

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 $j(U)$

where j is fresh.

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

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