A Categorical Interpretation of Landin's Correspondence Principle

Anindya Banerjee and David A. Schmidt
Department of Computing and Information Sciences
Kansas State University *
(banerjee, schmidt)@cis.ksu.edu

Abstract

Many programming languages can be studied by desugaring them into an intermediate language, namely, the simply-typed λ- calculus. In this manner Landin and Tennent discovered a "correspondence" between the semantics of definition bindings and parameter bindings such that the semantics of free identifiers becomes independent of their mode of definition.

In this paper we consider programming languages with modules and we desugar modules into records. A categorical model for the simply-typed λ- calculus with records is then freely generated. The record construction becomes a tensor product, the lambda abstraction construction becomes a function space, and if the language satisfies the correspondence principle, then the categorical exponentiation diagram commutes. A converse result is also proved. The framework for defining the model is of interest because it defines a hierarchy of call-by-value λ-calculi, of which call-by-name is the weakest form of call-by-value calculus.

Applications to compiling are given.

1 Introduction

In his seminal paper on the next 700 programming languages [9], Landin suggested that a programming language might satisfy a correspondence in the semantics of its definition and parameter constructions. That is, the semantics of binding a body, `U`, to a name, `i`, as seen in:

```
define i = U in V
```

should be the same as that of binding an actual parameter, `U`, to a formal parameter, `i`, as seen in:

```
define j(i) = V in call j(U)
```

*Manhattan, Kansas 66506, USA. Part of this work was supported by NSF under grant CCR-9102625.

where j is fresh.
Tennent [23] titled this the *correspondence principle* and suggested that it be used as a design guide for programming languages. The primary benefit from the correspondence principle is that a program phrase containing free identifiers can be understood without concern as to whether the identifiers were bound by definitions or parameters. For example, ... `A` ... means the same whether it appears in:

```
define A = 4 in ...A...
```

or in:

```
define G(A) = ...A... in G(4)
```

1.1 Correspondence in higher order, modular languages

The importance of the correspondence principle increases when a programming language is *higher-order*, that is, abstractions can be arguments and results of other abstractions. Consider the following example:

```
function g(a) = (function f(b) = ...a...b... in return f)
```

A call to `g(somevalue)` returns `f` with a binding to `a`. The semantics of `f` is explained as: `define a = somevalue in function f(b) = ...a...b...`. For this explanation to make sense, correspondence *must* hold.

Finally, languages with modules need correspondence to ensure proper behavior: a module, in the sense of Ada and Standard ML, is a set of definitions. Modules can be built hierarchically, one module importing another (*cf.* SML's "functors" [11, 12]):

```
module m = (define i = something)
in  module n(x) = (use x; define j = ...i...)
in  ...use n(m)...
```

or they can be written "flat":

```
module n = (define i = something; define j = ...i...)
in  ...use n...
```

Correspondence ensures that the semantics of a hierarchical module equals the semantics of a flat one with the same set of definitions. For example, if the "`something`" in the above example was a looping expression, and module parameters like `m` were evaluated eagerly but module importations like `use n` were done lazily, then hierarchical module construction would be a futile endeavor.

1.2 This paper

We show that Landin's correspondence principle, as it arises in the above examples, can be formalized in Category Theory: a language's definition construct defines a tensor product construction, its parameter construct defines a function space construction, and correspondence ensures that the exponentiation diagram commutes. That is, the category freely generated by a programming language with correspondence has an associative, commutative tensor product (with a unit) and it has weak categorical exponentiation (the fill-in morphism might not be unique). Hence, it is a symmetric monoidal weakly closed category. The significance is that the tensor product ensures that sets of bindings (*i.e.*, modules) behave like sets, and exponentiation ensures that definition bindings behave like parameter bindings. Both properties are crucial to properly designed, modular, higher-order programming languages. We also show a converse result: if the semantics of a programming language fits a "usual" format, weak exponentiation in the model implies correspondence.

In this way, a fundamental intuitive programming language criterion is characterized as a fundamental categorical one. This fits within Reynolds' program of "Semantics ... [as] applied mathematics; it seeks profound definitions rather than difficult theorems ... the application of such concepts directly reveals regularity in linguistic behaviour and strengthens and objectifies our intuitions of simplicity and uniformity." ([25], page 3).

A programming language with correspondence can use call-by-name or call-by-value binding, so the framework for our proof must accommodate both. For this reason, we prove the result for a hierarchy of call-by-value λ-calculi, of which call-by-name is the weakest call-by-value calculus. In this light, our result can be viewed as a generalization of the cartesian closedness of models for call-by-name, simply-typed λ-calculus [7] to monoidal closedness for call-by-value λ-calculi. Our result also reveals that the reason why categorical exponentiation holds is because the form of binding defined by a product construction corresponds with the form of binding defined by the function space construction.

In the rest of this paper, we define our metalanguage, outline the proof of the correspondence theorem, and state an application to compiling.

2 The Metalanguage

One way to obtain correspondence is to force it upon a language. Landin [8], Reynolds [18, 19], and Tennent [23, 24] observed that correspondence *must* hold if both definition binding and parameter binding are desugared purely into λ-abstractions:

`define i = U in V`	desugars to	$(\lambda\,$`i`$\,.\,$`V`$)\,$`U`
`define j(i) = V in j(U)`	desugars to	`define j =` $\lambda\,$`i.V` `in j(U)` which
		desugars to $(\lambda\,$`i`$\,.\,$`V`$)\,$`U`, when $\lambda\,$`i`$\,.\,$`V` is copied for `j`

Thus, the semantics of λ-abstraction — whether it be call-by-name or call-by-value semantics — defines the semantics of both definition and parameter binding. An example like:

(1)
```
const k = 0, alias x = location1
in procedure p(y:int) = x:= @x+y
   in x:= k; p(k)
```

is desugared into:

$((\lambda$`k`$:int.\lambda$`x`$:intloc.(\lambda$`p`$:int \rightarrow comm.$`x := k; (p k))`$(\lambda$`y`$:int.$`x := @x+y)0)location1`

We work with statically typed languages. For simplicity, we use `alias` definitions here rather than `var` declarations. A `var` declaration is a binding of an identifier to a location with the side effect of allocating the location in storage. One might desugar `var x:intloc in e` by `new`(λ`x` : *intloc*. `e`), where `new` is a storage allocation operator [4, 13, 14, 19]. Also, the `@` symbol denotes dereferencing. Notice that compound declarations, like `const k = 0, alias x = location1`, are desugared into curried bindings.

Since the desugaring pattern is regular and simple, Tennent [23] derived an *abstraction principle*, stating that a definition construction (*e.g.*, constant, alias, function, procedure, module, ...) can be introduced for each of a language's syntax domains (*e.g.*, numerals, locations, expressions, commands, declarations, ...). Each definition construct is desugared into a λ-abstraction. Similarly, Schmidt [22] proposed a *parameterization principle*, stating that parameter constructions (*e.g.*, numeral parameters, location parameters, expression parameters, command parameters, declaration parameters, ...) can be introduced for each of a language's syntax domains. Again, each formal parameter construct is desugared into a λ-abstraction. Properly applied, the two principles extend systematically a core programming language into a language for programming in-the-large [1, 21, 24, 26].

2.1 The need for records

Desugaring both definitions and parameters into purely λ-abstractions confuses definitions with parameters, which is problematic when a Pascal-like language is studied. Indeed, for the correspondence principle to be of value as a language design criterion, it must be possible for it to *fail.* More importantly, the desugaring of Ada/Standard ML-style modules, which are sets of bindings, as in:

(2)
```
begin module m = (const k = 0, alias x = location1)
in begin use m, function f(a:integer) = a + 2
   in x := k + f(@x) end end
```

can not be modelled easily by simply-typed λ-abstractions if at all. The problem is that modules are "packages of bindings," which are not λ-abstraction-like. For these reasons, we will maintain the integrity of definitions and parameters by desugaring them into records and λ-abstractions, respectively. The metalanguage we use is defined in Figure 1. The metalanguage is reminiscent of the one in Lambek and Scott [7] in its extension of the simply-typed λ-calculus by a product construction, but here we use records rather than tuples. Unlike Lambek and Scott's construction, however, the records are motivated by the pragmatic reasons stated above, not by an explicit desire to discover cartesian closedness.

$$\begin{array}{rcl}
\tau & \in & \text{Type-expression} \\
\iota & \in & \text{Primitive-type (for core language constructs)} \\
\pi & \in & \text{Type-assignment} \\
e & \in & \text{Expression} \\
i & \in & \text{Identifier} \\
op & \in & \text{Core-language-operator}
\end{array}$$

$\tau ::= \iota \mid \tau_1 \rightarrow \tau_2 \mid \pi$
$\pi ::= \{i : \tau_i\}_{i \in I}$, where I is a finite set of distinct identifiers
$e ::= i \mid \lambda\, i : \tau\, .\, e \mid (e_1\ e_2) \mid op\,(e_1, ..., e_n) \mid \{i = e\} \mid \mathit{with}\ e_1\ \mathit{do}\ e_2 \mid e_1, e_2$

$$\pi \vdash i : \tau \ \text{ where } (i : \tau) \in \pi \qquad\qquad \frac{\pi + (i : \tau_1) \vdash e : \tau_2}{\pi \vdash \lambda\, i : \tau_1\, .\, e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\pi \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \pi \vdash e_2 : \tau_1}{\pi \vdash (e_1\ e_2) : \tau_2}$$

$$\frac{\pi \vdash e_1 : \iota_1 \ldots \pi \vdash e_n : \iota_n}{\pi \vdash op\,(e_1, ..., e_n) : \iota} \qquad\qquad \frac{\pi \vdash e : \tau}{\pi \vdash \{i = e\} : \{i : \tau\}}$$

$$\frac{\pi \vdash e_1 : \{i : \tau_i\}_{i \in I} \qquad \pi + (i_1 : \tau_1) + ... + (i_n : \tau_n) \vdash e_2 : \tau}{\pi \vdash \mathit{with}\ e_1\ \mathit{do}\ e_2 : \tau} \ \text{ where } I = \{i_1, ..., i_n\}$$

$$\frac{\pi \vdash e_1 : \{i : \tau_i\}_{i \in I} \qquad \pi \vdash e_2 : \{j : \tau_j\}_{j \in J}}{\pi \vdash e_1, e_2 : \{k : \tau_k\}_{k \in I \cup J}} \ \text{ where } I \cap J = \emptyset$$

Note: $\pi + (i : \tau) = (\pi - \{(i : \tau') | (i : \tau') \in \pi\}) \cup \{i : \tau\}$
$\{i_1 = e_1, i_2 = e_2, \ldots, i_n = e_n\}$ abbreviates
$\{i_1 = e_1\}, \{i_2 = e_2\}, ..., \{i_n = e_n\}$

Figure 1: Meta-language

By convention, we refer to that part of the programming language consisting of primitive arithmetic, logical, and storage operators as the language's *core*; in Figure 1, $op\,(e_1, ..., e_n)$ represents those operators. Definitions and parameters are extensions to the core. Definitions are modelled by records, which are sets of identifier-expression bindings [2]: $\{i = e\}$ is a one field record, e_1, e_2 is record append, and $\mathit{with}\ e_1\ \mathit{do}\ e_2$ makes visible to e_2 the bindings in e_1. The example programs (1) and (2) stated above are desugared respectively into:

with $\{\texttt{k} = \texttt{0}, \texttt{x} = \texttt{location1}\}$ do
    with $\{\texttt{p} = \lambda\,\texttt{y} : \mathit{int}\,.$ `x := @x+y`$\}$ do
        `x := k; (p k)`

and

```
with {m = {k = 0, x = location1}} do
     with m, {f = λ a : int . a + 2} do
          x := k + (f (@ x))
```

This makes clear that binding by definitions, as modelled by records, is inherently different from binding by parameters, as handled by λ-abstraction. In particular, modules are revealed to be records, and importing a module is achieved by a *with* expression.

Records are product-like, and λ-abstractions are exponentiation-like. This leads us to explore their categorical relationship.

3 The Correspondence Theorem

3.1 Computation Rules

Say that a programming language is desugared into the language of Figure 1. The computation rules for the core language constructs are preserved, but the computation rules for definition and parameter binding must desugar into the computation rules for application and *with*, respectively. In a call-by-name calculus, the computation rules for application and *with* have strikingly similar forms:

Definition 1 (Call-by-name reduction)
β-name: $((\lambda\, i : \tau\,.\, e_1)\; e_2) \;\; \triangleright \;\; [e_2/i]e_1$
ρ-name: $\mathit{with}\ \{i_1 = e_1, \ldots, i_n = e_n\}\ \mathit{do}\ e \;\; \triangleright \;\; [e_1/i_1, \ldots, e_n/i_n]e$
where $[e_1/i_1, \ldots, e_n/i_n]e$ is parallel substitution.[1]

Both treat binding as substitution. Indeed, the semantics of the two forms of binding *correspond* in Landin's sense. In a call-by-value language, the computation rules correspond again:[2]

Definition 2 (Call-by-value reduction)
β-value: $((\lambda\, i : \tau\,.\, e_1)\; e_2) \;\; \triangleright \;\; [e_2/i]e_1$, *where e_2 is a value*
ρ-value: $\mathit{with}\ \{i_1 = e_1, \ldots, i_n = e_n\}\ \mathit{do}\ e \;\; \triangleright \;\; [e_1/i_1, \ldots, e_n/i_n]e$,
where $e_1, \ldots, e_n$ are values and the notion of value is predefined, e.g., [17].

But say that the language uses a β-name rule and a ρ-value rule—correspondence fails.

---

[1] The definition of parallel substitution is the usual one, but note that:

$$[e_i/i]_{i\in I}(\mathit{with}\ e_1\ \mathit{do}\ e_2) \;=\; \mathit{with}\ [e_i/i]_{i\in I}e_1\ \mathit{do}\ ([e_i/i]_{i\in I-J})e_2$$

where $\pi \vdash e_1 : \{j : \tau_j\}_{j\in J}$, and $[e_i/i]_{i\in I-J}$ is $[e_i/i]_{i\in I}$ less those substitutions e_j/j, where $j \in J$. This demands that we work only with well-typed programs and computation rules that preserve typing.

[2] Rather than view call-by-value reduction as the application of the call-by-name $\beta\rho$-rules with a fixed reduction strategy, we follow [17] and restrict the $\beta\rho$-rules. This leads to a pleasant equational theory.

Correspondence is aesthetically pleasing, but there is also theoretical justification for it: when it holds, regardless of the binding strategy employed, the category freely generated from the programming language and its computation rules is a symmetric monoidal weakly closed category, that is, the category has an associative, commutative tensor product (with unit) and it has weak exponentiation. These categorical properties formally ensure that records behave correctly (order of construction of subrecords and order of fields are unimportant) and λ-abstractions behave correctly (parameter binding is the same as definition binding).

3.2 The Value set

Let us now summarize the proof of the "correspondence theorem" stated in the previous paragraph. We begin by assuming that the correspondence principle is characterized by the *$\beta\rho$-value* computation rules of Definition 2, where the notion of "value" can be varied, depending on the desired binding strategy. For example, in the call-by-name calculus, *all* expressions are values. In a call-by-value calculus, only some proper subset of the expressions are values.

The set of expressions termed as "values" must satisfy the following conditions:

Definition 3 (Value) *Let Value $\subseteq$ Expression; Value is* well-defined *if the following conditions hold:*

(i) *for all $e \in$ Expression, $\lambda\, i : \tau\, .\, e \in$ Value;*

(ii) *for all $e_i \in$ Expression, $1 \leq i \leq n$,*
if all $e_i \in$ *Value*, then $\{i_1 = e_1, ..., i_n = e_n\} \in$ *Value;*

(iii) *if $e \in$ Value, then for all visible subexpressions, e' within e, $e' \in$ Value. (A subexpression, e', within e, is* visible *if it is not contained inside e_0 of some $(\lambda\, i : \tau\, .\, e_0)$ and it is not contained inside e_2 of some (with e_1 do e_2).)*

The intuition behind Definition 3 goes as follows. All call-by-value calculi treat λ-abstractions as closures, hence Clause (i). If an expression, e, is a value, then packaging it inside a record, $\{i = e\}$, should preserve its value-ness, hence Clause (ii). Clause (iii) is the converse, generalized, of Clause (ii).

We say that an expression, e, *is a value* if $e \in$ *Value*; e *has a value* if $e \equiv e'$ and e' is a value; else e *has no value*. ($e \equiv e'$ means that e is convertible to e' by the computation rules, $\rhd$.) Also, we assume that the computation rules for the core language operators have the form: $op(e_1, ..., e_n) \rhd e$, *where $e_1, ..., e_n$ are values.* Given the above terminology, we add this last clause to Definition 3:

(iv) if $\pi \vdash e : \{i : \tau_i\}_{i \in I}$ and e has a value, then, for all $i \in I$, $e \downarrow i$ has a value. ($e \downarrow i$ abbreviates (*with e do i*)).

This clause, a minor addition, is needed to gain the main result.

For (closed) expressions e_1 and e_2, such that $\vdash e_1 : \tau$ and $\vdash e_2 : \tau$, $e_1 \approx e_2$ iff $e_1 \approx_\tau e_2$, where: $e_1 \approx_\tau e_2$ if e_1 and e_2 have no value, or else:

$e_1 \approx_\iota e_2$ if e_1 and e_2 have values, and $e_1 \equiv e_2$

$e_1 \approx_{\tau_1 \rightarrow \tau_2} e_2$ if e_1 and e_2 have values, and for all a and b such that $a \approx_{\tau_1} b$, $(e_1\ a) \approx_{\tau_2} (e_2\ b)$

$e_1 \approx_{\{i:\tau_i\}_{i \in I}} e_2$ if e_1 and e_2 have values, and for all $i \in I$, $e_1 \downarrow i \approx_{\tau_i} e_2 \downarrow i$,

Figure 2: Equivalence relation

When all the well-defined *Value* sets for *Expression* are ordered by subset inclusion, they form a complete lattice, where the bottom element is formed by the inductive closure over clauses **(i)** and **(ii)** (this is the usual call-by-value calculus, except that no core language expressions are values), and the top element is *Expression* itself (this is the usual call-by-name calculus). From here on, we work with only those λ-calculi whose *Value* sets are well-defined.

3.3 The equivalence relation and category

Our intention is to freely generate a category, **C**, from the language in Figure 1. (We assume acquaintance with elementary category theory [16].) The technique resembles that of Lambek and Scott [7]: objects of **C** are the elements of Type-expression; morphisms in $hom(\tau_1, \tau_2)$ are equivalence classes of (closed) λ-abstractions, $\lambda\, i : \tau_1 \,.\, e$, where $\vdash \lambda\, i : \tau_1 \,.\, e : \tau_1 \rightarrow \tau_2$ holds, with respect to the equivalence relation, $\approx$, defined in Figure 2.

For open terms, $\pi \vdash e_1 : \tau$ and $\pi \vdash e_2 : \tau$, $e_1 \approx^\pi_\tau e_2$ iff for all $(i : \tau_i) \in \pi$, for all $\vdash a_i : \tau_i$ and $\vdash b_i : \tau_i$ such that $a_i \approx_{\tau_i} b_i$, $[a_i/i]e_1 \approx_\tau [b_i/i]e_2$, that is, substitution of equivalent closed terms for free variables yields equivalent closed terms.

Proposition 1

For $\pi \vdash e_1 : \tau$ *and* $\pi \vdash e_2 : \tau$,

(i) $\approx$ *is an equivalence relation;*

(ii) $e_1 \equiv e_2$ *implies* $e_1 \approx^\pi_\tau e_2$;

(iii) $a \approx^{\pi'}_{\tau'} b$ *and* $e_1 \approx^\pi_\tau e_2$ *imply* $[a/x]e_1 \approx^{\pi'}_\tau [b/x]e_2$, *where* $\pi' = \pi - \{x : \tau'\}$;

(iv) $\lambda\, x : \tau' \,.\, e_1 \approx^{\pi'}_\tau \lambda\, y : \tau' \,.\, [y/x]e_1$, *where* $\pi' = \pi - \{x : \tau'\}$;

(v) *if e has a value, then so do all its visible subexpressions;*

The proofs are routine, although clauses **(i)** and **(ii)** must be proved simultaneously: $e \approx^{\pi}_{\tau} e$ with ($e_1 \equiv e_2$ *implies* $e_1 \approx^{\pi}_{\tau} e_2$). Proposition 1 implies:

$$((\lambda\, i : \tau \,.\, e)\; e') \approx^{\pi}_{\tau} \mathit{with}\ \{i = e'\}\ \mathit{do}\ e$$

which is the traditional statement of correspondence.

For convenience, we write a morphism, $[\lambda\, i : \tau_1 \,.\, e]_{\approx} \in \mathit{hom}(\tau_1,\, \tau_2)$, as just $\lambda\, i : \tau_1 \,.\, e$. For an object, τ, the identity morphism, id_{τ}, is $\lambda\, i : \tau \,.\, i$. For morphisms $f = (\lambda\, i_1 : \tau_1 \,.\, e_1) \in \mathit{hom}(\tau_1,\, \tau_2)$ and $g = (\lambda\, i_2 : \tau_2 \,.\, e_2) \in \mathit{hom}(\tau_2,\, \tau_3)$, their composition, $g \circ f \in \tau_1 \rightarrow \tau_3$, is $\lambda\, i_1 : \tau_1 \,.\, ((\lambda\, i_2 : \tau_2 \,.\, e_2)\; e_1)$. With Proposition 1 in hand, we can prove that these definitions give a category.

3.4 Correspondence implies exponentiation

The next, natural, step is to try to show that the category has a categorical product, $\tau_1 \times \tau_2 = \{\mathit{fst} : \tau_1,\, \mathit{snd} : \tau_2\}$; where $\pi_j = \lambda\, i : \{\mathit{fst} : \tau_1,\, \mathit{snd} : \tau_2\} \,.\, i \downarrow j$, for $j \in \{\mathit{fst},\, \mathit{snd}\}$; and $\langle f,\, g\rangle = \lambda\, i : \tau \,.\, \{\mathit{fst} = (f\ i),\, \mathit{snd} = (g\ i)\}$. But the projection laws fail in the case when there is a phrase in the language that has no value. For example, let $f = \lambda\, x : \mathit{int} \,.\, 0$, $g = \lambda\, y : \mathit{int} \,.\, \Omega$, and say that 0 is a value but Ω has no value. Then, $\pi_{\mathit{fst}} \circ \langle f,\, g\rangle \not\approx f$.[3] Instead, we define a tensor product:

$$\begin{aligned} \tau_1 \otimes \tau_2 &= \{\mathit{fst} : \tau_1,\, \mathit{snd} : \tau_2\} \\ f \otimes g &= \lambda\, i : \tau_1 \otimes \tau_3 \,.\, \{\mathit{fst} = f(i \downarrow \mathit{fst}), \mathit{snd} = g(i \downarrow \mathit{snd})\} \end{aligned}$$

for $f \in \tau_1 \rightarrow \tau_2,\ g \in \tau_3 \rightarrow \tau_4$

Proposition 2 *$\otimes$ is a bifunctor on* **C**.

Next, we define the families of functions:

$$a_{\tau_1\tau_2\tau_3} : (\tau_1 \otimes (\tau_2 \otimes \tau_3)) \rightarrow ((\tau_1 \otimes \tau_2) \otimes \tau_3)$$

$$c_{\tau_1\tau_2} : (\tau_1 \otimes \tau_2) \rightarrow (\tau_2 \otimes \tau_1)$$

for all τ_1, τ_2, τ_3 and show that they are natural isomorphisms. If desired, a new type expression, $\{\}$, can be added to the language, and $\pi \vdash () : \{\}$ can be stated as a new axiom. The natural isomorphism:

$$i_{\tau} : (\{\} \otimes \tau) \rightarrow \tau$$

completes the collection: $\otimes$ is associative, commutative, has a unit, and satisfies the MacLane-Kelly coherence conditions [10], hence **C** is a symmetric monoidal category.

[3] Recall that all phrases - even Ω - in a call-by-name calculus are values, so $\{\mathit{fst} : \tau_1,\, \mathit{snd} : \tau_2\}$ is categorical product in this case.

The final step is the validation that **C** is closed, that is, it has categorical exponentiation. With the obvious definitions:

$$
\begin{array}{lcl}
\tau_1 \Rightarrow \tau_2 & = & \tau_1 \rightarrow \tau_2 \\
apply & = & \lambda\, i : \{fst : \tau_1 \Rightarrow \tau_2,\ snd : \tau_1\} \,.\, ((i \downarrow fst)\ (i \downarrow snd)) \\
closure(f) & = & \lambda\, i_1 : \tau_1 \,.\, \lambda\, i_2 : \tau_2 \,.\, f\{fst = i_1,\ snd = i_2\}, \\
 & & \forall\, f : \{fst : \tau_1,\ snd : \tau_2\} \rightarrow \tau_3
\end{array}
$$

we can show the commutativity of the exponentiation diagram:

$$f \approx apply \circ (closure(f) \otimes id) \tag{1}$$

for all $f \in hom(\tau_1, \tau_2)$. This gives us weak exponentiation.

Theorem 1 *The category freely generated by Figures 1 and 2 is a symmetric monoidal weakly closed category.*

We use the term "freely generated" in the theorem, because the extensionality conditions in Figure 2 are for all practical purposes necessary to make morphism composition associative and $\otimes$ a functor. (Indeed, we could formalize the previous remark by a suitable proof of initiality, but that is secondary to our goals here).

We would like to show that the exponentiation is "strong," that is, the choice of $closure(f)$ is unique. But this can fail when there are phrases of function type that have no value. For example, say that there is a family of phrases, Ω_τ, for all types τ, and no Ω_τ has a value. Then, $closure(\lambda\, r : \{fst : \tau_{11},\ snd : \tau_{12}\} \,.\, \Omega_{\tau_2})$ could be either $(\lambda\, a : \tau_{11} \,.\, \lambda\, b : \tau_{12} \,.\, \Omega_{\tau_2})$ or $(\lambda\, a : \tau_{11} \,.\, \Omega_{\tau_{12} \rightarrow \tau_2})$, but $(\lambda\, b : \tau_{12} \,.\, \Omega_{\tau_2}) \not\approx \Omega_{\tau_{12} \rightarrow \tau_2}$, since the former has a value and the latter does not. If all phrases of function type have values, however, strong exponentiation holds. This is the case in the call-by-name calculus, and also in those call-by-value calculi that possess a lifting type, $(\tau)_\perp$, where $\perp$ represents those phrases that have no value. Then, $\vdash \Omega_{\tau_1 \rightarrow \tau_2} : (\tau_1 \rightarrow \tau_2)_\perp$

3.5 Exponentiation implies correspondence

Under restrictive conditions, the existence of weak exponentiation in a language's semantics definition implies that correspondence holds. Say that the category underlying the semantics definition has the weak exponentiation property as in Figure 3. That is, $e_1 \approx apply \circ (closure(e_1) \otimes id)$. The usual reading of the property is that e_1 can be given one of its inputs (the one of type π) and then be packaged into a closure. When the closure is unpackaged and applied to its other input, the result is exactly the same as e_1 applied to both its inputs. Now suppose that a programming language is written in desugared form and its semantics matches the pattern in Figure 4. Then the reading that is appropriate to Figure 3 goes as follows: e_1 is a code fragment that requires definitions of π- and τ_0-typed values. (π denotes the type of the nonlocal definitions; τ_0 is the type of the local definition.) Just the π-definitions can be supplied, giving a code fragment, $closure(e_1)$, which

$$\begin{tikzcd}
\pi \otimes \pi \arrow[r, "id \otimes e_0"] & \pi \otimes \tau_0 \arrow[r, "closure(e_1) \otimes id"] \arrow[dr, "e_1"'] & (\tau_0 \Rightarrow \tau_1) \otimes \tau_0 \arrow[d, "apply"] \\
 & & \tau_1
\end{tikzcd}$$

Figure 3: Weak Exponentiation

Assume that ρ is an environment compatible with π [25].

$$\llbracket \pi \vdash \lambda\, i : \tau_1 \,.\, e : \tau_1 \rightarrow \tau_2 \rrbracket \rho = closure \llbracket \pi + (i : \tau_1) \vdash e : \tau_2 \rrbracket \rho$$
$$\llbracket \pi \vdash (e_1\ e_0) : \tau_2 \rrbracket \rho = apply\{fst = \llbracket \pi \vdash e_1 : \tau_1 \rightarrow \tau_2 \rrbracket \rho\ ,\ snd = \llbracket \pi \vdash e_0 : \tau_1 \rrbracket \rho\}$$
$$\llbracket \pi \vdash \{i = e\} : \{i : \tau\} \rrbracket \rho = \{i = \llbracket \pi \vdash e : \tau \rrbracket \rho\}$$
$$\llbracket \pi \vdash with\ e_0\ do\ e_1 : \tau_1 \rrbracket \rho = \llbracket \pi + (i : \tau_0) \vdash e_1 : \tau_1 \rrbracket (\rho \oplus (\llbracket \pi \vdash e_0 : \{i : \tau_0\} \rrbracket \rho)),$$
where $(\rho \oplus \{i = v\}) = \{fst = \rho\ ,\ snd = v\}$
$$\llbracket \pi \vdash i : \tau \rrbracket \rho = \rho(\text{deBruijn-index-of}(i))$$

Figure 4: Environment Semantics

requires a τ_0-typed parameter. When the parameter is supplied, the result is the same as e_1 with all its definitions for π and τ_0. Figure 3 tells us: $\llbracket \pi \vdash with\ \{i = e_0\}\ do\ e_1 : \tau_2 \rrbracket \rho = \llbracket \pi \vdash ((\lambda\, a : \tau_1 \,.\, e_1)\ e_0) : \tau_2 \rrbracket \rho$. Since the semantics in Figure 4 preserves meaning under substitution: $\llbracket \pi \vdash [e_1/i]e_2 : \tau_2 \rrbracket \rho = \llbracket \pi + (i : \tau_1) \vdash e_2 : \tau_2 \rrbracket (\rho \oplus (i = \llbracket \pi \vdash e_1 : \tau_1 \rrbracket \rho))$, for $e_1 \in Value$, then we immediately derive the soundness of the β-val and the ρ-val reduction rules – correspondence holds. The format in Figure 4 matches that used for lazy imperative languages and functional languages. Of course, a programming language can have correspondence even if its semantics does not match the format in the figure.

3.6 Extensions

Say there is a phrase, Ω_ι, of primitive type that has no value. This implies that phrases without value exist for all types τ: Ω_τ is $((\lambda\, a : \iota \,.\, e_0)\ \Omega_\iota)$, for a closed phrase, e_0, of type τ. This phenomenon prevents a proof of categorical product and strong exponentiation. On the positive side, it means that addition of higher order constants, like *fix*, do not impact the results already proved. We can add a family of binary *fix* operators, with the computation rule:

$$fix\ (i : \tau)\ e \quad \triangleright \quad [(fix\ (i : \tau)\ e)/i]e$$

Although our proof does not require it, in practice, the definition of *Value* should be "monotonic" with respect to substitution and computation in the sense that: (i) if e_1, $e_2 \in \mathit{Value}$, then $[e_1/i]e_2 \in \mathit{Value}$; and (ii) if $e \in \mathit{Value}$ and $e \rhd e'$, then $e' \in \mathit{Value}$ as well. A consequence is that, if the computation rules are orthogonal [6], the rule set possesses the closure property and is confluent [5, 6, 15].

Finally, the computation rules we use can be restricted so that they perform *weak reduction*, that is, an expression is a redex it if is visible and it matches the left-hand side of a computation rule. The proofs of the previous results carry through unaltered, since the reasoning in the proofs is extensional in nature. (But note that the previous remarks regarding confluence do not hold [3].)

4 Applications to Compiling

The obvious impact of correspondence on an implementation of a programming language is that the same implementation of binding can be used for both definitions and parameters. But the framework used to produce the results in Section 3 is of significance in itself because Definition 3 and the $\beta\rho$-value rules provide a natural style of compilation of a program. The idea is simple but important, because virtually all compilers exploit it: the binding of an identifier, i, to an expression, $e \in \mathit{Value}$, can be performed at compile-time. Indeed, this activity might be considered the essence of compiling [20, 27]. Here is an initial, significant example: regardless of a language's binding strategy, a collection of declarations of parameterized subroutines can be processed at compile-time because they form a record of λ-abstractions, which *must* be a value, by Definition 3. For example, regardless of binding strategy, the code segment:

```
begin module m = { procedure p(a:int) = x := a }
in begin use m, function f(b:int) = b + 2
   in call p(f(@x))  end end
```

which desugars to:

$\mathit{with}\ \{m = \{p = \lambda\, a : \mathit{int}\, .\, x := a\}\}\ \mathit{do}$

$\quad \mathit{with}\ m,\ \{f = \lambda\, b : \mathit{int}\, .\, b + 2\}\ \mathit{do}\ (p\ (f\ (@\ x)))$

can be evaluated by a compiler to: $((\lambda\, a : \mathit{int}\, .\, x := a)\ ((\lambda\, b : \mathit{int}\, .\, b + 2)\ (@\ x)))$. This matches the usual compile-time processing. (Of course, a compiler copies addresses to the code for `p` and `f`, rather than the code itself. If a compiler is given additional information, *e.g.*, that numerals are values, then definitions like `const a = 2` can also be evaluated at compile-time.

4.1 Commands as values

If commands are also values, then unparameterized procedures can be evaluated at compile-time, as in the following example:

```
begin const a = 2
in begin procedure p = x := f + a
   in call p; call p end end
```

This desugars to: $with\ \{a = 2\}\ do\ with\ \{p = x := f + a\}\ do\ p; p$ and evaluates at compile-time to: $x := f + 2;\ x := f + 2$. A corresponding example with a command parameter would read:

```
begin const a = 2  in
   begin procedure p = (x := f + a) , procedure q(r:comm) = (r; r)
      in call q(call p)   end end
```

This desugars to:

$$with\ \{a = 2\}\ do\ with\ \{p = (x := f + a), q = (\lambda r : comm.r; r)\}\ do\ (q\ p)$$

and evaluates at compile-time to $x := f+2;\ x := f+2$ as well. Commands, like `x:= f+a`, are values in imperative-style call-by-name languages like Algol-60 as well as in imperative-style call-by-value languages like Pascal. The latter point is often overlooked by Pascal programmers, but a Pascal compiler is well aware of it.

Functional-style call-by-value languages like Scheme and SML usually disallow commands as values. An SML-like program fragment such as:

```
begin function f(a:comm) = 2  in  x:= 0;  f(x:=@x+1)  end
```

is desugared to $with\ \{f = \lambda a : comm.\ 2\}\ do\ x := 0; f(x := @x + 1)$ and is compile-time evaluated to $x := 0; (\lambda a : comm.\ 2)(x := @x + 1)$, but the actual parameter, which is not a value, can not be bound to the formal parameter until run-time. Only when the run-time storage vector is available can the command be evaluated to a value. (In SML, the resulting "value" is ().) Thus, the compiler must generate object code for making the run-time binding.

4.2 Expressions as values

A related situation arises with arithmetic expressions. In a call-by-name language, all expressions are values, and the examples:

```
begin function f = @x + 1   in  x := f ; x := f  end

begin procedure p(f:int) = (x := f ; x := f)  in  call p(@x+1)  end
```

both compile to $x := @x + 1;\ x := @x + 1$. This is an example of the classic Algol-60 copy rule in action. In contrast, in a typical call-by-value language, numerals are values, but compound arithmetic expressions, like $@x + 1$, are not. The best a compiler can do with the previous examples is: $with\ \{f = @x + 1\}\ do\ x := f;\ x := f$ and $(\lambda f : int.\ x := f;\ x := f)(@x + 1)$, respectively. The compiler can not copy the body of f for its invocations. The reason should be clear: the evaluation of `@x+1` to a numeral fixes `f`'s value for all its subsequent uses. Since `@x+1` requires the run-time store for its evaluation, the compiler must generate object code to evaluate the expression and bind its result into (a cell for) `f`.

The above example should not be read as suggesting necessarily that call-by-name is inherently better than call-by-value, but it *is* different. One should note from the example, however, that confusion easily arises when the semantics of expression definitions (*i.e.*, functions) does not correspond to the semantics of expression parameters.

A compiler that folds constants can evaluate arithmetic expressions like 1+2 to values like 3 and copy the results at compile-time.

4.3 Declarations as values

Modules are declarations that are records, and Clauses (ii) and (iii) of Definition 3 ensure that a module is a value exactly when all its components are. The consequences are straightforward. But in imperative languages, variable declarations in modules can affect sharing. Say that a declaration, `var x`, is desugared to `alias x=` *allocate*, where *allocate* needs the run-time store to evaluate. The sharing of variable `x` by modules `n` and `p` in:

```
begin module m = { var x } ,
  module n(a:{x:intloc}) = begin use a in
                            { procedure p = x:=0 }
                           end ,
  module p(b:{x:intloc}) = begin use b in
                            { procedure q = x:=@x+1 }
                           end
in  use n(m) , use p(m)
end
```

is directly dependent upon whether or not the *allocate* operation is a value. This example makes clear why compilers typically "evaluate" *allocate* to a relative address.

5 Conclusion

We have shown the importance of the correspondence principle to modular, higher-order programming languages, and we have validated correspondence by proving it is weak exponentiation in a symmetric monodial closed category. A simple perspective to the main result, the correspondence theorem (Theorem 1), is that the theorem is the natural generalization of the cartesian closedness property for the call-by-name simply-typed λ-calculus [7] to monoidal closedness for call-by-value λ-calculi. But it is only because of the correspondence between product and function space that the generalization is possible. The variety of correspondence (*i.e.*, which variant of call-by-value) is unimportant—what *is* important is the correspondence principle itself.

Acknowledgements

Olivier Danvy, Pascal Fradet and the referees contributed many helpful suggestions. Thanks are also due to Andrzej Filinski for perceptive comments which led to a generalization of the main result. Peter O'Hearn's comments benefitted

Section 3. Naz Karjian's help in making slides out of Figure 1 for the MFPS presentation is acknowledged. The categorical diagram of Section 3 was typeset using John Reynolds' category-theory macros.

References

[1] Anindya Banerjee, Olivier Danvy, and David A. Schmidt. A programming language workbench. Technical Report CIS-92-4, Kansas State University, Manhattan, Kansas, November 1991.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17:471–522, 1985.

[3] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. Rapports de Recherche 1617, I.N.R.I.A., February 1992.

[4] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[5] Jan W. Klop. *Combinatory Reduction Systems*. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

[6] Jan W. Klop. Term rewriting systems. Technical Report CS-R9073, CWI: Center for Mathematics and Computer Science, Amsterdam, 1990.

[7] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge studies in advanced mathematics*. Cambridge University Press, 1986.

[8] Peter J. Landin. A correspondence between ALGOL60 and Church's lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965.

[9] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[10] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[11] David B. MacQueen. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207, Austin, Texas, August 1984.

[12] David B. MacQueen. Using dependent types to express modular structure. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 277–286, January 1986.

[13] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional programming languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

[14] Ian A. Mason and Carolyn L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2):167–215, 1992.

[15] Michael O'Donnell. *Computing in Systems Described by Equations*. Number 58 in Lecture Notes in Computer Science. Springer-Verlag, 1977.

[16] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. Foundation of Computing Series. The MIT Press, 1991.

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

[18] John C. Reynolds. GEDANKEN — a simple typeless language based on the principles of completeness and the reference concept. *Communications of the ACM*, 13:308–319, 1970.

[19] John C. Reynolds. The essence of Algol. In van Vliet, editor, *International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, 1982. North-Holland.

[20] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1988.

[21] David A. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, Cambridge, MA. To appear.

[22] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[23] Robert D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977.

[24] Robert D. Tennent. *Principles of Programming Languages*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[25] Robert D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.

[26] David Watt. *Programming Languages Concepts and Paradigms*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[27] Stephen Weeks and Matthias Felleisen. On the orthogonality of assignments and procedures in algol. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 20–35, Charleston, South Carolina, January 1993.