m_1 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \\ & \mid & a_2 :: \langle \mathit{imp}(a_1 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_2 \otimes \Sigma^m_2 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \mid N_L \end{array}$$

Assume now that a new computation round for a_0 is over, where it fires value 0, producing the state:

$$\begin{array}{rcl} N_3 & = & a_0 :: \langle \mathit{imp}(a_1 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_0 \otimes \Sigma^m_0 \otimes \Sigma^T_0 \rangle \mathsf{e}^p \mid a_0 \twoheadrightarrow [\Sigma^T_0] a_1 \\ & \mid & a_1 :: \langle \mathit{imp}(a_0 \mapsto \Sigma^T_\infty) \otimes \mathit{imp}(a_2 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_1 \otimes \Sigma^m_1 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \\ & \mid & a_2 :: \langle \mathit{imp}(a_1 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_2 \otimes \Sigma^m_2 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \mid N_L \end{array}$$

As node a_1 receives the corresponding value, it overwrite its import, so we move to:

$$\begin{array}{rcl} N_4 & = & a_0 :: \langle \mathit{imp}(a_1 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_0 \otimes \Sigma^m_0 \otimes \Sigma^T_0 \rangle \mathsf{e}^p \\ & \mid & a_1 :: \langle \mathit{imp}(a_0 \mapsto \Sigma^T_0) \otimes \mathit{imp}(a_2 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_1 \otimes \Sigma^m_1 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \\ & \mid & a_2 :: \langle \mathit{imp}(a_1 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_2 \otimes \Sigma^m_2 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \mid N_L \end{array}$$

Now the computation round of a_1 proceeds until completion, and reaching value 10, obtained by the minimum sum of neighbour distance and corresponding vector length:

$$\begin{array}{rcl} N_5 & = & a_0 :: \langle \mathit{imp}(a_1 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_0 \otimes \Sigma^m_0 \otimes \Sigma^T_0 \rangle \mathsf{e}^p \\ & \mid & a_1 :: \langle \mathit{imp}(a_0 \mapsto \Sigma^T_0) \otimes \mathit{imp}(a_2 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_1 \otimes \Sigma^m_1 \otimes \Sigma^T_{10} \rangle \mathsf{e}^p \\ & \mid & \quad \mid a_1 \twoheadrightarrow [\Sigma^T_{10}] a_0 \mid \ \mid a_1 \twoheadrightarrow [\Sigma^T_{10}] a_2 \\ & \mid & a_2 :: \langle \mathit{imp}(a_1 \mapsto \Sigma^T_\infty) \otimes \Sigma^S_2 \otimes \Sigma^m_2 \otimes \Sigma^T_\infty \rangle \mathsf{e}^p \mid N_L \end{array}$$

The computation may then proceed forward to compute value 20 for a_2, and the values will continue being computed thus until the sensor or network state changes.

5. Findings and Implications for Proto

In addition to establishing a reference for implementations of Proto, our work in establishing a formal semantics has produced a number of advancements in the language itself, ranging from major upgrades to small bug fixes.

5.1. Functions

The most significant advance in the Proto language, as a result of this work, is the semantics for function calls, which depend on the memory tree structure. The basic problem, which we have introduced the memory tree structure to address, is how to ensure that neighbor exports can be properly matched between neighboring devices, despite the fact that the program may execute differently on each device. Prior to the work reported in this paper, the discrete semantics of Proto assumed that all neighbor exports and state could be identified and uniquely enumerated at compile time. Every instance of the **`nbr`** operator in the program was assigned an index, and the export was a preallocated block of memory, the same size on every device, that was read from and written to using these indices. When a piece of code was not executed due to an **`if`**, a special mechanism marked the exports within it as "dead." A similar mechanism aligned state within a device from round to round. This is the model used in the core semantics we reported previously in [19].

The explicit indexing semantics was accomplished in the Proto compiler by inlining all function calls, such that indices could be easily calculated and hardwired in the ProtoKernel virtual machine code that it generated. This had two major negative consequences: first, programs were restricted to only those where total inlining was possible, eliminating the possibility of recursive functions, higher order functions like mapping, and functions as first class objects. Second, every instance of a function call resulted in another nearly identical copy of the function code, significantly inflating the binary size of compiled programs.

The memory tree model, coupled with a function call model for computation on manifolds in Proto's continuous abstraction [30], makes proper function calls possible in Proto, remedying most of the these shortcomings. We are now upgrading ProtoKernel to support memory trees; once this is complete, recursive functions and higher order functions could possibly be added to the language, alongh with the ability of passing functions as values.

Functions will not yet entirely be first-class objects, however, due to two other challenges revealed by this formal semantics: closure and determining when to share data between function instances. To illustrate these problems, consider the following expression:

```
(fold-hood*
  (let ((x (mid))) (def foo (a b) (min a b x)))
  Inf
  ((elt
    (tup (fun () (nbr (+ 1 (mid))))
         (fun () (nbr (- 1 (mid)))))
    (< (rnd 0 1) 0.5))))
```

The problem of closure is illustrated by the function `foo`, which references a variable `x` that is assigned outside of the function. The function `foo` is not evaluated, however, until outside of the scope in which `x` was defined, so the current semantics will be unable to find the assignment for `x`. Closures are a well-understood problem of functional languages, with a number of well-established solutions. Because of Proto's distributed nature, however, it is not immediately obvious whether these solutions also apply to Proto or whether they need to be modified.

The problem of determining when to share data between function instances is illustrated by the functions randomly selected from the tuple. Under the semantics we have presented, these functions would occupy the same location in the memory and thus use each other's **`nbr`** exports. This is clearly a problem if the functions are performing unrelated computations, but is appropriate and even necessary if the two functions are the same, but happen to have been originated on different devices. The problem is how to determine whether two functions should share state or not, and again it is not clear whether the solutions developed for other systems can be applied directly in the case of Proto.

We have previously explored some of these questions in [25]; the memory tree semantics now provide a solid foundation for further investigation of first-class functions across space-time. Since these issues are not yet resolved, the semantics presented in this paper simply prohibit the movement of first-class function objects into situations where these issues can apply.

5.2. Neighbor/Feedback Delay

Our work on semantics formalization has also uncovered a previously unnoticed suboptimality of Proto's design: unnecessary delay in expressions that combine neighborhood and feedback.

Consider the following expression, where a value of **true** spreads through neighborhoods from randomly chosen devices:

```
(rep x (< (rnd 0 1) 0.01) (fold-hood* max 0 (nbr x)))
```

This is an example of the most frequent design pattern in Proto, where a global interaction is expressed as a state variable chaining through neighborhoods.

Under the continuous space-time abstraction of Proto, the (**nbr** x) means that neighbors get access to the value of x as quickly as it can move through space. This is not, however, what the discrete semantics actually implement. When rule [REP-UPDATE] is executed, the variable assignment $\mathtt{x} := \mathcal{L}$ will change to $\mathtt{x} := \mathcal{L}'$. When the subexpression (**fold-hood* max** 0 (**nbr** x)) is evaluated, its lookup of x will find the old variable assignment $\mathtt{x} := \mathcal{L}$, which means that rule [NBR] will place $\mathcal{L}$ into the memory tree. Each round of evaluation after the first will thus produce a memory tree of the form: $(\textbf{true}, \mathtt{x} := \mathcal{L}', nbr(\mathcal{L}))$, where $\mathcal{L}$ is the value of x from the previous round of evaluation. What this means is that there is an extra round of delay before the value of x is actually exported to the neighbors.

This is not a flaw in our formalization of Proto's semantics, but a suboptimality in the design of Proto. It is somewhat remarkable that this was not noticed before, but close examination of ProtoKernel revealed two implementation bugs which largely cancelled this effect in most ordinary cases. Note that this double-delay is not an inconsistent behavior: when the density of devices and frequency of execution increase without bound, the discrete behavior does converge to the continuous model. In addition, although the case we are considering is a common case, there are others where attempting to accelerate the export may not be correct. However, this extra round of delay is suboptimal in many common cases, and it is an open question how best to modify the discrete semantics so that they can operate more quickly and more closely match the continuous abstraction.

5.3. Minor Changes

Besides these major contributions, our formalization of semantics has revealed three small flaws, which have now been dealt with:

- It was previously possible to use an **if** expression in a neighborhood computation. We found, though, that an **if** expression must not return

Field values, as it is possible to create undefined values by applying [FIELD-MATH] to combine fields that share no devices. This restriction is embodied in the [IF] rule returning only **Local** values and we have added corresponding error checking to the MIT Proto compiler.

- It was previously possible to apply an actuator function to a **Field** value. A device can only directly control its own actuators, however, so we restricted the [ACT] rule to require **Local** values and have added corresponding error checking to the MIT Proto compiler.
- It makes little sense for the initialization expressions of feedback operations (**`rep`**, **`letfed`**) to contain feedback, branching, or neighborhood operations, as any such state established will be discarded rather than used in the next evaluation. Allowing it is harmless, however, and no sensible program is likely to ever do so, so we do not attempt to exclude this in the semantics or the compiler.

6. Contributions

We have presented a complete operational semantics for programs written in the Proto spatial computing language, as executed on an asynchronous network of fast message-passing devices. This semantics covers both the interpretation of programs on individual devices, and a model of how devices interact with one another and their environment. In addition, the process of developing a formal operational semantics has created significant advancements for the Proto language and its implementation in MIT Proto.

We believe this formalization paves the way to a number of future works, first of all concerned with improving design, analysis and implementation of spatial computing applications. First, the presented operational semantics aims at a reference for implementers of Proto, which will be key to guaranteeing coherence across existing and emerging implementations for diverse execution platforms. Second, as Proto is being used to study self-organization patterns (e.g., [32]), the operational semantics can aid analysis by acting as an executable specification or as a basis for stochastic model-checking or reachability analysis. Likewise, it should be possible to prove behavioral properties (like confluence and deadlock freedom) formally, helping us to better predict and control the global behaviour of a system. In particular, we believe this task could be simplified by relying on a smaller calculus covering a significant fragment of Proto. Finally, as has already been the case, this semantics will be an aid to the continued development of Proto.

References

[1] J. Beal, J. Bachrach, Infrastructure for engineered emergence in sensor/actuator networks, IEEE Intelligent Systems 21 (2006) 10–19.

[2] J. Beal, S. Dulman, K. Usbeck, M. Viroli, N. Correll, Organizing the aggregate: Languages for spatial computing, in: M. Mernik (Ed.), Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global, 2013, Ch. 16, pp. 436–501. doi:10.4018/978-1-4666-2092-6.ch016.

[3] M. D. Rosa, S. C. Goldstein, P. Lee, J. D. Campbell, P. Pillai, Programming modular robots with locally distributed predicates, in: IEEE International Conference on Robotics and Automation (ICRA '08), 2008, pp. 3156–3162.

[4] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, P. Pillai, Meld: A declarative approach to programming ensembles, in: IEEE International Conference on Intelligent Robots and Systems (IROS '07), 2007, pp. 2794–2800.

[5] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications: The tota approach, ACM Transactions on Software Engineering Methodologies 18 (4) (2009) 1–56. doi:http://doi.acm.org/10.1145/1538942.1538945.

[6] M. Viroli, D. Pianini, J. Beal, Linda in space-time: an adaptive coordination model for mobile ad-hoc environments 7274 (2012) 212–229.

[7] M. Viroli, M. Casadei, S. Montagna, F. Zambonelli, Spatial coordination of pervasive services through chemical-inspired tuple spaces, ACM Transactions on Autonomous and Adaptive Systems 6 (2) (2011) 14:1 – 14:24. doi:10.1145/1968513.1968517.
URL `http://doi.acm.org/10.1145/1968513.1968517`

[8] M. Viroli, D. Pianini, S. Montagna, G. Stevenson, Pervasive ecosystems: a coordination model based on semantic chemistry, in: S. Ossowski,

P. Lecca, C.-C. Hung, J. Hong (Eds.), 27th Annual ACM Symposium on Applied Computing (SAC 2012), ACM, Riva del Garda, TN, Italy, 2012, pp. 295–302.

[9] S. Montagna, M. Viroli, J. L. Fernandez-Marquez, G. Di Marzo Serugendo, F. Zambonelli, Injecting self-organisation into pervasive service ecosystems, Mobile Networks and Applications (2012) 1–15Online first, available through DOI: 10.1007/s11036-012-0411-1. doi:10.1007/s11036-012-0411-1.
URL http://www.springerlink.com/content/x3j4776323717w7h/

[10] R. Gummadi, O. Gnawali, R. Govindan, Macro-programming wireless sensor networks using kairos., in: Distributed Computing in Sensor Systems (DCOSS), 2005, pp. 126–140.

[11] R. Newton, M. Welsh, Region streams: Functional macroprogramming for sensor networks, in: First International Workshop on Data Management for Sensor Networks (DMSN), 2004, pp. 78–87.

[12] J.-L. Giavitto, C. Godin, O. Michel, P. Prusinkiewicz, Computational models for integrative and developmental biology, Tech. Rep. 72-2002, Univerite d'Evry, LaMI (2002).

[13] J. Beal, R. Schantz, A spatial computing approach to distributed algorithms, in: 45th Asilomar Conference on Signals, Systems, and Computers, 2010, pp. 1–5.

[14] J. Bachrach, J. Beal, Programming a sensor network as an amorphous medium, in: Distributed Computing in Sensor Systems (DCOSS) 2006 Poster, 2006, pp. 1–6, available at http://jakebeal.com/.

[15] J. Bachrach, J. Beal, J. McLurkin, Composable continuous space programs for robotic swarms, Neural Computing and Applications 19 (6) (2010) 825–847.

[16] J. Beal, J. Bachrach, Cells are plausible targets for high-level spatial languages, in: Spatial Computing Workshop, 2008, pp. 284–291.

[17] J. Beal, T. Lu, R. Weiss, Automatic compilation from high-level languages to genetic regulatory networks, PLoS ONE 6 (8) (2011) E22490.

[18] MIT Proto, software available at http://proto.bbn.com/ (Retrieved January 1st, 2012).

[19] M. Viroli, J. Beal, M. Casadei, Core operational semantics of Proto, in: M. J. Palakal, C.-C. Hung, W. Chu, W. E. Wong (Eds.), 26th Annual ACM Symposium on Applied Computing (SAC 2011), Vol. II: Artificial Intelligence & Agents, Information Systems, and Software Development, ACM, Tunghai University, TaiChung, Taiwan, 2011, pp. 1325–1332.

[20] J. Bachrach, J. Beal, Building spatial computers, Tech. Rep. MIT-CSAIL-TR-2007-017, MIT (March 2007).

[21] M. Mernik, J. Heering, A. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys (CSUR) 37 (4) (2005) 316–344.

[22] P. R. Henriques, M. J. V. Pereira, M. Mernik, M. Lenic, J. Gray, H. Wu, Automatic generation of language-based tools using the LISA system, Software, IEE Proceedings - 152 (2) (2005) 54–69. doi:10.1049/ip-sen:20041317.
URL http://dx.doi.org/10.1049/ip-sen:20041317

[23] J. Beal, Amorphous medium language, in: Large-Scale Multi-Agent Systems Workshop (LSMAS), 2005, pp. 1–7, available at http://jakebeal.com/.

[24] J. Beal, Programming an amorphous computational medium, in: J.-P. Banatre, P. Fradet, J.-L. Giavitto, O. Michel (Eds.), Unconventional Programming Paradigms, Vol. 3566 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 121–136. doi:10.1007/11527800_10.
URL http://dx.doi.org/10.1007/11527800_10

[25] J. Beal, Dynamically defined processes for spatial computers, in: Spatial Computing Workshop, 2010, pp. 206–211.

[26] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, part I, Information and Computation 100 (1) (1992) 1–40.

[27] A. Ricci, M. Viroli, G. Piancastelli, simpA: An agent-oriented approach for programming concurrent applications on top of

Java, Science of Computer Programming 76 (1) (2011) 37–62. doi:10.1016/j.scico.2010.06.012.

[28] D. Turner, The polymorphic pi-calculus: Theory and implementation, Ph.D. thesis, University of Edinburgh (1995).

[29] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (2001) 396–450.

[30] J. Beal, K. Usbeck, B. Benyo, On the evaluation of space-time functions, The Computer Journal. doi:10.1093/comjnl/bxs099.

[31] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.

[32] J. Beal, J. Bachrach, D. Vickery, M. Tobenkin, Fast self-healing gradients, in: R. L. Wainwright, H. Haddad (Eds.), Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008, ACM, 2008, pp. 1969–1975.