On the Evaluation of Space-Time Functions

Jacob Beal
BBN Technologies
Cambridge, MA, USA, 02138
Email: jakebeal@bbn.com

Kyle Usbeck
BBN Technologies
Cambridge, MA, USA, 02138
Email: kusbeck@bbn.com

***Abstract*—The Proto spatial programming language abstracts the distributed execution of programs as evaluation of space-time functions over dynamically defined subspaces on a manifold. Previously, however, function evaluation has always been defined in terms of a complete inlining of expressions during compilation. This simplified the definition of programs, at the cost of limiting expressiveness and duplicating code in compiled binaries. In this paper, we address these shortcomings, producing a model of in-place function evaluation and analysis of its implications for Proto. We have extended the MIT Proto compiler and ProtoKernel virtual machine to implement this model, and empirically verified the reduction of compiled binary size.**

I. Introduction

Every distributed programming framework must address the question of how to control where (and when) a process executes. One way of controlling process execution is to view the distributed system as a spatial computer—a collection of inter-connected devices where the difficulty in communicating between devices is strongly-dependent on the distance between them. Proto [1] uses a continuous space abstraction to view the network of devices as a discrete approximation of continuous space. Paired with a dataflow model of computation, this allows Proto to describe distributed algorithms in terms of operators evaluated over continuous regions of space-time.

Previously, Proto implemented calls to user-defined functions with syntactic inlining during compilation, in order to simplify the global-to-local compilation process. Syntactic inlining caused duplicate code in compiled binaries and prevented desirable language features such as function binding and recursive function calls. This paper addresses these shortcomings, resulting in a model of in-place function evaluation and a reference implementation. The key contributions are:

- Formal substitution and in-place models for evaluating functions over space-time manifolds (Section III).
- Criteria for well-defined evaluation of space-time operators, with analysis of implications for Proto, including for function evaluation (Section IV and V).
- Description of a reference implementation for space-time function calls, along with empirical validation of decreased binary size (Section VI).

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.

II. Background: Proto

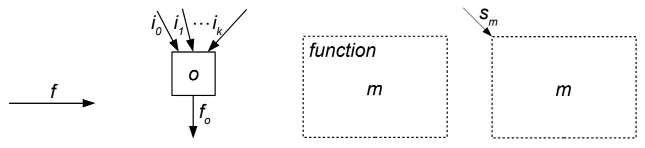
Proto [1], [2] is one of a number of programming models that have recently been developed for spatial computers. In Proto, programs are described in terms of dataflow field operators and information flow over regions of continuous space-time. Closely related to Proto are MGS [3], which performs computation and topological surgery on the cells of a k-dimensional CW-complex, and Regiment [4], which operates on data streams collected from space-time regions. A number of "pattern languages", such as Growing Point Language [5] and Origami Shape Language [6], also use continuous-space abstractions, but have limited expressiveness. There are also a number of discrete-model languages, such as TOTA [7], which uses a viral tuple-passing model, or LDP [8] and MELD [9], which implement a distributed logic programming model. These discrete languages are typically more tightly tied to particular assumptions about scale and communication than the 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 [1] 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 (functions assigning a value to each point in space). Careful selection of operators allows these programs to be automatically transformed for discrete approximation on a network of communicating devices—typically as a ProtoKernel virtual machine [10] binary.

Proto uses four families of 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 neighbor state a space-time measures. For a full explanation see [1] or the MIT Proto documentation and tutorial [2].

A. 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.



(a) Field (b) Operator Instance (c) Manifold (d) Sub-manifold

Fig. 1. Proto programs contain manifolds (spaces), fields that assigning values to every points in a manifold and operator instances that compute fields

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 are the space-time region and sub-regions[1] over which the program executes. 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 in zero or more input fields and producing precisely one output field ($o : i_0 \times i_1 \times ... \times i_k \rightarrow f_o$, where $i_*, f_o \in F$). 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 $\{p \in m' | s(p) = true\}$, the part of m' where s_m is true. 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.

Figure 1 shows diagram symbols for manifolds, fields, and operator instances. A field is an arrow going from the operator instance that produces it to the instances that use it as input. Operator instances are boxes, with inputs entering in order, left to right, across the top edge, and output exiting the bottom. Manifolds are large dashed-line boxes, indicating the domain of all fields produced within them. Root manifolds are labelled 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. Figure 2(a) shows an example diagram of an increment function:

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

evaluated in the expression:

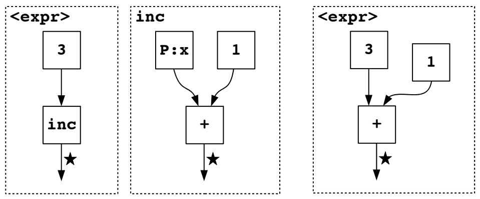
```
(inc 3) ;; returns scalar value 4
```

For a Proto program to be well-defined, we require that every field be guaranteed to have a well-defined value for every point in its domain. Ultimately, this boils down to ensuring an appropriate match on the domains of input and output fields to each operator instance. For each of Proto's four sets of space-time operators, well-definedness has a different criteria. The rest of the paper will examine these well-definedness

[1]Technically, the sub-regions may be CW-complexes, since they may include portions of their boundary, whereas a manifold does not contain its boundary and a manifold with boundary contains its entire boundary. In our analysis, we will maintain this generality, but for clarity of presentation we will refer to these regions as manifolds.

M	All manifolds in a Proto program
m_x	The manifold associated with element x
p	Point in a manifold
(x, t)	Space (x) and time (t) coordinates of a point
O	All operator instances in a Proto program
o	Some particular operator instance
F	All fields in a Proto program
f_x	The field associated with element x
i_j	The jth input field to an operator instance
s_m	Selector field for a sub-manifold m
r_d	Return value for a root manifold
d	Definition of a function
V	Any Proto data value
$\mathcal{E}_{d,o}(x)$	Value of x in function d evaluated for call o

TABLE I
REFERENCE TABLE FOR SYMBOLS USED IN THIS PAPER.



(a) Function Call (b) After Substitution

Fig. 2. The substitution model of space-time function evaluation copies the contents of a function definition in place of the function call operator instance.

criteria in detail, along with their relationship to space-time function evaluation, in order to enable a more powerful and less conservative model of function evaluation in Proto.

III. Models of Space-Time Function Evaluation

Previously in Proto, there has been no explicit model of space-time function evaluation. Instead, all function evaluation has been defined at the syntactic level, and resolved by complete inlining at compile time because it simplified the compiler's global-to-local transformation. Let us 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.

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. The function's definition d is a set of three things: a root manifold m_d that contains the sub-program for the function, a set of pairs (j, f_j), mapping the jth input to a "parameter" operator instance f_j, and a return value r_d. We will say that an operator instance o' is used by d if $m_{o'} \subseteq m_d$.

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 jth parameter, f_j, with the jth input, 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 2(b) shows an example of substitution for the example call of `(inc 3)`.

The converse model, of a function call in place, follows from the substitution model in a straightforward manner. We begin by setting m_d to be equal to m_o, then copy the values of the inputs fields 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.

We wish to use 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 $\mathcal{E}_{d,o}(*)$ is the value of any element $*$ under evaluation. Our task is to ensure that evaluation gives a well-defined value for every point in every field.

IV. Restriction and External References

We begin with the simplest case: pointwise operators. When a function references external variables, however, the domain may not match. We solve this using restriction operators.

A. 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 i_j and the output field f_o, must all have the same manifold m as their domain (excepting the `mux` operator, discussed below).

We can prove that well-defined pointwise operators in functions will remain well-defined when the function is evaluated:

Theorem 1. *If o is a well-defined pointwise operator instance used by function definition d, then $\mathcal{E}_{d,o'}(o)$ is also well-defined for any evaluation of d in the context of operator instance o'.*

Proof: The output field f_o of operator instance o has a domain 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, in all cases $\mathcal{E}_{d,o'}(o)$ is well-defined. ■

This 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) (+ y x))     ;; function: increments a scalar by x
  (inc-x (inc-x 4)))          ;; returns (4 + x) + x
```

As shown in Figure 3(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 failing to notice and handle it caused 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.

B. 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 not in the same sub-manifold, as the well-definedness condition would be violated (Figure 3(c)).

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

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

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

We disallow the programmer from using restriction directly, instead using it in an `if` syntactic construct of form:

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

This form creates two precisely complementary sub-manifolds, then combines the results of their expressions with a `mux` using the same selector field, following the template in Figure 3(d).

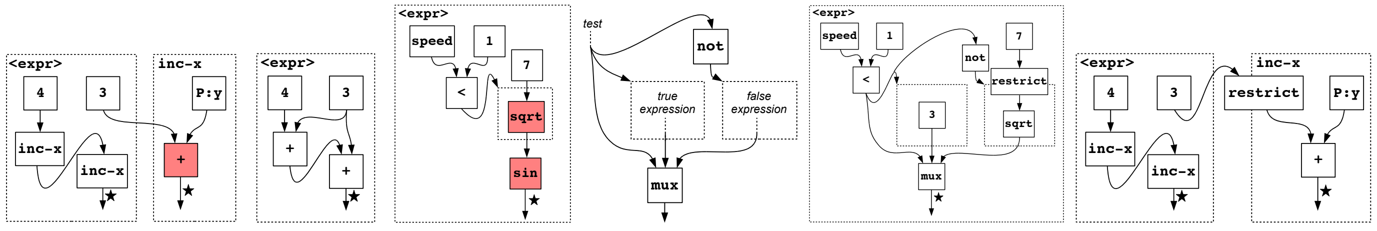
When an `if` expression refers to an external variable, as in:

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

then a `restrict` operator instance is inserted into the reference, limiting its domain (Figure 3(e):). This operator takes one input i_0, and produces an output f_o whose value is equal to 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.

C. References to External Variables

References in functions can be handled similarly, by inserting `restrict` operators wherever a function definition references an external variable (e.g., Figure 3(f)). We also generalize well-definedness for `restrict` for functions: $\mathcal{E}_{d,o'}(m_o) \subseteq m_{i_0}$ 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:



(a) Bad Reference (b) Substitution (c) Bad Restriction (d) `if` template (e) Safe Restriction Example (f) Safe Reference

Fig. 3. Direct references from a function to external fields mismatch domains (a, mismatch shown as red), even though substitution evaluation will sometimes be correct (b). Restrictions face a similar problem (c), which is addressed by a syntactic form (d) that ensures complementary sub-manifolds and inserts a domain-changing `restrict` operator instance on external references (e). Similar insertion for function references (f) is valid under lexically scoping.

Theorem 2. *Let d be the definition for a function using only pointwise and restriction expressions, and o and o' be operator instances. If o is used by d then o and $\mathcal{E}_{d,o'}(o)$ are well-defined.*

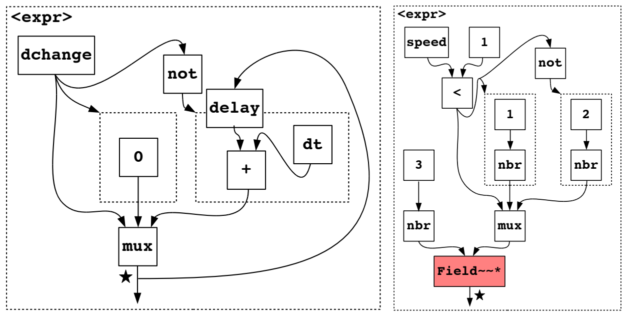
Proof: If o is used by d, then by assumption either o is an instance of `restrict`, `mux`, or another pointwise operator. If o is a pointwise operator other than `mux`, then by our syntactic assumptions, all of its inputs and outputs must have the same domain. Thus, o is well-defined and, by Theorem 1, its evaluation must be as well. A `mux` is the same, unless it was generated by an `if`, in which case the synactic construct still guarantees that it is well-defined and, by the same logic as Theorem 1, will remain so when evaluated. Likewise, the `if` construct guarantees that a `restrict` is well-defined if its input i_0 comes from another operator instance used by d.

This leaves only the case 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 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`, and thus every operator instance is well-defined and remains so under evaluation. ■

Note that the use of lexical scoping in the proof implies that space-time functions cannot in general be first-class objects—at least not in the sense of being values of a field. 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:

```
(let ((test (< (speed) 1))) ;; If a device is moving slowly
  (let ((fun-diff ;; fun-diff names a 'ceiling' function. Otherwise,
    (if test ceil sqrt))) ;; fun-diff names the 'sqrt' function.
    (let ((fun-all      ;; fun-all names the nearest slow-moving
      (broadcast test fun-diff)))        ;; device's 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.



(a) Feedback (b) Neighborhood

Fig. 4. Feedback syntactic forms ensure that `delay` operators are well defined (a). Operators over neighborhood values may not be well defined if restricted domains are combined (b).

Some of the desirable properties of first-class functions can be obtained through dynamic allocation of processes (see [11]). 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.

V. Feedback and Neighborhood Operators

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.

A. 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. For example, Figure 4(a) shows a simple timer:

```
(rep v 0         ;; initialize v to 0
   (+ v (dt)))  ;; update v by adding the time delta between steps
```

where `v` is the state variable, `0` is its initial value, and the update increases its value by time elapsed (measured by the `dt` operator). This allows feedback loops to specify general continuous-time state evolution functions, including for discrete valued functions where there is not always a derivative (see [12]).

Because manifold 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 i_0. Since the value of Δ_t is not resolved at compile-time, to be well-defined, it must be provable that some Δ_t can exist:

$$\exists \Delta_t > 0 \text{ s.t. } \forall (x,t) \in m_o, (x, t-\Delta_t) \in m_{i_0}$$

where m_o and m_{i_0} are the domains of f_o and i_0 and where (x,t) is a point decomposed into its space coordinates x and time coordinate t.

The intuition of the implications of this definition are simple: the input domain for a delay must include the initial values of the state, but the output domain must not. 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.

Proof that this ensures `delay` operator instances are always well-defined, including under function evaluation, can be derived similarly to Theorem 2, above, but is notationally much more complicated, so we do not present it here.

B. 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 like ordinary pointwise operators, except they operate over the values in the field that is the value of each point in the domain. For example, `Field~~*` computes products of neighborhood values.[2]
- Summary operators transform fields of neighborhood values back into ordinary data-valued fields, by applying summary operators like integral or minimum.

The well-definedness criteria for state-gathering and summary operators is just like that for pointwise operators. The pointwise field operators, on the other hand, have a stronger criteria. In addition to the normal pointwise criteria (the input fields i_j and the output field f_o have the same domain m), an operator instance o must satisfy the pointwise well-definedness criteria for each set of neighborhood values. In other words, for every point $p \in m$, the domain of $f_o(p)$ must be equal to the domain of $i_j(p)$ for all inputs.

[2]The "`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.

This means that fields gathered in different sub-manifolds cannot be safely combined. Consider, for example, an innocuous-looking statement like the following (Figure 4(b)):

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

The solution is simple: restrict the input types of `mux` to non-neighborhood values, preventing neighborhood-valued fields from exiting an `if` construct. Branching over neighborhood-valued fields is thus only handled by `Field~~mux`, 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 extended to restrict the domains of the neighborhoods as well.

By making these changes, have we lost any expressiveness in neighborhood computations? The differences between `if`-based computations and `mux`-based computations are only 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 has unpredictable effects. We have now added type-checking that eliminates this bug in the MIT Proto implementation.

As with feedback operators, we do not present the proof of well-definedness, as it is lengthy but similar to Theorem 2.

VI. 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 required to implement function evaluation in Proto and its effect on the size of the executable binary.

A. Implementation in MIT Proto

The ProtoKernel virtual machine [10] 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. An implementation of ProtoKernel is included in MIT Proto, along with a ProtoKernel code emitter, which linearizes Proto programs into executable ProtoKernel binaries.

In order to implement function evaluation for MIT Proto, we made two key additions to ProtoKernel and the code emitter: a new `FUNCALL_OP` instruction, and a preprocessor that "decurrys" external references into extra function parameters.

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

Function calls are implemented by a new `FUNCALL_OP` instruction, parameterized 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.

2) "De-currying" 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 a "de-currying" 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.

B. Effect of Function-Calls on Binary Size

Function calls should significantly reduce the size of ProtoKernel 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 5 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.

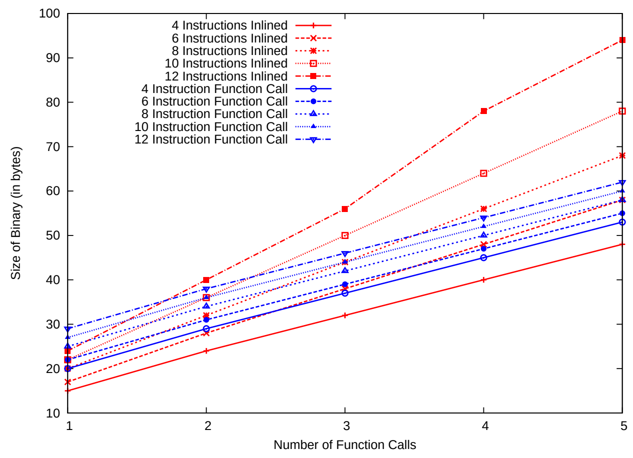


Fig. 5. Function calls (blue) add overhead, but scale better than inlined (red).

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.

VII. 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 also turned up 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.

Our future plans capitalize on these improvements to support general recursion and first-class space-time processes. Currently, ProtoKernel uses a static memory allocation policy where global, environment, and stack sizes are computed at compile time. This restriction prevents recursion from working when any neighborhood or feedback operators are involved. Future work will enable ProtoKernel to allocate memory at runtime and therefore support extra language features. Similarly, although we have shown that first-class functions cannot in general be data values of a field, we believe that space-time processes [11] may be able to provide the same capabilities.

References

[1] J. Beal and J. Bachrach, "Infrastructure for engineered emergence in sensor/actuator networks," *IEEE Intelligent Systems*, vol. 21, pp. 10–19, March/April 2006.

[2] "MIT Proto," software available at `http://proto.bbn.com/`, Retrieved November 22, 2010.

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

[4] R. Newton and M. Welsh, "Region streams: Functional macroprogramming for sensor networks," in *First International Workshop on Data Management for Sensor Networks (DMSN)*, Aug. 2004.

[5] D. Coore, "Botanical computing: A developmental approach to generating inter connect topologies on an amorphous computer," Ph.D. dissertation, MIT, Cambridge, MA, USA, 1999.

[6] R. Nagpal, "Programmable self-assembly: Constructing global shape using biologically-inspired local interactions and origami mathematics," Ph.D. dissertation, MIT, Cambridge, MA, USA, 2001.

[7] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: the TOTA approach," *ACM Transactions on Software Engineering and Methodology*, 2008.

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

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

[10] J. Bachrach and J. Beal, "Building spatial computers," MIT, Tech. Rep. MIT-CSAIL-TR-2007-017, March 2007.

[11] J. Beal, "Dynamically defined processes for spatial computers," in *Spatial Computing Workshop*, 2009.

[12] J. Bachrach, J. Beal, and T. Fujiwara, "Continuous space-time semantics allow adaptive program execution," in *IEEE SASO 2007*, July 2007.