A Syntactic Theory of Transparent Parameterization

Stanley Jefferson [*]
Shinn-Der Lee [†]
Daniel P. Friedman [‡]

Computer Science Department
Indiana University
Bloomington, Indiana 47405

Abstract
We present a calculus for Lamping's programming system of transparent and orthogonal parameterization. The calculus is shown to be consistent, to have a standardization procedure, and to correspond with an operational semantics obtained from the denotational semantics by viewing the semantic equations as state transition rules. Lamping's system is remarkable because it is small, having only four constructions, yet it can easily express a wide variety of parameterization mechanisms including lexical variables, dynamic variables, procedure calls, first-class environments, modules, and method lookup and inheritance mechanisms of object-oriented systems. Due to its orthogonal and transparent parameterization mechanisms, every object, including data and code, in Lamping's programming system can be parameterized, and a parameterized object can be manipulated as if it were a ground object. This blurs the distinction between data and code, allowing one to think of data as code and vice versa.

1 Introduction

Parameterization mechanisms in programming languages provide for the expression of potential dependencies of a program unit on the values of some parameters. Most languages incorporate different parameterization mechanisms for different circumstances. For instance, in a language such as Modula-2, the result of an expression depends on the values of its free variables, the result of a function application depends on the values of the arguments, and the behavior of a module depends on the behaviors of the modules it imports. Lamping [4,5] introduces a programming system, having only four

[*]Supported by the National Science Foundation under grant CCR 89-01919.
[†]Supported by the National Science Foundation under grant CCR 87-02117.
[‡]Supported by the National Science Foundation under grants CCR 87-02117 and CCR 89-01919, and by the Air Force Office of Scientific Research under grant AFOSR 89-0186.

constructions, that can easily express a wide variety of parameterization mechanisms including lexical variables, dynamic variables, procedure calls, first-class environments, modules, and objects, classes, method lookup and inheritance in object-oriented systems. Moreover, Lamping's programming system is capable of expressing recursion and parallel let. It is small and simple, having only one more construction than that of the λ-calculus, and it is computationally equivalent to the λ-calculus. The versatility of his system is a consequence of its ability to express sets of bindings and to parameterize over them, which in turn is a consequence of the transparency and orthogonality of its parameterization mechanisms. Orthogonality allows any component of a construction to be replaced by a parameter. Transparency allows the value of a parameter to pass through a construction, making it accessible to the components of the construction. Due to its orthogonal and transparent parameterization mechanisms, every object, including data and code, in Lamping's programming system can be parameterized, and a parameterized object can be manipulated as if it were a ground object. This blurs the distinction between data and code, allowing one to think of data as code and vice versa.

This paper presents a calculus for Lamping's system. The calculus is shown to be consistent, to have a standardization procedure, and to correspond with an operational semantics obtained from the denotational semantics by viewing the semantic equations as state transition rules. Consistency and correspondence imply that the calculus and the operational semantics produce the same unique result for every program that terminates with a ground constant; standardization provides a particular order of reduction for determining that result. With Lamping's denotational definition, reasoning occurs on the metalevel of domains and continuous functions. With our calculus, reasoning is performed algebraically on a syntactic level. The calculus, thus, can serve as a symbolic reasoning system.

The remainder of this paper is organized as follows. Section 2 is an informal overview of Lamping's programming system, culminating with an object-oriented programming example. Section 3 develops a calculus for the system. Section 4 gives an example of reasoning with the calculus. Section 5 proves that the calculus is consistent and that it has a standardization procedure. Section 6 reproduces Lamping's denotational semantics and gives the correspondence between the calculus and an operational semantics obtained from the denotational semantics. Section 7 is the conclusion.

2 Overview

We now present the syntax of Lamping's system, followed by an informal description of the semantics and several examples illustrating the capabilities of the system. The reader familiar with denotational semantics may also want to refer to Lamping's denotational definition, reproduced in Figure 6.

The pure system has a core syntax consisting of four categories of expressions: variables, let expressions, supply expressions, and data expressions. Ground constants (*e.g.*, $\mathsf{true}, \mathsf{false}, 0, 1, \ldots$) and primitive operators (*e.g.*, $\wedge, \vee, \neg, +, -, *, /, \ldots$, $\mathsf{if\ then\ else}, \ldots$) may be added to the pure system. We consider a language Π whose syntax is displayed in Figure 1. It is essentially the same as that given by Lamping except that we have added a special constant ϵ which denotes an error value. The results of this paper could

Syntactic categories :

$$\begin{array}{rcll} c & \in & \mathcal{C} = \{\epsilon, 0, 1, 2, \ldots\} & \text{(constants)} \\ x & \in & \mathcal{V} & \text{(variables)} \\ M, N & \in & \Pi & \text{(expressions)} \end{array}$$

Abstract syntax :

$$\begin{array}{rl} M ::= & x \\ & \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \\ & \mid \mathsf{supply}\ x = M\ \mathsf{to}\ N \\ & \mid \mathsf{data}\ x : M \\ & \mid c \\ & \mid M + N \end{array}$$

Figure 1: **Syntax of the language Π**

be extended to a language with additional ground constants and primitive operators, but doing so would increase the length of the proofs and obscure the main ideas.

We adopt the following conventions: a, b, c are metavariables for $\mathcal{C}$; w, x, y are metavariables for $\mathcal{V}$; uppercase $A, \ldots, Z$ denote Π-expressions; $\equiv$ denotes syntactic equivalence; and parentheses are used to resolve ambiguity.

From here on, we shall distinguish between the use of the terms *variable* and *parameter*: *variable* refers to syntax, while *parameter* refers to semantics. The system has two classes of parameters: lexically scoped *lexical parameters* and dynamically scoped *data parameters*. Lexical parameters are the same as those found in conventional lexically scoped languages. Data parameters are similar to conventional function parameters in that they provide an abstraction mechanism for expressions. Unlike function parameterization, data parameterization is a transparent abstraction mechanism. Depending on the context, an occurrence of a variable can denote either a lexical parameter or a data parameter.

The expression let $x = M$ in N specifies value M for lexical parameter x in expression N. The following expression, which evaluates (x + y) at x = 3 and y = 2, yielding 5, demonstrates lexical parameterization:

```
let x = 1 in
  let y = 2 * x in
    let x = 3 in
      (x + y)
```

Data parameterization is notated by data $x : M$. It indicates a potential dependency of the expression M on the parameter x; any free occurrence of x in M is a place holder for a value to be supplied. Moreover, it declares x to be a data parameter. For example, data x : (2 * x) is an expression which when supplied a value n for the data parameter x results in the value $2n$. The notation for supplying value M for data parameter x to expression N is supply $x = M$ to N. A supply expression is the only way to associate a value with a data parameter. In the next example, the expression on the left associates the data parameter x with 3 and evaluates to 6. It is equivalent to the expression on the right.

```
supply x = 3 to              let f = data x : (2 * x) in
  data x : (2 * x)             supply x = 3 to f
```

The following two expressions show the different handling of overriding by the two mechanisms:

```
let x = 1 in                 supply x = 1 to
  let y = 2 * x in             let y = data x : (2 * x) in
    let x = 3 in                 supply x = 3 to
      y                            y
```

Although the third let overrides the first one in the expression on the left, the lexical parameter x in expression 2 * x is bound to 1. Therefore, the result is 2. The expression on the right has a result of 6, because the second supply overrides the first one and the value supplied for data parameter x is determined at the time when expression data x : (2 * x) is used.

The expression data x : (2 * x) in the previous example evaluates to a parameterized object. Transparency necessitates the explicit naming of data parameters when a parameterized object is "called."

Transparency has some interesting properties. First, data parameters can be supplied in any order. The following two expressions are equivalent:

```
supply x = 3 to                           supply y = 4 to
  supply y = 4 to                           supply x = 3 to
    data y : data x : (3 * x + 4 * y)         data y : data x : (3 * x + 4 * y)
```

Second, one supply matches multiple data:

```
let f = data x : (2 * x) in
  let g = data x : (x * x) in
    supply x = 3 to
      (f + g)
```

In this example, (f + g) is equivalent to (data x : (2 * x)) + (data x : (x * x)). The supply supplies 3 for the x's in (f + g). Consequently, the result is 15.

Since parameterization is transparent to all constructions, the supply distributes over the primitive operator + in the above example. That is, the following two expressions are equivalent:

```
supply x = 3 to                           (supply x = 3 to data x : (2 * x))
  ((data x : (2 * x)) + (data x : (x * x)))   + (supply x = 3 to data x : (x * x))
```

It is, therefore, possible to write (f + g). In the λ-calculus, (f + g) would have to be expressed as $(\lambda x.(f\ x)+(g\ x))$, since parameterization is not transparent to the addition operator +.

$$\text{data } x_1, \ldots, x_n : M \equiv \text{data } x_1 : \ldots : \text{data } x_n : M$$

$$\text{supply } x_1 = M_1, \ldots, x_n = M_n \text{ to } N$$
$$\equiv \text{supply } x_1 = M_1 \text{ to} \ldots \text{supply } x_n = M_n \text{ to } N$$

Figure 2: **Multiple data and supply**

Consider the expression

```
B ≡ data body :
      supply x = 3 to
        supply y = 4 to
          supply dist = $\sqrt{(\text{data x : x})^2 + (\text{data y : y})^2}$ to
            supply closer = (data dist : dist) <
                              (data point : supply body = (data dist : dist) to point) to
              body
```

which supplies values for the data parameters x, y, dist, and closer to the value denoted by body. This is essentially a parameterized set of bindings which can be made to affect the value supplied for body. For instance,

$$\text{supply body} = \text{data dist : dist to } B$$

retrieves the value $\sqrt{(\text{data x : x})^2 + (\text{data y : y})^2}$ bound to dist in the set of bindings B. To completely evaluate $\sqrt{(\text{data x : x})^2 + (\text{data y : y})^2}$, the values bound to x and y are retrieved from the set of bindings B. The result is, therefore, $\sqrt{3^2 + 4^2} = 5$.

We define some syntactic extensions, given by Lamping, for programming with sets of bindings. Figure 2 defines two syntactic extensions for versions of multiple data and supply. Figure 3 defines syntactic extensions for manipulating sets of bindings. The transmit operation makes the set of bindings M affect the expression N by supplying N as a value for the data parameter body to M. The $\circ$ operation combines two sets of bindings M and N by making M affect the result of making N affect body. The key properties are

$$\text{transmit } \{x_1 = M_1, \ldots, x_n = M_n\} \text{ to } N$$
$$= \text{supply } x_1 = M_1, \ldots, x_n = M_n \text{ to } N$$

and

$$\{x_1 = M_1, \ldots, x_n = M_n\} \circ \{y_1 = N_1, \ldots, y_m = N_m\}$$
$$= \{x_1 = M_1, \ldots, x_n = M_n, y_1 = N_1, \ldots, y_m = N_m\}$$

provided that the data parameter body is reserved only for the definitions of sets of bindings.

To illustrate the use of the definitions, we rewrite the expression B from above:

```
B ≡ { x = 3,
      y = 4,
      dist = $\sqrt{(\text{data x : x})^2 + (\text{data y : y})^2}$,
      closer = (data dist : dist) < (data point : transmit point to (data dist : dist)) }
```

$\{x_1 = M_1, \ldots, x_n = M_n\}$
$\quad \equiv$ data body : supply $x_1 = M_1, \ldots, x_n = M_n$ to body

transmit M to $N \equiv$ supply body $=$ N to M

$M \circ N \equiv$ data body : transmit M to transmit N to body

Figure 3: **Binding definitions**

class$(x_1, \ldots, x_n)$ is $M \equiv$ data $x_1, \ldots, x_n$: M

subclass$(x_1, \ldots, x_n)$ of M is $N \equiv M \circ$ class$(x_1, \ldots, x_n)$ is N

with $x_1 = M_1, \ldots, x_n = M_n$ instantiate N
$\quad \equiv$ supply $x_1 = M_1, \ldots x_n, = M_n$ to N

$M.x \equiv$ transmit M to data x : x when $M \not\equiv$ self

self.$x \equiv$ data x : x

Figure 4: **Object-oriented programming definitions**

The set of bindings B can be considered as an instance of a cartesian point class: x and y are coordinates of the point, dist is the distance of the point from the origin, and closer compares the distances of this point and another given point point.

In the remainder of this section we give an example illustrating object-oriented programming in Π. Figure 4 gives definitions that will be used in the object-oriented programming example. An instance is represented as a set of bindings. A class is a parameterized set of bindings. A subclass of a class is formed by combining the set of new bindings representing the subclass with the set of bindings representing the class. Supplying values for the parameters of a class yields an instance of that class. The expression $M.x$ denotes the value bound to x in instance M, and corresponds to sending the message x to M. In an instance, the keyword self refers to the instance itself.

Using the definitions of Figure 4, a class cpclass of cartesian points can be defined as:

```
cpclass ≡ class(a, b) is
                { x = a,
                  y = b,
                  dist = $\sqrt{(\text{self.x})^2 + (\text{self.y})^2}$,
                  closer = self.dist < (data point : point.dist) }
```

An instance, cp, of a cartesian point at coordinates (3, 4) can be defined by

$$\mathsf{cp} \equiv \mathsf{with\ a\ =\ 3,\ b\ =\ 4\ instantiate\ cpclass}$$

Consider a subclass mpclass of cpclass that inherits all bindings of cpclass except that the distance is redefined as the Manhattan distance from the origin:

$$\mathsf{dist} = \mathsf{self.x} + \mathsf{self.y}$$

As a result, the self.dist in the definition of closer should use the newly defined dist. The subclass mpclass can be defined as

$$\mathsf{mpclass} \equiv \mathsf{subclass\ ()\ of\ cpclass\ is\ \{dist} = \mathsf{self.x} + \mathsf{self.y\}}$$

Since the new dist overrides the old one and closer uses the latest binding of dist, we get the desired behavior.

In summary, the expression

```
let cpclass = class(a, b) is
                { x = a,
                  y = b,
                  dist = √((self.x)² + (self.y)²),
                  closer = self.dist < (data point : point.dist) } in
  let mpclass = subclass () of cpclass is {dist = self.x + self.y} in
    let cp = with a = 3, b = 4 instantiate cpclass in
      let mp = with a = 3, b = 4 instantiate mpclass in
        supply point = mp to
          cp.closer
```

has a value of true since the distance of a cartesian point at coordinates (3, 4) is 5 and the distance of a Manhattan point at coordinates (3, 4) is 7. In section 4 we prove, using the calculus defined in section 3, that a Manhattan point at coordinates (3, 4) evaluates to 7 when sent the message dist.

3 A Calculus

In this section we give a calculus for the language Π. First, we define the notions of free lexical variables, free data variables, lexically closed expressions, programs, and substitution.

The notion of the set of free lexical variables of an expression M is an extension of the same notion in the λ-calculus. The set of *free lexical variables* of an expression M, denoted by $FLV(M)$, is defined inductively:

$$\begin{aligned}
FLV(x) &= \{x\}, \\
FLV(\mathsf{let}\ x = P\ \mathsf{in}\ Q) &= (FLV(Q) - \{x\}) \cup FLV(P), \\
FLV(\mathsf{supply}\ x = P\ \mathsf{to}\ Q) &= FLV(P) \cup FLV(Q), \\
FLV(\mathsf{data}\ x : P) &= FLV(P) - \{x\}, \\
FLV(c) &= \emptyset, \\
FLV(P + Q) &= FLV(P) \cup FLV(Q).
\end{aligned}$$

Similarly, the set of *free data variables* of an expression M, denoted by $FDV(M)$, is defined inductively:

$$\begin{aligned} FDV(x) &= \emptyset, \\ FDV(\mathsf{let}\ x = P\ \mathsf{in}\ Q) &= FDV(P) \cup FDV(Q), \\ FDV(\mathsf{supply}\ x = P\ \mathsf{to}\ Q) &= (FDV(P) \cup FDV(Q)) - \{x\}, \\ FDV(\mathsf{data}\ x : P) &= FDV(P) \cup \{x\}, \\ FDV(c) &= \emptyset, \\ FDV(P + Q) &= FDV(P) \cup FDV(Q). \end{aligned}$$

A *program* is an expression that does not have free lexical variables or free data variables. A lexically closed expression is an expression that does not have free lexical variables.

The *substitution* of lexically closed expression M for variable x in expression N, denoted by $N[x := M]$, is defined inductively according to the structure of N:

$$\begin{aligned} x[x := M] &\equiv M, \\ y[x := M] &\equiv y \quad \text{if } x \not\equiv y, \\ (\mathsf{let}\ x = P\ \mathsf{in}\ Q)[x := M] &\equiv \mathsf{let}\ x = (P[x := M])\ \mathsf{in}\ Q, \\ (\mathsf{let}\ y = P\ \mathsf{in}\ Q)[x := M] &\equiv \mathsf{let}\ y = (P[x := M])\ \mathsf{in}\ Q[x := M] \quad \text{if } x \not\equiv y, \\ (\mathsf{supply}\ w = P\ \mathsf{to}\ Q)[x := M] &\equiv \mathsf{supply}\ w = (P[x := M])\ \mathsf{to}\ Q[x := M], \\ (\mathsf{data}\ x : P)[x := M] &\equiv \mathsf{data}\ x : P, \\ (\mathsf{data}\ y : P)[x := M] &\equiv \mathsf{data}\ y : (P[x := M]) \quad \text{if } x \not\equiv y, \\ c[x := M] &\equiv c, \\ (P + Q)[x := M] &\equiv (P[x := M]) + (Q[x := M]). \end{aligned}$$

Note that any free occurrence of the variable x in the body P of the expression $\mathsf{data}\ x : P$ is a free data variable, and not a free lexical variable. Consequently, $(\mathsf{data}\ x : P)[x := M] \equiv (\mathsf{data}\ x : P)$.

The following definition gives the basic notion of reduction for our calculus:

Definition 3.1 (Basic π-Reduction) The *basic π-reduction* is

$$\pi = \pi_1 \cup \pi_2 \cup \pi_3 \cup \pi_4 \cup \pi_5 \cup \pi_6$$

where, with $FLV(M) = \emptyset$, (read $A \rightarrow B$ as a relation between redex A and its contractum B)

$$\begin{aligned} \pi_1 &: \quad \mathsf{let}\ x = M\ \mathsf{in}\ P \rightarrow P[x := M] \\ \pi_2 &: \quad \mathsf{supply}\ x = M\ \mathsf{to}\ \mathsf{data}\ x : P \rightarrow \mathsf{supply}\ x = M\ \mathsf{to}\ P[x := M] \\ \pi_3 &: \quad \mathsf{supply}\ y = M\ \mathsf{to}\ \mathsf{data}\ x : P \rightarrow \mathsf{data}\ x : \mathsf{supply}\ y = M\ \mathsf{to}\ P \quad \text{if } x \not\equiv y \\ \pi_4 &: \quad \mathsf{supply}\ x = M\ \mathsf{to}\ c \rightarrow c \\ \pi_5 &: \quad \mathsf{supply}\ x = M\ \mathsf{to}\ (P + Q) \rightarrow (\mathsf{supply}\ x = M\ \mathsf{to}\ P) + (\mathsf{supply}\ x = M\ \mathsf{to}\ Q) \\ \pi_6 &: \quad a + b \rightarrow c \quad \text{if } [\![a]\!] + [\![b]\!] = [\![c]\!]. \end{aligned}$$

Consider the expression $\mathsf{supply}\ x = M\ \mathsf{to}\ \mathsf{data}\ x : P$ in π_2. The data parameter x in P is supplied the lexically closed expression M. This can be done by substituting free x's in P by M. Since data parameters are dynamically scoped, the data parameter x might be needed in $P[x := M]$, so we must retain the supply in the contractum.

Consider the expression $\mathsf{supply}\ y = M\ \mathsf{to}\ \mathsf{data}\ x : P$ in π_3. The expression $\mathsf{data}\ x : P$ is expecting a value for x. The supply, on the other hand, supplies a value for y. Thus, π_2 is not applicable. But transparent data parameterization requires that expression $\mathsf{supply}\ y = M\ \mathsf{to}\ \mathsf{data}\ x : P$ be equivalent to expression $\mathsf{data}\ x : \mathsf{supply}\ y = M\ \mathsf{to}\ P$. Since M has no free lexical variables, there is no danger of turning any free lexical variable x in M into a free data variable. This basic π-reduction serves the purpose of moving the data outward in order to find a match with the closest supply that supplies a value for the data parameter x.

Reduction π_4 says that a constant does not depend on data parameters. Transparency requires that data parameters distribute over the primitive operator $+$; this is formalized in reduction π_5. The denotation of a constant symbol c is given by $[\![c]\!]$. Reduction π_6 simulates the behavior of the primitive operator $+$ extended to be strict with respect to error.

Based on the basic π-reduction, we define one-step π-reduction, π-reduction, and π-equality in the usual way [1]:

(i) *One-step π-reduction*, denoted by $\rightarrow_\pi$, is the compatible closure of π:

$$M \rightarrow N \Rightarrow M \rightarrow_\pi N,$$
$$M \rightarrow_\pi N \Rightarrow \mathsf{let}\ x = M\ \mathsf{in}\ P \rightarrow_\pi \mathsf{let}\ x = N\ \mathsf{in}\ P,$$
$$P \rightarrow_\pi Q \Rightarrow \mathsf{let}\ x = M\ \mathsf{in}\ P \rightarrow_\pi \mathsf{let}\ x = M\ \mathsf{in}\ Q,$$
$$M \rightarrow_\pi N \Rightarrow \mathsf{supply}\ x = M\ \mathsf{to}\ P \rightarrow_\pi \mathsf{supply}\ x = N\ \mathsf{to}\ P,$$
$$P \rightarrow_\pi Q \Rightarrow \mathsf{supply}\ x = M\ \mathsf{to}\ P \rightarrow_\pi \mathsf{supply}\ x = M\ \mathsf{to}\ Q,$$
$$M \rightarrow_\pi N \Rightarrow \mathsf{data}\ x : M \rightarrow_\pi \mathsf{data}\ x : N,$$
$$M \rightarrow_\pi N \Rightarrow M + P \rightarrow_\pi N + P,$$
$$P \rightarrow_\pi Q \Rightarrow M + P \rightarrow_\pi M + Q.$$

(ii) *π-reduction*, denoted by $\twoheadrightarrow_\pi$, is the reflexive and transitive closure of $\rightarrow_\pi$.

$$M \rightarrow_\pi N \Rightarrow M \twoheadrightarrow_\pi N,$$
$$M \twoheadrightarrow_\pi M,$$
$$M \twoheadrightarrow_\pi N, N \twoheadrightarrow_\pi P \Rightarrow M \twoheadrightarrow_\pi P.$$

(iii) *π-equality* (*π-convertibility*, *π-calculus*), denoted by $=_\pi$, is the least equivalence relation generated by $\twoheadrightarrow_\pi$.

$$M \twoheadrightarrow_\pi N \Rightarrow M =_\pi N,$$
$$M =_\pi M,$$
$$M =_\pi N \Rightarrow N =_\pi M,$$
$$M =_\pi N, N =_\pi P \Rightarrow M =_\pi P.$$

4 Reasoning with the Calculus

Using the calculus, we show that expression mp.dist, where mp is the Manhattan point

```
{ x = 3,
  y = 4,
  dist = self.x + self.y,
  closer = self.dist < (data point : point.dist)},
```

is equivalent to the constant expression 7:

```
mp.dist ≡ transmit mp to (data dist : dist)                              by definition
        ≡ supply body = (data dist : dist) to
            data body :
              supply x = 3, y = 4 to
                supply dist = (data x : x) + (data y : y) to
                  supply closer = (data dist : dist) < (data point : point.dist) to
                    body                                                 by definition
        →π supply body = (data dist : dist) to
              supply x = 3, y = 4 to
                supply dist = (data x : x) + (data y : y) to
                  supply closer = (data dist : dist) < (data point : point.dist) to
                    data dist : dist                                     by π2
        ↠π supply body = (data dist : dist) to
              supply x = 3, y = 4 to
                supply dist = (data x : x) + (data y : y) to
                  supply closer = (data dist : dist) < (data point : point.dist) to
                    (data x : x) + (data y: y)                by π3 and then π2
```

By a series of π_5 reductions, the above expression can be reduced to $M + N$ where subexpression M corresponds to (data x : x) and subexpression N corresponds to (data y : y). Subexpression M can be reduced to the constant expression 3:

```
M ≡ supply body = (data dist : dist) to
      supply x = 3, y = 4 to
        supply dist = (data x : x) + (data y : y) to
          supply closer = (data dist : dist) < (data point : point.dist) to
            (data x : x)
  ↠π supply body = (data dist : dist) to
        supply x = 3, y = 4 to
          supply dist = (data x : x) + (data y : y) to
            supply closer = (data dist : dist) < (data point : point.dist) to
              3                                     by repeated π3 and then π2
```

Then, by a series of five π_4 reductions, the above expression reduces to the constant expression 3.

Similarly, subexpression N reduces to the constant expression 4. Therefore, $M + N$ reduces to 3 + 4, which reduces to 7 by π_6.

$(P1)\ M \underset{1}{\twoheadrightarrow} M$

$(P2)\ M \underset{1}{\twoheadrightarrow} M_1, N \underset{1}{\twoheadrightarrow} N_1, FLV(M) = \emptyset \Rightarrow \mathsf{let}\ x = M\ \mathsf{in}\ N \underset{1}{\twoheadrightarrow} \mathsf{let}\ x = M_1\ \mathsf{in}\ N_1$
$(P3)\ M \underset{1}{\twoheadrightarrow} M_1, N \underset{1}{\twoheadrightarrow} N_1, FLV(M) = \emptyset \Rightarrow \mathsf{supply}\ x = M\ \mathsf{to}\ N \underset{1}{\twoheadrightarrow} \mathsf{supply}\ x = M_1\ \mathsf{to}\ N_1$
$(P4)\ M \underset{1}{\twoheadrightarrow} M_1 \Rightarrow \mathsf{data}\ x : M \underset{1}{\twoheadrightarrow} \mathsf{data}\ x : M_1$
$(P5)\ M \underset{1}{\twoheadrightarrow} M_1, N \underset{1}{\twoheadrightarrow} N_1 \Rightarrow M + N \underset{1}{\twoheadrightarrow} M_1 + N_1$

$(P6)\ M \underset{1}{\twoheadrightarrow} M_1, N \underset{1}{\twoheadrightarrow} N_1, FLV(M) = \emptyset \Rightarrow \mathsf{let}\ x = M\ \mathsf{in}\ N \underset{1}{\twoheadrightarrow} N_1[x := M_1]$
$(P7)\ M \underset{1}{\twoheadrightarrow} M_1, N \underset{1}{\twoheadrightarrow} N_1, FLV(M) = \emptyset$
$\quad \Rightarrow \mathsf{supply}\ x = M\ \mathsf{to}\ (\mathsf{data}\ x : N) \underset{1}{\twoheadrightarrow} \mathsf{supply}\ x = M_1\ \mathsf{to}\ N_1[x := M_1]$
$(P8)\ M \underset{1}{\twoheadrightarrow} M_1, N \underset{1}{\twoheadrightarrow} N_1, FLV(M) = \emptyset, y \not\equiv x$
$\quad \Rightarrow \mathsf{supply}\ y = M\ \mathsf{to}\ (\mathsf{data}\ x : N) \underset{1}{\twoheadrightarrow} \mathsf{data}\ x : (\mathsf{supply}\ y = M_1\ \mathsf{to}\ N_1)$
$(P9)\ N \underset{1}{\twoheadrightarrow} c, FLV(M) = \emptyset \Rightarrow \mathsf{supply}\ x = M\ \mathsf{to}\ N \underset{1}{\twoheadrightarrow} c$
$(P10)\ M \underset{1}{\twoheadrightarrow} M_1, N \underset{1}{\twoheadrightarrow} N_1 + N_2, FLV(M) = \emptyset$
$\quad \Rightarrow \mathsf{supply}\ x = M\ \mathsf{to}\ N \underset{1}{\twoheadrightarrow} (\mathsf{supply}\ x = M_1\ \mathsf{to}\ N_1) + (\mathsf{supply}\ x = M_1\ \mathsf{to}\ N_2)$
$(P11)\ M \underset{1}{\twoheadrightarrow} a, N \underset{1}{\twoheadrightarrow} b, [\![a]\!] + [\![b]\!] = [\![c]\!] \Rightarrow M + N \underset{1}{\twoheadrightarrow} c$

Figure 5: **Parallel reductions**

5 Consistency and Standardization

Given a calculus, two immediate concerns are its consistency and its standardization procedure. Consistency is implied by a Church-Rosser theorem which shows the confluence of two reduction paths that proceed in two different directions from the same expression. It says that if a program has a result then the result is unique no matter what order of reduction we choose. The derivation in the previous section is meaningful only if the calculus is Church-Rosser. Standardization is a particular order of reduction that provides a semi-effective procedure for finding the result of a program. It says that if a program has a result then the result can be obtained by that particular order of reduction.

For the proof of the Church-Rosser theorem, we follow Tait-Martin-Löf's strategy for the corresponding work on the λ-calculus [1]. We define a parallel reduction which is a superset of the one-step π-reduction and a subset of the π-reduction. Thus, its reflexive and transitive closure is the π-reduction. Then, we show that this parallel reduction satisfies the diamond property. From this, we infer that its reflexive and transitive closure also satisfies the diamond property. Since the reflexive and transitive closure of the parallel reduction is the π-reduction, we have the π-reduction satisfying the diamond property also. Hence, the calculus is Church-Rosser.

The inductive definition of the parallel reduction from an expression A to an expression B, denoted by $A \underset{1}{\twoheadrightarrow} B$, is given in Figure 5. Parallel reductions can be classified into three categories. The first category parallel reduces an expression to itself. This is (P1). The second category parallel reduces an expression A to an expression B with B being derived from A by parallel reducing every top-level subexpression of A. Consequently, the top-level structure of B is the same as that of A. There are six classes of expressions in the language, but constants and variables can only reduce to themselves, so we have four classes left. They are dealt with in (P2)–(P5). The third category parallel reduces an expression A to an expression C with C being derived from A by parallel reducing A to an intermediate expression B, as in the second category, then applying a basic π-reduction to B to get C. Since there are six basic π-reductions, this category has six cases: (P6)–(P11).

Theorem 5.1 (Church-Rosser) *π-reduction satisfies the diamond property. That is, if $M \twoheadrightarrow_\pi M_1$ and $M \twoheadrightarrow_\pi M_2$ then there is an M_3 such that $M_1 \twoheadrightarrow_\pi M_3$ and $M_2 \twoheadrightarrow_\pi M_3$.*

Outline of proof : By an induction on the length of proof of $M \underset{1}{\twoheadrightarrow} M_1$, we can show that $\underset{1}{\twoheadrightarrow}$ satisfies the diamond property. From that, by a simple diagram chase, we can show that $\underset{1}{\twoheadrightarrow}^*$ also satisfies the diamond property. Then, $\rightarrow_\pi \subset \underset{1}{\twoheadrightarrow}$ implies $\twoheadrightarrow_\pi = \rightarrow_\pi^* \subset \underset{1}{\twoheadrightarrow}^*$. Similarly, $\underset{1}{\twoheadrightarrow} \subset \twoheadrightarrow_\pi$ implies $\underset{1}{\twoheadrightarrow}^* \subset \twoheadrightarrow_\pi^* = \twoheadrightarrow_\pi$. Therefore, $\underset{1}{\twoheadrightarrow}^* = \twoheadrightarrow_\pi$. Hence $\twoheadrightarrow_\pi$ satisfies the diamond property. □

For the proof of the standardization theorem, we follow the strategies in [2,3,6]. We begin by defining the notion of outermost reduction. Informally, an outermost reduction always reduces an outermost redex.

Definition 5.2 (Evaluation Contexts) (i) An *evaluation context* $C[\,]$ is defined inductively:

- $[\,]$ is an evaluation context,
- if $FLV(M) = \emptyset$ and $C[\,]$ is an evaluation context then supply $x = M$ to $C[\,]$ is an evaluation context,
- if $C[\,]$ is an evaluation context then $C[\,] + M$ and $M + C[\,]$ are evaluation contexts.

(ii) If $C[\,]$ is an evaluation context and M is an expression then $C[M]$ denotes the result of replacing the $[\,]$ of $C[\,]$ by M.

Definition 5.3 (Outermost Reduction) *Outermost reduction*, denoted by $\mapsto_\pi$, is the least relation between Π-expressions such that whenever $M \mapsto_\pi N$, there is an evaluation context $C[\,]$ and a basic π-reduction $P \rightarrow Q$ with $M \equiv C[P]$ and $C[Q] \equiv N$.

Next, we allow a redex other than an outermost one to be reduced. But once a redex within an outermost redex is reduced, we can never go back to reduce the outermost redex. A sequence of expressions derived using this strategy is called a standard reduction sequence. With this, we can show that if $M \twoheadrightarrow_\pi N$ then there is a standard reduction sequence $P_1, \ldots, P_p$ such that $M \equiv P_1$ and $P_p \equiv N$.

Definition 5.4 (Standard Reduction Sequences) *Standard reduction sequences* (srs's) are defined inductively: if $M_1, \ldots, M_m$ and $N_1, \ldots, N_n$ are srs's then
(S1) x is a srs,
(S2) (let $x = M_1$ in N_1), …, (let $x = M_1$ in N_n), …, (let $x = M_m$ in N_n) is a srs,
(S3) (supply $x = M_1$ to N_1), …, (supply $x = M_1$ to N_n), …, (supply $x = M_m$ to N_n) is a srs,
(S4) (data $x : M_1$), …, (data $x : M_m$) is a srs,

(S5) c is a srs,
(S6) $(M_1 + N_1), \ldots, (M_m + N_1), \ldots, (M_m + N_n)$ is a srs,
(S7) if $M_0 \mapsto_\pi M_1$ then $M_0, \ldots, M_m$ is a srs.

Theorem 5.5 (Standardization) *$M \twoheadrightarrow_\pi N$ iff there is a srs $P_1, \ldots, P_p$ with $M \equiv P_1$ and $P_p \equiv N$.*

Outline of proof : ($\Leftarrow$) Clearly if there is a srs $P_1, \ldots, P_p$ such that $M \equiv P_1$ and $P_p \equiv N$ then $M \twoheadrightarrow_\pi N$ is provable in the calculus.

($\Rightarrow$) If $A_0 \underset{1}{\twoheadrightarrow} A_1$ and $A_1, \ldots, A_n$ is a srs then, by a lexicographic induction on n, on the number of one-step π-reduction used in the proof of $A_0 \underset{1}{\twoheadrightarrow} A_1$, and on the structure of A_0, we can show that there is a srs $B_0, \ldots, B_m$ with $A_0 \equiv B_0$ and $B_m \equiv A_n$.

$M \twoheadrightarrow_\pi N$ implies there are expressions $M_1, \ldots, M_m$ such that $M \equiv M_1 \underset{1}{\twoheadrightarrow} \cdots \underset{1}{\twoheadrightarrow} M_m \equiv N$. From $M_{m-1} \underset{1}{\twoheadrightarrow} M_m$ and M_m being a srs (any expression is a srs), we can find a srs $P_1, \ldots, P_p$ such that $M_{m-1} \equiv P_1$ and $P_p \equiv M_m$. From $M_{m-2} \underset{1}{\twoheadrightarrow} P_1$ and $P_1, \ldots, P_p$ being a srs, we can find a srs $Q_1, \ldots, Q_q$ such that $M_{m-2} \equiv Q_1$ and $Q_q \equiv P_p \equiv M_m$. Repeat this procedure from M_m back up to M_1 and we are done. □

If M is a program and $M \twoheadrightarrow_\pi c$ then the srs is a sequence of outermost reductions:

Corollary 5.6 *Let M be a program. Then $M \twoheadrightarrow_\pi c$ iff $M \mapsto_\pi^* c$.*

6 Operational Semantics and Correspondence

In order to relate the π-calculus to Lamping's denotational semantics, we prove that the calculus corresponds to an operational semantics that was informally derived from the denotational semantics. The correspondence states that the calculus and the operational semantics produce the same result for every program that terminates with a ground constant. This notion of correspondence is the same as in [2,3,6].

Lamping's denotational semantics is reproduced in Figure 6. The operational semantics, defined in Figure 7, is a simple state transition system. It is informally derived from the denotational semantics by viewing the equations of the denotational definition as state transition rules. A state is either a basic state or a compound state. A compound state is a pair of states σ_1 and σ_2 written as $(\sigma_1 + \sigma_2)$, and serves as a mechanism for evaluating addition expressions. A concrete environment represents an environment as an association list. The arid concrete environment, represented by $\mathbf{\Omega}$, binds every variable to the concrete parameterized object $<\epsilon, \mathbf{\Omega}>$. A concrete parameterized object corresponds to a lexically closed expression.

Theorem 6.1 (Correspondence) *Let M be a program. Then*

$$\ll M, \mathbf{\Omega}, \mathbf{\Omega} \gg \vdash^* \ll c, \mathbf{\Omega}; \mathbf{\Omega} \gg \quad \textit{iff} \quad M \mapsto_\pi^* c.$$

Outline of proof : ($\Rightarrow$) We show that each transition rule can be simulated by a sequence of standard reductions. With this and the standardization theorem, it is straightforward to show that the operational semantics can be simulated by the calculus.

($\Leftarrow$) First, by induction on the number of transition steps, we can show that for any state σ, if $\sigma \vdash^* \ll c, \mathbf{\Omega}, \mathbf{\Omega} \gg$ and $\sigma \vdash^* \sigma'$, then $\sigma' \vdash^* \ll c, \mathbf{\Omega}, \mathbf{\Omega} \gg$. Next, let $M_0, \ldots, M_m$ be

Semantic domains:

$c \in \mathcal{C}$	(constants)
$x \in \mathcal{V}$	(variables)
$\rho, \delta \in \mathcal{U} = \mathcal{V} \rightarrow \mathcal{O}$	(environments)
$\mathcal{O} = \mathcal{U} \rightarrow \mathcal{C}$	(parameterized objects)

Valuation function: $\mathcal{M} : \Pi \rightarrow \mathcal{U} \rightarrow \mathcal{U} \rightarrow \mathcal{C}$:

$$\begin{aligned}
\mathcal{M}\,[\![x]\!]\,\rho\,\delta &= \rho(x)\ \delta \\
\mathcal{M}\,[\![\mathsf{let}\ x = M\ \mathsf{in}\ N]\!]\,\rho\,\delta &= \mathcal{M}\,[\![N]\!]\,(\rho[(\mathcal{M}\,[\![M]\!]\,\rho)/x])\,\delta \\
\mathcal{M}\,[\![\mathsf{supply}\ x = M\ \mathsf{to}\ N]\!]\,\rho\,\delta &= \mathcal{M}\,[\![N]\!]\,\rho\,(\delta[(\mathcal{M}\,[\![M]\!]\,\rho)/x]) \\
\mathcal{M}\,[\![\mathsf{data}\ x : M]\!]\,\rho\,\delta &= \mathcal{M}\,[\![M]\!]\,(\rho[\delta(x)/x])\,\delta \\
\mathcal{M}\,[\![c]\!]\,\rho\,\delta &= [\![c]\!] \\
\mathcal{M}\,[\![M + N]\!]\,\rho\,\delta &= (\mathcal{M}\,[\![M]\!]\,\rho\,\delta) + (\mathcal{M}\,[\![N]\!]\,\rho\,\delta)
\end{aligned}$$

Figure 6: **Denotational Semantics of Π**

Data structures:

$\omega \in \mathcal{O} = \Pi \times \mathcal{U}$	(concrete parameterized objects)
$\rho, \delta \in \mathcal{U} = \{\mathbf{\Omega}\} + (\mathcal{U} \times \mathcal{V} \times \mathcal{O})$	(concrete environments)
$\beta \in \mathcal{B} = \Pi \times \mathcal{U} \times \mathcal{U}$	(basic states)
$\sigma \in \mathcal{S} = \mathcal{B} + (\mathcal{S} \times \{+\} \times \mathcal{S})$	(states)

Environment extension: $\bullet[\bullet \leftarrow \bullet] : \mathcal{U} \times \mathcal{V} \times \mathcal{O} \rightarrow \mathcal{U}$:

$$\rho[x \leftarrow \omega] = (\rho, x, \omega).$$

Environment lookup: $\mathsf{lookup}(\bullet, \bullet) : \mathcal{U} \times \mathcal{V} \rightarrow \mathcal{O}$:

$$\begin{aligned}
&\mathsf{lookup}(\mathbf{\Omega}, x) = <\epsilon, \mathbf{\Omega}>, \\
&\mathsf{lookup}(\rho[x \leftarrow \omega], x) = \omega, \\
&\mathsf{lookup}(\rho[y \leftarrow \omega], x) = \mathsf{lookup}(\rho, x) \quad \text{if } x \not\equiv y,
\end{aligned}$$

Transition rules: $\bullet \vdash \bullet : \mathcal{S} \rightarrow \mathcal{S}$:

$$\begin{aligned}
\ll x, \rho, \delta \gg &\vdash \ll M, \rho', \delta \gg && \text{if } \mathsf{lookup}(\rho, x) = <M, \rho'> \\
\ll \mathsf{let}\ x = M\ \mathsf{in}\ N, \rho, \delta \gg &\vdash \ll N, \rho[x \leftarrow <M, \rho>], \delta \gg \\
\ll \mathsf{supply}\ x = M\ \mathsf{to}\ N, \rho, \delta \gg &\vdash \ll N, \rho, \delta[x \leftarrow <M, \rho>] \gg \\
\ll \mathsf{data}\ x : M, \rho, \delta \gg &\vdash \ll M, \rho[x \leftarrow \mathsf{lookup}(\delta, x)], \delta \gg \\
\ll c, \rho, \delta \gg &\vdash \ll c, \mathbf{\Omega}, \mathbf{\Omega} \gg && \text{if } \rho \neq \mathbf{\Omega} \quad \text{or} \quad \delta \neq \mathbf{\Omega} \\
\ll M + N, \rho, \delta \gg &\vdash (\ll M, \rho, \delta \gg + \ll N, \rho, \delta \gg) \\
(\ll a, \mathbf{\Omega}, \mathbf{\Omega} \gg + \ll b, \mathbf{\Omega}, \mathbf{\Omega} \gg) &\vdash \ll c, \mathbf{\Omega}, \mathbf{\Omega} \gg && \text{if } [\![a]\!] + [\![b]\!] = [\![c]\!]
\end{aligned}$$

$$\begin{aligned}
\sigma_1 \vdash \sigma_2 &\Rightarrow (\sigma_1 + \sigma) \vdash (\sigma_2 + \sigma) \\
\sigma_1 \vdash \sigma_2 &\Rightarrow (\sigma + \sigma_1) \vdash (\sigma + \sigma_2)
\end{aligned}$$

Convention: $\vdash^*$ denotes the reflexive and transitive closure of binary relation $\vdash$.

Figure 7: **Operational semantics of Π**

a sequence of expressions with $M \equiv M_0 \mapsto_\pi \cdots \mapsto_\pi M_m \equiv c$. For each i, we can show that there is a state σ such that $\ll M_{i-1}, \mathbf{\Omega}, \mathbf{\Omega} \gg \vdash^* \sigma$ and $\ll M_i, \mathbf{\Omega}, \mathbf{\Omega} \gg \vdash^* \sigma$. Now, if $\ll M_i, \mathbf{\Omega}, \mathbf{\Omega} \gg \vdash^* \ll c, \mathbf{\Omega}, \mathbf{\Omega} \gg$, then $\sigma \vdash^* \ll c, \mathbf{\Omega}, \mathbf{\Omega} \gg$ also. Hence, $\ll M_{i-1}, \mathbf{\Omega}, \mathbf{\Omega} \gg \vdash^* \sigma \vdash^* \ll c, \mathbf{\Omega}, \mathbf{\Omega} \gg$. With $\ll M_m, \mathbf{\Omega}, \mathbf{\Omega} \gg \equiv \ll c, \mathbf{\Omega}, \mathbf{\Omega} \gg$ as the basis, by an induction on m back up to 0, we have $\ll M, \mathbf{\Omega}, \mathbf{\Omega} \gg \equiv \ll M_0, \mathbf{\Omega}, \mathbf{\Omega} \gg \vdash^* \ll c, \mathbf{\Omega}, \mathbf{\Omega} \gg$. □

7 Conclusion

Lamping's system of transparent parameterization provides a unified view of parameterization which can easily express a wide range of parameterization mechanisms. We have developed a calculus for this system. The calculus is consistent, has a standardization procedure, and corresponds to an operational semantics directly obtained from the denotational semantics. It provides a simple symbolic reasoning system for the language Π, which can be used to determine the result of a program and to prove the equivalence of programs. The calculus is small, having only three conversion rules for the core syntax. We believe the calculus provides additional insight into Lamping's system of parameterization. In particular, it clarifies the relationship between lexical substitution and data parameterization.

References

[1] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, second edition, 1984.

[2] Matthias Felleisen and Daniel P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989.

[3] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[4] John O. Lamping. A unified system of parameterization for programming languages. In *1988 ACM Conference On Lisp and Functional Programming*, pages 316–326, July 1988.

[5] John O. Lamping. *A Unified System Of Parameterization For Programming Languages.* PhD thesis, Stanford University, April 1988.

[6] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer Science*, 1:125–159, 1975.

[7] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1981.