
On the Evaluation of Space-Time Functions

JACOB BEAL, KYLE USBECK, BRETT BENYO

BBN Technologies Cambridge, MA, USA, 02138

Email: jakebeal@bbn.com, kusbeck@bbn.com, bbenyo@bbn.com

In an environment increasingly saturated with computing devices, it is desirable for some services to be distributed, executing via local interactions between devices. Creating fast, flexible, and dynamic distributed services requires a general model of function calls distributed over space-time. Prior models, however, have either depended strongly on large-scale Internet infrastructure or have restrictions in the scope or resolution of space-time for inputs, outputs, or evaluation of the function. We address this by providing a formal general model of function calls over space-time. We then fully realize a practical model of space-time function calls, based in the Proto language, and present both theoretical and empirical results. Finally, we show how our results for Proto generalize into implications for any model of distributed computing.

Keywords: Spatial computing, distributed algorithms, amorphous computing, distributed function calls

Received 15 Dec 2011; revised 7 March 2012, 17 May 2012

1. INTRODUCTION

Every day, the world around us becomes more saturated with computing devices—particularly as mobile devices like cell phones and tablets are becoming cheaper, more capable, and more widespread. With this increased saturation comes the ability to take advantage of the massive amount of computing power that these devices can provide. However, the programming models currently available take little advantage of their potential for distributed computation. Instead, most computation between physically proximate devices takes place inside of data centers hundreds or thousands of kilometers away. This process distribution model becomes insufficient when infrastructure is unavailable or inadequate—such as during natural disasters, terrorist attacks, large festivals, and sporting events.

Consider, for example, a distributed disaster response system running on people’s mobile phones in the aftermath of an earthquake. Traditional centralized infrastructure (e.g., cellular and satellite networks) tends to fail in these situations as a result of hardware destruction or network overload. However, people still might want to communicate information about fires and survivors to rescue workers, navigate while avoiding blocked or dangerous areas, rendezvous with friends and loved ones, or coordinate volunteer efforts. What all of these services have in common is that a distributed implementation requires a strong *many-to-many* model of distributed computation. One can imagine a “safe navigation” program, for example,

being invoked by many people. This program requires both local and non-local information to be considered, and in a disaster scenario, would pragmatically have to have its computation and data distributed across many devices.

This is only one of many examples requiring computational processes distributed over space-time. As the density of embedded devices rises, it is likely that efficiency, effectiveness, and privacy may all be improved by distributing and localizing computation. For example, why should drivers looking for real-time traffic data rely on connectivity to a server thousands of miles away, especially when it is mostly relevant to other drivers nearby and could be computed and stored on their personal devices? Other similar *spatial computing* domains include large-scale sensor networks, robotic swarms, and agent-based simulations [1]. Prior work in spatial computing has yielded a number of models of computation distributed over space-time, such as those of Proto [2], MGS [3], TOTA [4], and Meld [5]. The sort of fast, flexible, and dynamically invoked services required by scenarios like disaster response, however, demand a much more general model of distributed function calls than has been provided by any prior work.

In this paper, we address this problem by providing a formal general model of function calls over space-time. We then fully realize a practical model of space-time function calls, based in the Proto language, and present both theoretical and empirical results.

The key contributions of this work are:

X	Space-time region for a computation
p	A point in space-time
(t, x)	Time (t) and space (x) coordinates of a point
V	Set of all possible data values
f	A field mapping subspace X_f to values in V
f_*	The field associated with element $*$
o	An instance of a computational operator
f_{i_n}	The n th input field to an operator instance
f_{o_n}	The n th output field to an operator instance
M	All manifolds in a Proto program
m_*	The manifold associated with element $*$
O	All operator instances in a Proto program
F	All fields in a Proto program
s_m	Selector field for a sub-manifold m
r_d	Return value for a root manifold
d	Definition of a function
$\mathcal{E}_{d,o}(*)$	Value of $*$ in function d evaluated for call o
$\gamma_\delta(\tau)$	Trajectory of device δ against local time τ

TABLE 1. Reference table for symbols used in this paper. The top half contains general notation, the bottom half Proto-specific notation.

- A formal definition of function calls distributed over arbitrary regions of space-time and development of two models (substitution and in-place) for evaluating such functions;
- Criteria for well-defined space-time operators, along with application to Proto and analysis of their implications; and
- A description of a reference implementation for space-time function calls.
- Generalization of Proto results to implications for any distributed computing model.

The remainder of this paper is organized as follows: Section 2 defines our notion of space-time function calls, lists their criteria for well-definedness, and discusses related work. Next, our formal notation for space-time operators is contained in Section 3, and the formal conditions for well-definedness are discussed in Section 4. Section 5 explains two models of distributed function calls, and their implementation progress is documented in Section 6 along with the new language features this work has allowed and its effect on runtime performance. Finally, we discuss the general implications of our results for all distributed computing models in Section 7 and our contributions are summarized in Section 8. See Appendix A for proofs of well-definedness for the models of distributed function calls.

2. WELL-DEFINED SPACE-TIME FUNCTION CALLS

In order to investigate the problem of function calls distributed over space-time, we need to have a precise definition of what we mean by a space-time function call. We will also need criteria for well-definedness that we can use to determine whether a proposed approach

to space-time function calls is valid.

2.1. Well-Definedness of Operators

We begin with the definition of a generic space-time operator, which we will then refine to provide a definition of a space-time function call. Recall that any computation can be viewed in terms of operators that manipulate data. On a spatial computer, this data is not kept at one location, but may be distributed widely over space and time. We will thus define space-time data as *fields* that specify what data values are where at what times, and we will define space-time computation in terms of *operators* that take fields as input and produce fields as output.

We formalize this generally, with the aim that it should be applicable to all models of computation over space-time, from conventional networking to discrete models such as cellular automata to continuous models such as those proposed in [6], [7], or [8]. For ease of reference, Table 1 collects all notation used in this paper.

Let X be the collection of all points p in space-time where a computation can occur; this might be continuous (e.g., a manifold) or discrete (e.g., a collection of events on a network). We can assign coordinates (t, x) to each point in space-time using any valid metric, where t is the time coordinate and x is the space coordinate.

A *field* associates data with points in space-time: we shall define a field f as a mapping

$$f : X_f \rightarrow V$$

that maps each point in its domain—some portion of space-time $X_f \subseteq X$ —to a data value within the range V of possible values. For example, a field **temperature** might be defined over all locations in space-time and map each point to the temperature observed at that place at that time.

Building from this definition, an *operator* is a higher-order function that maps from fields to fields, and an *operator instance* o is an application of an operator to relate a particular collection of fields:

$$o : f_{i_1} \times f_{i_2} \times \dots \rightarrow f_{o_1} \times f_{o_2} \times \dots$$

where f_{i_n} is the n th input field and f_{o_n} is the n th output field. There may be any number of input or output fields, including zero,¹ and any given field may potentially be used multiple times, both within and across operator instances. For example, a **sqrt** operator might be defined to take one field of numbers as input and return another field of numbers as output, where the number at each point of space-time in the output field is equal to the square root of the number at each point in the input field.

¹Operators with no outputs are generally not very useful, though; actuators are often better represented as functions with some special semantics attributed to their outputs.

Given any set of primitive operators, arbitrarily complex computations can be constructed by composing operators mathematically. This is done by setting the output fields of some operator instances to be the input fields of others.

Not all such compositions are reasonable, of course. Each operator has *well-definedness criteria* that can be applied to test whether its set of input and output fields are legal.² In this paper, we will focus on the well-definedness criteria for domains (ranges are readily handled by conventional type theory). For example, the `sqrt` operator requires that its input and output fields have the same domain: every point of space-time with a number in the output field requires one in the input field as well, and vice versa. It is invalid to take a square root of a number that does not exist, or to take the square root of a number and produce no result—not even an error.

Importantly for our purposes, this notion of well-definedness is a compositional property: thus, in order to tell if a computation as a whole is well-defined, we need only check whether each of its operator instances is well-defined.

2.2. Space-Time Functions

A function, then, is a special type of operator, which takes its input fields and applies a composition of operator instances to them in order to evaluate the function and produce its outputs. The challenge is that, unlike with other operators, the fields cannot be fully defined when the function is defined. This is because the domain where the function is evaluated is set by the context in which the function is called. Likewise, the function call determines the fields that will give values to the function's parameters and the fields that will take their values from the function's outputs. Even more challenging, when function calls are driven by data (e.g., requests from users, observations of the environment), even the existence of a function call can often only be determined at run-time. How, then, can we ensure that a function call over space-time will be well-defined?

The previous answer has been to enforce well-definedness by limiting the degree to which a function call can be distributed over space-time. For example, client-server systems often collapse distribution at the server where all function calls arrive: the inputs and outputs at the clients may be at arbitrary points in space-time, but the evaluation of the function takes place on the server at a single location in space. Distributed function call APIs typically use an inverse model: the input and output are at single points in space-time, though the evaluation may be arbitrarily distributed. Transactional systems often use a third model: inputs, outputs, and evaluation can all be

²We could annotate this formally if we wished to do so, e.g., $W_o : F^k \rightarrow \{true, false\}$, but this does not add useful content for our purposes.

distributed in space, but time resolution is severely limited by strict communication requirements on the progress of evaluation. There are other models as well, but they all share the following property: each model restricts the space-time extent of the inputs, outputs, or evaluation of function calls, and each model is well-suited for some types of computation and poorly-suited for others.

Our task in this paper is to describe an effective general model of space-time function calls that has no such restrictions, yet can always be proven to be well-defined. This will allow a richer and more flexible approach to distributed programming, where the extent of each function call is limited only by the nature of the computation that it performs.

2.3. Relation to Existing Models of Distributed Computing

Models of distributed computation have existed for many years. We organize our presentation of the existing models in one of three categories: (i) many-to-one, (ii) one-to-many, and (iii) many-to-many, where their cardinality refers to the origin and execution points of the distributed function call.

The *many-to-one* model is essentially the standard client/server model, where client machines have their computations resolved by a single server. This evades the problems of space-time function calls by collapsing to non-distributed computation at the server.

Most distributed computation frameworks, on the other hand, employ *one-to-many* cardinality where a single machine divides a highly-parallel problem among many devices, analyzing or combining their results once they complete. Examples include Google MapReduce [9], Hadoop [10], Condor [11], BOINC [12], and the Oracle Grid Engine [13], among a vast number of others. These typically evade the problems of space-time function calls by collapsing to non-distributed or tightly synchronized computation in the work dispatching device or devices.

The third category, of *many-to-many* distributed computation, is rapidly growing as distributed systems become more complex. This many-to-many model is studied under a variety of other names and forms, e.g., message-passing, load-balancing, peer-to-peer (P2P), and distributed consensus.

The most basic many-to-many message-passing models like Message Passing Interface [14], and Erlang [15], allow distributed function calls via arbitrary message exchange. While this approach is flexible enough to be used to implement any notion of distributed function calls, it is very low level and provides no coherent model of a many-to-many function call, thus effectively ignoring the space-time function call problem.

Load-balancing is often used as a tool for distributing computation among a group of devices that provide

a similar service. For example, large-scale websites employ load-balancing to maintain responsiveness and reliability while servicing requests from a multitude of clients. In reality, load-balancing is improving scalability by shifting the distribution cardinality from a traditional website's *many-to-one* to a more-scalable *many-to-many*. According to [16], load-balancing can be accomplished in a number of manners including client-side proxies, Cluster DNS, packet rewriting, request redirection, and more. However, all these methods of load-balancing lack the dynamism required to operate outside of a conventional, mostly-static, Internet environment. Put another way: they only operate on space-time regions with highly constrained structure.

Another category of *many-to-many* distributed computation research is termed *peer-to-peer* (P2P). P2P is an alternative to client/server architectures where every device is capable of being both a resource producer and a resource consumer, and where resources can include data, bandwidth, or computational power. Unfortunately, while the P2P architecture enables generic resource distribution, many current or recent P2P applications (e.g., file sharing tools such as BitTorrent [17], Tribler [18], and Napster [19]; message-based systems such as Skype [20], and Data Distribution Service [21]; distributed hash-table implementations such as Chord [22]; and many more) are ad-hoc data-sharing or messaging tools; such tools lack the ability to distribute *computation* among peers—an important criteria for our distributed processing model.

An important exception is gossip and population protocols [23] and approximate consensus (e.g., [24]), which provide a model of peer-to-peer distributed function evaluation for certain special classes of functions and network assumptions. The class of functions with known feasible methods of implementation, however, is small.

Exact consensus has also long been studied as a means of enabling many-to-many distributed computation, with a rich history of algorithms and strong impossibility bounds [25]. Of particular interest is the recent thrust of research in the area of “virtual nodes” where a collection of nearby devices use consensus to form a larger virtual device on which arbitrary computation can be computed independent of any particular physical devices (e.g., [26]). As noted above, however, these systems require guarantees on the integrity of computational transactions, which typically leads to slow execution, high communication costs, and problems ensuring that an algorithm can progress—in other words, limits on the time resolution of computation and/or the permissible structure of space-time. Recent work coupling these concepts with self-stabilization [27] offers some possible paths forward, however.

Spatial computing systems, however, have produced a number of distributed computing models (reviewed

below in Section 3.1), which provide partial solutions to the problem of space-time function calls. Typically, these have been limited either in scope (e.g., OSL [28], which produces only origami constructions) or in the degree of guarantees (e.g., TOTA [4], which offers a viral process model, but little in the way of encapsulation). Proto [2], however, offers both an aggregate model of space-time computation and an informal model of function calls, which will form the basis for our approach in this paper.

3. APPROACH

Our approach is to consider an extreme case for distributed function calls, in which the region of space-time X is continuous. The input and output fields of a function call will be subspaces of X that may contain an infinite number of points, as will the fields used in evaluating the function. If we can produce a generalized model of function calls that can scale up to infinite numbers of points, then we can also scale it down to any smaller system.

In particular, we have chosen to work with the Proto spatial computing language [2]. Proto is a purely functional language where programs are specified using a model of computation over space-time manifolds. These programs are then compiled for distributed approximation on networks of discrete devices. Proto is well-suited for our investigation because its continuous model and functional composition match the problem we wish to address, and its approximation model means that our results should be applicable to discrete systems as well.

Proto is particularly general—approaching space-time universality [6]. We thus expect that any of the distributed computing models discussed above should be able to be implemented in Proto (though some would be quite inefficient). For example, [29] shows one way that a distributed publish subscribe system (a many-to-many model) can be implemented using Proto. Our results from working with Proto will thus have implications for any distributed computing model, which we discuss below. Furthermore, working with a continuous model of space-time simplifies our definitions and proofs, while simultaneously avoiding tying our results to any particular network model.

We will first formally establish the well-definedness of Proto programs: although continuous space [30], time [31], and discrete approximation [32] semantics have previously been specified for Proto, the well-definedness of arbitrary compositions of Proto operators has not yet been proven. We will then develop a model of space-time function calls for Proto and prove that all such function calls are guaranteed to be well-defined. This model supports recursive functions, higher-order functions, and first-class functions that can be resolved at compile time (fully first-class functions remain a challenge for future work).

Proto’s continuous-to-discrete mapping then means that mechanisms that we produce for Proto should be able to serve as a basis for generalized distributed function calls in other more conventional languages.

3.1. Background: Proto

Proto [2, 33] is one of a number of programming models that have recently been developed for spatial computers. In Proto, programs are described in terms of dataflow field operators and information flow over regions of continuous space-time. Closely related to Proto are MGS [34], which performs computation and topological surgery on the cells of a k -dimensional CW-complex, and Regiment [35], which operates on data streams collected from space-time regions. A number of “pattern languages”, such as Growing Point Language [36] and Origami Shape Language [28], also use continuous-space abstractions, but have limited expressiveness. There are also a number of discrete-model spatial languages, such as TOTA [4], which uses a viral tuple-passing model, or LDP [37] and MELD [5], which implement a distributed logic programming model. These discrete languages are typically more tightly tied to particular assumptions about scale and communication than languages that use a continuous abstraction.

In Proto, programs are described in terms of operators over regions of continuous space, using the amorphous medium abstraction. An amorphous medium [2] is a manifold with a computational device at every point, where each device can access the recent past state of a neighborhood of other nearby devices. Computations are structured as a dataflow graph of operators on fields. Careful selection of operators allows these programs to be automatically transformed for discrete approximation on a network of communicating devices—typically as a Proto Virtual Machine (VM) binary.

We consider four families of Proto operators: *pointwise*, *restriction*, *feedback*, and *neighborhood*. Pointwise operators (e.g., `+`, `sqrt`, `3`) involve neither space nor time. Restriction operators (e.g., `if`) limit program execution to a subspace. Feedback operators (e.g., `rep`) remember state and specify how it changes over time. Neighborhood operators (e.g., `nbr`, `int-hood`) encapsulate all interaction between devices, computing over space-time measures and neighboring device state.

We will discuss each of the four families of operators, pointwise, restriction, feedback, and neighborhood, in detail in the next section as we establish the well-definedness of Proto. For a full explanation see [2] or the MIT Proto documentation and tutorial [33].

4. WELL-DEFINEDNESS OF PROTO

In this section, we formalize the well-definedness conditions for each family of space-time operators

in Proto and establish that arbitrary compositions of Proto operators are well-defined, so long as they specify finitely-approximable computations. These well-definedness conditions may be summarized as:

- Pointwise operators (except `mux`, which is used to re-combine sub-spaces into a single, coherent space) must have equal domains for all fields.
- a `mux` operation’s first input and output domain must be equal, and its second and third input domains must cover all true and false values in the first input, respectively.
- a `restrict` operation, which changes the domain of a field, must have its input domain must contain its output domain.
- a `delay` operation, used for creating state memory for feedback operations, must have its input domain contain past locations or initialization values for all points in its output domain.
- Operators over neighborhoods must have equal domains for all fields and also equal domains for the field-values at each point in all domains.

In the process of formalizing well-definedness, we also discovered two flaws in the prior operator definitions. These flaws previously led to a lack of well-definedness in certain cases, causing subtle problems in the execution of Proto programs. In the appropriate subsections, we will explain these flaws and how they have been corrected.

4.1. Formal Notation

Proto dataflow programs can be formally represented in an equivalent manner using either mathematical notation or dataflow diagrams. In this paper, we will use both: diagrams for intuition and mathematical notation for analysis and proofs.

In either case, a Proto program may be represented as a collection of *manifolds* (M), *fields* (F), and *operator instances* (O), and *return value* associations between fields and manifolds (R). The manifolds, fields, and operator instances are straightforward specializations of the definitions from Section 2.

The manifolds³ are the space-time region and sub-regions over which the program executes (specializations of X and the set of X_f from Section 2). One of these manifolds must always be the equivalent of X , covering the whole space: we term this the *root manifold*.

The fields are the variables and values of the program, each field ($f \in F$) assigning a value to every point in some region of space-time ($f : m \rightarrow V$, where $m \in M$ and V is any data value). The operator instances are the computations being done to produce the fields, with each operator instance ($o \in O$) taking zero or more input fields and producing precisely one output field

³Technically, we will require all of these to be manifolds with boundary, for reasons explained in Section 4.5.

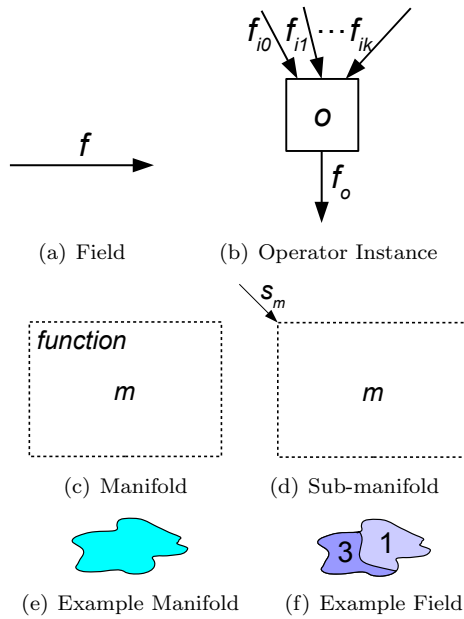


FIGURE 1. Proto programs contain manifolds (spaces), fields that assign values to every point in a manifold, and operator instances that compute fields. We also show examples of evaluation against a manifold at an instant in time, to better illustrate evaluation and domains, as in the example shown of an irregular manifold (e) and a field over that manifold (f) where some points are mapped to 3 and others are mapped to 1.

($o : f_{i_0} \times f_{i_1} \times \dots \times f_{i_k} \rightarrow f_o$, where $f_{i_*}, f_o \in F$). Fields can also be *selectors* for subspaces: given manifold m' and a Boolean-valued field $s_m \in F$ with m' as its domain, we can define sub-manifold m as the closure of the collection of points $\{p \in m' | s_m(p) = true\}$, the part of m' where s_m is true (the reason we use the closure will be explained in Section 4.5).

Finally, return values are pairs $r = (m, f)$, associating a root manifold m (the entire space associated with a function or program), with some field that has m as its domain. Right now, there is precisely one root manifold, but when we introduce function definitions there will be many.

Figure 1 illustrates symbols for manifolds, fields, and operator instances. Operator instances are boxes, with inputs entering in order, left to right, across the top edge, and output exiting the bottom. A field is an arrow going from the operator instance that produces it to the instances that use it as input. Manifolds are large dashed-line boxes, indicating the domain of all fields produced within them. Root manifolds are labeled with a name, while sub-manifolds are nested inside their parent manifold and take a selector field as input. Return values are indicated by a star next to the field's arrow. For pedagogical purposes, we will also show snapshot examples of evaluation against a manifold: each example will also show an example irregular space and fields that might be produced by

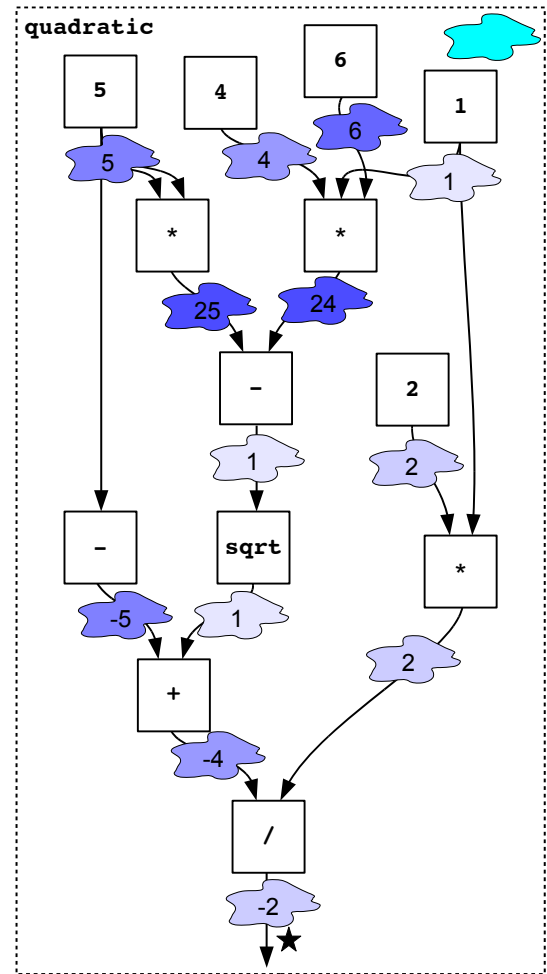


FIGURE 2. Diagram of a Proto expression computing the quadratic formula with $a = 1$, $b = 5$, and $c = 6$, and example of evaluation on an irregular manifold.

evaluation over that manifold at an instant in time.

Figure 2 shows a simple example diagram combining these elements, an expression that computes the quadratic formula with $a = 1$, $b = 5$, and $c = 6$:

```
(let ((a 1) (b 5) (c 6)) ;; define a, b, and c
  (/ (+ (- b)
        (sqrt (- (* b b)
                  (* 4 a c))))
     (* 2 a)))
```

4.2. Pointwise Operators

Pointwise operators are the “normal” operators that each device can execute independently, without considering space or time. Examples include constants (e.g., 3, true), numerical operations (e.g., +, log), sensors and actuators, and function parameters. Because these operators act over all space-time identically, the condition for a pointwise operator instance o to produce a well-defined field is that every input field f_{i_j} and the output field f_o , must all have

the same manifold m as their domain (except the `mux` operator, discussed in Section 4.3).

If a Proto program is composed entirely of well-defined pointwise operators, then it is trivial to prove that the entire program is well-defined.

4.3. Restriction Operators

Restriction expressions create a sub-manifold and evaluate a target expression in that sub-manifold. For example:

```
(restrict (sqrt 3) ;; compute square root of 3
 (< (speed) 1)) ;; on devices moving slower than 1 m/s
```

selects a sub-manifold of slow-moving devices and computes the square root of 3 across that sub-manifold. Such an expression, however, cannot directly use fields from or be used as an input by operator instances in different sub-manifolds, as the well-definedness condition would be violated (a similar example is shown in Figure 3(a)).

The output side of the problem is resolved by means of a pointwise multiplexing operator, `mux`, with a relaxed well-definedness condition. An instance o of the `mux` operator takes three inputs. The range of f_{i_0} is a Boolean: for each point p in its domain m_{i_0} where $f_{i_0}(p)$ is true, $f_o(p) = f_{i_1}(p)$; otherwise, $f_o(p) = f_{i_2}(p)$. We can thus relax the well-definedness condition for `mux` to be:

$$\begin{aligned} m_{i_0} &= m_o \\ m_{i_1} &\supseteq \{p \in m_{i_0} \mid f_{i_0}(p) = \text{true}\} \\ m_{i_2} &\supseteq \{p \in m_{i_0} \mid f_{i_0}(p) = \text{false}\} \end{aligned}$$

In other words: any points whose values will not be used need not be part of the domain of the second and third inputs.

The problem of managing where restriction expressions can be safely used is resolved by disallowing the programmer from using restriction directly. Instead, restriction is made available in an `if` syntactic construct of form:

```
(if test true-expression false-expression)
```

This form creates two complementary sub-manifolds, then combines the results of their expressions with a `mux` using the same selector field, following the template in Figure 3(b). In other words, we prevent the programmer from using `restrict` directly, in favor of using an `if` construct, as a way of ensuring well-definedness of the domain restriction.

Expressions using `if` and `mux` thus provide two forms of branching that look similar, but differ greatly in their semantics. The `if` and `mux` operators share the same syntax in that they both have a selection (a.k.a., test) expression, and two branch expressions — a *true* branch and a *false* branch. Further, for any given point in the

manifold, both operators evaluate and return the result from the branch expression that corresponds to the value of the selection expression at that point (i.e., if the selection evaluates to *true*, then the *true-expression* is executed and returned). However, where the behaviors of the `if` and `mux` operators differ is in their treatment of the unselected branch expression. The `if` operator does **not** execute the unselected branch expression (i.e., it uses `restrict` to limit the domain of the operators and fields in the branch), whereas the unselected branch expression **is** evaluated with the `mux` operator (which is not a syntactic operator and thus cannot affect the evaluation conditions of its branch expressions).

The `if/mux` distinction has important implications for how branching interacts with the state established by feedback operators and the information sharing of neighborhood operators. As we will see below, with `if`, the domain restriction resets state and prevents information from flowing; with `mux`, a state can be retained and information can be shared. A typical distributed computation will need to use both. For example, publish-subscribe needs to differentiate behavior but share information between publishers and subscribers (using `mux`), but for efficiency must restrict the propagation of information to only those who need it (using `if`).

When an `if` expression refers to an external variable, as in:

```
(let ((x 7)) ;; define x as 7
 (if (< (speed) 1) ;; if device is moving slower than 1 m/s
 3 ;; return 3.
 (sqrt x))) ;; Otherwise, return sqrt of x
```

then a `restrict` operator instance is inserted into the reference, limiting its domain (as shown in Figure 3(c)) to be the sub-manifold where the reference occurs. This operator takes one input f_{i_0} , and produces an output f_o whose value is equal to f_{i_0} , but whose domain may be smaller. The well-definedness condition for `restrict` is simply that $m_o \subseteq m_{i_0}$, which is guaranteed by lexical scoping and the `if` construct.

LEMMA 1. Any Proto program composed of pointwise and restriction expressions is well-defined.

Proof. See Appendix A. □

4.4. Space-Time Trajectories of Devices

The other two families of Proto operators, feedback and neighborhood, compute with values from more than one point in space-time. The conditions for these operator instances being well-defined are thus about ensuring that domains of the input include all of the points needed.

Defining these operators also requires us to introduce a notion of the trajectory of a device, since a moving device carries state with it, and may communicate with different sets of other devices at different times.

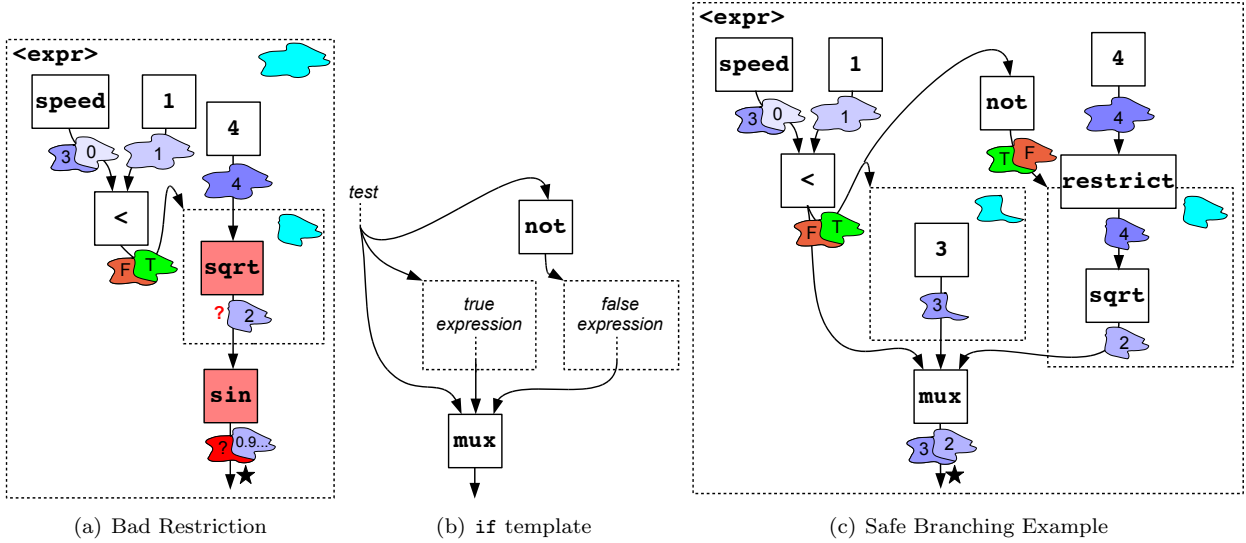


FIGURE 3. Restriction of a computation to a sub-manifold can lead to problems with domain mismatch (a, mismatch shown as red), and example of evaluation on an irregular manifold. Safe restriction is provided by a special form branch construct that creates complementary sub-manifolds whose fields can be multiplexed together to produce a well-defined output over the original manifold (b). External references are passed through a domain-changing `restrict` operator instance, as in the example in (c).

The trajectory of any given device δ over time will be expressed as $\gamma_\delta(\tau)$, a function that maps from a local time τ to coordinates (t, x) in time and space (i.e., the “world-line” of the device, to borrow terminology from relativity; an exploration of the implications of relativity for Proto is beyond the scope of this paper, however). We will assume that a valid collection of such trajectories exists, and that when bundled together they form the entire space-time X . Any given trajectory must be closed, but need not cover the entirety of time (i.e., devices can be created or destroyed). For purposes of this paper, however, we will assume that devices never join, split, or intersect, such that any given point p belongs to precisely one device δ .

4.5. Feedback Operators

State is created in Proto through state evolution functions, which specify an initial value and its evolution over time. This is implemented using a feedback loop with a `delay` operator, which time-shifts values across an arbitrarily small positive time Δ_t (which may further vary by location and time). For example, Figure 4 shows a simple timer:

```
(rep v 0           ;; initialize v to 0, then each time step...
 (+ v (dt)))      ;; ... update v by adding the time delta
```

where v is the state variable, 0 is its initial value, and the update increases the value of v by the elapsed time, Δ_t (measured by the `dt` operator). This allows feedback loops to specify general continuous-time state evolution functions, including discrete valued functions where there is not always a derivative (see [31]).

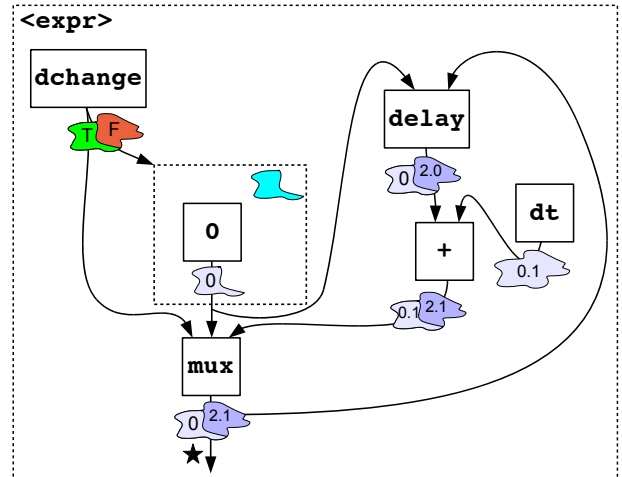


FIGURE 4. Well-definedness of feedback `delay` operators is assured with a syntactic form that guarantees that an initialization expression supplies values whenever there are no past values to be delayed. We also show an example of evaluation on an irregular manifold, where the left portion of the space is just being initialized.

Because manifolds may have different spatial scope at different points in time, the criteria for a `delay` operator o to be well-defined is similar to that of `restrict`: all of the points in the output field f_o must have corresponding time-shifted points in the input field f_{i_0} . Since the value of Δ_t is not resolved at compile-time, to be well-defined, it must be provable that, for every device δ :

$$\forall \delta, \tau \text{ s.t. } \gamma_\delta(\tau) \in m_o,$$

$$\gamma_\delta(\tau) \in m_v \text{ or } \exists \Delta_t > 0 \text{ s.t. } \gamma_\delta(\tau - \Delta_t) \in m_{i_0}$$

where m_o and m_{i_0} are the domains of f_o and f_{i_0} , m_v is the domain of the initial value field, and γ_δ is the trajectory of a particular device δ over time.

The intuition of this definition is simple: the output of a delay must get its values either from values earlier in time or, if there is no earlier in time, from initialization values. We thus introduce a pointwise operator **dchange** that selects the minimum-time surfaces of a manifold, and use a restriction construct, similar to **if**, in which the initialization expression is evaluated on the minimum-time surfaces and the update expression is evaluated elsewhere.

We are not quite done, however, since we need to ensure that the minimum-time surface of a manifold actually exists. Consider, for example, this Proto expression:

```
;; Make a sine wave
(let ((wave (sin (rep x 0 (+ x (dt))))))
  (if (>= wave 0) ;; When the wave is zero or more...
    (rep y 0 (+ y (dt))) ;; ... run a timer.
    (rep z 0 (- z (dt)))) ;; ... same when below zero.
```

In this program, each device times how long a sine wave has been above or below zero. The two branches are not quite identical, however: the regions where (\geq **wave 0**) is true are a collection of closed intervals, since they include zero, while the regions where it is false (below zero) are a collection of open intervals, since they do not include zero. If we switched from \geq to $>$, the two sets of intervals would swap which is open and which is closed.

It is for this reason that we have required all manifolds to be manifolds with boundary and that we have defined a sub-manifold to be the closure of the space where its selector field is true. Open intervals would be a problem, since they do not have a minimum time—the limit is the point just before the interval. Using closed manifolds with boundary ensures that the minimum does exist.⁴

Note that this does mean that there will generally be a measure-zero region in space-time (though frequently non-zero in the spatial dimension) where values are computed in both branches of an **if** statement, but the **mux** operator is still well-defined for this case. More to the point, such replication will not change the result of any *finitely-approximable* function. This concept, defined in detail in [6], says effectively that, for any discrete approximation (e.g., implementation of the program on a real-world collection of devices), as the resolution of the approximation increases, the value computed by the program should converge.

⁴Technically, this is slightly stronger than needed, since we only need minimum-time points, but defining the requirement this way is simple and elegant.

This leads us to another and more subtle condition for well-definedness, which we shall not attempt to enforce: although the condition specified so far ensures that there is always some time-shifted point from where **delay** may obtain values, the overall computation must be finitely-approximable, or else there is nothing to ensure that the value found at that point is reasonable. Consider, for example, the expression:

```
(rep x 0 (- 1 x)) ;; oscillates between 0 and 1
```

The domains will be well-defined for this expression, just as in our previous examples, but the value of **x** depends critically on the sequence of Δ_t steps that are taken to get to a particular point. This is an example of a function that is not finitely-approximable, since its value does not converge as Δ_t goes to zero. At present, however, we are leaving it to the programmer to prevent such problems, since that way lies the intractable difficulty of general program analysis.

Closure of sub-manifolds is a change for Proto, and implies a change in how feedback operations are approximated, so that the update expression is evaluated immediately after initialization. Even though its value is discarded, this means that any feedback or neighborhood operations within the update expression are also initialized, rather than waiting a potentially long time for the next round of approximation.

We can now prove well-definedness for finitely-approximable programs that may include feedback, using a proof similar to that of Theorem 1, above.

LEMMA 2. Any finitely-approximable Proto program composed of pointwise, restriction, and feedback expressions is well-defined.

Proof. See Appendix A. □

4.6. Neighborhood Operators

Neighborhood operators fall into three sub-categories:

- *State-gathering operators* produce output fields where the value at each point is a field over a local neighborhood in space-time. For example, **nbr** collects values from neighbors and **nbr-range** collects distances to neighbors.
- *Pointwise field operators* are much like ordinary pointwise operators, except that the value of each point in the input and output fields is itself a field of values over the local neighborhood. The operators then act pointwise on these neighborhood values. For example, **Field~sqrt** computes square roots of neighborhood values.⁵
- *Summary operators* transform fields of neighborhood values back into ordinary data-valued fields,

⁵The “**Field~**” prefix is specific to the MIT Proto implementation, which generates pointwise field operators from ordinary pointwise operators, using the reserved character “~” in the generated name.

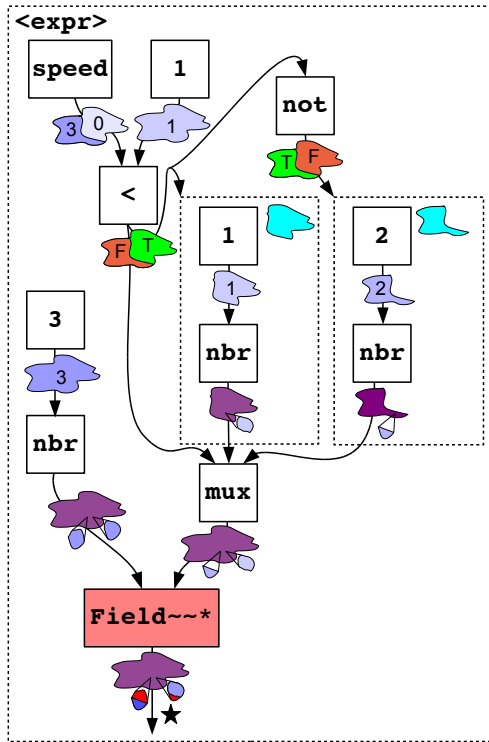


FIGURE 5. Neighborhood operators must all operate on the same domain, as otherwise it is possible to violate the pointwise well-definedness criteria within a neighborhood. In the case shown above, well-definedness fails in the red `Field~*~*` operator instance for points near the boundary of the sub-manifolds, when their domain-restricted neighborhoods are multiplied by neighborhoods gathered in the larger space.

by applying summary operators like integral or minimum.

We will not attempt to constrain the particulars of these neighborhoods, except to note that we assume precisely one neighborhood $n(p)$ to be defined for each space-time point p in the manifold, that this neighborhood is causal (i.e., not using points that are not accessible to p , such as those in the future), and that the neighborhood includes at least the point p itself.

The well-definedness criteria for state-gathering and summary operators is the same as that for pointwise operators: all the input fields f_{i_j} and the output field f_o have the same domain m . The pointwise field operators, on the other hand, have a stronger criteria. In addition to the normal pointwise criteria, an operator instance o must satisfy the pointwise well-definedness criteria for each set of neighborhood values. In other words, for every point $p \in m$, all of the values $f_o(p)$ are expected to be fields mapping from $m_{f_o}(p) \rightarrow V$, where $m_{f_o}(p) \subseteq n(p)$ is a closed subspace of the neighborhood. This is likewise the case for all input fields $f_{i_j}(p)$. For such a pointwise field operation to be well defined, the domain of $f_o(p)$ must be equal to the domain of $f_{i_j}(p)$ for all inputs.

This means that fields gathered in different sub-manifolds cannot be safely combined. Consider, for example, an innocuous-looking statement like the following (Figure 5):

```
(* (nbr 3)           ;; multiply the neighbor-3 field
  (if (< (speed) 1)  ;; depending on the device's speed
      (nbr 1) (nbr 2))) ;; by nbr field from a sub-manifold
```

Near the boundary between the sub-manifolds delineated by `< (speed) 1`, the neighborhoods gathered within each sub-manifold are truncated, omitting values from points in the complementary sub-manifold. The `(nbr 3)` expression, however, can gather values from the full neighborhoods. As such, well-definedness fails for points near the boundary of the sub-manifolds, when their domain-restricted neighborhoods are multiplied by neighborhoods gathered in the larger space.

Preventing this is simple: we constrain the input types of `mux` to non-neighborhood values, which prevents neighborhood-valued fields from exiting an `if` or feedback construct. Branching over neighborhood-valued fields is thus only handled by `Field~*~*`, the pointwise field analogue of `mux`. On the other side, the `restrict` operator can be safely applied to neighborhood-valued fields, as long as its definition is taken to also include restricting the domains of the neighborhoods.

The restriction on types `mux` and application of `restrict` to neighborhood domains are changes for Proto. By making these changes, have we lost any expressiveness in neighborhood computations? The differences between `if`-based computations and `mux`-based computations can only be observed in the behavior of neighborhood, `delay`, and actuator operator instances within their sub-expressions. Proto already prohibits neighborhoods of neighborhoods and delay of neighborhood-valued fields. Actuation over neighborhood-valued fields was not previously prohibited, but this was incorrect: multiple actuator calls in the discrete approximation have ill-defined and unpredictable effects. Thus this definition change only removes bugs and does not change functionality.

We can now prove well-definedness for all finitely-approximable programs:

THEOREM 3. Any finitely-approximable Proto program composed of pointwise, restriction, feedback, and neighborhood operators is well-defined.

Proof. See Appendix A. □

5. FUNCTION CALLS IN PROTO

We now return to our original problem: to create a model of space-time function evaluation for Proto and to prove that well-definedness is guaranteed for all such function calls. Previously in Proto, there has been no explicit model of space-time function evaluation. Instead, function evaluation was defined at the syntactic

level, and resolved by complete inlining at compile time because it simplified the compiler's global-to-local transformation. This inlining had detrimental effects on the Proto compiler and language. First, function inlining unnecessarily increased the size of the binary files that were produced by the Proto compiler because the binary representation of the function was duplicated for every instance of its invocation. Second, function inlining prevented Proto from taking advantage of desirable language features, such as recursion.

We will now, however, formally define space-time function evaluation using a substitution model, then create a model of in-place evaluation with reference to this substitution model. The result of all of the work on the well-definedness of space-time functions presented in Section 4 is that these models be elegantly simple, both to define and to implement (as will be shown in Section 6. As an additional benefit, the function call semantics we develop do not differ in syntax or expected behavior for any form of function call that was previously supported by inlining.

5.1. Defining Functions

Before we can call functions, we must define them. Given Proto code for a function, we produce a function definition in three stages:

1. Create a new root manifold m_d for the function definition.
2. For each input to the function, create a "parameter" operator instance that returns the values of that input.
3. Evaluate the expression(s) for the function body in this environment.

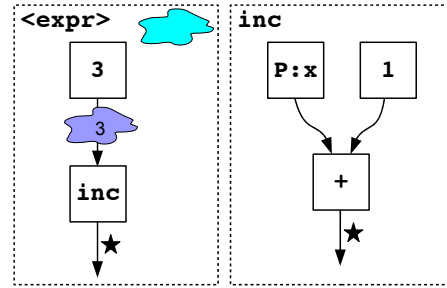
This yields a function definition $d = (m_d, \{(j, f_j)\}, r_d)$ that is a tuple of three values: first, the root manifold m_d that contains the sub-program for the function. Second, a set of pairs (j, f_j) , one per input, which map the j th input to parameter operator instance f_j . Finally, a return value r_d that designates which field in the function will become the source of values for the output field of any function instance (this designation is needed because the output cannot be determined *a priori* from field/operator relations, but must be drawn from expressions structure). Given these definitions, we will say that an operator instance o' is used by d if $m_{o'} \subseteq m_d$.

Figure 6(a) shows an example diagram of an increment function:

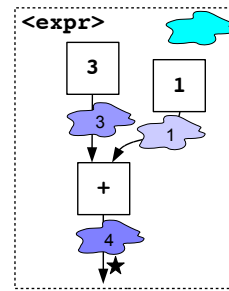
```
(def inc (x) ;; define an increment function (inc) that
  (+ x 1)) ;; returns its argument plus one
```

evaluated in the expression:

```
(inc 3) ;; returns scalar value 4
```



(a) Function Call



(b) After Substitution

FIGURE 6. The substitution model of space-time function evaluation copies the contents of a function definition in place of the function call operator instance. We show an example of evaluation on an irregular manifold both before (a) and after (b) substitution.

Note that the `inc` function and the base expression `<expr>` are each contained in an independent manifold.

5.2. Substitution Model

Consider an instance o of a function call. The output of this function call is a field, which maps points in manifold m_o to data values. To evaluate the function call by substitution, we begin by copying all of the operator instances, fields, and sub-manifolds in d , substituting m_o for m_d anywhere that it occurs. The only elements not copied are the parameters and their fields: for each copied operator instance, we replace all instances of the field output from the j th parameter, f_j , with the j th input, f_{i_j} . Next, we take the function's return value $r_d = (m_d, f_d)$ and replace all instances of the output of the function call f_o with f_d . Finally, the operator instance o and its output f_o may be discarded, completing the substitution.

This model is a straight-forward extension of the substitution models commonly used in conventional functional programming languages. The only difference is the inclusion of the manifold in the mapping. Figure 6(b) shows an example of substitution for the example call of `(inc 3)`.

5.3. In-Place Model

The converse model, of a function call in place, follows from the substitution model in a straightforward

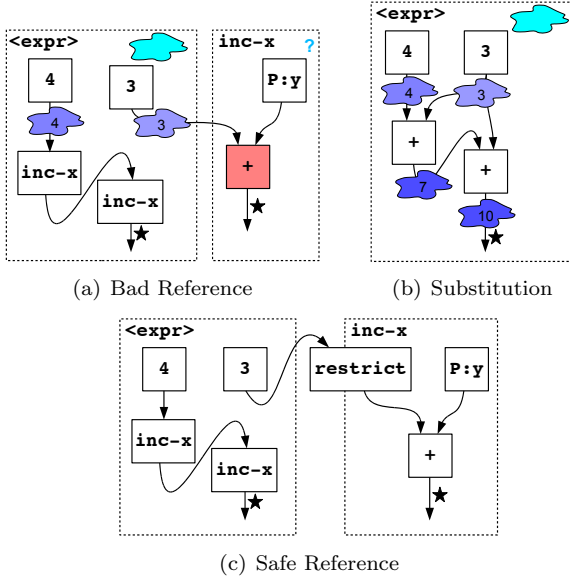


FIGURE 7. Direct references from a function to external fields mismatch domains (a, mismatch shown as red), even though substitution evaluation will sometimes be correct (b). This problem can be addressed by inserting a domain-changing **restrict** operator instance on external references (c), similarly to a restriction operator. An example of evaluation is shown in Figure 8

manner. We begin by setting m_d to be equal to m_o , then copy the values of the input fields f_{i_j} into their corresponding parameter fields, f_j . All of the values in d may then be calculated, and the return values copied from f_d into f_o .

The advantage of the in-place model over the substitution model is that it allows a function to be evaluated without modifying the program—only the *values* of manifolds and fields are manipulated. We thus choose the in-place model for dynamic evaluation of space-time functions in Proto. We will denote such an evaluation of function definition d in the context of operator instance o as $\mathcal{E}_{d,o}$, such that $\mathcal{E}_{d,o}(\ast)$ is the value of any element \ast under evaluation. Our task is to ensure that evaluation gives a well-defined value for every point in every field.

5.4. References to External Variables

The model presented so far suffices for self-contained functions of pointwise operators, like the example of **inc** above. But what if the function references an external variable? Consider this example:

```
(let ((x 3))           ;; define x as 3
  (def inc-x (y)      ;; define a function
    (+ y x))         ;; that increments a scalar by x
  (inc-x (inc-x 4)))  ;; returns (4 + x) + x
```

As shown in Figure 7(a), this type of a reference is problematic: any pointwise operator instance o that uses an external field f as an input is not well-defined,

since the field's domain m_f is not related to function's root manifold m_d until the function is evaluated. Previously, this problem was partially masked by the complete inlining conducted at compile-time, but still led to a bug in the MIT Proto implementation, where variables within **if** clauses behaved differently when referenced once or multiple times.

Our solution comes from the part of Proto designed for domain changes: restriction expressions. References in functions can be handled similarly, by inserting **restrict** operators wherever a function definition references an external variable (e.g., Figure 7(c)). We also generalize well-definedness for **restrict** for functions to be $\mathcal{E}_{d,o'}(m_o) \subseteq m_{i_o}$, meaning that domains must match only when the function is evaluated in the context of operator o' . We can now prove that functions incorporating restriction and reference will be well-defined.

THEOREM 4. Let d be the definition for a Proto function, and o and o' be operator instances. If o is used by d then o and $\mathcal{E}_{d,o'}(o)$ are well-defined.

Proof. See Appendix A. □

An unusual and important aspect of the proof for this theorem is that correctness depends critically on lexical scoping: it is because declaring variables and restricting domains via **if** both obey the same scoping rules, a reference to a variable can never have a larger domain than the variable itself.

This is problematic for making truly first-class functions, however, because it means that closures are not possible. This is because evaluation becomes ill-defined when the domain of the evaluation is not a subspace of the domain of external fields referred to in the function definition. For example, consider the Proto expression:

```
;; slow is true for slow-moving devices
(let ((slow (< (speed) 1)))
  ;; fun-diff names a function that differs based on device
  ;; speed. On slow devices, add unique ID to the argument.
  ;; On other devices, take the square-root of the argument.
  (let ((fun-diff (if slow
    (let ((x (mid)))
      (fun (y) (+ x y)))
    sqrt)))
    ;; fun-all equals fun-diff of the nearest slow device.
    (let ((fun-all (broadcast slow fun-diff)))
      (apply fun-all 4)))) ;; Now call fun-all.
```

The **broadcast** would produce a field with the function **(fun (y) (+ x y))** at every point throughout the root manifold. Yet the field named **x** is only defined in the sub-manifold where **test** is true. Thus, such passing of functions as field values cannot be permitted.

Nevertheless, many of the desirable properties of first-class functions can be obtained through dynamic allocation of processes (see [38]). There are also a limited set of cases where it is safe to pass functions as values. The challenge of obtaining the functionality

of first-class functions in Proto is still an open investigation, so we will not discuss it in detail at present.

6. REFERENCE IMPLEMENTATION

We have extended MIT Proto to implement space-time function evaluation, as well as changing the handling of `mux`, `feedback`, and `actuator` operators to match the descriptions above so that well-definedness can be assured. This section discusses the changes to Proto that were required to implement space-time function evaluation and its effect on the size of the executable binary and available language features. These changes are currently included in the reference implementation of Proto that is freely available at [39].

6.1. Implementation in MIT Proto

The Proto virtual machine [40] is a stack-based virtual machine with two stacks: a “data” stack used by most instructions and an “environment” stack used for storing local variables. The Proto virtual machine also contains local storage of state variables and flags. An implementation of the Proto VM is included in MIT Proto, along with a Proto code emitter, which linearizes Proto programs into executable binaries.

In order to implement function evaluation for MIT Proto, we made three key additions to the Proto VM and the code emitter: a new `FUNCALL_OP` instruction, a preprocessor that uncurries external references into extra function parameters, and support for recursion.

6.1.1. Emitting Function Calls

During emission, functions are linearized one at a time into a sequence of Proto VM instructions. The order in which they are linearized implies their location in the global memory of a running Proto VM, which the emitter tracks as it executes.

Function calls are implemented by a new `FUNCALL_OP` instruction, parametrized with the number of arguments to be consumed. A function of k inputs takes $k + 1$ arguments from the data stack. The first is a reference to the function’s location in memory (which the emitter obtains from its function name map). The rest are the function arguments, which are placed into the environment stack and accessed like any other local variable. The program counter is then set to the start of the function and executed, returning a value on the data stack.

6.1.2. Uncurrying Restrictions

References to fields computed outside of a function present a problem: if they are stored in either the data stack or the environment stack, their depth in the stack is determined by the context of the call. To deal with this problem, we use an uncurrying method where every domain-conversion operator adds an implicit

parameter to the function. This is implemented by means of a preprocessing pass that, for each external-reference `restrict` encountered, 1) adds a parameter to the function (compound operator) definition, and 2) converts the `restrict` operator instance into a parameter operator instance.

Neighborhood operators are implemented using function calls, which transform a collection of neighborhood operations into a single map/reduce operation to be performed over the values of each neighbor in turn. These present a case where this uncurrying method does not work, because the virtual machine implementation of the map/reduce operation expects functions with a fixed number of inputs. Thus, if the automatically generated functions used to implement the neighborhood operations contain external references (including values that are merely not part of the neighborhood computation), the above uncurrying method cannot be used. Instead, since this function call is a lambda function generated by the compiler, we can be sure that no other portion of the program can call it and that it will only be used by the neighborhood folding operator. We instead set up the environment stack before the neighborhood operation, placing all external parameters to the map/reduce operation on the stack, so that they are available to the function as input parameters. Since the neighborhood operator is a single instruction, we can ensure that no other operator will modify the environment stack until the neighborhood is finished, at which point we remove the parameters and continue.

6.1.3. Closure of Feedback Sub-manifolds

The sub-manifolds used by feedback operations must be closed (See Section 4.5), to ensure well-definedness. To implement this, we add a flag to the feedback initialization operator when the initialization function executes. When this flag is active, we force value of Δ_t to be 0 during subsequent execution of the update function. Once the value of the feedback variable is stored and the feedback operation is complete, we remove this flag and allow Δ_t to be calculated as normal.

6.1.4. Recursion

The Proto compiler previously relied on a function to be defined before it could be used. This caused errors when a function was called in the body of its own definition, thus blocking recursion. Implementing late function binding for recursion simply required inserting an empty function “placeholder” in the function lookup table upon its first reference. The placeholder is replaced when the function is properly defined.

Implementing late function binding in the compiler was necessary for adding recursion as a Proto language feature, but the VM also needed a change to support recursion. The first instruction of every Proto binary

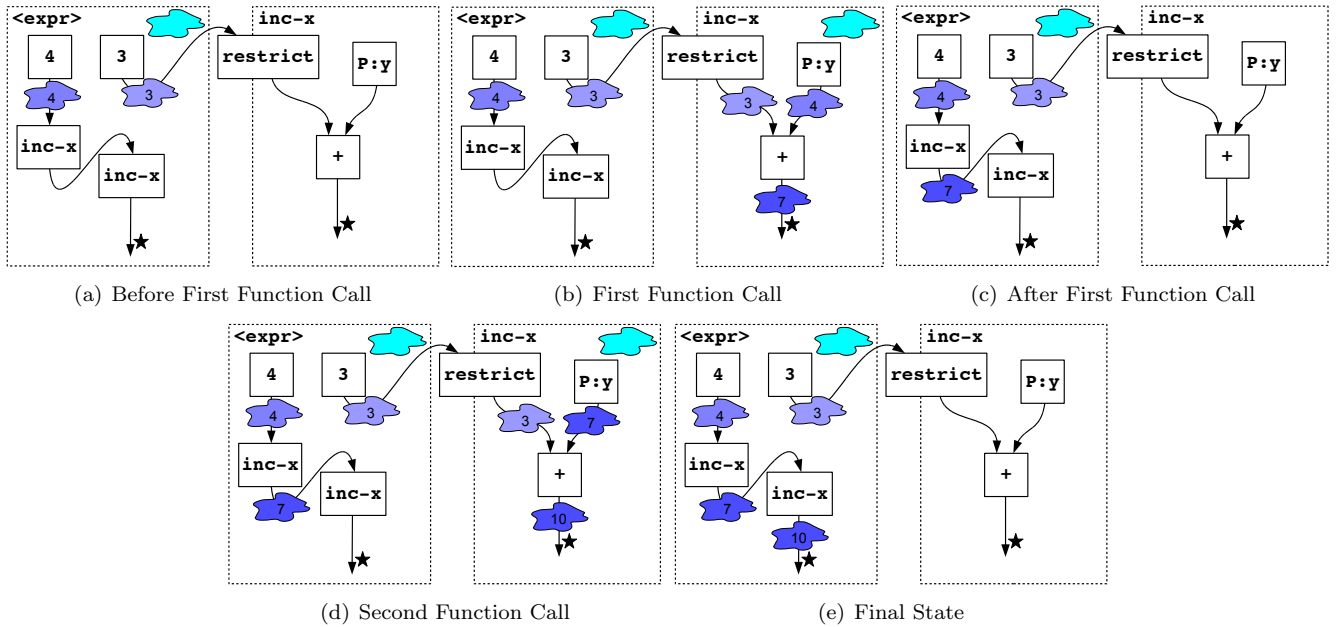


FIGURE 8. Example of evaluating the program from Figure 7(c) against an irregular manifold using in-place evaluation. At each function call, the manifold and inputs are copied over to the function’s root manifold, the function is evaluated, and the return value copied back.

previously specified the size of the stacks necessary for executing the localized Proto program. However, in the case of recursion, the depth of the stack cannot be known until runtime and can vary on every device (e.g., (*factorial (mid)*)). Therefore, we implemented a dynamically-sized stack to store the necessary stack memory for the virtual machine.

6.2. Effect of Function-Calls on Binary Size

Function calls should significantly reduce the size of Proto VM binaries, since the function code need not be duplicated for each use. We verify this empirically by comparing the binary sizes of programs using the new function-call method versus completely inlined programs.

Figure 9 shows how the size of the compiled binary scales with function size and number of calls. As expected, function call overhead means inlining is smaller for very small or infrequently called functions, but when a non-trivial function is called multiple times, the function-call strategy produces smaller binaries and the benefits scale approximately linearly.

At present, the MIT Proto compiler decides whether to inline based on the number of operator instances in the function. When there are less than a fixed threshold (default 10), the function is inlined. Our validation experiment shows that this heuristic should also include the number of function-calls, and this simple improvement is planned as future work.

The next experiment compares the size of compiled binaries for recursive functions using both the inlining and substitution methods. We use *factorial* as our

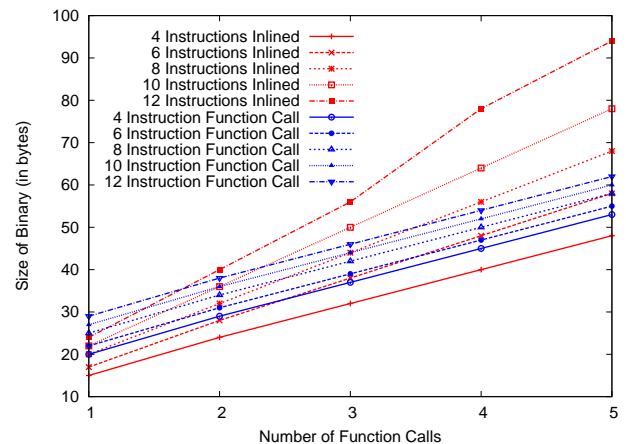


FIGURE 9. A comparison of the compiled binary size for a varying number and size of function calls.

recursive function, implemented in Proto as:

```
(def factorial (n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

One complication with this test was dealing with Proto’s compile-time optimizations. When Proto compiles the program (*factorial 5*), it performs all calculations at compile time and returns the answer, 120. On the other hand, when Proto compiles the program (*factorial (mid)*), all calculations must be evaluated at runtime since that is the earliest time when a machine’s ID is known. However, it is impossible to inline a recursive function call whose termination point

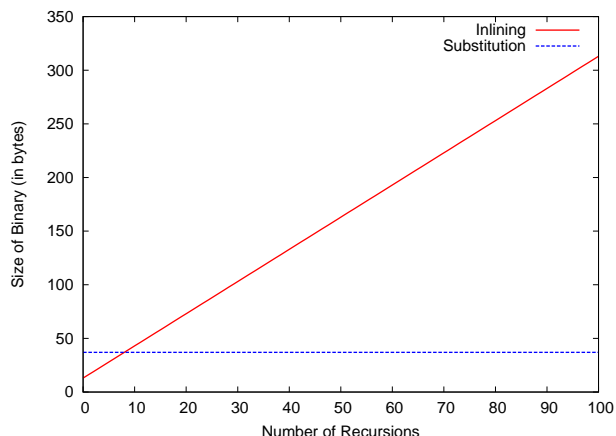


FIGURE 10. A comparison of the binary size for the recursive implementation of *factorial*, compiled using the inlining and substitution methods.

is unknown at compile-time. Thus, for the purposes of this experiment, we manually construct the inlined recursive function evaluations like so for the example, $n = 5$:

```
(* 5 (* 4 (* 3 (* 2 (* 1 (mid))))))
```

Note that `(mid)` replaces the recursion value, which should be 1 for a proper factorial computation to prevent Proto from computing the result of the expression at compile-time, however it will not affect the size of the binary since the literal value 1 is represented by the same number of opcodes as the `(mid)` expression.

Figure 10 shows how the size of the compiled binary varies as n increases for inlined versus substituted evaluations of `(factorial n)`. As expected, the results show that the substitution method produces smaller binaries than inlining for $n > 8$. Where the substitution method maintains a constant binary size for increasing depth of recursive calls, function inlining exhibits linear growth in the binary size. Thus, deep recursion with function inlining produces needlessly large Proto VM binaries.

7. DISCUSSION: IMPLICATIONS FOR OTHER DISTRIBUTED COMPUTATION MODELS

Finally, let us return to the question of more general implications for distributed computing models. We have examined the problem of distributed function calls in an extreme environment: infinite numbers of devices embedded throughout a continuous region of space-time. Moreover, the computational model we investigate is distributed at every stage: a function call may originate on many devices, be evaluated on many devices, and return values to many devices.

Doing so has forced us to encounter a number of problems that might otherwise be deferred or restricted out of the scope of a less general distributed computing

model. The challenges to well-definedness that we have discussed for Proto are not special cases, but may be restated as general problems for distributed computing, not all of which currently have solutions.

From the results for Proto presented above, we may thus identify four general challenges that any distributed computing model must address:

- Domains for Communication and State:** Throughout Section 4 and Section 5, we found potential problems that needed to be addressed in ensuring that no operator ever is missing a value on a device where it is executed. This is a problem for all distributed computation models, as distributed computations frequently need to be able to restrict which devices share information and how long state should be retained. If a computation imposes such restrictions, then its subroutines must be limited by the same restrictions. Otherwise, a computation may be rendered incorrect by contamination from stale state or by information leaking from devices that should not be under consideration. For functional abstraction to be possible and safe, this context information needs to be implicitly supplied and enforced—otherwise, a function may disobey the restrictions of the context in which it is called. Proto provides one solution, using the `if/mux` dichotomy to manage domain restriction. This approach does not require continuous space-time, and thus could be adapted for other distributed computing models as well.
- Well-Defined Branching:** In Section 4.3 and 4.6, we found that branches could have ill-defined values if variables used values from different sets of devices. This is also a general problem, as in any distributed computing model a branch will in general be taken in different directions by different devices. Any branch thus forms a case of restriction, where the domains are the subsets of devices evaluating each branch. It is necessary to ensure that these restrictions are imposed uniformly on all variables used by computations within the branch—whether or not the variable was initially computed within the branch. Otherwise, incorrect behavior may arise from interactions along the boundaries between branch domains. In Proto, this is enforced by means of implicit restriction on variables external to a branch, and by applying additional limits on restriction of neighborhood values. These methods could be adapted for other computational models as well.
- Initialization and Update:** In Section 4.5, we showed that in stateful computations in Proto, it is necessary for the state update to occur at the same time as state initialization. Failing to do so leads to well-definedness problems in any nested state function because of the offset

between initialization and update. This indicates a problem that can afflict any stateful distributed computation. If initialization and updates do not occur simultaneously, then nested computations may be offset from one another, potentially causing elements of the computation to become problematically decoupled.

- **Scoping:** Scoping of distributed computations is a critical open problem. The two standard solutions developed for local computation, lexical scoping and dynamic scoping, are both inadequate models for distributed computation.

This becomes quite obvious in the context of function closures, as discussed in Section 5.4. For both dynamic and lexical scoping, restriction of domain means that a variable may not have a defined value at the location in space-time where the function is evaluated—even if a definition of that variable is within the syntactic scope. If a value is transported from elsewhere (a natural fix to attempt with lexical scoping), which of potentially many possible sources with different values should be used?

All of these challenges may, of course, be evaded or ignored in any given distributed computing model. Indeed, most existing models either include a centralized element in their semantics (thereby evading the necessity of facing some of these challenges) or else leave them as a problem for the programmer—which the programmer is no better equipped to deal with.

These problems, however, must not be ignored. They are entangled with issues at the very foundations of software engineering: abstraction, branches, state, and scoping. We contend that the fact that most distributed computing models lack good mechanisms for addressing these issues is a major impediment to the construction of complex distributed systems. Conversely, when these issues are addressed, we expect the reliability, scope, and accessibility of distributed systems engineering to improve markedly.

8. CONTRIBUTIONS

We have formalized Proto’s model of function evaluation and extended it to include in-place evaluation. This allows significant reduction in the size of the compiled binary and dynamic function calls (e.g., for recursion). Our analysis of Proto semantics to ensure correctness of function calls revealed other challenges to well-defined program execution, which we have addressed as well. We have upgraded MIT Proto, implementing function calls, and verified that function calls reduce the size of the compiled binary as expected, and have showed how our results generalize beyond Proto to the engineering of any distributed computation.

Our future plans capitalize on these improvements to support higher-order functions and first-class space-time processes. One enhancement will support higher

order functions, such as map and reduce, which use functions in their input or output, but which do not attempt to move them around as data. Additionally, although we have shown that first-class functions cannot in general be data values of a field, we believe that space-time processes [38] may be able to provide the same capabilities, and this is an important open area of investigation, as noted in the previous section. As with this work, we expect that our results will both improve Proto and show the way for more general improvement of distributed computing models.

ACKNOWLEDGMENT

Work partially sponsored by DARPA DSO under contract W91CRB-11-C-0052; the views and conclusions contained in this document are those of the authors and not DARPA or the U.S. Government.

REFERENCES

- [1] Beal, J., Dulman, S., Usbeck, K., Viroli, M., and Correll, N. (2012) Organizing the aggregate: Languages for spatial computing. *CoRR*, **abs/1202.5509**.
- [2] Beal, J. and Bachrach, J. (2006) Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, **21**, 10–19.
- [3] Giavitto, J.-L., Michel, O., Cohen, J., and Spicher, A. (2004) Computation in space and space in computation. Technical Report 103-2004. Universite d’Evry, LaMI, Evry, France.
- [4] Mamei, M. and Zambonelli, F. (2008) Programming pervasive and mobile computing applications: the TOTA approach. *ACM Transactions on Software Engineering and Methodology*, **18**, 15:1–15:56.
- [5] Ashley-Rollman, M. P., Goldstein, S. C., Lee, P., Mowry, T. C., and Pillai, P. (2007) Meld: A declarative approach to programming ensembles. *IEEE International Conference on Intelligent Robots and Systems (IROS ’07)*, New York, October, pp. 2794–2800. IEEE.
- [6] Beal, J. (2010) A basis set of operators for space-time computations. *Spatial Computing Workshop*, New York, September, pp. 91–97. IEEE.
- [7] Woods, D. and Naughton, T. (2005) An optical model of computation. *Theoretical Computer Science*, **334**, 227–258.
- [8] MacLennan, B. (1990) Continuous spatial automata. Technical Report Department of Computer Science Technical Report CS-90-121. University of Tennessee, Knoxville, Knoxville, Tennessee.
- [9] Dean, J. and Ghemawat, S. (2008) Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, **51**, 107–113.
- [10] Apache Hadoop (Retrieved March 6, 2012). <https://hadoop.apache.org/>.
- [11] Condor High Throughput Computing (Retrieved March 6, 2012). <http://research.cs.wisc.edu/condor/>.
- [12] BOINC Open-source software for volunteer computing and grid computing (Retrieved March 6, 2012). <https://boinc.berkeley.edu/>.

- [13] Oracle Grid Engine (Retrieved March 6, 2012). <http://www.oracle.com/us/products/tools/>.
- [14] The Message Passing Interface (MPI) standard (Retrieved March 6, 2012). <http://www.mcs.anl.gov/research/projects/mpi/>.
- [15] Version 5.9 (2011) *Erlang Reference Manual User's Guide*. Ericsson AB. Stockholm, Sweden.
- [16] Cardellini, V., Colajanni, M., and Yu, P. (1999) Dynamic load balancing on web-server systems. *Internet Computing, IEEE*, **3**, 28–39.
- [17] BitTorrent (Retrieved March 6, 2012). <http://www.bittorrent/>.
- [18] Tribler (Retrieved March 6, 2012). <http://www.tribler.com>.
- [19] Napster (Retrieved March 6, 2012). <http://www.napster.com>.
- [20] Skype (Retrieved March 6, 2012). <http://www.skype.com>.
- [21] OMG Data Distribution Service Portal (Retrieved March 6, 2012). <http://portals.omg.org/dds/>.
- [22] Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H. (2001) Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, **31**, 149–160.
- [23] Aspnes, J. and Ruppert, E. (2009) An Introduction to Population Protocols. In Garbinato, B., Miranda, H., and Rodrigues, L. (eds.), *Middleware for Network Eccentric and Mobile Applications*. Springer, New York, NY, USA.
- [24] Olfati-Saber, R., Fax, J. A., and Murray, R. M. (2007) Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, **95**, 215–233.
- [25] Lynch, N. (1996) *Distributed Algorithms*. Morgan Kaufmann, San Francisco, USA.
- [26] Dolev, S., Gilbert, S., Lynch, N. A., Schiller, E., Shvartsman, A. A., and Welch, J. L. (2004) Virtual mobile nodes for mobile ad hoc networks. *Distributed Computing (DISC)*, New York, NY, USA, July, pp. 230–244. Springer.
- [27] Dolev, S. and Tzachar, N. (2011) Self-stabilizing and self-organizing virtual infrastructures for mobile networks. In Nikolettseas, S. and Rolim, J. D. (eds.), *Theoretical Aspects of Distributed Computing in Sensor Networks* Monographs in Theoretical Computer Science. An EATCS Series, pp. 621–653. Springer Berlin Heidelberg, Berlin.
- [28] Nagpal, R. (2001) Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics. PhD thesis MIT Cambridge, MA, USA.
- [29] Usbeck, K. and Beal, J. (2011) An agent framework for agent societies. *Proceedings of Systems, Programming, Languages and Applications: Software for Humanity*, Portland, Oregon, USA, Oct, pp. 201–212. SPLASH.
- [30] Beal, J. and Bachrach, J. (2007) Programming manifolds. In DeHon, A., Giavitto, J.-L., and Gruau, F. (eds.), *Computing Media and Languages for Space-Oriented Computation*, Dagstuhl Seminar Proceedings, **06361**. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany.
- [31] Bachrach, J., Beal, J., and Fujiwara, T. (2007) Continuous space-time semantics allow adaptive program execution. *IEEE SASO 2007*, New York, July, pp. 315–319. IEEE.
- [32] Viroli, M., Beal, J., and Casadei, M. (2011) Core operational semantics of proto. *ACM Symposium on Applied Computing*, New York, NY, USA, March, pp. 1325–1332. ACM.
- [33] MIT Proto (Retrieved March 6, 2012). <http://proto.bbn.com>.
- [34] Giavitto, J.-L., Godin, C., Michel, O., and Prusinkiewicz, P. (2002) Computational models for integrative and developmental biology. Technical Report 72-2002. Universite d'Evry, LaMI, Evry, France.
- [35] Newton, R. and Welsh, M. (2004) Region streams: Functional macroprogramming for sensor networks. *First International Workshop on Data Management for Sensor Networks (DMSN)*, New York, August, pp. 78–87. ACM Press.
- [36] Coore, D. (1999) Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer. PhD thesis MIT Cambridge, MA, USA.
- [37] Rosa, M. D., Goldstein, S. C., Lee, P., Campbell, J. D., and Pillai, P. (2008) Programming modular robots with locally distributed predicates. *IEEE International Conference on Robotics and Automation (ICRA '08)*, New York, May, pp. 3156–3162. IEEE.
- [38] Beal, J. (2009) Dynamically defined processes for spatial computers. *Spatial Computing Workshop*, New York, September, pp. 206–211. IEEE.
- [39] (Retrieved November 22, 2010). MIT Proto. software available at <http://proto.bbn.com/>.
- [40] Bachrach, J. and Beal, J. (2007) Building spatial computers. Technical Report MIT-CSAIL-TR-2007-017. MIT, Cambridge, MA.

APPENDIX A. PROOFS

Appendix A.1. Well-Definedness of Proto

LEMMA 1. Any Proto program composed of pointwise and restriction expressions is well-defined.

Proof. Consider any operator instance $oinO$. By assumption, either o is an instance of **restrict** or **mux**, or another pointwise operator.

Assume that o is a pointwise operator instance besides **mux**. By our syntactic assumptions, the only way for a domain to change is via the **if** construct. This means that o 's output can only differ from the input if one of its inputs is a variable defined outside of the innermost **if** construct containing it—all other variables in scope are in the same construct and therefore have the same domain. In this case, however, we have defined that a **restrict** operator is inserted into the reference, limiting the domain of the variable to be equal to that of o . Thus o is well-defined.

If o is a **restrict** operator instance, then, by our syntactic assumptions, it can only have been produced by such an external variable reference. If m_o is the

domain of its output and m_{i_0} is the domain of its input, and there are n `if` constructs nested between the definition and the reference, then let s_{m_k} be the selector field for the k th submanifold. We then have the following relations:

$$m_{n-1} = C(\{p \in m_{i_0} \mid s_{n-1}(p) = \text{true}\})$$

$$m_{k-1} = C(\{p \in m_k \mid s_{k-1}(p) = \text{true}\})$$

$$m_o = C(\{p \in m_1 \mid s_{m_o}(p) = \text{true}\})$$

where $C(*)$ is the closure of set $*$. In other words, m_o is generated as a chain of successive domain reductions. Concatenating these relations together, we have $m_o \subseteq m_{i_0}$, meaning that o is well-defined.

Finally, if o is a `mux` operator instance, then either it was declared directly by the programmer or it was generated as part of an `if` construct. If declared directly, it behaved just as any other pointwise operator, and is well-defined by the same logic as above. If generated by an `if` construct, then by the structure of the construct we have $m_o = m_{i_0}$, the domain of f_{i_1} is guaranteed to be $C(\{p \in m_{i_0} \mid f_{i_0} = \text{true}\})$ and the domain of f_{i_2} is guaranteed to be $C(\{p \in m_{i_0} \mid f_{i_0} = \text{false}\})$. Since $C(*) \supseteq *$, we then know that o is well-defined.

Thus, every operator instance is well-defined and therefore the program as a whole is well-defined. \square

LEMMA 2. Any finitely-approximable Proto program composed of pointwise, restriction, and feedback expressions is well-defined.

Proof. We begin by splitting the proof into two cases: either o is a pointwise or restriction operation, or else o is one of the new functions introduced: `delay`, `dchange`, or `dt`.

In the first case, for pointwise or restriction operations, the same logic as in Lemma 1 holds: because of the syntactic constraints that we have imposed, o is always well-defined: even when an input comes from one of the new functions, its domain is syntactically constrained to the same expected by the pointwise or restriction operation that consumes it.

We next note that, although they are newly introduced and have special meanings that take careful implementation, the `dchange` and `dt` operations are both pointwise functions with no inputs, and therefore are automatically always well-defined.

This leaves only `delay` operators. Syntactically, we know that the output domain m_o is always equal to the input domain m_{i_0} , since the `dchange` operator is true only on a measure zero boundary. Since we assume the function is finitely-approximable, it remains only to prove that the well-definedness condition:

$$\forall \delta, \tau \text{ s.t. } \gamma_\delta(\tau) \in m_o,$$

$$\gamma_\delta(\tau) \in m_v \text{ or } \exists \Delta_t > 0 \text{ s.t. } \gamma_\delta(\tau - \Delta_t) \in m_{i_0}$$

will necessarily hold.

Consider any point $p \in m_o$. Because devices are non-intersecting, p belongs to precisely one device δ , which implies that there is precisely one local time τ such that $\gamma_\delta(\tau) = p$. We now consider the interval $\gamma_\delta \cap m_{i_0}$ where the device trajectory intersects the input domain. By the definition of selectors, we know this interval is closed, and includes a point with the minimum local time τ_- . If $\tau > \tau_-$, then any $\Delta_t \leq \tau - \tau_-$ will satisfy the well-definedness condition. Otherwise, we know that $\tau = \tau_-$, and therefore the value of `dchange` for $\gamma_\delta(\tau)$ is true. This then implies that $\gamma_\delta(\tau) \in m_v$, also satisfying the well-definedness condition.

Thus, every operator instance is well-defined and therefore the program as a whole is well-defined. \square

THEOREM 3. Any finitely-approximable Proto program composed of pointwise, restriction, feedback, and neighborhood operators is well-defined.

Proof. As before, we begin by splitting the proof into two cases: either o is one of the new neighborhood operations, or it is a pointwise, restriction, or feedback operation. For all pointwise, restriction, or feedback operations, the same logic as in Lemma 2 holds: because of the syntactic constraints and closure assumptions that we have imposed, o is always well-defined.

If o is a state-gathering operation, it works exactly the same as the previous case, since neighborhoods are always assumed to be defined. Likewise, if o is a pointwise field operation, it operates similarly, except that we must also show that the domains of the neighborhoods for every point p are identical for all input fields f_{i_j} and the output field f_o . By the same logic as before, we can show that the domains of all f_{i_j} and f_o are the same manifold m_o . By the extended definition of `restrict`, this means that the domains of the neighborhoods for every point p have also been restricted to m_o . By the input constraint on `mux`, we know that the original source of these must have been some state-gathering operation with domain $m' \supseteq m_o$. No input field, therefore, can contain a point in its neighborhood that is not in all of the others.

Finally, if o is a summary operation, it operates similarly, except that we also need to show that the field value for every point p in the input domain m_{i_0} will always contain at least one point in its neighborhood. Ultimately, the computation leading to this input field must have originated with state-gathering operations, which means that the originating fields contained at least p itself. Any well-defined pointwise field operation will not have changed the domain; only a restriction can have done that. By the input constraint on `mux`, we know that the final neighborhood domain must be the result of a sequence of zero or more selections on the original domain, and by the extended definition of `restrict`, we know that the neighborhood must still contain p , or else it would not be in m_{i_0} . Thus summary operations are guaranteed to be well-defined as well.

Thus, every operator instance is well-defined and therefore the program as a whole is well-defined. \square

Appendix A.2. Function Evaluation

THEOREM 4. Let d be the definition for a Proto function, and o and o' be operator instances. If o is used by d then o and $\mathcal{E}_{d,o'}(o)$ are well-defined.

Proof. We begin by showing that o is well-defined. The process for constructing d uses the same evaluation methods as the process for constructing any other Proto expression, so by the same logic Theorem 3, we know that every operator o in d is well-defined except for **restrict** operations whose input domain is not contained within the root manifold m_d . These, however, are external references whose well-definedness depends on evaluation, so we may defer dealing with them until later in the proof.

We now need to show that well-definedness of o is preserved under evaluation. We split the remainder of this proof into two cases: either o is a **restrict**, a **mux**, or any other operator instance.

For any other operator instance, the fact that o is well-defined guarantees that its input and output fields have the same domain m_o . By our syntactic assumptions, we know that m_o is either equal to the root manifold m_d of d , or to some sub-manifold $m' \subseteq m_d$. If the domain is m_d , then the domain of $\mathcal{E}_{d,o'}(f_o)$ is defined to be $m_{o'}$. Because o is well-defined, every input field i_j of o also has domain m_d , and therefore $\mathcal{E}_{d,o'}(i_j)$ is also $m_{o'}$. On the other hand, if the domain is some other m' , then the domain $\mathcal{E}_{d,o'}(f_o)$ is still m' , and by the well-definedness of o , this is the case for every input field i_j as well. Thus, $\mathcal{E}_{d,o'}(o)$ is well-defined.

Similar logic applies if o is a **mux** or **restrict** operator instance generated by the **if** construct. In either case, defining m_d to be $m_{o'}$ does not change the relations between fields, so the same logic as in Lemma 1 still applies. Thus, $\mathcal{E}_{d,o'}(o)$ is well-defined.

This leaves only the case of a **restrict** whose input field has a domain $m_{i_0} \not\subseteq m_d$. The question, then, is whether the domain m_{i_0} of the input contains the domain $\mathcal{E}_{d,o'}(m_{f_o})$ of the output f_o when it is evaluated. We show this by relating both to $m_{o'}$, the domain of the function call. First, f_o either has a domain equal to the root manifold m_d of d , or to some sub-manifold $m' \subseteq m_d$. When evaluation sets $m_d = m_{o'}$, we have $\mathcal{E}_{d,o'}(m_{f_o}) \subseteq m_{o'}$. Since Proto is lexically scoped, the construct defining f_{i_0} must also contain expressions for both function definition d and function call o' . Since the **if** construct changes domain only on sub-expressions, this means that $m_{o'} \subseteq m_{i_0}$. Thus we have $\mathcal{E}_{d,o'}(m_f) \subseteq m_{i_0}$, fulfilling well-definedness for **restrict**. Thus every operator instance is well-defined and remains so under evaluation. \square