PUTTING IT IN CONTEXT: A SYNTACTIC THEORY OF INCREMENTAL PROGRAM CONSTRUCTION

Shinn-Der Lee

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science

Indiana University

June 1996

This report is based on work supported in part by the National Science Foundation grants numbered CCR 93-02114 and CDA 93-12614.

Acknowledgements

I would like to thank my advisor Daniel P. Friedman for introducing me to the enchanting world of programming languages, for fascinating me with his relentless pursuit of new ways to program, for his interest and encouragement in my work, and for his patience and supervision during the writing of this thesis.

I would also like to thank the other members of my doctoral committee: Christopher T. Haynes, Paul W. Purdom, and George Springer. Their support and advise have been enormously valuable to the completion of this thesis.

I am grateful to Matthias Felleisen for answering all my questions about programming languages and lambda calculi. I wish to thank Stan Jefferson. Through numerous conversations, he has sharpened my ability to formalize my ideas about programming. He has also guided me through the early stages of my endeavor in English technical writing.

My fellow graduate students have not only provided insightful criticisms on my work, but have also been friends. Thanks go to Mike Ashley, Mayer Goldberg, Julia Lawall, Jenq-Kuen Lee, Anurag Mendhekar, Jon Rossie, John Simmons, Jonathan Sobel, and Simon Tung.

Finally, I wish to thank my family. I am grateful to my parents, whose unwavering trust has been a constant source of motivation for me to carry on. I am especially indebted to my loving and caring wife Wei and our son Daniel. Without their faith in me, my graduate studies would not have been possible.

Abstract

The ability to construct programs in an incremental fashion is a premise of popular programming paradigms such as modular programming, object-oriented programming, and interactive programming. Incremental program development amounts to filling the holes of well-planned program contexts with new experimental components, or assembling new programs from previously developed and thoroughly tested components, or a combination of both. Its popularity lies in the flexibilities it provides for reusing existing program components and program structures. The benefits of constructing programs incrementally are reduced demand on human and computing resources, faster turn-around time, and enhanced program reliability.

We present a schema for enhancing programming languages with incremental program construction capabilities based on the notion of program contexts. By perceiving fully-evolved program contexts (proper parse trees) as compiled code, partially-evolved program contexts (improper parse trees with holes as non-terminal leaves) as compilation operators, and context hole filling as the means to link together compiled code, the schema conservatively extends programming languages with mechanisms capable of modeling the incremental construction, linking, and loading of compiled program components.

We use the schema to enrich three λ-calculi. The enriched pure λ-calculus is capable of expressing a metacircular compiler for itself. The enriched λ-calculus with elaborate variable definition notations is a module manipulation language suitable for modular programming. The enriched λ-calculus with elaborate variable referencing devices is capable of expressing variable references whose linking relation can be altered when their context evolves. Such relinkable variable references can provide the late binding behavior required by interactive and object-oriented programming.

Our language design schema is unique in three aspects. First, it distinguishes the construction of programs from the execution of programs. Linking is described strictly as a compile-time operation, rather than in terms of run-time computational steps such as environment lookups or record field selections. Second, the language design methodology our schema introduces is modular. The induced incremental program construction operations can be added to a language one at a time. Hence, extensibility and modifiability are available at the language design level. Third, the two basic code editing functions required by our schema are the copying of code that models the reuse of existing program components and the renaming of variables that models the linking of existing program components. The incremental program construction capabilities derived from our schema thus mimic source code editing if programs were constructed incrementally by hand.

Contents

List of Figures

Chapter 1

Introduction

We present a schema for incorporating incremental program construction capabilities into programming languages.

The ability to build programs in an incremental fashion is a premise of many programming systems. Source code editing using a text editor is sufficient for the job. The approach has not flourished into a mainstream programming paradigm for obvious reasons. Human intervention is error prone. The repetitive text editing chore can be more effectively handled by machines. Furthermore, recompilation of source code is a terrible waste of computing resources. More critically, the source code of some widely used utility programs are not available because of commercial or proprietary restrictions. Realistic incremental program construction must therefore be based on some form of machine code, rather than on source code.

Among the prominent alternatives is modular programming. Modules are tightly encapsulated program components that communicate with one another only through clearly specified import and export interfaces. Programs are formed by linking separately-developed modules together. The linking of modules is static. Once a link is established, it can never be broken. This stringent constraint is not the best for incremental program construction. Time and again, a program can be obtained by modifying only a few links of an existing system of modules. There is no need to take apart the entire system to accommodate the few necessary changes.

Object-oriented programming is an enhancement of modular programming. Objects have encapsulation capabilities similar to modules. They also have the static linking features of modules. Above and beyond, objects embrace a progressive modification mechanism allowing for the relinking of import attributes often explained in terms of late binding variable references.

Another popular form of incremental program development is interactive programming. An interactive evaluator uses an incrementally constructed environment to keep track of previously evaluated definitions. A new program is constructed each time a definition is submitted to the evaluator for evaluation. The interactive environment is used in the evaluation of the submitted definition. The result is then used to extend or override the existing interactive environment to yield a new environment. The interactive environment therefore constitutes a growing program in which mutually dependent definitions are added in an incremental fashion.

The popularity of the above incremental programming paradigms stems from the degree of flexibility they provide in the reuse of existing program code and program structures. The obvious benefits of reuse are reduced demand on human and computing resources as well as faster turn-around time. Moreover, if reuse of some existing code is in high demand, it is economically feasible to verify the code vigorously. Consequently, by reusing proven existing code, program reliability can be greatly enhanced.

Incremental program development amounts to filling in the blanks of well-planned program structures with new experimental components, or assembling new programs from previously developed and thoroughly tested components, or a combination of both. To illustrate, the following example, which is taken from Abelson and Sussman [4], shows an interactive Scheme [21] programming session that develops a program for computing the square root of a given number **x** using Newton's method of successive approximations (the interactive Scheme evaluator issues a prompt (>) preceded by a number added for reference purposes when it is ready to accept the next input expression):

```
1> (define sqrt (lambda (x) (sqroot 1.0 x)))
2> (define sqroot
     (lambda (g x)
       (if (good? g x) g (sqroot (improve g x) x))))
3> (define improve (lambda (g x) (/ (+ g (/ x g)) 2)))
4> (define good? (lambda (g x) (< (abs (- (* g g) x)) 0.001)))
5> (sqrt 0.0001)
   0.0323
6> (define good?
     (lambda (g x)
       (< (abs (/ (- (improve g x) g) g)) 0.001)))
7> (sqrt 0.0001)
   0.0100
```

The idea is to start with a guess **g** for the square root of **x**. We are done if the guess is good enough for our needs; otherwise, we can repeat the process with an improved guess that is the average of **g** and **x/g**. The function **sqrt** is the intended program for computing square roots, the function **sqroot** implements the iterative approximation process, the function **improve** computes the next guess, and the predicate **good?** decides when the guess is good enough for the approximation process to terminate.

Later on, we discover that a better **good?**-predicate is necessary to work with very small numbers such as **0.0001**; see the obviously unacceptable response to the fifth prompt. We can do so by revising the definition of **good?**, hence the input to the sixth prompt. Such a redefinition of **good?** is seen by the function **sqroot** and so a new program for computing square roots is developed, as witnessed by the result of the seventh prompt.

The incremental construction of the square root program can be described as follows:

0> Initially, the evaluator holds a **letrec**-expression with a blank $\square_1$ in the location for its recursive bindings and another blank $\square_2$ in the position for its body expression:

```
(letrec ($\square_1$) $\square_2$)
```

1> Evaluating the definition of **sqrt** amounts to filling in $\square_1$ with the binding of **sqrt** and another occurrence of $\square_1$, hence yielding:

```
(letrec ((sqrt (lambda (x) (sqroot 1.0 x)))
         $\square_1$)
  $\square_2$)
```

Likewise, evaluating the next three **define**-expressions incrementally fills in $\square_1$ with bindings of **sqroot**, **improve**, and **good?**:

```
(letrec ((sqrt (lambda (x) ··· sqroot ···))
         (sqroot (lambda (g x) ··· good? ··· sqroot ··· improve ···))
         (improve (lambda (g x) ···))
         (good? (lambda (g x) ···))
         $\square_1$)
  $\square_2$)
```

In the process, the free variable references **sqroot**, **improve**, and **good?** used in the functions **sqrt** and **sqroot** are incrementally linked to their latest binding due to the recursive nature of the **letrec**-expression.

5> Evaluating a non-**define**-expression means replacing the blank $\square_2$ with the expression after removing the blank $\square_1$. So, computing the square root of **0.0001** amounts to constructing the following program:

```
(letrec ((sqrt (lambda (x) ··· sqroot ···))
         (sqroot (lambda (g x) ··· good? ··· sqroot ··· improve ···))
         (improve (lambda (g x) ···))
         (good? (lambda (g x) ···)))
  (sqrt 0.0001))
```

6> The redefinition of **good?** fills in the blank $\square_1$ with a new binding for **good?**; moreover, the previous binding of **good?** is hidden by converting its name to something inaccessible:

```
(letrec ((sqrt (lambda (x) ··· sqroot ···))
         (sqroot (lambda (g x) ··· good? ··· sqroot ··· improve ···))
         (improve (lambda (g x) ···))
         (_ (lambda (g x) ···))
         (good? (lambda (g x) ··· improve ···))
         $\square_1$)
  $\square_2$)
```

Consequently, the free variable reference **good?** in the definition of **sqroot** is linked to the new binding of **good?** to yield a new square root function.

As shown by the example above, a key machinery needed in the incremental construction of programs is an expressive linking mechanism to facilitate easy addition or modification of existing program components. It is the aim of this thesis to deepen the understanding of such a notion of linking. Our contribution is a schema for enhancing programming languages with incremental program construction capabilities based on the ability to link together separately-compiled program components. The schema is unique in three aspects. First, it distinguishes the incremental construction of programs from the execution of programs. As a result, linking is not described in terms of run-time computational steps such as environment lookups or record field selections. Instead, it is modeled strictly as a compile-time operation. Second, the language design methodology our schema introduces is modular. Incremental program construction operations can be added to a language one at a time. This is made possible in part by the fact that our schema distinguishes between compile-time linking and run-time computation. Third, the linking capabilities derived from our schema mimic source code editing. The two basic editing functions required by our schema are consistent with those that would have been used most often if editing were done by hand. They are the copying (inclusion) of code that models the reuse of existing program components and the renaming of variables that models the linking of existing program components.

1.1 Incremental Program Construction

We address the difficulties involved in enhancing a programming language with incremental program construction capabilities in terms of the pure λ-calculus [7]. We choose the λ-calculus as a representative to convey our design methodology for two reasons. First, the λ-calculus possesses all the basic obstacles to incremental program construction inherently associated with most languages. Second, the syntactic nature of the λ-calculus helps highlight the similarities between the linking features induced by our schema and the most intuitive form of incremental program construction, *i.e.*, source code editing. We refer the reader not familiar with the basic inner workings of the λ-calculus to Chapter 2.

1.1.1 Incremental Machine Code Construction

The term language of the untyped (pure) λ-calculus has the following abstract syntax:

$$e \quad ::= \quad x \mid \lambda x.e \mid e\ e$$

That is, a λ-term e is either

- a variable reference x that refers to the closest function parameter named x,
- an abstraction $\lambda x.e$ that models a function with parameter x and body e, or
- an application $e_1\ e_2$ that models the application of the function denoted by the term e_1 to the argument denoted by the term e_2.

Traditionally, the λ-calculus is perceived as a substitution-based computational machine in which λ-terms are the machine code. A machine instruction is then a β-redex of the form $(\lambda x.e_1)\ e_2$. It models the application of the function $\lambda x.e_1$ to the argument e_2. Its execution is modeled by substituting the argument e_2 for every reference to the function parameter x, which is denoted as

$$[e_2/x]e_1$$

Based on the way λ-terms are defined and the correspondence between machine code and λ-terms, one might think that incremental machine code construction capabilities are readily built into the λ-calculus. This is unfortunately a misconception. The culprit is that the λ-calculus is a statically-scoped language. It means that the linking relation between a function parameter and the variable references that refer to the parameter is static. It cannot be altered by any means. This vital characteristic of static scope is upheld in the λ-calculus by the α-conversion and β-substitution meta-operations. The former says that the name of a function parameter can be replaced as long as the parameter's linking relation is maintained. Hence, changing the parameter names x and y of $\lambda x.\lambda y.(x\ y)$ to w and z, thus yielding the λ-term $\lambda w.\lambda z.(w\ z)$, does not affect the function's behavior. The two λ-terms actually denote the same function.

The β-substitution meta-operation complies with static scope by employing α-conversion to avoid inadvertent alteration of any function parameter's linking relation. For instance, substituting the term $\lambda y.z$ for the variable x in $\lambda z.(z\ x)$ must not simply replace the x of $\lambda z.(z\ x)$ with $\lambda y.z$. It would yield the term $\lambda z.(z\ \lambda y.z)$ in which the static linking relation of the function parameter z is violated. Instead, the name of the function parameter z is changed to some fresh z' to yield the result $\lambda z'.(z'\ \lambda y.z)$ where the function parameter's static linking relation persists.

Consequently, any attempt to define a function in the λ-calculus that would take any machine code e and construct out of it the function $\lambda x.e$ in which the parameter x would link with the free occurrences of the variable x in e is destined to fail. It would require the changing of the linking relation of the function parameter x (or equivalently, the linking relation of the free variable references x in e), an anomaly known as variable capture that is outlawed by the nature of static scope. Unfortunately, such an anomalous feature is precisely the essence of the linking capabilities needed to construct programs incrementally at the machine code level.

1.1.2 Incremental Source Code Construction

The notion of capture does exist in the theoretical study of the λ-calculus, though not with λ-terms. It is associated with a separate meta-notion called contexts [7]. The abstract syntax of λ-contexts is:

$$\begin{array}{rcl} \mathbf{h} & \in & \mathit{Holes} = \{\mathbf{h}_1, \mathbf{h}_2, \ldots\} \\ \mathbf{C} & ::= & \mathbf{h} \mid \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C} \mid \mathbf{C}\ \mathbf{C} \end{array}$$

That is, a λ-context $\mathbf{C}$ is either

- a hole $\mathbf{h}$,
- an identifier $\mathbf{x}$,
- an abstraction $\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}$ of a context $\mathbf{C}$ over the identifier $\mathbf{x}$, or
- an application $\mathbf{C}_1\ \mathbf{C}_2$ of a context $\mathbf{C}_1$ to another context $\mathbf{C}_2$.

Although λ-terms e and λ-contexts $\mathbf{C}$ are built out of the same syntactic structures, they are based on distinct notions of names: variables x for λ-terms versus identifiers $\mathbf{x}$ and holes $\mathbf{h}$ for λ-contexts. We thus typeset λ-contexts in boldface to distinguish them from λ-terms.

Whereas λ-terms model machine code, λ-contexts can be characterized as representing incomplete source code—incomplete in the sense that their parse trees may have *non*-terminal leaves designated by holes. Thus, variables of λ-terms correspond to machine locations while identifiers of λ-contexts are source program symbols.

The analogue of β-substitution $[e_2/x]e_1$ for λ-contexts is *hole filling*, which is denoted as

$$[\mathbf{C}_2/\mathbf{h}]\mathbf{C}_1$$

It substitutes the λ-context $\mathbf{C}_2$ for occurrences of the hole $\mathbf{h}$ in $\mathbf{C}_1$. Conceptually, $[\mathbf{C}_2/\mathbf{h}]\mathbf{C}_1$ amounts to replacing each non-terminal leaf designated by the name $\mathbf{h}$ in the parse tree $\mathbf{C}_1$ with the parse tree $\mathbf{C}_2$. In contrast to β-substitution, hole filling $[\mathbf{C}_2/\mathbf{h}]\mathbf{C}_1$ has the effect of *literally* replacing occurrences of $\mathbf{h}$ in $\mathbf{C}_1$ with $\mathbf{C}_2$. Identifier capture is indeed solicited. For instance, filling the hole $\mathbf{h}$ of the λ-context $\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}$ with the λ-context $\mathbf{x}$ yields $\boldsymbol{\lambda}\mathbf{x}.\mathbf{x}$, not $\boldsymbol{\lambda}\mathbf{x}'.\mathbf{x}$ for some identifier $\mathbf{x}'$ distinct from $\mathbf{x}$. That is, the replacement context $\mathbf{x}$ for the hole $\mathbf{h}$ gets linked to the parameter $\mathbf{x}$ of the abstraction context $\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}$.

Using hole filling, we can construct any application λ-context $\mathbf{C}_1\ \mathbf{C}_2$ out of the λ-contexts $\mathbf{C}_1$ and $\mathbf{C}_2$ by filling the holes $\mathbf{h}_1$ and $\mathbf{h}_2$ of the application λ-context $\mathbf{h}_1\ \mathbf{h}_2$ with $\mathbf{C}_1$ and $\mathbf{C}_2$, respectively. Likewise, by filling the hole $\mathbf{h}$ of the abstraction λ-context $\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}$ with $\mathbf{C}$, we can construct any abstraction λ-context $\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}$ in which free occurrences of $\mathbf{x}$ in $\mathbf{C}$ are linked to the parameter $\mathbf{x}$ of $\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}$ via identifier capture. Intuitively, there are only two categories of terminal node in a parse tree, namely, identifiers and holes. Furthermore, there are only two categories of internal node in a parse tree. They correspond to application contexts and abstraction contexts. Hence, by the inductive nature of the way λ-contexts are defined, starting with holes $\mathbf{h}$ and identifiers $\mathbf{x}$, any composite λ-context can be incrementally constructed using the two basic hole-filling operations described.

So, with λ-contexts, incremental program construction capabilities are readily available, but only at the source code level. It is then our goal to integrate into a single calculus the capabilities of λ-contexts and λ-terms. The task is non-trivial since β-substitution and hole filling cannot operate on the same domain. In particular, functions may no longer be statically scoped because of the existence of holes. As a result, we cannot simply transplant the α-conversion and β-reduction of λ-terms to λ-contexts, or, conversely, the hole filling of λ-contexts to λ-terms. For example, we cannot rename the parameter y of $\lambda y.(x\ \mathbf{h})$ to yield $\lambda z.(x\ \mathbf{h})$ because we do not have a complete picture of its scope; indeed, we have no idea what the hole $\mathbf{h}$ will be filled with. Consequently, we cannot treat the application

$$(\lambda x.\lambda y.(x\ \mathbf{h}))\ \lambda x.y$$

as a β-redex and substitute $\lambda x.y$ for the variable x in $\lambda y.(x\ \mathbf{h})$. That would require renaming the parameter y of the function $\lambda y.(x\ \mathbf{h})$.

1.1.3 Incremental Compiled Code Construction

The problem is that we overlook the distinction between the purpose of λ-terms and λ-contexts, one is for modeling computation and the other is for modeling program construction. So, instead of forcing every term in the integrated calculus to play the role of both machine code (λ-terms) and source code (λ-contexts) at the same time, we separate the domains of β-substitution and hole filling. The integrated calculus is therefore a mixture of machine code and source code. Of course, there is then the need to compile source code into machine code. For that, we must incorporate compilation mechanisms into the integrated calculus. But then, why can't we simply treat source code as compiled code since they are subject to compilation anyway. So, rather than extending the calculus of λ-terms with λ-contexts, our alternative is to translate λ-contexts into compiled code and extend the λ-calculus with compiled λ-contexts, thus obtaining incremental program construction capabilities at the compiled code level.

The general schema is as follows. We distinguish between λ-contexts with and without holes since they have different compilation interpretations. A *(fully-)evolved* λ-context $\mathbf{e}$ has no holes:

$$\mathbf{e} \ ::= \ \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{e} \mid \mathbf{e}\ \mathbf{e}$$

It is a piece of complete source code ready for compilation. It is not generally feasible to compile an evolved λ-context into some machine code, however. At issue are the free identifiers of the context. We cannot simply translate them into free λ-term variables. Once that is done, we lose the capacity to link them later since free variable capture is not expressible with statically-scoped λ-terms. Hence, we need new terms in the integrated calculus to model compiled but not yet fully-linked version of evolved contexts. Moreover, there should also be a compiled code loading mechanism to transform compiled code into machine code once the compiled code is fully-linked.

A *partially-evolved* λ-context $\mathbf{C}^+$, on the other hand, is a λ-context that contains at least one hole:

$$\mathbf{C}^+ \quad ::= \quad \mathbf{h} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C}^+ \mid \mathbf{e}\ \mathbf{C}^+ \mid \mathbf{C}^+\ \mathbf{e} \mid \mathbf{C}^+\ \mathbf{C}^+$$

When the holes of a partially-evolved context are filled with other contexts, new contexts are constructed. A partially-evolved context can therefore be seen as a compilation operator that constructs new compiled code out of existing compiled code. Consequently, depending on the evolvedness of the filler contexts, hole filling can be perceived as the application or composition of such compiled code constructors. Furthermore, by the inductive nature of their definitions, composite contexts can be constructed from filling the holes of the two most basic partially-evolved contexts, namely, $\mathbf{h}_1\ \mathbf{h}_2$ and $\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}$. We therefore require only two basic incremental compiled code constructors, one for each of the two basic composite contexts.

To summarize, our schema of enriching the λ-calculus is to add to the λ-calculus these compilation mechanisms to simulate the behavior of λ-contexts:

- new terms for modeling the compiled but not yet fully-linked code of evolved λ-contexts,
- a loading operation for assimilating compiled code into machine code, and
- a basic compilation operator for modeling the incremental construction of each category of composite λ-contexts.

The integrated calculus is therefore a mixture of machine code, compiled code, and incremental compiled code constructors. It still models program execution with β-reductions. In addition, the extended calculus is capable of expressing the construction, linking, and loading of separately-developed compiled code in an incremental fashion.

It is our intention to demonstrate that the context-enriching schema described above does indeed enhance a programming language with incremental program construction capabilities. In the first half of this thesis we show the feasibility of the context-enriching schema on the pure λ-calculus. We develop in detail the necessary compilation mechanisms, study their novel features with emphasis on their linking capabilities, and investigate their interaction with the computational aspects of the λ-calculus. Also included are proofs that the induced compilation mechanisms are in fact capable of simulating the behavior of λ-contexts.

The applicability of our schema is not restricted to the pure λ-calculus. It can be easily adapted to real programming languages. Obviously, more basic compilation operators must be added to accommodate extra core constructs. Likewise, new categories of compiled code must be introduced to model additional categories of evolved contexts. There is no change to the fundamentals of their linking mechanisms, however. In particular, linking can always be modeled as variable capture. To demonstrate our point, we apply the context-enriching schema to languages with more descriptive variable defining and referencing mechanisms in the second half of the thesis.

The rest of this chapter provides an overview of our work. Each section below is a short summary of one of the succeeding chapters.

1.2 Lambda Calculus

As a starting point, we survey the basics of the untyped λ-calculus [7]. We pay special attention to two fundamental concepts of the λ-calculus that are most relevant to our work, namely, variable renaming and contexts.

Although variable renaming is traditionally used as a means to avoid inadvertent variable capture, it is not essential to the understanding of programming with the λ-calculus. By tinkering with the representation of λ-terms [9, 14, 34] or the definition of β-substitution [1], the β-reduction rule can be framed in a setting in which variable renaming is unnecessary. We have found that variable renaming is indispensable to our work, however. Our incremental compiled code construction operations rely exclusively on the renaming of variables to model the linking of separately-developed program components. A detailed description of variable renaming is a must.

Our schema is founded on the notion of contexts. A comprehensive description of the behavior of contexts is thus in order. Particularly, we review in detail the primary meta-operation on contexts, namely, hole filling. Furthermore, we demonstrate that allowing hole filling and β-substitution to operate on the same domain easily leads to the contradictory conclusion that all programs are equal, hence showing the potential difficulties involved in integrating the two programming notions in a single system.

1.3 Context-Enriched Lambda Calculus

We give a comprehensive description of the application of our context-enriching schema to the pure λ-calculus. The result, $\boldsymbol{\lambda}\mathbf{C}$, is a conservative extension of the λ-calculus with incremental compiled code construction capabilities.

The key mechanism needed in the development of $\boldsymbol{\lambda}\mathbf{C}$ is a compiled code representation of evolved λ-contexts (source code). Let q be a one-to-one function that assigns a distinct variable name $q(\mathbf{x})$ to each identifier $\mathbf{x}$. Then, the image of an evolved λ-context $\mathbf{e}$ is the λ-term $im(\mathbf{e})$ defined inductively on the structure of $\mathbf{e}$ as follows:

$$\begin{array}{rcl} im(\mathbf{x}) & \equiv & q(\mathbf{x}) \\ im(\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}) & \equiv & \lambda q(\mathbf{x}).im(\mathbf{e}) \\ im(\mathbf{e}_1\ \mathbf{e}_2) & \equiv & im(\mathbf{e}_1)\ im(\mathbf{e}_2) \end{array}$$

That is, $im(\mathbf{e})$ has the same structure as $\mathbf{e}$ but with each identifier $\mathbf{x}$ replaced by $q(\mathbf{x})$. For example, $im(\boldsymbol{\lambda}\mathbf{x}.(\mathbf{y}\ (\mathbf{x}\ \mathbf{z})))$ is the λ-term $\lambda x.(y\ (x\ z))$, where x, y, and z are the unique variable names assigned to $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ by q. An evolved λ-context $\mathbf{e}$ is then compiled into the following *free identifier abstraction*

$$\Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.im(\mathbf{e})$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are the free identifiers of $\mathbf{e}$ and $x_1, \ldots, x_n$ are the unique variable names $q(\mathbf{x}_1), \ldots, q(\mathbf{x}_n)$. Such a free identifier abstraction models compiled but not yet fully-linked code. The specification $\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}$ indicates that free occurrences of the variables $x_1, \ldots, x_n$ in the machine code e are *temporary* placeholders for the unlinked free identifiers $\mathbf{x}_1, \ldots, \mathbf{x}_n$.

Like λ-parameters, the parameter variables $x_1, \ldots, x_n$ of a free identifier abstraction $\Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e$ are considered the abstraction's bound variables; hence, in the wake of β-substitution, they can be renamed to avoid inadvertent variable capture. Unlike λ-parameters, Φ-parameter variables are *quasi-statically scoped* [42]. The free occurrences of x_i in e are not statically linked to the Φ-parameter $\mathbf{x}_i : x_i$; they can be *relinked*. Such quasi-statically scoped Φ-parameter variables facilitate the sought-after variable linking mechanism needed in incremental program construction.

According to our schema, there are two incremental compiled code construction operators, one for constructing the compiled code of evolved λ-abstractions and one for constructing the compiled code of evolved applications. Linking is needed in the former operation. The compilation operator involved is $\mathbf{lam}_{\mathbf{X}}$ (one for each identifier $\mathbf{x}$ to be exact). Metaphorically speaking, it is a unary operator that incrementally constructs the compiled code of a λ-context $\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}$ from the compiled code of a λ-context $\mathbf{e}$. In particular, let the free identifier abstraction $\Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e$ be the compiled code of the λ-context $\mathbf{e}$. Then,

$$\mathbf{lam}_{\mathbf{X}}\ \Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e \ \ \rightarrow \ \ \Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.\lambda x.e$$

The resulting free identifier abstraction $\Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.\lambda x.e$ is the compiled code of the λ-context $\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}$. Notice that in the process, the operator $\mathbf{lam}_{\mathbf{X}}$ takes advantage of the quasi-statically-scoped nature of the free occurrences of x (x is x_i if $\mathbf{x}$ is some $\mathbf{x}_i$; otherwise, x is a fresh variable name) in e and relinks them to the parameter x of the newly constructed λ-abstraction, thus mimicking the identifier capture behavior of context hole filling.

The other incremental compiled code construction operator **app** is a binary operator that constructs the compiled code of a λ-context $\mathbf{e}_1\ \mathbf{e}_2$ from the compiled code of λ-contexts $\mathbf{e}_1$ and $\mathbf{e}_2$. Let $\Phi\rho_1.e_1$ and $\Phi\rho_2.e_2$ be the respective compiled code of the λ-contexts $\mathbf{e}_1$ and $\mathbf{e}_2$. Then,

$$\mathbf{app}\ \Phi\rho_1.e_1\ \Phi\rho_2.e_2 \quad \rightarrow \quad \Phi\rho_1 \uplus \rho_2.(e_1\ e_2)$$

The result is the compiled code of the λ-context $\mathbf{e}_1\ \mathbf{e}_2$. The notation $\rho_1 \uplus \rho_2$ denotes a variant of the union of the free identifier parameter sets $\rho_1 \equiv \{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_m : x_m\}$ and $\rho_2 \equiv \{\mathbf{y}_1 : y_1, \ldots, \mathbf{y}_n : y_n\}$ that uses variable renaming to ensure that an identifier and its variable can occur in the union once only. It models the fact that free occurrences of the same identifier in both $\mathbf{e}_1$ and $\mathbf{e}_2$ should be assigned the same temporary placeholder in both e_1 and e_2.

1.4 Simulations of Lambda Contexts

Our schema enriches the λ-calculus with compilation capabilities to model the behavior of λ-contexts. It is thus imperative that the notion of context hole filling is indeed expressible in $\boldsymbol{\lambda}\mathbf{C}$. We give not one but two simulations.

The first simulation is in essence an incremental compiler for the λ-calculus. An evolved λ-context $\mathbf{e}$ is translated into a constant function that always returns its compiled code in the form of the Φ-abstraction

$$\Phi\{\mathbf{x}_1 : q(\mathbf{x}_1), \ldots, \mathbf{x}_n : q(\mathbf{x}_n)\}.im(\mathbf{e})$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are the free identifiers of $\mathbf{e}$. A partially-evolved λ-context $\mathbf{C}^+$, on the other hand, is encoded as some composition of the two compiled code construction operators $\mathbf{lam}_\mathbf{x}$ and **app** of $\boldsymbol{\lambda}\mathbf{C}$. A hole filling operation $[\mathbf{C}_2/\mathbf{h}]\mathbf{C}_1$ is then interpreted as a composition of the encodings of $\mathbf{C}_1$ and $\mathbf{C}_2$.

The second simulation does not rely on distinguishing evolved and partially-evolved contexts. Every λ-context $\mathbf{C}$ is translated into its image $im(\mathbf{C})$ defined as follows (the one-to-one function q has been extended to assign a unique variable name $q(\mathbf{h})$ to each hole $\mathbf{h}$ as well):

$$\begin{array}{rcl}
im(\mathbf{h}) & \equiv & q(\mathbf{h}):\{\mathbf{x}_1:q(\mathbf{x}_1),\ldots,\mathbf{x}_n:q(\mathbf{x}_n)\} \\
im(\mathbf{x}) & \equiv & q(\mathbf{x}) \\
im(\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}) & \equiv & \lambda q(\mathbf{x}).im(\mathbf{C}) \\
im(\mathbf{C}_1\ \mathbf{C}_2) & \equiv & im(\mathbf{C}_1)\ im(\mathbf{C}_2)
\end{array}$$

Each hole $\mathbf{h}$ is represented by its unique variable $q(\mathbf{h})$ *annotated* with the identifiers it intends to capture, namely, $\mathbf{x}_1,\ldots,\mathbf{x}_n$. The annotation $\{\mathbf{x}_1:q(\mathbf{x}_1),\ldots,\mathbf{x}_n:q(\mathbf{x}_n)\}$ is a linking mechanism that maps identifiers (source code symbols) $\mathbf{x}_1,\ldots,\mathbf{x}_n$ to variables (machine code locations) $q(\mathbf{x}_1),\ldots,q(\mathbf{x}_n)$. It states that when the hole $\mathbf{h}$ is filled with some context $\mathbf{C}'$, free occurrences of the identifiers $\mathbf{x}_1,\ldots,\mathbf{x}_n$ in $\mathbf{C}'$ are replaced by their variables $q(\mathbf{x}_1),\ldots,q(\mathbf{x}_n)$, thus accomplishing the desired linking effect. Hence, in the second simulation, every context is fully compiled. They are then linked together using the above hole annotation mechanism. In other words, the second simulation is an incremental linker for λ-terms.

1.5 Twice Context-Enriched Lambda Calculus

By applying our incremental program construction capability enhancing schema to the pure λ-calculus, we obtain a context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{C}$. As a second demonstration, we apply the schema to $\boldsymbol{\lambda}\mathbf{C}$ to yield the twice context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{CC}$, and then to $\boldsymbol{\lambda}\mathbf{CC}$ to yield the thrice context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{CCC}$, and so forth. Interestingly, the twice context-enriched calculus $\boldsymbol{\lambda}\mathbf{CC}$ is the "fixpoint" of the repetitive applications of our schema to the λ-calculus.

Intuitively, the $\boldsymbol{\lambda}\mathbf{C}$-calculus extends the λ-calculus with free identifier abstractions $\Phi\{\mathbf{x}_1:y_1,\ldots,\mathbf{x}_n:y_n\}.e$ to model the compiled code of evolved λ-contexts. The evolved

$\boldsymbol{\lambda}\mathbf{C}$-contexts corresponding to free identifier abstractions are of the form:

$$\boldsymbol{\Phi}\{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}.\mathbf{e}$$

They can be compiled into nested Φ-abstractions of the form

$$\Phi\{\mathbf{z}_1 : z_1, \ldots, \mathbf{z}_m : z_m\}.\Phi\{\mathbf{x}_1 : y_1, \ldots, \mathbf{x}_n : y_n\}.e$$

The machine code e is the image of the source code $\mathbf{e}$. Free occurrences of the identifiers $\mathbf{y}_1, \ldots, \mathbf{y}_n$ in the source code $\mathbf{e}$ are replaced by their respective placeholders $y_1, \ldots, y_n$ in the machine code e. Similarly, the free identifiers $\mathbf{z}_1, \ldots, \mathbf{z}_m$ of $\boldsymbol{\Phi}\{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}.\mathbf{e}$ are compiled into their respective placeholders $z_1, \ldots, z_m$ in the machine code e.

Thus, to obtain the twice context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{CC}$ from the once context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{C}$, we need to introduce only an extra compiled code construction operator to build the compiled code of $\boldsymbol{\Phi}\{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}.\mathbf{e}$ from the compiled code of $\mathbf{e}$. Moreover, such an operator is a primitive involving no free variables, *i.e.*, it is a constant. As a result, further context-enrichment of the $\boldsymbol{\lambda}\mathbf{CC}$-calculus introduces no new compilation mechanisms, thus converging the process to $\boldsymbol{\lambda}\mathbf{CC}$.

The context-enriched version of a λ-calculus is capable of compiling the λ-contexts incrementally, a goal designed into our schema. It is therefore possible to express the compilation of the $\boldsymbol{\lambda}\mathbf{CC}$-contexts in the $\boldsymbol{\lambda}\mathbf{CC}$-calculus. That is, there is a self-compiler [8, 52] for $\boldsymbol{\lambda}\mathbf{CC}$. Furthermore, we can show that the self-compiler is metacircular [45]. That is, each category of evolved $\boldsymbol{\lambda}\mathbf{CC}$-contexts can be translated directly into the machine code of the same category of $\boldsymbol{\lambda}\mathbf{CC}$-terms, no auxiliary notions or mechanisms are necessary.

The context-enriched calculi $\boldsymbol{\lambda}\mathbf{C}$ and $\boldsymbol{\lambda}\mathbf{CC}$ demonstrate that it is indeed possible to conservatively extend the λ-calculus with incremental program construction capabilities. In the second part of this thesis we apply the schema to intricate variable defining and referencing mechanisms to provide the advanced linking capabilities required by popular incremental programming paradigms.

1.6 Context-Enriched Calculus of Definitions

We extend the λ-calculus with definitions d, which are more elaborate variable defining mechanisms [59]:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e\ e \mid let\ d\ in\ e \\ d & ::= & \{x_1 = e, \ldots, x_n = e\} \end{array}$$

The computational behavior of the new λ-terms $let\ d\ in\ e$ can be explained in terms of the following syntactic expansion:

$$let\ \{x_1 = e_1, \ldots, x_n = e_n\}\ in\ e \quad \equiv \quad \begin{cases} (\lambda x_1 \ldots x_n.e)\ e_1 \cdots e_n & \text{if } n > 0 \\ e & \text{otherwise} \end{cases}$$

Hence, definitions merely add a shorthand for expressing some commonly used programming idioms. We then apply our context-enriching schema to the extended calculus.

The contexts of the λ-calculus with definitions are:

$$\begin{array}{rcl} \mathbf{C} & ::= & \mathbf{h} \mid \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C} \mid \mathbf{C}\ \mathbf{C} \mid \mathbf{let\ D\ in\ C} \\ \mathbf{D} & ::= & \mathbf{h} \mid \{\mathbf{x}_1 = \mathbf{C}, \ldots, \mathbf{x}_n = \mathbf{C}\} \end{array}$$

When we see the free identifiers of a $\mathbf{D}$-context $\{\mathbf{x}_1 = \mathbf{C}_1, \ldots, \mathbf{x}_n = \mathbf{C}_n\}$ as import identifiers, the defining identifiers $\mathbf{x}_1, \ldots, \mathbf{x}_n$ become export identifiers. By assigning a new category of compiled code abstractions for evolved $\mathbf{D}$-contexts, we have abstractions that import and export variables through identifiers (external names). They are modules [73]. The context-enriched calculus of definitions $\boldsymbol{\lambda}\mathbf{DD}$ is therefore a module manipulation language. The operation that incrementally constructs the compiled code of the context **let D in C** from the compiled code of the contexts $\mathbf{D}$ and $\mathbf{C}$ is the means to import the module of $\mathbf{D}$ into the compiled code of $\mathbf{C}$. The operation that incrementally constructs the compiled code of the context $\{\mathbf{x}_1 = \mathbf{C}_1, \ldots, \mathbf{x}_n = \mathbf{C}_n\}$ from the compiled code of the contexts $\mathbf{C}_1, \ldots, \mathbf{C}_n$ provides the mechanism for building modules from scratch.

Additional module construction operations can be easily derived from commonly used compound definitions. For instance, the compound definition *seq* d_1 d_2 combines the two definitions d_1 and d_2 such that we have the following syntactic expansion:

$$\textit{let}\ (\textit{seq}\ d_1\ d_2)\ \textit{in}\ e \quad \equiv \quad \textit{let}\ d_1\ \textit{in}\ \textit{let}\ d_2\ \textit{in}\ e$$

That is, the bindings of d_1 are visible to the bindings of d_2; moreover, the bindings of d_2 override those of d_1 in case of conflicts. The corresponding module operator **seq** derived from our context-enriching schema is capable of incrementally constructing the module of $\mathbf{seq}\,\mathbf{D}_1\,\mathbf{D}_2$ from the modules of $\mathbf{D}_1$ and $\mathbf{D}_2$ by linking the import variables of $\mathbf{D}_2$ to the export variables of $\mathbf{D}_1$.

A point worth stressing here again is that the compilation mechanisms introduced by our context-enriching schema are strictly about the incremental construction of programs; they do not interfere with the computational behavior of the underlying calculus. As a result, the addition of new module operators to $\boldsymbol{\lambda}$**DD** can be done in a modular fashion. Indeed, the complexities involved in the process stem mainly from the syntactic expansion of the derived definitions such as *seq* d_1 d_2. The induced module operators actually have relatively straightforward semantic descriptions that incur only simple identification of import and export variables.

1.7 Context-Enriched Calculus of Relinkables

As another exercise, we extend the λ-calculus with relinkable variable references r, which are more elaborate variable referencing mechanisms:

$$\begin{array}{rcl} k & \in & \{1, 2, \ldots\} \\ r & ::= & [x^k, x^k, \ldots] \\ e & ::= & r \mid \lambda x.e \mid e\ e \end{array}$$

A relinkable variable reference $[x_1^{k_1}, x_2^{k_2}, \ldots]$ is a (possibly infinite) sequence of variable references $x_1^{k_1}, x_2^{k_2}, \ldots$, *e.g.*, $[x^2, y^3, z^1]$. Each constituent x^k is a variable reference x

with a lexical address k [9, 10, 14]. It refers to the kth nearest enclosing function parameter named x. Hence, the reference x^2 in $\lambda \underline{x}.\lambda y.\lambda \overline{x}.[x^2]$ refers to the underlined function parameter, not the one overlined. Semantically, a relinkable variable reference $[x_1^{k_1}, x_2^{k_2}, \ldots]$ is equivalent to

```
if $x_1^{k_1}$ is bound then $x_1^{k_1}$
    else if $x_2^{k_2}$ is bound then $x_2^{k_2}$
        $\ddots$
            else $[x_1^{k_1}, x_2^{k_2}, \ldots]$ is unbound
```

If the first constituent $x_1^{k_1}$ is bound, its denotation dominates the others. Otherwise, if the second constituent $x_2^{k_2}$ is bound, its denotation dominates the rest, and so forth. In other words, a relinkable variable reference is a dispatch based on the boundness of its constituents.

A relinkable variable reference is so called since we can relink it to different λ-parameters as its enclosing context grows. For instance, the same relinkable variable reference $[x^2, y^1, x^1, x^3]$ in the following successive terms changes its link to refer to different λ-parameters (the constituent variable reference and the function parameter to which it refers are underlined):

$$\begin{aligned}
\lambda \underline{x}.[x^2, y^1, \underline{x^1}, x^3] \\
\lambda \underline{y}.\lambda x.[x^2, \underline{y^1}, x^1, x^3] \\
\lambda \underline{x}.\lambda y.\lambda x.[\underline{x^2}, y^1, x^1, x^3] \\
\lambda x.\lambda \underline{x}.\lambda y.\lambda x.[\underline{x^2}, y^1, x^1, x^3]
\end{aligned}$$

The semantics of $[x^2, y^1, x^1, x^3]$ is therefore quite sensitive to its surrounding context.

The potential of relinkable references is unleashed when they are enriched with the notion of contexts. The result is a calculus $\boldsymbol{\lambda}$**RR** capable of expressing variable references with adaptive behavior. These are the kind of variable references needed in object-oriented programming and interactive programming.

1.8 Incremental Programming

The module manipulation calculus $\boldsymbol{\lambda}$**DD** and the calculus of relinkable variable references $\boldsymbol{\lambda}$**RR** together are expressive enough to encode the adaptive behavior fundamental to these common incremental programming paradigms in practice: object-oriented programming and interactive programming.

The three most essential features of object-oriented programming are object encapsulation, inheritance, and late binding [31]. Object encapsulation is about information hiding and modularity. Objects interact with one another only through clearly specified import and export interfaces. Object inheritance implements code reuse and code organization. It is the mechanism by which new and enhanced objects can be defined in terms of existing objects. Late binding provides the necessary means for existing objects to adapt to their ever-changing context.

An object is therefore a $\boldsymbol{\lambda}$**DD**-module. The export interface of a module specifies an object's public attributes (an attribute is either a method or an instance variable). The import interface specifies an object's dependencies on other objects. Existing objects are (re)used to construct new objects using the module construction operations of $\boldsymbol{\lambda}$**DD**. Late-bound virtual references essential to object inheritance are relinkable variable references of $\boldsymbol{\lambda}$**RR**. A relinkable variable reference $[x^\infty, \ldots, x^1]$ is a virtual reference whose link is subject to change as many times as necessary. A relinkable variable reference $[x^2, x^1]$ is a virtual reference that can be relinked only once. A relinkable variable reference $[y^2, x^2, y^1, x^1]$ is a virtual reference that can be linked to either of the two attributes denoted by x or y.

Hence, unlike Smalltalk [30], but similar to C++ [68], our object system does not rely on some pseudo-variable named **self** to accomplish intra-object virtual attribute references. Instead, the pseudo-variable is strictly reserved for the purpose of object self-reference. Furthermore, our virtual references can be sealed off individually so that their links are not subject to future modifications. Moreover, virtual references can decide by themselves on which of the many versions of future modifications to take as their permanent denotation. All in all, we are able to show that object-oriented programming can be modeled with compile-time linking, rather than run-time environment lookup or record field selection [2, 5, 22, 23, 24, 32, 37, 51, 57, 60, 74].

Lisp [66] and its dialects such as Scheme [4, 21] employ an interactive evaluator as a means for incremental program development. An interactive evaluator uses an ever-growing environment to keep track of the bindings produced by previously evaluated **define**-expressions. Each **define**-expression submitted to the evaluator is evaluated in the scope of the current interactive environment to yield a binding. The binding is then used to extend or override the existing interactive environment to yield a new environment for the evaluation of the next expression. The interactive environment thus constitutes an incrementally constructed program.

Traditionally, the adaptive behavior of an interactive system is explained in terms of side effects [21] or dynamic binding [4]. We are able to describe the adaptive nature of interactive programming as yet another form of incremental program construction based on relinkable variable references. The ever-growing environment of an interactive evaluator is a module whose exports are the previously evaluated **define**-expressions and whose imports are free variables expressed as relinkable variable references. The binding produced by each **define**-expression is a modifier used to transform the current interactive environment module to a new environment module.

1.9 Finale

We have presented a schema for enhancing programming languages with incremental program construction capabilities and demonstrated its usefulness. It is appropriate to summarize key contributions, to compare with other work, and to plot future plans.

Programming language design is about the abstraction of recurring programming idioms. The expressiveness of a programming language is tied to the idioms it can fluently express. In this sense, our context-enriched λ-calculi are highly desirable because

- They cover the basics of many prominent programming paradigms.
- Their linking mechanisms involve only the copying of code and the renaming of variables.
- Their compilation mechanisms are orthogonal to the computational devices of the enhanced calculus.
- They are derived from a modular language design methodology.

There are many issues concerning our context-enhancing schema left unexplored. Among them are:

typing The typing of object-oriented programming languages is currently a very active research area [2, 32, 51, 57]. Some of the crucial notions underlying the programming paradigm are shared by our context-enriched calculi. It is therefore tempting to investigate if our schema simplifies or complicates typing issues.

implementation Efficient implementation should not be the deciding factor of the practicality of our schema, but a stronger case can be presented if we have one. (Build it and they will come.) Many of the clever implementation techniques employed in some popular functional [17, 27, 56, 62] and object-oriented languages [20, 25, 26] may be applicable to our context-enriched calculi.

The above issues, as well as related work, are discussed in Chapter 9. It is now time to start filling in the holes.

Chapter 2

Lambda Calculus

The λ-calculus has made an indelible imprint on the study of programming languages. As a mathematical notion, it is used for defining programming language semantics. A case in point is the denotational approach of Scott and Strachey [48, 61, 67]. As a computational notion, the λ-calculus is Turing-complete; every computable function is definable in the λ-calculus [38]. As a programming notion, its features form the core of many contemporary high-level programming languages [21, 49]. One of the reasons that make λ-calculus the quintessential representative used in conveying new programming notions is its concise and expressive syntax. In our case, the λ-calculus also happens to exhibit all the obstacles to incremental program construction that are inherent to statically-scoped languages.

There are many variants of the untyped λ-calculus. Each of them possesses particular idiosyncrasies inherent to the type of computation it is intended to model. Examples are λ-calculi that employ call-by-name or call-by-value evaluation strategies [58], λ-calculi that include constants and sophisticated datatypes [35], λ-calculi that embody imperative features such as assignments [29, 53] and continuations [29]. Our schema applies to them all. Of concern to us is how to construct programs incrementally. The way computation is actually modeled has little impact on our schema. We therefore choose to illustrate our schema on the λ-calculus that has the least complicated description—the call-by-name untyped pure λ-calculus. For an in-depth treatment of the λ-calculus, the reader is advised to consult the literature [7, 33, 34]. In this chapter we briefly overview only its basic concepts and notations that are relevant to our work.

2.1 Term Language

The syntax of the pure λ-calculus is constructed from the following alphabet:

$x, y, z, \ldots \in \mathit{Vars}$	variable names
λ	lambda
$.$	dot
$(,)$	parentheses

The set of λ-terms Λ is defined inductively as follows:

variable: if $x \in \mathit{Vars}$ then $x \in \Lambda$;

abstraction: if $x \in \mathit{Vars}$ and $e \in \Lambda$ then $(\lambda x.e) \in \Lambda$;

application if $e_1 \in \Lambda$ and $e_2 \in \Lambda$ then $(e_1\ e_2) \in \Lambda$.

A common shorthand for the above inductive definition is the following abstract syntax:

$$\begin{array}{rcl} e & \in & \Lambda \\ e & ::= & x \mid (\lambda x.e) \mid (e\ e) \end{array} \qquad (\Lambda)$$

Intuitively, the three categories of λ-term encompass these programming notions:

statically-scoped function: An abstraction $(\lambda x.e)$ is a *statically-scoped* function. The variable x names the function's parameter. The term e is the function's body. The scope of the parameter x consists of the body e. Free occurrences of the variable x in e refer to the function's parameter.

function parameter reference: A variable reference x is a name used in the body of a function to refer to the function's parameter. A variable name used as a function parameter is called a *binding* occurrence; it is called an *applied* occurrence when used as a reference to a function parameter.

function application: An application $(e_1\ e_2)$ denotes function application, the computational engine of the λ-calculus. The term e_1 is the function part of the application. The term e_2 is the argument of the application.

To avoid the proliferation of parentheses, abstractions $(\lambda x.e)$ and applications $(e_1\ e_2)$ are generally written without their parentheses, namely, $\lambda x.e$ and $e_1\ e_2$. Application is left associative. Hence, $e_1\ e_2\ e_3$ is equivalent to the fully-parenthesized $((e_1\ e_2)\ e_3)$. Abstraction associates to the right. Hence, $\lambda x.\lambda y.e$ is $(\lambda x.(\lambda y.e))$ when fully parenthesized. Moreover, internal "λ"s and "."s of nested abstractions are suppressed. Hence, $\lambda xy.e$ is a shorthand for $\lambda x.\lambda y.e$, which is $(\lambda x.(\lambda y.e))$. The notation $e_1 \equiv e_2$ means that e_1 and e_2 are syntactically identical.

2.2 Term Equality Framework

The λ-calculi are theories of equality (convertibility) between λ-terms. In this section we give a general framework for defining such term equality relations, which is used later on to instantiate the λ-calculi of interest to us.

We first present the notion of *one-hole contexts.* They denote incomplete λ-terms. The one-hole contexts of the λ-calculus are constructed from the same alphabet as the λ-terms except for the addition of a hole denoted as $[]$. The set of one-hole contexts $\Lambda[]$ is defined by the following abstract syntax:

$$\begin{array}{rcl} C[] & \in & \Lambda[] \\ e & \in & \Lambda \\ C[] & ::= & [] \mid \lambda x.C[] \mid C[]\ e \mid e\ C[] \end{array} \qquad (\Lambda[])$$

Each one-hole context $C[]$ is a λ-term that has a hole $[]$ in it. The notation $C[e]$ denotes the λ-term resulting from replacing the hole of $C[]$ with the λ-term e. For instance, $\lambda x.[\lambda y.(x\ y)]$ is the λ-term $\lambda xy.(x\ y)$. One-hole contexts provide a convenient way for us to isolate a particular subterm e' out of a λ-term e, $e \equiv C[e']$.

2.2.1 Notion of Reduction

A *notion of reduction* (*reduction rule*) R on λ-terms is a binary relation on Λ:

$$R \subseteq \Lambda \times \Lambda$$

For each $(e_1, e_2) \in R$, e_1 is called an *R-redex*, e_2 is the *R-contractum* of e_1, and the reduction from e_1 to e_2 is an *R-contraction step.*

A notion of reduction R is *compatible* with the syntactic construction of λ-terms if $(e_1, e_2) \in R$ implies that $(C[e_1], C[e_2]) \in R$ for any one-hole context $C[]$. That is, two terms e_1 and e_2 are R-related if they differ only on a particular subterm and the different subterms are R-related. Since the λ-terms Λ are constructed inductively, every notion of reduction R induces a compatible closure. That is, there is always a smallest extension of R that is compatible. Formally, the *compatible closure* of R, denoted as $\rightarrow_R$, is a binary relation on Λ defined inductively as follows:

$$\begin{array}{rcl} (e_1, e_2) \in R & \Rightarrow & e_1 \rightarrow_R e_2 \\ e_1 \rightarrow_R e_2 & \Rightarrow & e_1\ e \rightarrow_R e_2\ e \\ e_1 \rightarrow_R e_2 & \Rightarrow & e\ e_1 \rightarrow_R e\ e_2 \\ e_1 \rightarrow_R e_2 & \Rightarrow & \lambda x.e_1 \rightarrow_R \lambda x.e_2 \end{array}$$

The compatible closure of R is also called the *one-step R-reduction.* We say that e_1 one-step R-reduces to e_2 if $e_1 \rightarrow_R e_2$.

The reflexive and transitive closure of $\rightarrow_R$ is the *R-reduction* relation $\twoheadrightarrow_R$:

$$\begin{array}{rrcl} & e_1 \rightarrow_R e_2 & \Rightarrow & e_1 \twoheadrightarrow_R e_2 \\ \text{reflexivity:} & & & e \twoheadrightarrow_R e \\ \text{transitivity:} & e_1 \twoheadrightarrow_R e_2, e_2 \twoheadrightarrow_R e_3 & \Rightarrow & e_1 \twoheadrightarrow_R e_3 \end{array}$$

In the case of $e_1 \twoheadrightarrow_R e_2$, we say that e_1 R-reduces to e_2 or e_2 is an R-reduct of e_1.

The least equivalence relation generated by $\twoheadrightarrow_R$ is the *R-equality* (*R-convertibility*) relation $=_R$:

$$\begin{array}{rrcl} & e_1 \twoheadrightarrow_R e_2 & \Rightarrow & e_1 =_R e_2 \\ \text{reflexivity:} & & & e =_R e \\ \text{symmetry:} & e_1 =_R e_2 & \Rightarrow & e_2 =_R e_1 \\ \text{transitivity:} & e_1 =_R e_2, e_2 =_R e_3 & \Rightarrow & e_1 =_R e_3 \end{array}$$

It is the theory of equality on the λ-terms Λ induced by the notion of reduction R. We say that e_1 is R-convertible (R-equivalent) to e_2 if $e_1 =_R e_2$.

A term is called an *R-normal form* if it does not contain as subterm an R-redex. That is, e is an R-normal form if we cannot partition it into a one-hole context $C[]$ and an R-redex subterm e' such that $e \equiv C[e']$. We say that a term e_1 has an R-normal form e_2 if e_2 is an R-normal form and e_1 is R-equivalent to e_2.

Computationally speaking, an R-redex e_1 is an instruction of a reduction machine and the contraction of e_1 to its contractum e_2 amounts to the execution of the machine instruction e_1 to yield the result e_2. An R-normal form is a program that has no more instructions to be executed. It is an *answer*. The compatible closure of R says that each R-instruction in a program can be *autonomously* executed regardless of its surrounding program context. The R-reduction $\twoheadrightarrow_R$ computes the answer of programs. The R-convertibility says that two programs are equivalent if they compute to the same answer.

2.2.2 Church-Rosser Property

Since the one-step R-reduction relation $\rightarrow_R$ underlying the R-equality relation $=_R$ is not obligated to contract any particular R-redex of a term, a term is reducible to possibly many different terms when it has several R-redexes as subterms. It is thus essential for the equality relation $=_R$ to be computationally well-behaved in the sense that regardless of which redex we choose to execute in each reduction step, if a program has an answer, it is always the same one. This is formalized below.

A binary relation $\succ$ on Λ satisfies the *diamond property*, notated as $\succ \models \diamond$, if for all λ-terms e, e_1, and e_2 such that $e \succ e_1$ and $e \succ e_2$, there exists a λ-term e_3 such that $e_1 \succ e_3$ and $e_2 \succ e_3$. A notion of reduction R is said to be *Church-Rosser* if its induced R-reduction relation $\twoheadrightarrow_R$ satisfies the diamond property. In other words, if e R-reduces to two different terms e_1 and e_2, then e_1 and e_2 have a common R-reduct e_3. Two immediate consequences of R being Church-Rosser are:

- a term can have at most one R-normal form; hence, if a program has an answer, the answer is unique,
- if a program has an answer, *i.e.*, it is R-equivalent to an R-normal form, then it is possible to R-reduce the program to its answer.

Given an existing reduction machine whose instructions are modeled by some notion of reduction R_1, in many cases we can obtain a new reduction machine by adding instructions represented by a second notion of reduction R_2. The collective notion of reduction R for the new machine is then the union of R_1 and R_2:

$$\begin{aligned} R &= R_1 \cup R_2 \\ &= \{(e_1, e_2) \mid (e_1, e_2) \in R_1 \text{ or } (e_1, e_2) \in R_2\} \end{aligned}$$

To ensure that the new machine remains in good computational behavior, we must show that the collective notion of reduction R is Church-Rosser. A modular approach to such a proof is attributed to Hindley and Rosen:

Lemma 2.1 (Hindley-Rosen) *Let $\succ_1$ and $\succ_2$ be binary relations on some set X and let $\succ^+$ be the transitive closure of the union of $\succ_1$ and $\succ_2$. Suppose that*

- *Both relations $\succ_1$ and $\succ_2$ satisfy the diamond property individually.*
- *The relations $\succ_1$ and $\succ_2$ commute with each other, i.e., for all elements x, x_1, and x_2 of X, if $x \succ_1 x_1$ and $x \succ_2 x_2$, then there exists an element x_3 of X such that $x_1 \succ_2 x_3$ and $x_2 \succ_1 x_3$.*

Then, the binary relation $\succ^+$ satisfies the diamond property.

So, to prove that the collective notion of reduction R of the new machine is Church-Rosser, we need only show that:

- each of the two notions of reduction R_1 and R_2 is Church-Rosser, *i.e.*, $\twoheadrightarrow_{R_1} \models \diamond$ and $\twoheadrightarrow_{R_2} \models \diamond$, and
- the reduction relations $\twoheadrightarrow_{R_1}$ and $\twoheadrightarrow_{R_2}$ commute with each other.

The lemma is particularly useful since we already know that the notion of reduction R_1 of the existing machine is Church-Rosser. We thus need only show that the new notion of reduction R_2 is Church-Rosser and that the execution of the instructions of R_2 do not interfere with the execution of the instructions of R_1 to meet the commutativity requirement.

We are now ready to introduce the two fundamental λ-term equality relations of the λ-calculus: α- and β-convertibilities. The former captures the static scope nature of function parameters. The latter provides a vehicle for computing with functions. These λ-term equivalence relations are the subject of the next two sections.

2.3 Alpha Convertibility

The functions modeled by λ-calculus abstractions are statically scoped. That means the linking relation between a function's parameter and its associated references cannot be altered under any circumstances. The parameter name is used merely as a visual aid to express the linking relation. Consequently, we can change the parameter's name without changing the meaning of the function, as long as we also rename all its associated references accordingly. Thus, $\lambda x.\lambda y.(x\ y)$ and $\lambda w.\lambda y.(w\ y)$, which is the result of changing the parameter name x of $\lambda x.\lambda y.(x\ y)$ to w, denote the same function. In contrast, $\lambda y.\lambda y.(y\ y)$ does not result from renaming the parameter x of $\lambda x.\lambda y.(x\ y)$ to y since that would change the linking relation of parameters x and y.

A basic syntactic notion needed in the formal description of function parameter renaming is the set of free variables occurring in a term e, denoted as $fv(e)$. Intuitively, a variable reference x is *free* in a term e only if it does not refer to some function parameter. Thus, the underlined occurrence of x in the application term $(\lambda x.\overline{x})\ \underline{x}$ is free while the overlined occurrence is not. Formally, the set $fv(e)$ is defined inductively on the structure of e as follows:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(\lambda x.e) &= fv(e) \setminus \{x\} \\ fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) \end{aligned} \tag{fv}$$

The abstraction clause enforces static scope for function parameters. Free occurrences of x in the body e of a function $\lambda x.e$ refer to the function's parameter. They are not free beyond the function, hence the removal of x from the set $fv(e)$.

The fact that the actual name of a function's parameter is irrelevant to the function's behavior is expressed in the λ-calculus by the following notion of reduction known as *α-conversion* (*parameter renaming*):

$$\begin{aligned} \lambda x.e &\rightarrow \lambda y.\langle y/x\rangle e \\ &\quad \text{where } y \not\equiv x \text{ and } y \notin fv(e) \end{aligned} \tag{α}$$

which is an alternative expression of the notion of reduction α using the usual set notation:

$$\alpha = \{(\lambda x.e, \lambda y.\langle y/x\rangle e) \mid y \not\equiv x, y \notin fv(e)\}$$

It says that we can change the parameter's name x of a function $\lambda x.e$ to a new name y, as long as we also change the free references of x in e to y, which is what the notation $\langle y/x\rangle e$ stands for. There are two provisions in choosing the new name y. First, y is not x, the original name of the parameter; otherwise, the name change would be vacuous. Second, y is not the name of any of the free variables occurring in the body term e. It ensures that no linking relation between a function parameter and its references will be altered by the change of the parameter's name. Had $y \in fv(e)$, the free occurrences of y in e would have been linked to the renamed parameter. In essence, only the name used by a function's parameter and its references is changed by α-conversion, but not the linking relation.

The α-substitution meta-operation $\langle y/x\rangle e$ used in function parameter renaming replaces every free occurrence of x in e with the new name y:

$$\begin{array}{rcl}
\langle y/x\rangle z & \equiv & \begin{cases} y & \text{if } x \equiv z \\ z & \text{otherwise} \end{cases} \\
\langle y/x\rangle \lambda z.e & \equiv & \begin{cases} \lambda z.e & \text{if } x \equiv z \\ \lambda z.\langle y/x\rangle e & \text{if } x \not\equiv z \text{ and } z \not\equiv y \\ \lambda w.\langle y/x\rangle\langle w/z\rangle e & \text{if } x \not\equiv z \text{ but } z \equiv y \end{cases} \\
\langle y/x\rangle(e_1\; e_2) & \equiv & \langle y/x\rangle e_1\; \langle y/x\rangle e_2
\end{array} \qquad (\langle\cdot/x\rangle\cdot)$$

The abstraction clause is the most complex. There are three cases involving three variables, namely, the variable to be replaced x, the replacement y, and the function's parameter z. When z is x, x is not a free variable in $\lambda z.e$; hence, the renaming has no effect on $\lambda z.e$. When z is neither x nor y, the parameter z does not interfere with the renaming process. The α-substitution can therefore be carried into the body e without bothering the parameter. The last case, when z is not x but is the same as y, requires special attention. Renaming free occurrences of x in e to y naively would amount to linking the free x's in e to the parameter z, a flagrant violation of static scope. Consequently, α-conversion is used inductively first to rename the parameter z to a fresh variable w to avoid inadvertent capture of y. The freshness of w can be guaranteed by choosing it to be different from x and z, and not to be one of the free variables of e.

According to the λ-term convertibility framework described in Section 2.2, the notion of reduction α induces a term equivalence relation $=_\alpha$ that effectively says that two terms are α-equivalent if one can be made syntactically identical to the other by the renaming of its function parameters. But since the primary motivation behind the λ-calculus is the modeling of the behavior of functions, not their cosmetic appearances, such terms are identified because they actually denote the same program. Thus, $e_1 \equiv e_2$ is generalized to mean either that e_1 and e_2 are syntactically identical or that they are identical up to the renaming of some function parameters. This is the *α-congruence* variable convention adopted widely in the literature.

There is another reason to adopt the α-congruence variable convention. It has to do with the notion of normal forms, which are answers produced by the execution of programs. Since renaming is applicable to every function parameter, a term containing at least an abstraction as a subterm can never be an α-normal form and therefore technically cannot play the role of an answer. By identifying terms that differ only in their function parameter names, the problem disappears.

A second commonly used variable convention is the *hygiene* variable convention. It says that in a mathematical context, *e.g.*, a definition or a theorem, the function parameters of the terms involved are distinct from one another and are distinct from the free variables of the terms involved. Assumption of the hygiene convention greatly simplifies the presentation of many computational concepts and mechanisms expressed in terms of the λ-calculus. It is used throughout most of this thesis. There are exceptional cases in which we must identify some parameters, however. The reader will be alerted on such occasions.

2.4 Beta Convertibility

The second equivalence relation on λ-terms is induced by the following notion of reduction:

$$(\lambda x.e)\ e' \quad \rightarrow \quad [e'/x]e \qquad (\beta)$$

which is another way of expressing:

$$\beta \quad = \quad \{((\lambda x.e)\ e', [e'/x]e) \mid e, e' \in \Lambda\}$$

It invokes the function represented by the abstraction $\lambda x.e$ on the argument represented by the term e'. The intended behavior is for the parameter x to assume the argument e' while computing the answer of the function body e. This is modeled by substituting each free occurrence of x in e with e', denoted as $[e'/x]e$.

The β-substitution meta-operation $[e'/x]e$ is defined by induction on the structure of e as follows:

$$
\begin{aligned}
[e'/x]z &\equiv \begin{cases} e' & \text{if } x \equiv z \\ z & \text{otherwise} \end{cases} \\
[e'/x]\lambda z.e &\equiv \begin{cases} \lambda z.e & \text{if } x \equiv z \\ \lambda z.[e'/x]e & \text{if } x \not\equiv z \text{ and } z \notin \mathit{fv}(e') \\ \lambda w.[e'/x]\langle w/z\rangle e & \text{if } x \not\equiv z \text{ but } z \in \mathit{fv}(e') \end{cases} \qquad ([\cdot/x]\cdot) \\
[e'/x](e_1\ e_2) &\equiv [e'/x]e_1\ [e'/x]e_2
\end{aligned}
$$

In the abstraction clause, α-conversion is used to avoid inadvertent capture when the parameter z is a also a free variable of e'. The new name w is not the same as x or z, nor is it a free variable of e or e'. It is chosen as such to maintain the static linking relation of the function parameter. The abstraction clause can be simplified to

$$[e'/x]\lambda z.e \quad \equiv \quad \lambda z.[e'/x]e$$

when the variable conventions are assumed to ensure that the function parameter z is not the same as the to-be-replaced variable x nor any of the free variables of the argument term e', the replacement for x.

As we can see from the definition of the β-substitution above, α-conversion is needed to avoid inadvertent variable capture. It is not essential to the understanding of functional programming, however. Indeed, by tinkering with the representation of λ-terms and the definition of the β-reduction rule, the λ-calculus can be framed in a nameless setting and therefore rendering α-conversion unnecessary [1, 9, 10, 14]. We are obliged to discuss it in detail since our incremental compiled code construction operations rely exclusively on variable renaming to model the linking of separately-developed program components.

We should point out the similarity between the two substitution meta-operations $\langle y/x\rangle e$ and $[y/x]e$. In most treatments of the λ-calculus, the former is viewed as a special case of the latter. This is really an oversight. They model two very distinct notions about functions, namely, statically-scoped function parameters and function invocation. Hence, even though the syntactic description of α-substitution is a degenerate case of β-substitution, they are each given a distinct notation.

2.5 Lambda Calculus

The λ-calculus of interest is the term equality relation generated by the notion of reduction β. Thus, $e_1 \rightarrow_\beta e_2$ denotes the one-step β-reduction of e_1 to e_2, $e_1 \twoheadrightarrow_\beta e_2$ is the β-reduction of e_1 to e_2, and $e_1 =_\beta e_2$ means e_1 is β-equivalent to e_2.

Computationally, we can perceive the λ-calculus as a β-reduction machine. The λ-terms are machine code and the β-redexes are machine instructions. The execution of a machine instruction $(\lambda x.e)\ e'$ yields the result $[e'/x]e$. The execution of a program then continuously replaces each β-redex with its contractum. There are two possible outcomes. In one case there is no more β-redexes left to contract. The result is a β-normal form that is the answer of the program. In the other case there is always some β-redexes available for further contraction. The computation thus produces no answer.

Clearly, for the above intuition to hold, we must have the assurance that the notion of reduction β is Church-Rosser, *i.e.*, two β-reductions starting from the same term are confluent. This is indeed the case as is supported by the following theorem:

Theorem 2.2 (Church-Rosser) *The reduction relation $\twoheadrightarrow_\beta$ satisfies the diamond property, i.e., if $e \twoheadrightarrow_\beta e_1$ and $e \twoheadrightarrow_\beta e_2$, then there is a λ-term e_3 such that $e_1 \twoheadrightarrow_\beta e_3$ and $e_2 \twoheadrightarrow_\beta e_3$.*

There are many ways to prove the theorem. We will not repeat them here but refer the interested reader to the literature [7, 69].

2.6 Programming with Lambda Calculus

Programming with the λ-calculus is programming with functions. The computational power of the λ-calculus comes from its ability to represent datatypes as functions and to express iterations using recursive functions.

2.6.1 Datatypes as Functions

Basic datatypes such as integers and booleans can be encoded in the λ-calculus as special functions. To illustrate, the boolean values of true and false are often coded as the following λ-terms:

$$\begin{array}{rcl} \text{true}: & T & \equiv & \lambda x.\lambda y.x \\ \text{false}: & F & \equiv & \lambda x.\lambda y.y \end{array}$$

The conditional construct *if* e_1 *then* e_2 *else* e_3 found in most languages may then be defined as:

$$\textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3 \quad \equiv \quad e_1\ e_2\ e_3$$

Particularly, the boolean negation operator *not* is λ-definable as follows:

$$\begin{array}{rcl} \textit{not} & \equiv & \lambda x.(\textit{if } x \textit{ then } F \textit{ else } T) \\ & \equiv & \lambda x.(x\ F\ T) \end{array}$$

with the following expected behavior (where we have underlined the β-redex(es) contracted in each step):

$$\begin{array}{rcl} \textit{not } T & \equiv & \underline{(\lambda x.(x\ F\ T))\ T} \\ & \rightarrow_\beta & T\ F\ T \\ & \equiv & \underline{(\lambda x.\lambda y.x)\ F\ T} \\ & \twoheadrightarrow_\beta & F \end{array} \qquad \begin{array}{rcl} \textit{not } F & \equiv & \underline{(\lambda x.(x\ F\ T))\ F} \\ & \rightarrow_\beta & F\ F\ T \\ & \equiv & \underline{(\lambda x.\lambda y.y)\ F\ T} \\ & \twoheadrightarrow_\beta & T \end{array}$$

Compound datatypes such as tuples, lists (sequences), and records can also be represented as functions. For instance, n-tuples and their selectors are λ-definable as follows:

$$\begin{array}{rl} n\text{-tuple } \langle e_1, \ldots, e_n \rangle : & \lambda s.(s\ e_1 \cdots e_n) \\ & \qquad \text{where } s \notin \mathit{fv}(e_1) \cup \cdots \cup \mathit{fv}(e_n) \\ i\text{th element selector } \pi_i^n : & \lambda t.(t\ \lambda x_1 \cdots x_n.x_i) \end{array}$$

such that

$$\pi_i^n\ \langle e_1, \ldots, e_n \rangle \quad \equiv \quad \underline{(\lambda t.(t\ \lambda x_1 \cdots x_n.x_i))\ \lambda s.(s\ e_1 \cdots e_n)}$$

$$\begin{array}{ll}
\to_\beta & \underline{(\lambda s.(s\ e_1 \cdots e_n))\ \lambda x_1 \cdots x_n.x_i} \\
\to_\beta & \underline{(\lambda x_1 \cdots x_n.x_i)\ e_1 \cdots e_n} \\
\twoheadrightarrow_\beta & e_i
\end{array}$$

2.6.2 Applied Lambda Calculi

The above illustrations use clever coding tricks. An alternative is to augment the λ-calculus with basic constants and datatype constructors along with their associated notions of reduction (these notions of reduction are known as δ-rules). Such extensions of the λ-calculus are often called *applied λ-calculi* [35].

For instance, we may extend the term language of the λ-calculus to include integers and arithmetic operations (we use infix notation for arithmetic terms):

$$\begin{array}{lll}
k & \in & \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
e & ::= & x \mid \lambda x.e \mid e\ e \mid k \mid e + e \mid e - e \mid e * e \mid \cdots
\end{array}$$

and then add the following necessary notions of reduction to the extended calculus:

$$\begin{array}{rcll}
k_1 + k_2 & \to & k & \text{where } k = k_1 + k_2 \qquad (+) \\
k_1 - k_2 & \to & k & \text{where } k = k_1 - k_2 \qquad (-) \\
k_1 * k_2 & \to & k & \text{where } k = k_1 * k_2 \qquad (*) \\
& \cdots & &
\end{array}$$

The δ-rules need not operate only on constants, which is the case for the example above, they can be defined on λ-terms in general. For instance, we may add n-tuples $\langle e_1, \ldots, e_n \rangle$ and their selectors π_i^n to the term language of the λ-calculus:

$$e \ ::= \ x \mid \lambda x.e \mid e\ e \mid \langle e_1, \ldots, e_n \rangle \mid \pi_i^n$$

Of course, we should also extend the λ-calculus with the following notion of reduction:

$$\pi_i^n\ \langle e_1, \ldots, e_n \rangle \ \to \ e_i \qquad (\pi_i^n)$$

In any case, care must be taken to preserve the Church-Rosser property when new δ-rules are added to the λ-calculus [16, 39]. According to the Hindley-Rosen Lemma, the added δ-rules must be shown to be Church-Rosser; moreover, they must commute with the existing β-rule.

In the thesis, we demonstrate our context-enriching schema only on the pure λ-calculi, but will use the applied λ-calculi in examples. The addition of constants has no impact on the applicability or complexity of the schema. Intuitively, constants have no free variables, we can therefore isolate them from the subject of linking (variable capture).

2.6.3 Recursive Functions

Recursive functions are defined in the λ-calculus using fixpoint combinators (a combinator is a λ-term that has no free variables). A typical example is the Y combinator defined below:

$$Y \quad \equiv \quad \lambda f.((\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x))))$$

It has the unusual property that for any function G, one has:

$$\begin{array}{rcl} Y\ G & \equiv & \underline{(\lambda f.((\lambda x.(f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))))\ G} \\ & \rightarrow_\beta & \underline{(\lambda x.(G\ (x\ x)))\ (\lambda x.(G\ (x\ x)))} \\ & \rightarrow_\beta & G\ ((\lambda x.(G\ (x\ x)))\ (\lambda x.(G\ (x\ x)))) \\ & =_\beta & G\ (Y\ G) \end{array}$$

Thus, to define a recursive function

$$f \quad = \quad \lambda x.(\cdots f \cdots)$$

we only need to form the function

$$G \quad \equiv \quad \lambda f.\lambda x.(\cdots f \cdots)$$

and submit it to the Y combinator:

$$\begin{aligned}
f &=_\beta Y\ G \\
&=_\beta G\ (Y\ G) \\
&\equiv \underline{(\lambda f.\lambda x.(\cdots f \cdots))\ (Y\ G)} \\
&\rightarrow_\beta \lambda x.(\cdots (Y\ G) \cdots) \\
&=_\beta \lambda x.(\cdots f \cdots)
\end{aligned}$$

Following tradition, we use *fix* to denote collectively all fixpoint combinators that depict the necessary behavior shown above.

As an example, the following is a recursive function *fact* that computes $n!$:

$$\begin{aligned}
\mathit{fact} &\equiv \mathit{fix}\,\lambda f.\lambda n.(\mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n * (f\ (n-1))) \\
&=_\beta \lambda n.(\mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n * (\mathit{fact}\,(n-1)))
\end{aligned}$$

To illustrate,

$$\begin{aligned}
\mathit{fact}\,3 &\equiv (\lambda n.(\mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n * (\mathit{fact}\,(n-1))))\ 3 \\
&\twoheadrightarrow \mathit{if}\ 3 = 0\ \mathit{then}\ 1\ \mathit{else}\ 3 * (\mathit{fact}\,(3-1)) \\
&\twoheadrightarrow 3 * (\mathit{fact}\,2) \\
&\twoheadrightarrow 3 * (\mathit{if}\ 2 = 0\ \mathit{then}\ 1\ \mathit{else}\ 2 * (\mathit{fact}\,(2-1))) \\
&\twoheadrightarrow 3 * (2 * (\mathit{fact}\,1)) \\
&\twoheadrightarrow 3 * (2 * (\mathit{if}\ 1 = 0\ \mathit{then}\ 1\ \mathit{else}\ 1 * (\mathit{fact}\,(1-1)))) \\
&\twoheadrightarrow 3 * (2 * (1 * (\mathit{fact}\,0))) \\
&\twoheadrightarrow 3 * (2 * (1 * (\mathit{if}\ 0 = 0\ \mathit{then}\ 1\ \mathit{else}\ 0 * (\mathit{fact}\,(0-1))))) \\
&\twoheadrightarrow 3 * (2 * (1 * 1)) \\
&\twoheadrightarrow 6
\end{aligned}$$

2.7 Contexts

Programming with the λ-calculus constitutes a limited form of incremental program construction. Simple functions can be composed to form sophisticated functions. There is no machinery built into the λ-calculus to directly model the more general form of incremental program construction that we are looking for; that would require the capacity to link free variables. Indeed, a function L_x that takes a term e and produces the term $\lambda x.e$ such that free occurrences of x in the given e are linked to the parameter x of the newly constructed abstraction is not definable in the λ-calculus. That would require variable capture and the existence of L_x would render the λ-calculus inconsistent. In particular, we would be able to prove that the two boolean values $T \equiv \lambda x.\lambda y.x$ and $F \equiv \lambda x.\lambda y.y$ are equivalent!

Here is one way the contradictory conclusion can be reached. According to the α-congruence variable convention, we have the following α-equivalent representations of the same function:

$$F_1 \;\equiv\; \lambda f.\lambda x.(f\ x) \;\equiv\; \lambda f.\lambda y.(f\ y) \;\equiv\; F_2$$

Hence, we should have

$$F_1\ L_x \;=_\beta\; F_2\ L_x$$

But that would mean $T =_\beta F$ because

$$\begin{array}{lcl@{\qquad\qquad}lcl}
F_1\ L_x & \equiv & (\lambda f.\lambda x.(f\ x))\ L_x & F_2\ L_x & \equiv & (\lambda f.\lambda y.(f\ y))\ L_x \\
 & \rightarrow_\beta & \lambda x.(L_x\ x) & & \rightarrow_\beta & \lambda y.(L_x\ y) \\
 & =_\beta & \lambda x.\lambda x.x & & =_\beta & \lambda y.\lambda x.y \\
 & \equiv & F & & \equiv & T
\end{array}$$

Consequently, we would be able to show that all λ-terms are equivalent because for any two terms e_1 and e_2, we have

$$\begin{array}{rcl}
T =_\beta F & \Rightarrow & (T\ e_1\ e_2) =_\beta (F\ e_1\ e_2) \\
 & \Rightarrow & e_1 =_\beta e_2
\end{array}$$

In other words, all programs are equal, thus rendering the λ-calculus useless as a reasoning system.

To model linking, we turn to the notion of contexts [7]. Recall the one-hole contexts $C[]$ introduced earlier in Section 2.2 to characterize the compatible closure of notions of reduction. A one-hole context $C[]$ is a λ-term with a single hole $[]$ in it. The λ-term generated by filling the hole of a one-hole context $C[]$ with a term e is written as $C[e]$. It replaces the hole of $C[]$ with e. For instance, filling the one-hole context $\lambda x.[]$ with the term $\lambda y.(x\ y)$ yields the λ-term $\lambda xy.(x\ y)$. Notice that the free variable x of the filler term $\lambda y.(x\ y)$ is no longer free in the result. This variable capture feature of hole filling is the basis of our schema.

It is clear that context hole filling cannot be an operation on λ-terms; otherwise, we would be able to define the function L_x alluded to earlier. Not surprisingly, contexts remain strictly as a meta-notion in the study of the λ-calculus. It is our goal to incorporate the linking capability of hole filling as a programming notion into languages.

In this thesis we deal with contexts with multiple holes each of which can have multiple occurrences. The syntax of the contexts of the λ-calculus are constructed from the following alphabet:

$\mathbf{h} \in \mathit{Holes} = \{\mathbf{h}_1, \mathbf{h}_2, \ldots\}$	holes
$\mathbf{x}, \mathbf{y}, \mathbf{z}, \ldots \in \mathit{Idents}$	identifier names
$\boldsymbol{\lambda}$	lambda
$.$	dot
$(,)$	parentheses

The set of λ-contexts $\mathbf{\Lambda}$ has the following abstract syntax:

$$\begin{array}{rcl} \mathbf{C} & \in & \mathbf{\Lambda} \\ \mathbf{C} & ::= & \mathbf{h} \mid \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C} \mid \mathbf{C}\ \mathbf{C} \end{array}$$

Although λ-terms e and λ-contexts $\mathbf{C}$ are built from the same syntactic structures, they are based on distinct categories of names: λ-terms use variables x while λ-

contexts employ identifiers $\mathbf{x}$. We thus typeset λ-contexts in boldface to distinguish them from λ-terms.

Following our intuition that λ-terms are machine code, contexts can be perceived as source code. Consequently, whereas variables are machine locations, identifiers are source program symbols. Technically speaking, λ-contexts represent potentially incomplete source code—incomplete in the sense that their parse trees may have non-terminal leaves designated by holes $\mathbf{h}$.

The basic operation on λ-contexts that provides the sought-after linking capabilities is the *hole filling* meta-operation $[\mathbf{C}'/\mathbf{h}]\mathbf{C}$. It *literally* substitutes the λ-context $\mathbf{C}'$ for occurrences of the hole $\mathbf{h}$ in $\mathbf{C}$, thus amounting to replacing each non-terminal leaf designated by the hole $\mathbf{h}$ in the parse tree $\mathbf{C}$ with the parse tree $\mathbf{C}'$. Hole filling $[\mathbf{C}'/\mathbf{h}]\mathbf{C}$ is defined inductively on the structure of $\mathbf{C}$ as follows:

$$\begin{aligned}
[\mathbf{C}'/\mathbf{h}]\mathbf{h}' &\equiv \begin{cases} \mathbf{C}' & \text{if } \mathbf{h} \equiv \mathbf{h}' \\ \mathbf{h}' & \text{otherwise} \end{cases} \\
[\mathbf{C}'/\mathbf{h}]\mathbf{x} &\equiv \mathbf{x} \\
[\mathbf{C}'/\mathbf{h}]\boldsymbol{\lambda}\mathbf{x}.\mathbf{C} &\equiv \boldsymbol{\lambda}\mathbf{x}.[\mathbf{C}'/\mathbf{h}]\mathbf{C} \\
[\mathbf{C}'/\mathbf{h}](\mathbf{C}_1\ \mathbf{C}_2) &\equiv [\mathbf{C}'/\mathbf{h}]\mathbf{C}_1\ [\mathbf{C}'/\mathbf{h}]\mathbf{C}_2
\end{aligned} \qquad ([\cdot/\mathbf{h}]\cdot)$$

Capture of free identifiers is solicited; see the abstraction clause. Hence, contrasting to

$$[y/x]\lambda y.x \;\equiv\; \lambda y'.y$$

for some fresh variable name y',

$$[\mathbf{y}/\mathbf{h}]\boldsymbol{\lambda}\mathbf{y}.\mathbf{h} \;\equiv\; \boldsymbol{\lambda}\mathbf{y}.\mathbf{y}$$

Finally, we must point out that there are no equivalents of α- and β-substitutions in λ-contexts. For example, the abstraction $\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}$ is not α-equivalent to $\boldsymbol{\lambda}\mathbf{y}.\mathbf{h}$. Filling each of their holes $\mathbf{h}$ with the same $\mathbf{x}$ yields incomparable results:

$$\begin{array}{lcl}
[\mathbf{x}/\mathbf{h}]\boldsymbol{\lambda}\mathbf{x}.\mathbf{h} & \qquad & [\mathbf{x}/\mathbf{h}]\boldsymbol{\lambda}\mathbf{y}.\mathbf{h} \\
\quad \equiv\ \boldsymbol{\lambda}\mathbf{x}.\mathbf{x} & & \quad \equiv\ \boldsymbol{\lambda}\mathbf{y}.\mathbf{x}
\end{array}$$

Intuitively, because of the hole **h**, the scope of the function parameter **x** is not fully known. It is therefore unwise to rename it to **y**. Thus, static scope does not apply to abstractions in λ-contexts. The β-reduction rule is not valid on λ-contexts either. For instance,

$$(\boldsymbol{\lambda}\mathbf{x}.\boldsymbol{\lambda}\mathbf{y}.(\mathbf{x}\ \mathbf{h}))\ \boldsymbol{\lambda}\mathbf{x}.\mathbf{y}$$

is not a β-redex since contracting it would require us to rename the parameter **y**, which has already been declared invalid.

The capability to express identifier capture through hole filling is what attracted us to consider the notion of contexts as a model of incremental program construction. In the following chapters, the feature is integrated into the λ-calculus. It is the first time that it has been done in its full capacity.

Chapter 3

Context-Enriched Lambda Calculus

We apply the context-enriching schema to the pure λ-calculus. The result is a conservative extension of the λ-calculus $\boldsymbol{\lambda}\mathbf{C}$ with incremental compiled code construction capabilities. A comprehensive description of $\boldsymbol{\lambda}\mathbf{C}$ is the subject of this chapter. To demonstrate the expressiveness of $\boldsymbol{\lambda}\mathbf{C}$, we show that program symbols and first-class environments are $\boldsymbol{\lambda}\mathbf{C}$-definable. The fact that $\boldsymbol{\lambda}\mathbf{C}$ is sufficient to simulate the behavior of λ-contexts is verified in Chapter 4.

3.1 Term Language

Our schema enriches a calculus of machine code with the notion of compiled code and incremental compiled code construction mechanisms. To enrich the λ-calculus, we thus add:

- a new category of compiled code abstraction to simulate the compiled version of evolved λ-contexts $\mathbf{e}$,
- a unary compiled code loading operator **load** to assimilate compiled code into machine code,

Syntactic Domains:

$$\begin{array}{rcll} x & \in & \mathit{Vars} & \text{(Variables)} \\ \mathbf{x} & \in & \mathit{Idents} & \text{(Identifiers)} \\ \tilde{\mathbf{x}} & \in & \mathit{Unlnks} & \text{(Unlinked Identifier Indicators)} \end{array}$$

Abstract Syntax:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e\ e \mid \Phi\rho.e \mid \boldsymbol{\delta} \\ \boldsymbol{\delta} & ::= & \mathbf{load} \mid \tilde{\mathbf{x}} \mid \mathbf{lam}_{\mathbf{x}} \mid \mathbf{app} \\ \rho & ::= & \{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\} \\ & & \quad \mathbf{x}_1, \ldots, \mathbf{x}_n \text{ pairwise distinct,} \\ & & \quad x_1, \ldots, x_n \text{ pairwise distinct} \end{array}$$

Figure 3.1: Term language of context-enriched λ-calculus $\lambda\mathbf{C}$

- unary compiled code operators $\mathbf{lam}_{\mathbf{x}}$, one for each identifier $\mathbf{x}$, to incrementally build the compiled code of abstraction λ-contexts $\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}$, and

- a binary compiled code operator **app** to incrementally build the compiled code of application λ-contexts $\mathbf{e}_1\ \mathbf{e}_2$.

The syntax of the context-enriched λ-calculus $\lambda\mathbf{C}$ is summarized in Figure 3.1.

An evolved λ-context $\mathbf{e}$ is source code ready for compilation. Let q be a one-to-one function that assigns a distinct variable name $q(\mathbf{x})$ to each identifier $\mathbf{x}$. Let the *image* λ-term of an evolved λ-context $\mathbf{e}$, denoted as $im(\mathbf{e})$, be defined inductively on the structure of $\mathbf{e}$ as follows:

$$\begin{array}{rcl} im(\mathbf{x}) & \equiv & q(\mathbf{x}) \\ im(\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}) & \equiv & \lambda q(\mathbf{x}).im(\mathbf{e}) \\ im(\mathbf{e}_1\ \mathbf{e}_2) & \equiv & im(\mathbf{e}_1)\ im(\mathbf{e}_2) \end{array}$$

That is, $im(\mathbf{e})$ has the same structure as $\mathbf{e}$ but with each identifier $\mathbf{x}$ replaced by $q(\mathbf{x})$. Compilation is then straightforward for a *closed* evolved λ-context, which is an evolved λ-context that has no free identifiers. It can be translated into its image, *e.g.*, compiling $\boldsymbol{\lambda}\mathbf{x}.\boldsymbol{\lambda}\mathbf{y}.(\mathbf{x}\ \mathbf{y})$ into $\lambda q(\mathbf{x}).\lambda q(\mathbf{y}).(q(\mathbf{x})\ q(\mathbf{y}))$, or equivalently $\lambda x.\lambda y.(x\ y)$. The task becomes non-trivial for an *open* evolved λ-context that has free identifiers whose linking relation is still undetermined. For instance, we cannot translate $\mathbf{y}\ \boldsymbol{\lambda}\mathbf{x}.(\mathbf{x}\ \mathbf{y})$ into

$$\begin{aligned} im(\mathbf{y}\ \boldsymbol{\lambda}\mathbf{x}.(\mathbf{x}\ \mathbf{y})) &\equiv q(\mathbf{y})\ \lambda q(\mathbf{x}).(q(\mathbf{x})\ q(\mathbf{y})) \\ &\equiv q(\mathbf{y})\ \lambda x.(x\ q(\mathbf{y})) \end{aligned}$$

It is unclear what the linking relation of the free variable references $q(\mathbf{y})$ should be.

We therefore require *free identifier abstractions*

$$\Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e$$

to play the role of potentially not yet fully linked compiled code. The specification $\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}$, where the identifiers $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are pairwise distinct and so are the variables $x_1, \ldots, x_n$, indicates that free occurrences of the variables $x_1, \ldots, x_n$ in the machine code e are *temporary* placeholders for the unlinked free identifiers $\mathbf{x}_1, \ldots, \mathbf{x}_n$. Each parameter $\mathbf{x}_i : x_i$ has an identifier $\mathbf{x}_i$ and a variable x_i. The parameter identifier $\mathbf{x}_i$ comes from the evolved λ-context (source code) of e. The parameter variable x_i, like the parameter of a λ-abstraction, denotes a placeholder in the machine code e. The idea then is to compile an evolved λ-context $\mathbf{e}$ into a free identifier abstraction

$$\Phi\{\mathbf{x}_1 : q(\mathbf{x}_1), \ldots, \mathbf{x}_n : q(\mathbf{x}_n)\}.im(\mathbf{e})$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are the free identifiers of $\mathbf{e}$ and $q(\mathbf{x}_1), \ldots, q(\mathbf{x}_n)$ are their respective placeholders in the machine code $im(\mathbf{e})$. (The compiler is defined in Chapter 4.) That is, each free identifier $\mathbf{x}_i$ of the source code $\mathbf{e}$ becomes a Φ-bound free variable x_i of the machine code $im(\mathbf{e})$. To illustrate, the compiled version of the closed evolved λ-context $\boldsymbol{\lambda}\mathbf{x}.\boldsymbol{\lambda}\mathbf{y}.(\mathbf{x}\ \mathbf{y})$ is

$$\Phi\{\}.\lambda q(\mathbf{x}).\lambda q(\mathbf{y}).(q(\mathbf{x})\ q(\mathbf{y})) \quad \equiv \quad \Phi\{\}.\lambda x.\lambda y.(x\ y)$$

and the compiled code of the open evolved λ-context $\mathbf{y}\ \boldsymbol{\lambda}\mathbf{x}.(\mathbf{x}\ \mathbf{y})$ is

$$\Phi\{\mathbf{y}\!:\!q(\mathbf{y})\}.(q(\mathbf{y})\ \lambda q(\mathbf{x}).(q(\mathbf{x})\ q(\mathbf{y}))) \quad\equiv\quad \Phi\{\mathbf{y}\!:\!y\}.(y\ \lambda x.(x\ y))$$

The parameter variables of a free identifier abstraction $\Phi\rho.e$ need not coincide with the free variables of e. There can be more parameter variables specified in ρ than are actually referred to in e; the unreferenced ones are simply ignored. An example is $\Phi\{\mathbf{x}\!:\!x\}.\lambda y.y$, which could be the result of optimizing the compiled code $\Phi\{\mathbf{x}\!:\!x\}.\underline{((\lambda w.\lambda y.y)\ \lambda z.x)}$ by contracting the underlined β-redex. There can also be free variables in machine code e that are not specified in ρ; they simply obey static scope, a design decision we have consciously made. It allows us to embed machine code in source code, so to speak. For instance, we may express the compiled code of $\boldsymbol{\lambda}\mathbf{z}.(x\ (\mathbf{z}\ \mathbf{y}))$ as $\Phi\{\mathbf{y}\!:\!y\}.\lambda z.(x\ (z\ y))$. Hence, the e-part of compiled code $\Phi\rho.e$ can be parameterized, *e.g.*, $\lambda x.\Phi\{\mathbf{y}\!:\!y\}.\lambda z.(x\ (z\ y))$. Furthermore, Φ-abstractions may be nested, *e.g.*, $\Phi\{\mathbf{x}\!:\!x\}.\Phi\{\mathbf{y}\!:\!y\}.\lambda z.(x\ (z\ y))$.

The parameter specification $\{\mathbf{x}_1\!:\!x_1, \ldots, \mathbf{x}_n\!:\!x_n\}$ of a free identifier abstraction forms a set. The order of its elements is immaterial to the behavior of the abstraction. As a result, we do not distinguish free identifier abstractions that differ only in the syntactic ordering of their parameters. Hence, $\Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.(x\ y)$ and $\Phi\{\mathbf{y}\!:\!y, \mathbf{x}\!:\!x\}.(x\ y)$ are considered identical.

We adopt the following notational convention for Φ-abstractions throughout the thesis. Let $\bar{\mathbf{x}}$ denote a series of pairwise distinct identifiers $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and let $\bar{x}$ be a series of the same number of pairwise distinct variables $x_1, \ldots, x_n$. Then, $\Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e$ abbreviates $\Phi\{\mathbf{x}_1\!:\!x_1, \ldots, \mathbf{x}_n\!:\!x_n\}.e$.

3.2 Alpha Convertibility

The first order of business in defining the $\boldsymbol{\lambda}\mathbf{C}$-calculus is the formalization of α-conversion. A λ-abstraction is a variable binding construct. It is thus possible to rename the parameter x of $\lambda x.e$ to a fresh variable y without changing the behavior of the abstraction:

$$\lambda x.e \quad \rightarrow \quad \lambda y.\langle y/x\rangle e \tag{α-λ}$$

where the fresh variable y is neither x nor one of the free variables of e. A free identifier abstraction is also a variable binding form. Its parameter variables are subject to α-conversion as well:

$$\Phi\{\ldots,\mathbf{x}_i\!:\!x_i,\ldots\}.e \quad \rightarrow \quad \Phi\{\ldots,\mathbf{x}_i\!:\!y,\ldots\}.\langle y/x_i\rangle e \tag{α-Φ}$$

The fresh variable y should not be x_i, the parameter variable it replaces. It should not be any of the free variables in the body e so that no inadvertent capture occurs. Neither should it be one of the other parameter variables x_j; otherwise the resulting parameter specification $\{\ldots,\mathbf{x}_i\!:\!x_j,\ldots,\mathbf{x}_j\!:\!x_j,\ldots\}$ is not well-formed. The notion of reduction α for parameter renaming is then the union of the two basic notions of reduction α-λ and α-Φ:

$$\alpha \quad = \quad \alpha\text{-}\lambda \cup \alpha\text{-}\Phi \tag{α}$$

Two fundamental syntactic notions needed in the definition of α-conversion are the notion of free variables and the α-substitution meta-operation $\langle y/x\rangle e$. An occurrence of a variable x in e is free if it is not in linking relation with any λ- or Φ-abstraction parameter. The set of free variable of a $\boldsymbol{\lambda}\mathbf{C}$-term e, denoted as $fv(e)$, conservatively extends its λ-term counterpart:

$$\begin{aligned}
fv(x) &= \{x\} \\
fv(\lambda x.e) &= fv(e) \setminus \{x\} \\
fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) \\
fv(\Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e) &= fv(e) \setminus \{\bar{x}\} \\
fv(\boldsymbol{\delta}) &= \emptyset
\end{aligned}$$

The compiled code related primitives $\boldsymbol{\delta}$ contain no free variable references; they are constants. The scope of the parameter variables $\bar{x}$ of a free identifier abstraction $\Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e$ is the term e. Hence, free occurrences of $\bar{x}$ in e are not considered free beyond the abstraction.

The α-substitution meta-operation $\langle y/x\rangle e$ that replaces the variable name y for the free occurrences of the variable name x in the $\boldsymbol{\lambda}\mathbf{C}$-term e is also a conservative extension of its λ-term counterpart:

$$\begin{array}{rcl}
\langle y/x\rangle z & \equiv & \begin{cases} y & \text{if } x \equiv z \\ z & \text{otherwise} \end{cases} \\
\langle y/x\rangle \lambda z.e & \equiv & \lambda z.\langle y/x\rangle e \\
\langle y/x\rangle (e_1\ e_2) & \equiv & \langle y/x\rangle e_1\ \langle y/x\rangle e_2 \\
\langle y/x\rangle \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e & \equiv & \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.\langle y/x\rangle e \\
\langle y/x\rangle \boldsymbol{\delta} & \equiv & \boldsymbol{\delta}
\end{array}$$

Since the renaming of parameter variables is universally applicable, we can assume that the parameter variables $\bar{x}$ in the Φ-abstraction clause are distinct from x and y.

The above simplification is a result of the following variable conventions:

α-congruence Two $\boldsymbol{\lambda}\mathbf{C}$-terms that are syntactically identical up to the renaming of their parameter variables are identified ($\equiv$).

hygiene The parameter variables of the $\boldsymbol{\lambda}\mathbf{C}$-terms involved in any mathematical contexts are mutually distinct from one another and are distinct from the free variables as well.

3.3 Reduction Rules

The context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{C}$ has four reduction rules. In addition to the β-reduction rule for modeling function invocation, there is one rule for defining each of the three compiled code operators **load**, **app**, and $\mathbf{lam}_{\mathbf{X}}$.

3.3.1 Function Invocation

The β-reduction rule is still of the form:

$$(\lambda x.e)\ e' \ \rightarrow\ [e'/x]e \qquad (\beta)$$

The β-substitution meta-operation $[e'/x]e$ is conservatively extended to include the newly introduced free identifier abstractions $\Phi\rho.e$ and constants $\boldsymbol{\delta}$:

$$\begin{aligned}
[e'/x]z &\equiv \begin{cases} e' & \text{if } x \equiv z \\ z & \text{otherwise} \end{cases} \\
[e'/x]\lambda z.e &\equiv \lambda z.[e'/x]e \\
[e'/x](e_1\ e_2) &\equiv [e'/x]e_1\ [e'/x]e_2 \\
[e'/x]\Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e &\equiv \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.[e'/x]e \\
[e/x]\boldsymbol{\delta} &\equiv \boldsymbol{\delta}
\end{aligned}$$

We have used the variable conventions to ensure that β-substitution does not cause inadvertent variable capture. In particular, in the Φ-abstraction clause, x is not one of the parameter variables $\bar{x}$ and none of the parameter variables are free in e'.

3.3.2 Loading Compiled Code

The operator **load** loads compiled code modeled by free identifier abstractions. Ideally, when $\Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e$ is ready for loading, the parameter variables $\bar{x}$ should no longer occur free in e, they should have been linked in the process. If that is always the case, loading $\Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e$ can be accomplished by simply stripping off the abstractor $\Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}$ to yield the machine code e:

$$\mathbf{load}\ \Phi\rho.e \quad \rightarrow \quad e$$

This is a difficult condition to verify, however. We thus incorporate constants $\tilde{\mathbf{x}}$, one for each identifier $\mathbf{x}$, into the calculus to serve as *unlinked free identifier indicators.* They are a unique category of constants whose interpretation is orthogonal to the rest of the calculus. The reduction rule for compiled code loading is:

$$\mathbf{load}\ \Phi\{\mathbf{x}_1\!:\!x_1,\ldots,\mathbf{x}_n\!:\!x_n\}.e \quad \rightarrow \quad [\tilde{\mathbf{x}}_n/x_n]\cdots[\tilde{\mathbf{x}}_1/x_1]e \qquad (\mathbf{load})$$

It removes the abstractor $\Phi\{\mathbf{x}_1\!:\!x_1,\ldots,\mathbf{x}_n\!:\!x_n\}$ after substituting each free occurrence of x_i in the machine code e with the unlinked identifier indicator $\tilde{\mathbf{x}}_i$ of $\mathbf{x}_i$. (A mechanism to provide unlinked parameter variables with default denotations other than $\tilde{\mathbf{x}}$ can be found in Chapter 7.)

For instance,

$$\begin{aligned} \textbf{load}\ \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.\lambda z.y &\ \rightarrow\ [\tilde{\mathbf{x}}/x][\tilde{\mathbf{y}}/y]\lambda z.y \\ &\ \equiv\ \lambda z.\tilde{\mathbf{y}} \end{aligned}$$

The constant $\tilde{\mathbf{y}}$ in the contractum $\lambda z.\tilde{\mathbf{y}}$ indicates that the identifier $\mathbf{y}$ had not been linked when the compiled code was loaded.

3.3.3 Constructing Compiled Applications

The incremental compiled code construction operator **app** models the filling of the application λ-context $\mathbf{h}_1\ \mathbf{h}_2$ with the evolved λ-contexts $\mathbf{e}_1$ and $\mathbf{e}_2$. The reduction rule is:

$$\textbf{app}\ \Phi\rho_1.e_1\ \Phi\rho_2.e_2\ \rightarrow\ \Phi\rho_1 \uplus \rho_2.(e_1\ e_2) \qquad (\textbf{app})$$

It constructs the compiled version of the λ-context $\mathbf{e}_1\ \mathbf{e}_2$, expressed as $\Phi\rho_1 \uplus \rho_2.(e_1\ e_2)$, from the compiled code of $\mathbf{e}_1$ and $\mathbf{e}_2$ represented by $\Phi\rho_1.e_1$ and $\Phi\rho_2.e_2$, respectively.

The notation $\rho_1 \uplus \rho_2$ denotes a variant of the union of ρ_1 and ρ_2 that relies on α-conversion to meet these constraints:

(i) To maintain the well-formedness of the union, an identifier $\mathbf{x}$ can occur in both ρ_1 and ρ_2 as long as its variables are identical, that is,

$$(\mathbf{x}\!:\!x_1 \in \rho_1 \text{ and } \mathbf{x}\!:\!x_2 \in \rho_2) \text{ if and only if } x_1 \equiv x_2.$$

Similarly, a variable x can occur in both ρ_1 and ρ_2 as long as its corresponding identifiers are identical, that is,

$$(\mathbf{x}_1\!:\!x \in \rho_1 \text{ and } \mathbf{x}_2\!:\!x \in \rho_2) \text{ if and only if } \mathbf{x}_1 \equiv \mathbf{x}_2.$$

(ii) The parameter variables of ρ_1 do not capture the free variables of e_2 except for those specified in (i). That is, if $\mathbf{x}\!:\!x \in \rho_1$ and $\mathbf{x}$ is not a parameter identifier of

ρ_2, then x should not be a free variable of e_2. Likewise, the parameter variables of ρ_2 do not capture the free variables of e_1 except for those specified in (i).

Notice that this is the first time we depart from the hygiene variable convention and insist that some parameter variables of both ρ_1 and ρ_2 actually be identical under condition (i). Beyond that, condition (ii) is merely a rephrase of the hygiene convention.

The constraint $\rho_1 \uplus \rho_2$ is purely artificial since it can always be met by renaming the parameter variables of ρ_1 and ρ_2. Hence, every term of the form **app** $\Phi\rho_1.e_1\ \Phi\rho_2.e_2$ has an α-equivalent **app**-redex. For example,

$$\begin{aligned} &\mathbf{app}\ \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.(x\ y)\ \Phi\{\mathbf{x}\!:\!w, \mathbf{z}\!:\!y\}.(x\ w\ y) \\ \equiv\ \ &\mathbf{app}\ \Phi\{\mathbf{x}\!:\!v, \mathbf{y}\!:\!y\}.(v\ y)\ \Phi\{\mathbf{x}\!:\!v, \mathbf{z}\!:\!z\}.(x\ v\ z) \\ \rightarrow\ \ &\Phi\{\mathbf{x}\!:\!v, \mathbf{y}\!:\!y, \mathbf{z}\!:\!z\}.(v\ y\ (x\ v\ z)) \end{aligned}$$

To take the union of the two sets $\rho_1 \equiv \{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}$ and $\rho_2 \equiv \{\mathbf{x}\!:\!w, \mathbf{z}\!:\!y\}$, we identify the variables of $\mathbf{x}$ and distinguish the variables of $\mathbf{y}$ and $\mathbf{z}$. Hence, both x and w are renamed to v and the variable y of $\mathbf{z}$ is renamed to z. The resulting specifications $\{\mathbf{x}\!:\!v, \mathbf{y}\!:\!y\}$ and $\{\mathbf{x}\!:\!v, \mathbf{z}\!:\!z\}$ can then be combined to form the union $\{\mathbf{x}\!:\!v, \mathbf{y}\!:\!y, \mathbf{z}\!:\!z\}$. Intuitively, $\Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.(x\ y)$ and $\Phi\{\mathbf{x}\!:\!w, \mathbf{z}\!:\!y\}.(x\ w\ y)$ are the compiled code of $\mathbf{e}_1 \equiv \mathbf{x}\ \mathbf{y}$ and $\mathbf{e}_2 \equiv x\ \mathbf{x}\ \mathbf{z}$, respectively. The contractum $\Phi\{\mathbf{x}\!:\!v, \mathbf{y}\!:\!y, \mathbf{z}\!:\!z\}.(v\ y\ (x\ v\ z))$ is the compiled version of $\mathbf{e}_1\ \mathbf{e}_2 \equiv \mathbf{x}\ \mathbf{y}\ (x\ \mathbf{x}\ \mathbf{z})$, the result of filling the holes of $\mathbf{h}_1\ \mathbf{h}_2$ with $\mathbf{e}_1$ and $\mathbf{e}_2$.

3.3.4 Constructing Compiled Abstractions

The incremental compiled code construction operator $\mathbf{lam_x}$, one for each identifier $\mathbf{x}$, models the filling of the hole $\mathbf{h}$ of the λ-context $\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}$ with the evolved λ-context $\mathbf{e}$. The reduction rule is:

$$\mathbf{lam_x}\ \Phi\rho.e \ \ \rightarrow\ \ \Phi\rho.\lambda x.e \qquad \textbf{(lam)}$$
$$\mathbf{x}\!:\!x \in \rho \text{ or else } x \text{ is fresh}$$

It builds the compiled code $\Phi\rho.\lambda x.e$ of $\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}$ from the compiled code $\Phi\rho.e$ of $\mathbf{e}$. The choice of x in the contractum $\Phi\rho.\lambda x.e$ depends on whether $\mathbf{x}$ is specified in ρ. If so, $\mathbf{x}\!:\!x \in \rho$ and x is the parameter of the newly constructed abstraction $\lambda x.e$; otherwise, x is a fresh variable name not free in e. In both cases, a new abstraction is constructed. In the former, the free occurrences of x in e become linked to the parameter of the newly constructed abstraction. In other words, they are captured by the new parameter. Notice that this is the second time we have bypassed the hygiene variable convention and actually require the happening of variable capture instead. In the latter, the new parameter x is carefully chosen to be distinct from the free variables of e so as to avoid inadvertent capture since $\mathbf{x}$ is not a free identifier of $\mathbf{e}$.

To demonstrate, when $\mathbf{x}$ and $\mathbf{y}$ are distinct identifiers, the result of filling $\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}$ with $\mathbf{y}$ is $\boldsymbol{\lambda}\mathbf{x}.\mathbf{y}$. This is modeled as:

$$\mathbf{lam}_{\mathbf{x}}\ \Phi\{\mathbf{y}\!:\!y\}.y \quad\rightarrow\quad \Phi\{\mathbf{y}\!:\!y\}.\lambda x.y$$

The free identifier abstraction $\Phi\{\mathbf{y}\!:\!y\}.y$ is the compiled version of $\mathbf{y}$ and $\Phi\{\mathbf{y}\!:\!y\}.\lambda x.y$ is the compiled code of $\boldsymbol{\lambda}\mathbf{x}.\mathbf{y}$. In contrast,

$$\mathbf{lam}_{\mathbf{y}}\ \Phi\{\mathbf{y}\!:\!y\}.y \quad\rightarrow\quad \Phi\{\mathbf{y}\!:\!y\}.\lambda y.y$$

where the contractum is the compiled version of $\boldsymbol{\lambda}\mathbf{y}.\mathbf{y}$, a result of filling $\boldsymbol{\lambda}\mathbf{y}.\mathbf{h}$ with $\mathbf{y}$.

Hence, an incremental compiled code constructor $\mathbf{lam}_{\mathbf{x}}$ generates the compiled code of a function from the compiled code of the function's body. It relies on the fact that the parameter variables $x_1, \ldots, x_n$ of $\Phi\{\mathbf{x}_1\!:\!x_1, \ldots, \mathbf{x}_n\!:\!x_n\}.e$ are *quasi-statically scoped* [42]. The linking relation between the free occurrences of x_i in e and the Φ-parameter $\mathbf{x}_i\!:\!x_i$ is not static; the free variable references as a whole can be *relinked* to the parameter of a newly constructed encapsulating λ-abstraction.

3.4 Calculus of Compiled Code

The notion of reduction $\mathbf{c}$ underlying the extended calculus $\lambda\mathbf{C}$ is the union of the four reduction rules β, $\mathbf{load}$, $\mathbf{lam}$, and $\mathbf{app}$ described in the last section. They

$$
\begin{array}{rcll}
(\lambda x.e)\ e' & \to & [e'/x]e & (\beta) \\
\textbf{load}\ \Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e & \to & [\tilde{\mathbf{x}}_n/x_n] \cdots [\tilde{\mathbf{x}}_1/x_1]e & (\textbf{load}) \\
\textbf{app}\ \Phi\rho_1.e_1\ \Phi\rho_2.e_2 & \to & \Phi\rho_1 \uplus \rho_2.(e_1\ e_2) & (\textbf{app}) \\
\textbf{lam}_{\mathbf{x}}\ \Phi\rho.e & \to & \Phi\rho.\lambda x.e & (\textbf{lam}) \\
 & & \quad \mathbf{x} : x \in \rho \text{ or else } x \text{ is fresh} &
\end{array}
$$

Figure 3.2: Reduction rules of context-enriched λ-calculus $\lambda\mathbf{C}$

are collectively repeated in Figure 3.2. The one-step **c**-reduction relation $\to_{\mathbf{c}}$ is the compatible closure of **c**. That is, a term e_1 one-step **c**-reduces to the term e_2 if e_2 is the result of replacing a **c**-redex subterm e_1' of e_1 with its contractum e_2'. The reflexive and transitive closure of $\to_{\mathbf{c}}$ is the **c**-reduction relation $\twoheadrightarrow_{\mathbf{c}}$. The least equivalence relation generated by $\twoheadrightarrow_{\mathbf{c}}$ is the equational theory $\lambda\mathbf{C}$. Equivalence under $\lambda\mathbf{C}$ is written as $e_1 =_{\mathbf{c}} e_2$. We often omit the subscript **c** of the reduction relations and write $\to$, $\twoheadrightarrow$, and $=$ for $\to_{\mathbf{c}}$, $\twoheadrightarrow_{\mathbf{c}}$, and $=_{\mathbf{c}}$, instead.

To summarize, computation in $\lambda\mathbf{C}$ involves three categories of machine instruction:

- function invocation modeled by the β-rule,
- incremental compilation and linking modeled by the **app**- and **lam**-rules, and
- the incorporation of compiled code into machine code modeled by the **load**-rule.

What sets $\boldsymbol{\lambda}\mathbf{C}$ apart from languages like the λ-calculus is that incremental construction, linking, and loading of compiled code are readily available in a single system, instead of as additional programming environment tools. Furthermore, as promised, the incremental compilation mechanisms of $\boldsymbol{\lambda}\mathbf{C}$ use only two basic editing operations, namely, the copying of compiled code and the renaming of variables, to model the reuse and linking of existing program components.

As a sanity check, in the rest of this section we show that the notion of reduction **c** underlying the $\boldsymbol{\lambda}\mathbf{C}$-calculus is Church-Rosser. That is,

Theorem 3.1 *The reduction relation $\twoheadrightarrow$ satisfies the diamond property.*

It ensures that the $\boldsymbol{\lambda}$**C**-calculus as a computational model produces unique answers, if any, for programs. The proof of Theorem 3.1 uses the Hindley-Rosen Lemma; see Section 2.2.2. Let the notion of reduction γ be the union of the compiled code oriented reduction rules **lam**, **app**, and **load**:

$$\gamma \quad = \quad \mathbf{lam} \cup \mathbf{app} \cup \mathbf{load}$$

Let $\rightarrow_\gamma$ and $\twoheadrightarrow_\gamma$ denote the one-step γ-reduction and the γ-reduction relations. We can show that the notion of reduction γ is Church-Rosser:

Lemma 3.2 $\twoheadrightarrow_\gamma \models \Diamond$. *The γ-reduction relation $\twoheadrightarrow_\gamma$ satisfies the diamond property.*

Proof: We prove a stronger result that the one-step γ-reduction relation $\rightarrow_\gamma$ satisfies the diamond property. From which it is clear that the γ-reduction relation $\twoheadrightarrow_\gamma$ also satisfies the diamond property.

The one-hole $\boldsymbol{\lambda}$**C**-contexts $C[]$ have the following abstract syntax:

$$C[] \quad ::= \quad [] \mid \lambda x.C[] \mid C[]\ e \mid e\ C[] \mid \Phi\rho.C[]$$

Let $F \equiv C[\mathbf{lam_X}\ \Phi\rho.e]$ be a term that has a γ-redex $\mathbf{lam_X}\ \Phi\rho.e$ as a subterm. Let $F_1 \equiv C[\Phi\rho.\lambda x.e]$ be the result of contracting the γ-redex of F. There are three cases we must consider:

(a) F one-step γ-reduces to $F_2 \equiv C[\mathbf{lam_X}\ \Phi\rho.e']$ because of the one-step γ-reduction of e to e'. Then, $F_3 \equiv C[\Phi\rho.\lambda x.e']$ is the common one-step γ-reduct of F_1 and F_2:

$$F_1 \quad \equiv \quad C[\Phi\rho.\lambda x.e] \quad \rightarrow_\gamma \quad C[\Phi\rho.\lambda x.e'] \quad \equiv \quad F_3$$

as a direct consequence of $e \rightarrow_\gamma e'$, and

$$F_2 \quad \equiv \quad C[\mathbf{lam_X}\ \Phi\rho.e'] \quad \rightarrow_\gamma \quad C[\Phi\rho.\lambda x.e'] \quad \equiv \quad F_3$$

by contracting the redex $\mathbf{lam_X}\ \Phi\rho.e'$.

(b) F one-step γ-reduces to $F_2 \equiv C'[\mathbf{lam_X}\ \Phi\rho.e]$ because of the one-step γ-reduction of some redex in $C[]$ that is disjoint from the redex $\mathbf{lam_X}\ \Phi\rho.e$. Then, $F_3 \equiv C'[\Phi\rho.\lambda x.e']$ is the common one-step γ-reduct of F_1 and F_2:

$$F_1 \;\equiv\; C[\Phi\rho.\lambda x.e] \;\rightarrow_\gamma\; C'[\Phi\rho.\lambda x.e] \;\equiv\; F_3$$

as a direct consequence of contracting the same redex in $C[]$, and

$$F_2 \;\equiv\; C'[\mathbf{lam_X}\ \Phi\rho.e] \;\rightarrow_\gamma\; C'[\Phi\rho.\lambda x.e] \;\equiv\; F_3$$

by contracting the redex $\mathbf{lam_X}\ \Phi\rho.e$.

(c) F one-step γ-reduces to $F_2 \equiv C'[\mathbf{lam_X}\ \Phi\rho.\underline{e}]$ because of the one-step γ-reduction of some **load**-redex in $C[]$ that has the redex $\mathbf{lam_X}\ \Phi\rho.e$ as a subterm. The notation $\underline{e}$ denotes the result of substituting some of the free variables of e with unlinked identifier indicators as a result of contracting the **load**-redex. Then, $F_3 \equiv C'[\Phi\rho.\lambda x.\underline{e}]$ is the common one-step γ-reduct of F_1 and F_2:

$$F_1 \;\equiv\; C[\Phi\rho.\lambda x.e] \;\rightarrow_\gamma\; C'[\Phi\rho.\lambda x.\underline{e}] \;\equiv\; F_3$$

as a direct consequence of contracting the same redex in $C[]$, and

$$F_2 \;\equiv\; C'[\mathbf{lam_X}\ \Phi\rho.\underline{e}] \;\rightarrow_\gamma\; C'[\Phi\rho.\lambda x.\underline{e}] \;\equiv\; F_3$$

by contracting the redex $\mathbf{lam_X}\ \Phi\rho.\underline{e}$.

The other two cases in which the γ-redex of concern in F is either an **app**-redex, $F \equiv C[\mathbf{app}\ \Phi\rho_1.e_1\ \Phi\rho_2.e_2]$, or a **load**-redex, $F \equiv C[\mathbf{load}\ \Phi\rho.e]$. Their analyses can be handled in a similar fashion and are therefore omitted. □

The second step of Hindley-Rosen Lemma requires us to show that $\twoheadrightarrow_\gamma$ commutes with $\twoheadrightarrow_\beta$. We first prove the following intermediate result:

Lemma 3.3 *Let F, F_1, and F_2 be $\boldsymbol{\lambda}\mathbf{C}$-terms such that $F \rightarrow_\beta F_1$ and $F \rightarrow_\gamma F_2$. Then, there is a $\boldsymbol{\lambda}\mathbf{C}$-term F_3 such that $F_1 \twoheadrightarrow_\gamma F_3$ and $F_2 \rightarrow_\beta F_3$.*

Proof: Let $F \equiv C[(\lambda x.e_1)\ e_2]$ and $F_1 \equiv C[[e_2/x]e_1]$. There are four cases for $F \rightarrow_\gamma F_2$:

(a) $F_2 \equiv C[(\lambda x.e_1')\ e_2]$ and $F \rightarrow_\gamma F_2$ is a direct consequence of $e_1 \rightarrow_\gamma e_1'$. Then, the choice of F_3 is $C[[e_2/x]e_1']$:

$$F_1 \;\equiv\; C[[e_2/x]e_1] \;\rightarrow_\gamma\; C[[e_2/x]e_1'] \;\equiv\; F_3$$

as a direct consequence of contracting the same γ-redex in e_1, and

$$F_2 \;\equiv\; C[(\lambda x.e_1')\ e_2] \;\rightarrow_\beta\; C[[e_2/x]e_1'] \;\equiv\; F_3$$

by contracting the β-redex $(\lambda x.e_1')\ e_2$.

(b) $F_2 \equiv C[(\lambda x.e_1)\ e_2']$ and $F \rightarrow_\gamma F_2$ is a direct consequence of $e_2 \rightarrow_\gamma e_2'$. Then, the choice of F_3 is $C[[e_2'/x]e_1]$:

$$F_1 \;\equiv\; C[[e_2/x]e_1] \;\twoheadrightarrow_\gamma\; C[[e_2'/x]e_1] \;\equiv\; F_3$$

as a direct consequence of contracting each occurrence of the same γ-redex in e_2, and

$$F_2 \;\equiv\; C[(\lambda x.e_1)\ e_2'] \;\rightarrow_\beta\; C[[e_2'/x]e_1] \;\equiv\; F_3$$

by contracting the β-redex $(\lambda x.e_1)\ e_2'$.

(c) $F_2 \equiv C'[(\lambda x.e_1)\ e_2]$ and $F \rightarrow_\gamma F_2$ is a direct consequence of $C[] \rightarrow_\gamma C'[]$, reduction of a γ-redex in $C[]$ that is disjoint from the β-redex $(\lambda x.e_1)\ e_2$. Then, the choice of F_3 is $C'[[e_2/x]e_1]$:

$$F_1 \;\equiv\; C[[e_2/x]e_1] \;\rightarrow_\gamma\; C'[[e_2/x]e_1] \;\equiv\; F_3$$

as a direct consequence of contracting the same γ-redex in $C[]$, and

$$F_2 \;\equiv\; C'[(\lambda x.e_1)\ e_2] \;\rightarrow_\beta\; C'[[e_2/x]e_1] \;\equiv\; F_3$$

by contracting the β-redex $(\lambda x.e_1)\ e_2$.

(d) $F_2 \equiv C'[(\lambda x.\underline{e_1})\ \underline{e_2}]$ and $F \rightarrow_\gamma F_2$ is a direct consequence of $C[] \rightarrow_\gamma C'[]$, reduction of a **load**-redex in $C[]$ that has the β-redex $(\lambda x.e_1)\ e_2$ as a subterm. The notation $\underline{e_i}$ denotes the result of substituting some of the free variables of e_i with unlinked identifier indicators as a result of contracting the **load**-redex. The choice of F_3 is $C'[[\underline{e_2}/x]\underline{e_1}]$:

$$F_1 \equiv C[[e_2/x]e_1] \rightarrow_\gamma C'[[\underline{e_2}/x]\underline{e_1}] \equiv F_3$$

as a direct consequence of contracting the same **load**-redex in $C[]$, and

$$F_2 \equiv C'[(\lambda x.\underline{e_1})\ \underline{e_2}] \rightarrow_\beta C'[[\underline{e_2}/x]\underline{e_1}] \equiv F_3$$

by contracting the β-redex $(\lambda x.\underline{e_1})\ \underline{e_2}$. □

From Lemma 3.3 it is straightforward to show that $\twoheadrightarrow_\gamma$ commutes with $\rightarrow_\beta$. Thus, with a simple diagram chase, the commutativity between $\twoheadrightarrow_\gamma$ and $\twoheadrightarrow_\beta$ follows immediately:

Lemma 3.4 *$\twoheadrightarrow_\gamma$ commutes with $\twoheadrightarrow_\beta$.*

Proof of Theorem 3.1: We already know that the β-reduction relation $\twoheadrightarrow_\beta$ satisfies the diamond property, $\twoheadrightarrow_\beta \models \diamond$. We also know that the notion of reduction **c** is the union of β and γ. Thus, from Lemmas 3.2 and 3.4, and the Hindley-Rosen Lemma (Lemma 2.1), the notion of reduction **c** is Church-Rosser. □

We have thus far formally defined the context-enriched λ-calculus $\boldsymbol{\lambda}$**C**. The rest of the chapter is dedicated to discussions on novel properties of compiled code abstractions and programming examples using $\boldsymbol{\lambda}$**C**.

3.5 Redundant Parameters

Although free identifier abstractions are devised as a means to model the compiled code of evolved λ-contexts, the correspondence is not one-to-one. A λ-context can have many Φ-abstraction representations. For instance, both $\Phi\{\}.\lambda y.y$ and

$\Phi\{\mathbf{y}\!:\!y\}.\lambda y.y$ can serve as the compiled code of the λ-context $\boldsymbol{\lambda}\mathbf{y}.\mathbf{y}$. The reason is that they are not distinguishable by the compiled code operators **load**, $\mathbf{lam_x}$, and **app** of $\boldsymbol{\lambda}\mathbf{C}$.

In general, for any two free identifier abstractions $\Phi\rho_1.e$ and $\Phi\rho_2.e$ that differ only in their parameter specifications ρ_1 and ρ_2, let $\Phi\rho_1.e \approx \Phi\rho_2.e$ denote the following condition:

$$\text{for each free variable } x \text{ of } e,\ \mathbf{x}\!:\!x \in \rho_1 \text{ if and only if } \mathbf{x}\!:\!x \in \rho_2 \qquad (\approx)$$

That is, the two parameter specifications ρ_1 and ρ_2 agree on the free variables of e. Then, the two abstractions $\Phi\rho_1.e$ and $\Phi\rho_2.e$ are *indistinguishable* by the compiled code operations of $\boldsymbol{\lambda}\mathbf{C}$. In particular, we can show the following:

Theorem 3.5 *(i) If* $\Phi\rho_1.e \approx \Phi\rho_2.e$ *then* **load** $\Phi\rho_1.e =$ **load** $\Phi\rho_2.e$.

(ii) Let $\Phi\rho_1.e \approx \Phi\rho_2.e$ *and*

$$\begin{aligned} \mathbf{lam_x}\ \Phi\rho_1.e &\rightarrow \Phi\rho_1.\lambda x_1.e \\ \mathbf{lam_x}\ \Phi\rho_2.e &\rightarrow \Phi\rho_2.\lambda x_2.e \end{aligned}$$

Then, $\Phi\rho_1.\lambda x_1.e \approx \Phi\rho_2.\lambda x_2.e$.

(iii) Let $\Phi\rho_1.e \approx \Phi\rho_2.e$, $\Phi\rho'_1.e' \approx \Phi\rho'_2.e'$, *and*

$$\begin{aligned} \mathbf{app}\ \Phi\rho_1.e\ \Phi\rho'_1.e' &\rightarrow \Phi\rho_1 \uplus \rho'_1.(e\ e') \\ \mathbf{app}\ \Phi\rho_2.e\ \Phi\rho'_2.e' &\rightarrow \Phi\rho_2 \uplus \rho'_2.(e\ e') \end{aligned}$$

Then, $\Phi\rho_1 \uplus \rho'_1.(e\ e') \approx \Phi\rho_2 \uplus \rho'_2.(e\ e')$.

Proof: Part (i) is rather simple. Loading $\Phi\rho_1.e$ affects only free occurrences of the parameter variables of ρ_1 in e. The result is therefore identical to the loading of $\Phi\rho_2.e$ because the two specifications ρ_1 and ρ_2 agree on exactly those parameters that are of concern.

For part (ii), we need only show that $\lambda x_1.e \equiv \lambda x_2.e$. Then, with the fact that ρ_1 and ρ_2 agree on the free variables of e, we can conclude that ρ_1 and ρ_2 agree on the free variables of $\lambda x_1.e$ as well. There are three cases to consider:

(a) $\mathbf{x}$ is not a parameter identifier of ρ_1 and therefore x_1 is not a free variable of e. Then, either $\mathbf{x}$ is not a parameter identifier of ρ_2 and so x_2 is not free in e or $\mathbf{x}\!:\!x_2$ is a parameter of ρ_2 but x_2 is not a free variable of e (otherwise $\mathbf{x}\!:\!x_2$ must be a parameter of ρ_1 according to the definition of $\Phi\rho_1.e \approx \Phi\rho_2.e$). Hence, $\lambda x_1.e \equiv \lambda x_2.e$ holds because neither x_1 nor x_2 occurs free in e.

(b) $\mathbf{x}\!:\!x_1$ is a parameter of ρ_1 but x_1 is not free in e. Then, either $\mathbf{x}\!:\!x_2$ is a parameter of ρ_2 but x_2 is not a free variable of e or $\mathbf{x}$ is not a parameter identifier of ρ_2 and so x_2 does not occur free in e. Again, in both cases $\lambda x_1.e \equiv \lambda x_2.e$ is valid since neither x_1 nor x_2 is a free variable of e.

(c) $\mathbf{x}\!:\!x_1$ is a parameter of ρ_1 and x_1 is free in e. Then $\mathbf{x}\!:\!x_1$ must be a parameter of ρ_2 as well. Hence x_2 must be the same variable name as x_1 and so $\lambda x_1.e \equiv \lambda x_2.e$.

The validity of part (iii) is obvious. The compiled code constructor **app** does not involve the relinking or substitution of parameter variables. Hence, it can detect neither the distinction between $\Phi\rho_1.e$ and $\Phi\rho_2.e$ nor between $\Phi\rho_1'.e'$ and $\Phi\rho_2'.e'$. $\square$

To illustrate, the Φ-abstractions $\Phi\{\}.\lambda y.y$ and $\Phi\{\mathbf{y}\!:\!y\}.\lambda y.y$ are indistinguishable in $\boldsymbol{\lambda}\mathbf{C}$:

$$\begin{array}{lllll}
\mathbf{load}\ \Phi\{\}.\lambda y.y & & & & \mathbf{load}\ \Phi\{\mathbf{y}\!:\!y\}.\lambda y.y \\
 & & & \to & [\tilde{\mathbf{y}}/y]\lambda y.y \\
\equiv\ \lambda y.y & & = & \equiv & \lambda y.y
\end{array}$$

and

$$\begin{array}{lcl}
\mathbf{lam}_{\mathbf{y}}\ \Phi\{\}.\lambda y.y & & \mathbf{lam}_{\mathbf{y}}\ \Phi\{\mathbf{y}\!:\!y\}.\lambda y.y \\
\to\ \Phi\{\}.\lambda y.\lambda y.y & \approx & \to\ \Phi\{\mathbf{y}\!:\!y\}.\lambda y.\lambda y.y
\end{array}$$

and for any $\Phi\rho_1.e \approx \Phi\rho_2.e$,

$$\begin{array}{lcl}
\mathbf{app}\ \Phi\{\mathbf{y}\!:\!y\}.\lambda y.y\ \Phi\rho_1.e & & \mathbf{app}\ \Phi\{\}.\lambda y.y\ \Phi\rho_2.e \\
\to\ \Phi\{\mathbf{y}\!:\!y\}\uplus\rho_1.((\lambda y.y)\ e) & \approx & \to\ \Phi\rho_2.((\lambda y.y)\ e)
\end{array}$$

We have actually employed the term indistinguishability relation $\approx$ in the reduction rule for operators $\mathbf{lam_X}$:

$$\mathbf{lam_X}\ \Phi\rho.e \quad\rightarrow\quad \Phi\rho.\lambda x.e$$

The contractum always retains the parameter specification ρ intact. We could have opted to remove the element $\mathbf{x}\!:\!x$, if there is any, from ρ since x is no longer free in $\lambda x.e$. The decision is immaterial because the choices are indistinguishable in $\boldsymbol{\lambda}\mathbf{C}$.

In summary, for any free identifier abstraction $\Phi\rho.e$ we can remove from or add to ρ parameters that are not referenced in e without changing the abstraction's behavior. Moreover, as $\Phi\rho.e$ optimizes to $\Phi\rho.e'$ because of the reduction of e to e', we can further remove from $\Phi\rho.e'$ the parameters eliminated in the optimization process. Indeed, for each free identifier abstraction $\Phi\rho.e$, there is a canonical form $\Phi\rho'.e$ that satisfies the following condition:

$$\Phi\rho'.e \approx \Phi\rho.e \text{ such that for each } \mathbf{x}\!:\!x \in \rho',\ x \in \mathit{fv}(e)$$

It is the indistinguishable free identifier abstraction of $\Phi\rho.e$ that has no redundant parameters.

It is conceivable to add as a new notion of reduction to the $\boldsymbol{\lambda}\mathbf{C}$-calculus to reduce every free identifier abstraction to its canonical form. We have opted not to do so, but to state the equivalence relation on free identifier abstractions strictly beyond the $\boldsymbol{\lambda}\mathbf{C}$-calculus for one subjective reason. Such a notion of equivalence has more to do with the optimization of compiled code, but very little to do with incremental compiled code construction, the theme of our work.

Either way, when we add new compiled code operations to $\boldsymbol{\lambda}\mathbf{C}$ in the future, we must carefully maintain the indistinguishability property of free identifier abstractions. That means there are some seemingly harmless operations that we must reject. A counterexample is the addition of the following predicate that tells whether some identifier $\mathbf{x}$ is specified in a free identifier abstraction:

$$\mathbf{x?}\ \Phi\rho.e \quad\rightarrow\quad \begin{cases} T & \text{if } \mathbf{x}\!:\!x \in \rho \\ F & \text{otherwise} \end{cases}$$

With it, $\Phi\{\}.\lambda y.y$ and $\Phi\{\mathbf{y}:y\}.\lambda y.y$ would no longer be indistinguishable since we would have the following incompatible behavior:

$$\begin{array}{lll} \mathbf{y}?\ \Phi\{\}.\lambda y.y & \qquad & \mathbf{y}?\ \Phi\{\mathbf{y}:y\}.\lambda y.y \\ \quad \rightarrow \ F & & \quad \rightarrow \ T \end{array}$$

3.6 Transparency

Another novelty of $\boldsymbol{\lambda}\mathbf{C}$ is the *transparency* of free identifier abstractions. A function abstraction $\lambda x.e$ is opaque in the sense that there is no way to access its body e until the abstractor λx is peeled off via function invocation. In contrast, a compiled code abstraction $\Phi\rho.e$ is transparent. We can operate on its body e without having to load the compiled code first. That is, the barriers set up by free identifier abstractors are not as rigid as λ-abstractors. Metaphorically speaking, functions are running machine code; they should therefore be treated as black boxes. In contrast, free identifier abstractions model compiled code; to link them together, we should have access to their "representation."

As an example showing the distinction between opaque λ-abstractions and transparent Φ-abstractions, consider taking the sum of two binary functions(we use infix notation for arithmetic operations):

$$f + g \qquad \text{where} \qquad \begin{array}{rcl} f & \equiv & \lambda xy.\sqrt{x * y} \\ g & \equiv & \lambda xy.\sqrt{x^2 + y^2} \end{array}$$

It makes no sense since the arguments to $+$ are of the wrong type. A quick fix is to define a new addition operator

$$\hat{+} \ \equiv \ \lambda fg.\lambda xy.((f\ x\ y) + (g\ x\ y))$$

so that

$$\begin{array}{rcl} f \mathbin{\hat{+}} g & \twoheadrightarrow & \lambda xy.(\underline{((\lambda xy.\sqrt{x * y})\ x\ y)} + \underline{((\lambda xy.\sqrt{x^2 + y^2})\ x\ y)}) \\ & \twoheadrightarrow & \lambda xy.(\sqrt{x * y} + \sqrt{x^2 + y^2}) \end{array}$$

The solution is not universal. $\hat{+}$ does not work with functions of different numbers of parameters such as $\lambda xy.\sqrt{x*y}$ and $\lambda x.\sqrt{x+3}$. Indeed, each of the many possible combinations would require a distinct version of $\hat{+}$.

Transparency does not allow us to take the sum of two Φ-abstractions either:

$$f+g \qquad \text{where} \qquad \begin{array}{rcl} f & \equiv & \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.\sqrt{x*y} \\ g & \equiv & \Phi\{\mathbf{x}\!:\!x\}.\sqrt{x^2+y^2} \end{array}$$

The arguments to $+$ are still of the wrong type. We can define on top of $+$ an operator $+^1$ to compute the sum of the bodies of its Φ-abstraction arguments as follows, however:

$$+^1 \;\; \equiv \;\; \lambda fg.(\mathbf{app}\ (\mathbf{app}\ \Phi\{\}.\lambda xy.(x+y)\ f)\ g)$$

so that

$$\begin{array}{lll} f +^1 g & & \\ \twoheadrightarrow & \mathbf{app}\ \underline{(\mathbf{app}\ \Phi\{\}.\lambda xy.(x+y)\ \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.\sqrt{x*y})}\ \Phi\{\mathbf{x}\!:\!x\}.\sqrt{x^2+y^2} & (3.1) \\ \rightarrow & \underline{\mathbf{app}\ \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.((\lambda xy.(x+y))\ \sqrt{x*y})\ \Phi\{\mathbf{x}\!:\!x\}.\sqrt{x^2+y^2}} & (3.2) \\ \rightarrow & \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!z\}.\underline{((\lambda xy.(x+y))\ \sqrt{x*z}\ \sqrt{x^2+y^2})} & \\ \twoheadrightarrow & \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!z\}.(\sqrt{x*z}+\sqrt{x^2+y^2}) & \end{array}$$

The key to transparent free identifier abstractions is the compiled code operator **app**. We can use it to "skip over" free identifier abstractors, as exemplified by the two reduction steps 3.1 and 3.2 above. Unlike $\hat{+}$, $+^1$ works for free identifier abstractions with an arbitrary number of parameters. In addition, it has no dependency on the orientation of the parameters.

Thus far, we have focused on making operators transparent. The same technique can also be used to make operands transparent. In particular, to supply e as an argument to the operator f embedded in a free identifier abstraction, $\Phi\rho.f$, we may first embed e in a free identifier abstraction with an arid parameter specification, $\Phi\{\}.e$, and then use **app** to construct the desired application:

$$\mathbf{app}\ \Phi\rho.f\ \Phi\{\}.e \;\; \rightarrow \;\; \Phi\rho.(f\ e)$$

To illustrate, we can define for each identifier $\mathbf{x}$ a binding operator $\mathit{beta}_{\mathbf{x}}$ as follows:

$$\mathit{beta}_{\mathbf{x}} \equiv \lambda xy.(\mathbf{app}\ (\mathbf{lam}_{\mathbf{x}}\ y)\ \Phi\{\}.x)$$

It binds the parameter $\mathbf{x}$ of the free identifier abstraction denoted by y to the denotation referred to by x. Hence,

$$\begin{array}{rcl} \mathit{beta}_{\mathbf{x}}\ e_1\ \Phi\rho.e_2 & \twoheadrightarrow & \mathbf{app}\ \underline{(\mathbf{lam}_{\mathbf{x}}\ \Phi\rho.e_2)}\ \Phi\{\}.e_1 \\ & \rightarrow & \underline{\mathbf{app}\ \Phi\rho.\lambda x.e_2\ \Phi\{\}.e_1} \quad [\mathbf{x}\!:\!x \in \rho \text{ or else } x \text{ is fresh}] \\ & \rightarrow & \Phi\rho.\underline{((\lambda x.e_2)\ e_1)} \\ & \rightarrow & \Phi\rho.[e_1/x]e_2 \end{array}$$

That is, each parameter variable of $\mathbf{x}$ in e_2 gets the denotation e_1.

Since $\mathit{beta}_{\mathbf{x}}$ is an operator, we can make it transparent by defining on top of it the following operator:

$$\mathit{beta}^1_{\mathbf{x}} \equiv \lambda xy.(\mathbf{app}\ (\mathbf{app}\ \Phi\{\}.\mathit{beta}_{\mathbf{x}}\ x)\ y)$$

such that

$$\begin{array}{rcl} \mathit{beta}^1_{\mathbf{x}}\ \Phi\rho_1.e_1\ \Phi\rho_2.e_2 & \twoheadrightarrow & \mathbf{app}\ \underline{(\mathbf{app}\ \Phi\{\}.\mathit{beta}_{\mathbf{x}}\ \Phi\rho_1.e_1)}\ \Phi\rho_2.e_2 \\ & \rightarrow & \underline{\mathbf{app}\ \Phi\rho_1.(\mathit{beta}_{\mathbf{x}}\ e_1)\ \Phi\rho_2.e_2} \\ & \rightarrow & \Phi\rho_1 \uplus \rho_2.(\mathit{beta}_{\mathbf{x}}\ e_1\ e_2) \end{array}$$

In general, we can define for any operator f_n of arity n a version f^1_n that is transparent to one level of free identifier abstractors. But since f^1_n itself is an operator, we can apply the same technique to yield a version f^2_n that is transparent to two levels of free identifier abstractors, and so forth.

3.7 Additional Non-Binding Constructs

The applicability of our incremental compiled code construction capability enhancing schema can be easily extended to additional language constructs that do not involve variable binding semantics.

For instance, we may decide to add conditionals to the pure λ-calculus later on:

$$e \ ::= \ x \mid \lambda x.e \mid e\ e \mid \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e$$

The contexts of the above extended λ-calculus are:

$$\mathbf{C} \ ::= \ \mathbf{h} \mid \mathbf{x} \mid \lambda\mathbf{x}.\mathbf{C} \mid \mathbf{C}\ \mathbf{C} \mid \mathbf{if}\ \mathbf{C}\ \mathbf{then}\ \mathbf{C}\ \mathbf{else}\ \mathbf{C}$$

The context-enriched version of the extended λ-calculus is $\boldsymbol{\lambda}\mathbf{C}$ extended with the same conditionals and a new compiled code construction operator **if**:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e\ e \mid \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e \mid \Phi\rho.e \mid \boldsymbol{\delta} \\ \boldsymbol{\delta} & ::= & \mathbf{load} \mid \tilde{\mathbf{x}} \mid \mathbf{lam}_{\mathbf{X}} \mid \mathbf{app} \mid \mathbf{if} \end{array}$$

The new operator **if** models the incremental construction of the compiled code of **if** $\mathbf{e}_1$ **then** $\mathbf{e}_2$ **else** $\mathbf{e}_3$ from the compiled code of $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$. The reduction rule is:

$$\mathbf{if}\ \Phi\rho_1.e_1\ \Phi\rho_2.e_2\ \Phi\rho_3.e_3 \ \rightarrow \ \Phi\rho_1\uplus\rho_2\uplus\rho_3.(\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3)$$

Analogous to the **app**-rule for constructing application terms, the union $\rho_1\uplus\rho_2\uplus\rho_3$ ensures that the same free identifier in $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$ is given the same temporary placeholder in $\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3$. Hence, new compilation constructs can be added to the $\boldsymbol{\lambda}\mathbf{C}$-calculus in a modular fashion to accommodate additional non-variable binding λ-terms.

As another example, we may extend the pure λ-calculus with sequences (lists):

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e\ e \mid s \mid \mathit{cons} \mid \mathit{hd} \mid \mathit{tl} \\ s & ::= & [e_1, \ldots, e_n] \end{array}$$

The notation $[e_1, \ldots, e_n]$ denotes a sequence whose first element is e_1 and whose last element is e_n. The empty sequence is $[]$. The operation $\mathit{cons}\ e\ s$ builds a new sequence whose first element is e and the rest is the same as the sequence s. The operation $\mathit{hd}\ s$ yields the first element of the non-empty sequence s. The other operation $\mathit{tl}\ s$ yields the non-empty sequence s excluding its first element.

The contexts of the above extended λ-calculus are:

$$\begin{array}{rcl} \mathbf{C} & ::= & \mathbf{h} \mid \mathbf{x} \mid \lambda\mathbf{x}.\mathbf{C} \mid \mathbf{C}\ \mathbf{C} \mid \mathbf{s} \mid \mathbf{cons} \mid \mathbf{hd} \mid \mathbf{tl} \\ \mathbf{s} & ::= & [\mathbf{C}_1, \ldots, \mathbf{C}_n] \end{array}$$

The context-enriched version of the extended λ-calculus is $\boldsymbol{\lambda}\mathbf{C}$ extended with the same sequence operations:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e\ e \mid s \mid \mathit{cons} \mid \mathit{hd} \mid \mathit{tl} \mid \Phi\rho.e \mid \boldsymbol{\delta} \\ \boldsymbol{\delta} & ::= & \mathbf{load} \mid \tilde{\mathrm{x}} \mid \mathbf{lam}_{\mathbf{X}} \mid \mathbf{app} \end{array}$$

The compiled code of the contexts **cons**, **hd**, and **tl** are $\Phi\{\}.\mathit{cons}$, $\Phi\{\}.\mathit{hd}$, and $\Phi\{\}.\mathit{tl}$. The compiled code of **cons e s** can be incrementally constructed out of the compiled code $\Phi\rho.e$ of **e** and the compiled code $\Phi\rho'.s$ of **s** using the transparent *cons*-operator as follows:

$$\begin{array}{rcl} \mathbf{app}\ \underline{(\mathbf{app}\ \Phi\{\}.\mathit{cons}\ \Phi\rho.e)}\ \Phi\rho'.s & = & \underline{\mathbf{app}\ \Phi\rho.(\mathit{cons}\ e)\ \Phi\rho'.s} \\ & = & \Phi\rho\uplus\rho'.(\mathit{cons}\ e\ s) \end{array}$$

The incremental construction of the other categories of sequence operation can be handled in a similar fashion.

Hence, we have shown that adding sequences before or after the application of our schema to the pure λ-calculus produces the same result. In other words, our context-enriching schema is orthogonal to the enrichment introduced by applied λ-calculi. In particular, for whatever basic constants and datatype constructors we may add to the λ-calculus, we have their transparent versions for free, which is intuitively sensible since contexts are a notion orthogonal to basic constants and datatype constructors, which have no free variables. Indeed, transparency is what allows us to focus our schema on the three basic λ-terms of the pure λ-calculus without worrying about its applicability to any additional constructs introduced by applied λ-calculi.

3.8 Programming Examples

The incremental compiled code construction capabilities of $\boldsymbol{\lambda}\mathbf{C}$ allow us to express many practical programming mechanisms more intuitively than the λ-calculus. In this section we describe two, namely, program symbols and first-class environments.

3.8.1 Program Symbols

Program symbols are names used in source code; variables are names used in machine code; identifiers are names used in compiled code to relate variables to program symbols. Although we do not include identifiers $\mathbf{x}$ as proper terms of $\lambda\mathbf{C}$, we can simulate their behavior when used as program symbols.

A program symbol $\mathbf{x}$ can be encoded as the following pair:

$$\langle \mathbf{lam}_{\mathbf{x}}, \Phi\{\mathbf{x}\!:\!x\}.x\rangle$$

Intuitively, we use a program symbol $\mathbf{x}$ either as the syntactic representation of the parameter of a λ-abstraction or as the syntactic representation of a variable reference. The functionalities are covered by $\mathbf{lam}_{\mathbf{x}}$ and $\Phi\{\mathbf{x}\!:\!x\}.x$, explaining our choice. In the following we write $\mathbf{x}$ for the program symbol represented by such a pair and denote the two components $\mathbf{lam}_{\mathbf{x}}$ and $\Phi\{\mathbf{x}\!:\!x\}.x$ as $\mathit{lam}(\mathbf{x})$ and $\mathit{phi}(\mathbf{x})$, respectively.

The predicate *eq?* that determines the equality of two such program symbols is $\lambda\mathbf{C}$-definable as follows:

$$\mathit{eq?} \;\equiv\; \lambda xy.(\mathbf{load}\ (\mathit{lam}(x)\ (\mathit{lam}(y)\ \mathit{phi}(x)))\ F\ T)$$

where $T \equiv \lambda x.\lambda y.x$ and $F \equiv \lambda x.\lambda y.y$ are the boolean values of true and false, respectively. When $\mathbf{x}$ and $\mathbf{y}$ are distinct program symbols, we have

$$\begin{array}{rll}
\mathit{eq?}\,\mathbf{x}\,\mathbf{y} & \twoheadrightarrow & \mathbf{load}\ (\mathit{lam}(\mathbf{x})\ (\mathit{lam}(\mathbf{y})\ \mathit{phi}(\mathbf{x})))\ F\ T \\
 & \twoheadrightarrow & \mathbf{load}\ (\mathbf{lam}_{\mathbf{x}}\ \underline{(\mathbf{lam}_{\mathbf{y}}\ \Phi\{\mathbf{x}\!:\!x\}.x)})\ F\ T \\
 & \rightarrow & \mathbf{load}\ \underline{(\mathbf{lam}_{\mathbf{x}}\ \Phi\{\mathbf{x}\!:\!x\}.\lambda y.x)}\ F\ T \qquad [x \not\equiv y] \\
 & \rightarrow & \underline{\mathbf{load}\ \Phi\{\mathbf{x}\!:\!x\}.\lambda x.\lambda y.x}\ F\ T
\end{array}$$

$$\begin{aligned}
&\rightarrow \underline{(\lambda x.\lambda y.x)\ F\ T} \\
&\twoheadrightarrow F
\end{aligned}$$

On the other hand,

$$\begin{aligned}
\mathit{eq?}\ \mathbf{x}\ \mathbf{x} &\twoheadrightarrow \textbf{load}\ (\mathit{lam}(\mathbf{x})\ (\mathit{lam}(\mathbf{x})\ \mathit{phi}(\mathbf{x})))\ F\ T \\
&\twoheadrightarrow \textbf{load}\ (\mathbf{lam}_{\mathbf{x}}\ \underline{(\mathbf{lam}_{\mathbf{x}}\ \Phi\{\mathbf{x}{:}x\}.x)})\ F\ T \\
&\rightarrow \textbf{load}\ \underline{(\mathbf{lam}_{\mathbf{x}}\ \Phi\{\mathbf{x}{:}x\}.\lambda x.x)}\ F\ T \\
&\rightarrow \underline{\textbf{load}\ \Phi\{\mathbf{x}{:}x\}.\lambda x.\lambda x.x}\ F\ T \\
&\rightarrow \underline{(\lambda x.\lambda x.x)\ F\ T} \\
&\twoheadrightarrow T
\end{aligned}$$

With program symbols, many common symbol-related programming mechanisms can be encoded in $\boldsymbol{\lambda}\mathbf{C}$. In the following section we demonstrate one such programming mechanism, namely, first-class environments [47].

3.8.2 First-Class Environments

Our notion of an environment is a finite function mapping program symbols $\mathbf{x}$ to compiled code $\Phi\rho.e$ represented as the sequence

$$[\langle \mathbf{x}_1, \Phi\rho_1.e_1\rangle, \ldots, \langle \mathbf{x}_n, \Phi\rho_n.e_n\rangle]$$

where each pair $\langle \mathbf{x}_i, \Phi\rho_i.e_i\rangle$ models the binding of the symbol $\mathbf{x}_i$ with its denotation $\Phi\rho_i.e_i$.

The bindings of an environment can be imported into a piece of compile code $\Phi\rho.e$ using an operator *link* with the following behavior:

$$\begin{aligned}
&\mathit{link}\ [\langle \mathbf{x}_1, \Phi\rho_1.e_1\rangle, \ldots, \langle \mathbf{x}_n, \Phi\rho_n.e_n\rangle]\ \Phi\rho.e \\
&= \Phi\rho\uplus\rho_1\uplus\cdots\uplus\rho_n.((\lambda x_n \cdots x_1.e)\ e_n \cdots e_1) \\
&= \Phi\rho\uplus\rho_1\uplus\cdots\uplus\rho_n.[e_n/x_n]\cdots[e_1/x_1]e
\end{aligned}$$

where each x_i is the parameter variable corresponding to $\mathbf{x}_i$ specified in ρ or else it is fresh. Hence, we can define a run-time evaluator *eval* in the style of Lisp [66] and MIT Scheme [47] as follows:

$$\mathit{eval} \quad \equiv \quad \lambda xy.(\mathbf{load}\ (\mathit{link}\ y\ x))$$

where the arguments x and y should denote a piece of compiled code and a first-class environment, respectively.

The environment importing operator *link* has a straightforward recursive definition (pattern matching is used to simplify the presentation):

$$\begin{aligned} \mathit{link}\ []\ e' &= e' \\ \mathit{link}\ \langle \mathbf{x}, e\rangle :: s\ e' &= \mathbf{app}\ (\mathit{link}\ s\ (\mathit{lam}(\mathbf{x})\ e'))\ e \end{aligned}$$

The notation [] denotes the empty sequence; $\langle \mathbf{x}, e\rangle :: s$ is the sequence whose first element is $\langle \mathbf{x}, e\rangle$ and the remainder of the sequence is s.

The operator *link* provides a means to use the bindings of an environment. There should also be mechanisms to construct environments. In Lisp's terminology, our environments are association lists whose keys are program symbols and whose data are compiled code. Once we realize the analogy, there are more ways to build environments than we have room to describe. Here, we sketch just one such operator *rec* that constructs recursive environments. The definition of *rec* is given in Figure 3.3. We have omitted the derivation of the combinators A_n and B_n to make the complexity of the definition more manageable.

The behavior of *rec* is explained below. (The following derivations are routine examples involving list processing; the reader who is comfortable with transparent free identifier abstraction operations may wish to skip to the end of the section.) Let

$$D \quad = \quad [\langle \mathbf{x}_1, \Phi\rho_1.e_1\rangle, \ldots, \langle \mathbf{x}_n, \Phi\rho_n.e_n\rangle]$$

be an environment such that $\rho \equiv \rho_1 \uplus \cdots \uplus \rho_n$ is well-formed. (If not, one can always use α-conversion to rename the parameter variables of $\rho_1, \ldots, \rho_n$ so that ρ meets the

$$\mathit{rec}\ D \;=\; H\ D\ (B_n\ (\mathbf{app}\ \Phi\{\}.A_n\ (G\ (E\ D)\ D)))$$

with

$$\begin{aligned}
A_n &= \lambda f.(\mathit{fix}\ \lambda s.(f\ (\pi_n\ s) \cdots (\pi_1\ s))) \\
&\qquad \text{where } \pi_i = hd \circ \underbrace{tl \circ \cdots \circ tl}_{i-1} \\
B_n &= \lambda s.[(\pi_1^1\ s), \ldots, (\pi_n^1\ s)] \\
&\qquad \text{where } \pi_i^1 \equiv \lambda x.(\mathbf{app}\ \Phi\{\}.\pi_i\ x)
\end{aligned}$$

$$\begin{aligned}
E\ [] &= \Phi\{\}.[] \\
E\ \langle \mathbf{x}, e\rangle :: s &= \mathbf{app}\ (\mathbf{app}\ \Phi\{\}.\mathit{cons}\ e)\ (E\ s)
\end{aligned}$$

$$\begin{aligned}
G\ e'\ [] &= e' \\
G\ e'\ \langle \mathbf{x}, e\rangle :: s &= G\ (\mathit{lam}(\mathbf{x})\ e')\ s
\end{aligned}$$

$$\begin{aligned}
H\ []\ [] &= [] \\
H\ \langle \mathbf{x}, e\rangle :: s\ e' :: s' &= \langle \mathbf{x}, e'\rangle :: (H\ s\ s')
\end{aligned}$$

Figure 3.3: Constructing recursive first-class environments

requirement.) Then, we can show by induction that

$$E\ D \;=\; \Phi\rho.[e_1, \ldots, e_n] \tag{3.3}$$

Specifically,

$$\begin{aligned}
E\ D &= \mathbf{app}\ (\mathbf{app}\ \Phi\{\}.\mathit{cons}\ \Phi\rho_1.e_1)\ (E\ [\langle \mathbf{x}_2, \Phi\rho_2.e_2\rangle, \ldots, \langle \mathbf{x}_n, \Phi\rho_n.e_n\rangle]) \\
&= \mathbf{app}\ \Phi\rho_1.(\mathit{cons}\ e_1)\ \Phi\rho_2 \uplus \cdots \uplus \rho_n.[e_2, \ldots, e_n] \quad \text{[induction hypothesis]} \\
&= \Phi\rho_1 \uplus \rho_2 \uplus \cdots \uplus \rho_n.(\mathit{cons}\ e_1\ [e_2, \ldots, e_n]) \\
&= \Phi\rho_1 \uplus \rho_2 \uplus \cdots \uplus \rho_n.[e_1, e_2, \ldots, e_n]
\end{aligned}$$

With Equality 3.3, we can next verify that

$$\begin{aligned}
G\ (E\ D)\ D &= \mathit{lam}(\mathbf{x}_n)\ (\cdots\ (\mathit{lam}(\mathbf{x}_1)\ (E\ D))) \\
&= \mathbf{lam}_{\mathbf{x}_n}\ (\cdots\ (\mathbf{lam}_{\mathbf{x}_1}\ \Phi\rho.[e_1, \ldots, e_n])) \\
&= \Phi\rho.\lambda x_n \cdots x_1.[e_1, \ldots, e_n] \qquad (3.4)
\end{aligned}$$

where $x_1, \ldots, x_n$ are the parameter variables of $\mathbf{x}_1, \ldots, \mathbf{x}_n$ in ρ. That is, for each x_i, either $\mathbf{x}_i\!:\!x_i \in \rho$ or else x_i is fresh with respect to $[e_1, \ldots, e_n]$.

With Equality 3.4, we have the following derivation:

$$\begin{aligned}
\mathbf{app}\ \Phi\{\}.A_n\ (G\ (E\ D)\ D) &= \mathbf{app}\ \Phi\{\}.A_n\ \Phi\rho.\lambda x_n \cdots x_1.[e_1, \ldots, e_n] \\
&= \Phi\rho.(A_n\ \lambda x_n \cdots x_1.[e_1, \ldots, e_n]) \\
&= \Phi\rho.(\mathit{fix}\ \lambda s.((\lambda x_n \cdots x_1.[e_1, \ldots, e_n])\ (\pi_n\ s) \cdots (\pi_1\ s))) \\
&= \Phi\rho.S \qquad (3.5)
\end{aligned}$$

where S is the following recursively defined sequence:

$$\begin{aligned}
S &= \mathit{fix}\ \lambda s.((\lambda x_n \cdots x_1.[e_1, \ldots, e_n])\ (\pi_n\ s) \cdots (\pi_1\ s)) \\
&= (\lambda x_n \cdots x_1.[e_1, \ldots, e_n])\ (\pi_n\ S) \cdots (\pi_1\ S) \\
&= [([(\pi_1\ S)/x_1] \cdots [(\pi_n\ S)/x_n]e_1), \ldots, ([(\pi_1\ S)/x_1] \cdots [(\pi_n\ S)/x_n]e_n)]
\end{aligned}$$

Hence, we have the elements of the environment D recursively available to one another.

With Equality 3.5, next we form the sequence

$$\begin{aligned} B_n\ (\mathbf{app}\ \Phi\{\}.A_n\ (G\ (E\ D)\ D)) &= [(\pi_1^1\ \Phi\rho.S), \ldots, (\pi_n^1\ \Phi\rho.S)] \\ &= [\Phi\rho.(\pi_1\ S), \ldots, \Phi\rho.(\pi_n\ S)] \end{aligned} \tag{3.6}$$

where $\pi_i\ S = [(\pi_1\ S)/x_1] \cdots [(\pi_n\ S)/x_n]e_i$ is the ith element of S. Using the sequence of Equality 3.6 and the symbols of the environment D, the desired recursive environment can be constructed as follows:

$$\begin{aligned} \mathit{rec}\ D &= H\ D\ (B_n\ (\mathbf{app}\ \Phi\{\}.A_n\ (G\ (E\ D)\ D))) \\ &= H\ [\langle \mathbf{x}_1, \Phi\rho_1.e_1\rangle, \ldots, \langle \mathbf{x}_n, \Phi\rho_n.e_n\rangle]\ [\Phi\rho.(\pi_1\ S), \ldots, \Phi\rho.(\pi_n\ S)] \\ &= [\langle \mathbf{x}_1, \Phi\rho.(\pi_1\ S)\rangle, \ldots, \langle \mathbf{x}_n, \Phi\rho.(\pi_n\ S)\rangle] \end{aligned}$$

Put together, the above definition of *rec* is apparently way too complicated to be desirable. Programming language design is about giving a clean and straightforward description for recurring programming idioms. In Chapter 6 we present a more intuitive view of first-class environments.

Chapter 4

Simulations of Lambda Contexts

We have argued that our schema is a framework for enhancing a calculus with the expressiveness of contexts. It is time to verify that $\boldsymbol{\lambda}\mathbf{C}$ is indeed sufficient to simulate the behavior of λ-contexts. We show two simulations. The first simulation is an incremental compiler for the λ-calculus. An evolved λ-context, which is the source code of a λ-calculus program, is translated into a free identifier abstraction, our representation of compiled code. A partially-evolved λ-context, on the other hand, is an incremental compilation operator. It is encoded as the composition of the two compiled code construction operators of $\boldsymbol{\lambda}\mathbf{C}$, namely, $\mathbf{lam}_\mathbf{X}$ and $\mathbf{app}$. The second simulation is an incremental linker for compiled code. It does not distinguish between evolved and partially-evolved λ-contexts. Both are encoded as free identifier abstractions, *i.e.*, they are fully compiled. The simulation then relies on annotating each hole with a linker that maps identifiers (source code symbols) to variables (machine code locations) to model the identifier capture capabilities of hole filling. Together, the simulations fulfill our promise that adding the notion of contexts enhances a language with incremental compiled code construction capabilities.

4.1 Notions and Notations

We begin with notions and notations used in the simulations. To simplify the presentation, we assume that there is a one-to-one function q that assigns a distinct variable name $q(\mathbf{n})$ to each name $\mathbf{n}$, which is either a hole $\mathbf{h}$ or an identifier $\mathbf{x}$.

The λ-contexts $\mathbf{C}$ are:

$$\mathbf{C} \quad ::= \quad \mathbf{h} \mid \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C} \mid \mathbf{C}\ \mathbf{C}$$

They are either evolved $\mathbf{e}$ or partially-evolved $\mathbf{C}^{+}$:

$$\begin{array}{rcl} \mathbf{e} & ::= & \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{e} \mid \mathbf{e}\ \mathbf{e} \\ \mathbf{C}^{+} & ::= & \mathbf{h} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C}^{+} \mid \mathbf{e}\ \mathbf{C}^{+} \mid \mathbf{C}^{+}\ \mathbf{e} \mid \mathbf{C}^{+}\ \mathbf{C}^{+} \end{array}$$

The *image* of a λ-context $\mathbf{C}$ is a λ-term $im(\mathbf{C})$ defined inductively on the structure of $\mathbf{C}$ as follows:

$$\begin{array}{rcl} im(\mathbf{h}) & \equiv & q(\mathbf{h}) \\ im(\mathbf{x}) & \equiv & q(\mathbf{x}) \\ im(\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}) & \equiv & \lambda q(\mathbf{x}).im(\mathbf{C}) \\ im(\mathbf{C}_1\ \mathbf{C}_2) & \equiv & im(\mathbf{C}_1)\ im(\mathbf{C}_2) \end{array}$$

That is, $im(\mathbf{C})$ has the same structure as $\mathbf{C}$ but with each name $\mathbf{n}$ replaced by $q(\mathbf{n})$.

The set of holes *occurring* in a λ-context $\mathbf{C}$, $oh(\mathbf{C})$, is defined inductively on the structure of $\mathbf{C}$ as follows:

$$\begin{array}{rcl} oh(\mathbf{h}) & = & \{\mathbf{h}\} \\ oh(\mathbf{x}) & = & \emptyset \\ oh(\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}) & = & oh(\mathbf{C}) \\ oh(\mathbf{C}_1\ \mathbf{C}_2) & = & oh(\mathbf{C}_1) \cup oh(\mathbf{C}_2) \end{array}$$

Let $\{\mathbf{h}_1, \ldots, \mathbf{h}_n\} = oh(\mathbf{C})$ and $h_i = q(\mathbf{h}_i)$ be the unique variable name of $\mathbf{h}_i$. We write $\rho^{\mathbf{h}}_{\mathbf{c}}$, $\{\bar{\mathbf{h}}_{\mathbf{c}} : q(\bar{\mathbf{h}}_{\mathbf{c}})\}$, or $\{\bar{\mathbf{h}}_{\mathbf{c}} : \bar{h}_{\mathbf{c}}\}$ for the free identifier abstraction parameter specification $\{\mathbf{h}_1 : h_1, \ldots, \mathbf{h}_n : h_n\}$ generated from the holes $oh(\mathbf{C})$ occurring in the context $\mathbf{C}$.

The set of *applied* identifiers of a λ-context $\mathbf{C}$, $ai(\mathbf{C})$, is defined inductively on the structure of $\mathbf{C}$ as follows:

$$\begin{aligned} ai(\mathbf{h}) &= \emptyset \\ ai(\mathbf{x}) &= \{\mathbf{x}\} \\ ai(\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}) &= ai(\mathbf{C}) \\ ai(\mathbf{C}_1\ \mathbf{C}_2) &= ai(\mathbf{C}_1) \cup ai(\mathbf{C}_2) \end{aligned}$$

Let $ai(\mathbf{C})$ be $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ and $x_i = q(\mathbf{x}_i)$ be the unique variable name of $\mathbf{x}_i$. We then write $\rho^{\mathbf{a}}_{\mathbf{c}}$ for the free identifier abstraction parameter specification $\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}$.

The set of *free* identifiers in a λ-context $\mathbf{C}$, $\mathit{fi}(\mathbf{C})$, is defined inductively on the structure of $\mathbf{C}$ as follows ($\mathit{fi}(\mathbf{C})$ is always a subset of $ai(\mathbf{C})$):

$$\begin{aligned} \mathit{fi}(\mathbf{h}) &= \emptyset \\ \mathit{fi}(\mathbf{x}) &= \{\mathbf{x}\} \\ \mathit{fi}(\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}) &= \mathit{fi}(\mathbf{C}) \setminus \{\mathbf{x}\} \\ \mathit{fi}(\mathbf{C}_1\ \mathbf{C}_2) &= \mathit{fi}(\mathbf{C}_1) \cup \mathit{fi}(\mathbf{C}_2) \end{aligned}$$

Let $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} = \mathit{fi}(\mathbf{C})$ and $x_i = q(\mathbf{x}_i)$ be the unique variable name of $\mathbf{x}_i$. We write $\rho^{\mathbf{x}}_{\mathbf{c}}$, $\{\bar{\mathbf{x}}_{\mathbf{c}} : q(\bar{\mathbf{x}}_{\mathbf{c}})\}$, or $\{\bar{\mathbf{x}}_{\mathbf{c}} : \bar{x}_{\mathbf{c}}\}$ for the free identifier abstraction parameter specification $\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}$ generated from $\mathit{fi}(\mathbf{C})$.

Define for each hole $\mathbf{h}$ a binding operator $\mathit{bind}_{\mathbf{h}}$

$$\mathit{bind}_{\mathbf{h}} \equiv \lambda xy.(\mathbf{app}\ (\mathbf{lam}_{\mathbf{h}}\ y)\ x) \tag{4.1}$$

with the following intended behavior:

$$\begin{aligned} \mathit{bind}_{\mathbf{h}}\ \Phi\rho_1.e_1\ \Phi\rho_2.e_2 &\twoheadrightarrow \mathbf{app}\ (\mathbf{lam}_{\mathbf{h}}\ \Phi\rho_2.e_2)\ \Phi\rho_1.e_1 \\ &\twoheadrightarrow \Phi\rho_1 \uplus \rho_2.((\lambda h.e_2)\ e_1) \quad [\mathbf{h}{:}h \in \rho_2 \text{ or else } h \text{ is fresh}] \\ &\rightarrow \Phi\rho_1 \uplus \rho_2.[e_1/h]e_2 \end{aligned}$$

That is, each free occurrence of the parameter variable h corresponding to $\mathbf{h}$ in e_2 is replaced by the denotation e_1.

4.2 First Simulation

The first encoding of λ-contexts $\mathcal{R}_1$ is defined as follows:

$$\mathcal{R}_1(\mathbf{C}) \equiv \Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\mathcal{T}(\mathbf{C})$$

A λ-context $\mathbf{C}$ is represented as the free identifier abstraction $\Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\mathcal{T}(\mathbf{C})$ where $\rho^{\mathbf{h}}_{\mathbf{c}}$ is the Φ-parameterization of the holes occurring in $\mathbf{C}$ and the transformation $\mathcal{T}(\mathbf{C})$ is defined inductively on the structure of $\mathbf{C}$ as follows:

$$\begin{aligned} \mathcal{T}(\mathbf{h}) &\equiv q(\mathbf{h}) \\ \mathcal{T}(\mathbf{x}) &\equiv \Phi\{\mathbf{x}:q(\mathbf{x})\}.q(\mathbf{x}) \\ \mathcal{T}(\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}) &\equiv \mathbf{lam}_{\mathbf{x}}\ \mathcal{T}(\mathbf{C}) \\ \mathcal{T}(\mathbf{C}_1\ \mathbf{C}_2) &\equiv \mathbf{app}\ \mathcal{T}(\mathbf{C}_1)\ \mathcal{T}(\mathbf{C}_2) \end{aligned}$$

For instance, the λ-context $\boldsymbol{\lambda}\mathbf{x}.(\mathbf{h}\ (\mathbf{x}\ \mathbf{y}))$ is encoded as

$$\Phi\{\mathbf{h}:q(\mathbf{h})\}.(\mathbf{lam}_{\mathbf{x}}\ (\mathbf{app}\ q(\mathbf{h})\ (\mathbf{app}\ \Phi\{\mathbf{x}:q(\mathbf{x})\}.q(\mathbf{x})\ \Phi\{\mathbf{y}:q(\mathbf{y})\}.q(\mathbf{y})))) \tag{4.2}$$

The use of the function q in the definition of $\mathcal{R}_1$ is just a matter of convenience. The unique variable names that it produces are always bound in the resulting $\boldsymbol{\lambda}\mathbf{C}$-term $\mathcal{R}_1(\mathbf{C})$. They are therefore subject to α-conversion. Thus, the following encoding of $\boldsymbol{\lambda}\mathbf{x}.(\mathbf{h}\ (\mathbf{x}\ \mathbf{y}))$ is just as good:

$$\Phi\{\mathbf{h}:h\}.(\mathbf{lam}_{\mathbf{x}}\ (\mathbf{app}\ h\ (\mathbf{app}\ \Phi\{\mathbf{x}:x\}.x\ \Phi\{\mathbf{y}:y\}.y)))$$

Indeed, any $\Phi\rho.e$ that is indistinguishable from $\Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\mathcal{T}(\mathbf{C})$ with respect to the compiled code operations of $\boldsymbol{\lambda}\mathbf{C}$, $\Phi\rho.e \approx \Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\mathcal{T}(\mathbf{C})$, can serve as an encoding of $\mathbf{C}$.

The following theorem, whose proof is a straightforward induction on the structure of $\mathbf{e}$, shows that $\mathcal{T}$ transforms an evolved λ-context into its image:

Theorem 4.1 $\mathcal{T}(\mathbf{e}) \twoheadrightarrow \Phi\rho^{\mathbf{a}}_{\mathbf{e}}.im(\mathbf{e})$.

In other words, $\mathcal{T}$ compiles evolved λ-contexts. As an example, let $\mathbf{e}$ be $\boldsymbol{\lambda}\mathbf{x}.(\mathbf{x}\ \mathbf{y})$. Then,

$$\begin{aligned}
\mathcal{T}(\mathbf{e}) &\equiv \mathbf{lam_x}\ (\mathbf{app}\ \Phi\{\mathbf{x}\!:\!q(\mathbf{x})\}.q(\mathbf{x})\ \Phi\{\mathbf{y}\!:\!q(\mathbf{y})\}.q(\mathbf{y}))\\
&\equiv \mathbf{lam_x}\ \underline{(\mathbf{app}\ \Phi\{\mathbf{x}\!:\!x\}.x\ \Phi\{\mathbf{y}\!:\!y\}.y)}\\
&\rightarrow \underline{\mathbf{lam_x}\ \Phi\{\mathbf{x}\!:\!x,\mathbf{y}\!:\!y\}.(x\ y)}\\
&\rightarrow \Phi\{\mathbf{x}\!:\!x,\mathbf{y}\!:\!y\}.\lambda x.(x\ y)\\
&\equiv \Phi\{\mathbf{x}\!:\!q(\mathbf{x}),\mathbf{y}\!:\!q(\mathbf{y})\}.\lambda q(\mathbf{x}).(q(\mathbf{x})\ q(\mathbf{y}))\\
&\equiv \Phi\rho^{\mathbf{a}}_{\mathbf{e}}.im(\mathbf{e})
\end{aligned}$$

Based on Theorem 4.1 and that λ-contexts are either evolved or partially-evolved, we can redefine the transformation $\mathcal{T}$ as follows to emphasize that evolved λ-contexts can be compiled by the transformation:

$$\begin{aligned}
\mathcal{T}(\mathbf{e}) &\equiv \Phi\rho^{\mathbf{a}}_{\mathbf{e}}.im(\mathbf{e})\\
\mathcal{T}(\mathbf{h}) &\equiv q(\mathbf{h})\\
\mathcal{T}(\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}^+) &\equiv \mathbf{lam_x}\ \mathcal{T}(\mathbf{C}^+)\\
\mathcal{T}(\mathbf{C}^+\ \mathbf{e}) &\equiv \mathbf{app}\ \mathcal{T}(\mathbf{C}^+)\ \Phi\rho^{\mathbf{a}}_{\mathbf{e}}.im(\mathbf{e})\\
\mathcal{T}(\mathbf{e}\ \mathbf{C}^+) &\equiv \mathbf{app}\ \Phi\rho^{\mathbf{a}}_{\mathbf{e}}.im(\mathbf{e})\ \mathcal{T}(\mathbf{C}^+)\\
\mathcal{T}(\mathbf{C}^+_1\ \mathbf{C}^+_2) &\equiv \mathbf{app}\ \mathcal{T}(\mathbf{C}^+_1)\ \mathcal{T}(\mathbf{C}^+_2)
\end{aligned}$$

4.2.1 Simulating Hole Filling

Let $\mathbf{C}''$ be the result of filling the holes named $\mathbf{h}$ in the λ-context $\mathbf{C}$ with another λ-context $\mathbf{C}'$, $\mathbf{C}'' \equiv [\mathbf{C}'/\mathbf{h}]\mathbf{C}$. Then, the holes occurring in $\mathbf{C}''$ are a subset of the holes occurring in $\mathbf{C}$ and $\mathbf{C}'$ together, $oh(\mathbf{C}'') \subseteq oh(\mathbf{C}) \cup oh(\mathbf{C}')$. As a result, we have $\rho^{\mathbf{h}}_{\mathbf{c}''} \subseteq \rho^{\mathbf{h}}_{\mathbf{c}} \uplus \rho^{\mathbf{h}}_{\mathbf{c}'}$, a fact needed below.

With the $\mathcal{R}_1$-encoding of λ-contexts, hole filling can be accomplished as follows:

Theorem 4.2 *Let* $\Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\mathcal{T}(\mathbf{C})$, $\Phi\rho^{\mathbf{h}}_{\mathbf{c}'}.\mathcal{T}(\mathbf{C}')$, *and* $\Phi\rho^{\mathbf{h}}_{\mathbf{c}''}.\mathcal{T}(\mathbf{C}'')$ *be the encoding of* $\mathbf{C}$, $\mathbf{C}'$, *and* $\mathbf{C}''$*, respectively. Then,*

$$bind_{\mathbf{h}}\ \Phi\rho^{\mathbf{h}}_{\mathbf{c}'}.\mathcal{T}(\mathbf{C}')\ \Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\mathcal{T}(\mathbf{C}) \quad \approx \quad \Phi\rho^{\mathbf{h}}_{\mathbf{c}''}.\mathcal{T}(\mathbf{C}'')$$

Proof: Let $\mathbf{h}\!:\!h \in \rho^{\mathbf{h}}_{\mathbf{c}}$ or else h is fresh. Then,

$$\begin{array}{rcll}
\underline{bind_{\mathbf{h}}\ \Phi\rho^{\mathbf{h}}_{\mathbf{c}'}.\mathcal{T}(\mathbf{C}')\ \Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\mathcal{T}(\mathbf{C})} & \twoheadrightarrow & \mathbf{app}\ \underline{(\mathbf{lam}_{\mathbf{h}}\ \Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\mathcal{T}(\mathbf{C}))}\ \Phi\rho^{\mathbf{h}}_{\mathbf{c}'}.\mathcal{T}(\mathbf{C}') & \\
 & \rightarrow & \underline{\mathbf{app}\ \Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\lambda h.\mathcal{T}(\mathbf{C})\ \Phi\rho^{\mathbf{h}}_{\mathbf{c}'}.\mathcal{T}(\mathbf{C}')} & \\
 & \rightarrow & \Phi\rho^{\mathbf{h}}_{\mathbf{c}}\uplus\rho^{\mathbf{h}}_{\mathbf{c}'}.\underline{((\lambda h.\mathcal{T}(\mathbf{C}))\ \mathcal{T}(\mathbf{C}'))} & \\
 & \rightarrow & \Phi\rho^{\mathbf{h}}_{\mathbf{c}}\uplus\rho^{\mathbf{h}}_{\mathbf{c}'}.\underline{[\mathcal{T}(\mathbf{C}')/h]\mathcal{T}(\mathbf{C})} & \\
 & \equiv & \Phi\rho^{\mathbf{h}}_{\mathbf{c}}\uplus\rho^{\mathbf{h}}_{\mathbf{c}'}.\mathcal{T}(\underline{[\mathbf{C}'/\mathbf{h}]\mathbf{C}}) & (4.3) \\
 & \equiv & \Phi\underline{\rho^{\mathbf{h}}_{\mathbf{c}}\uplus\rho^{\mathbf{h}}_{\mathbf{c}'}}.\mathcal{T}(\mathbf{C}'') & \\
 & \approx & \Phi\rho^{\mathbf{h}}_{\mathbf{c}''}.\mathcal{T}(\mathbf{C}'') & (4.4)
\end{array}$$

Equality 4.3 follows from $\mathcal{T}([\mathbf{C}'/\mathbf{h}]\mathbf{C}) \equiv [\mathcal{T}(\mathbf{C}')/q(\mathbf{h})]\mathcal{T}(\mathbf{C})$, whose proof is a straightforward induction on the structure of $\mathbf{C}$. Equivalence 4.4 is valid since $\Phi\rho^{\mathbf{h}}_{\mathbf{c}}\uplus\rho^{\mathbf{h}}_{\mathbf{c}'}.\mathcal{T}(\mathbf{C}'')$ and $\Phi\rho^{\mathbf{h}}_{\mathbf{c}''}.\mathcal{T}(\mathbf{C}'')$ are indistinguishable in $\boldsymbol{\lambda}\mathbf{C}$ when the redundant parameters are removed from $\rho^{\mathbf{h}}_{\mathbf{c}}\uplus\rho^{\mathbf{h}}_{\mathbf{c}'}$. □

4.2.2 Compiling Lambda Contexts

Following our intuition that contexts are source code and terms are machine code, the transformation $\mathcal{T}$ qualifies as a compiler for evolved λ-contexts as is shown by Theorem 4.1. So, to express in $\boldsymbol{\lambda}\mathbf{C}$ a compiler for the λ-calculus, all we need then is an *abstract* source code representation [44] for λ-contexts:

- for the source code encoding of $\mathbf{e}$, there are predicates to tell whether $\mathbf{e}$ is an identifier, an abstraction, or an application;
- there are mechanisms to break the encoding of each composite evolved λ-context $\mathbf{e}$ down to (the encoding of) its components; and
- there is a predicate to determine the equality of identifiers via their encoding.

Based on the simulation of a program symbol $\mathbf{x}$ as the pair $\langle \mathbf{lam_x}, \Phi\{\mathbf{x}\!:\!x\}.x\rangle$ defined in Section 3.8.1, we can represent evolved λ-contexts $\mathbf{e}$ in $\boldsymbol{\lambda}\mathbf{C}$ as follows:

$$\begin{aligned}
[\![\mathbf{x}]\!]_c &\equiv \langle \mathbf{0}, \mathbf{x}\rangle \\
[\![\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}]\!]_c &\equiv \langle \mathbf{1}, \langle \mathbf{x}, [\![\mathbf{e}]\!]_c\rangle\rangle \\
[\![\mathbf{e}_1\ \mathbf{e}_2]\!]_c &\equiv \langle \mathbf{2}, \langle [\![\mathbf{e}_1]\!]_c, [\![\mathbf{e}_2]\!]_c\rangle\rangle
\end{aligned}$$

An evolved λ-context $\mathbf{e}$ is encoded as the $\boldsymbol{\lambda}\mathbf{C}$-term $[\![\mathbf{e}]\!]_c$. The representation schema $[\![\ ]\!]_c$ is abstract. For the encoding $[\![\mathbf{e}]\!]_c$, we can tell from the tag of $\mathbf{0}$, $\mathbf{1}$, or $\mathbf{2}$ whether $\mathbf{e}$ is an identifier, an abstraction, or an application. (The tags $\mathbf{0}$–$\mathbf{2}$ can be any program symbols as long as they are pairwise distinct.) Moreover, we have the tools to take $[\![\mathbf{e}]\!]_c$ apart to get to its components. Last but not least, we can determine if $\mathbf{x}$ and $\mathbf{y}$ are the same identifier by comparing their encodings $[\![\mathbf{x}]\!]_c$ and $[\![\mathbf{y}]\!]_c$ using the predicate *eq?* defined in Section 3.8.1. The representation schema $[\![\ ]\!]_c$ is also *generative* [67], the representation of an evolved λ-context is constructible from the encodings of the context's components. A generative encoding schema allows for incremental construction of source code.

The companion compilation function $\mathcal{C}$ for the above encoding schema $[\![\ ]\!]_c$ is defined as follows (again, pattern matching is used to simplify the presentation):

$$\begin{aligned}
\mathcal{C}\ \langle \mathbf{0}, e\rangle &= \mathit{phi}(e) \\
\mathcal{C}\ \langle \mathbf{1}, \langle e_1, e_2\rangle\rangle &= \mathit{lam}(e_1)\ (\mathcal{C}\ e_2) \\
\mathcal{C}\ \langle \mathbf{2}, \langle e_1, e_2\rangle\rangle &= \mathbf{app}\ (\mathcal{C}\ e_1)\ (\mathcal{C}\ e_2)
\end{aligned}$$

It is merely an adaptation of the transformation $\mathcal{T}$ to the specific representation schema $[\![\ ]\!]_c$. Indeed, we may devise other abstract source code representations for evolved λ-contexts. All that is required to define their companion compilers are mechanisms to derive the pair $\mathbf{lam_x}$ and $\Phi\{\mathbf{x}\!:\!x\}.x$ from the representation of a program symbol $\mathbf{x}$.

4.3 Second Simulation

The transformation $\mathcal{T}$ given in the first simulation is rather conservative. It translates a λ-context $\mathbf{C}$ to its image $im(\mathbf{C})$ when the context is evolved. On the other hand, a partially-evolved λ-context $\mathbf{C}^+$ is transformed into the composition of compiled code construction operators. Hence, the transformation of $(\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}_1)\ \mathbf{C}_2$, where either $\mathbf{C}_1$ or $\mathbf{C}_2$ is partially-evolved, is not a β-redex. For example,

$$\begin{aligned}\mathcal{T}((\boldsymbol{\lambda}\mathbf{x}.\mathbf{x})\ \mathbf{h}) &\equiv \mathbf{app}\ \Phi\{\mathbf{x}\!:\!x\}.\lambda x.x\ q(\mathbf{h})\\ \mathcal{T}((\boldsymbol{\lambda}\mathbf{x}.\mathbf{h})\ \mathbf{y}) &\equiv \mathbf{app}\ (\mathbf{lam}_{\mathbf{x}}\ q(\mathbf{h}))\ \Phi\{\mathbf{y}\!:\!y\}.y\end{aligned}$$

In this section we give another simulation showing that it is possible to transform any λ-context $\mathbf{C}$ into its image $im(\mathbf{C})$, except for its holes. In particular, the new transformation $\mathcal{S}$ translates λ-contexts in β-redex form into β-redexes of $\boldsymbol{\lambda}\mathbf{C}$, hence allowing the reduction of $(\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}_1)\ \mathbf{C}_2$ without any preconditions. For comparison,

$$\begin{aligned}\mathcal{S}((\boldsymbol{\lambda}\mathbf{x}.\mathbf{x})\ \mathbf{h}) &\equiv (\lambda x.x)\ \mathcal{S}(\mathbf{h})\\ &\to \mathcal{S}(\mathbf{h})\end{aligned}$$

and

$$\begin{aligned}\mathcal{S}((\boldsymbol{\lambda}\mathbf{x}.\mathbf{h})\ \mathbf{y}) &\equiv (\lambda x.\mathcal{S}(\mathbf{h}))\ \mathcal{S}(\mathbf{y})\\ &\to [\mathcal{S}(\mathbf{y})/x]\mathcal{S}(\mathbf{h})\end{aligned}$$

4.3.1 Binding Structures

The second simulation of λ-contexts is based on Talcott's binding structures [70, 71], which combine λ-contexts and λ-terms at the meta-langauge level. The binding structures have the following abstract syntax (using our notation):

$$e \ ::= \ x \mid \lambda x.e \mid e\ e \mid \mathbf{x} \mid \mathbf{h}\!:\{\mathbf{x}_1\!:\!e, \ldots, \mathbf{x}_n\!:\!e\}$$

In addition to the three λ-terms, there are the identifiers $\mathbf{x}$ and the annotated holes $\mathbf{h}\!:\{\mathbf{x}_1\!:\!e, \ldots, \mathbf{x}_n\!:\!e\}$ where $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are pairwise distinct identifiers.

An annotated hole $\mathbf{h}:\{\mathbf{x}_1:e_1,\ldots,\mathbf{x}_n:e_n\}$ denotes "suspended" hygienic substitutions of e_i for $\mathbf{x}_i$ that are activated when $\mathbf{h}$ is filled. The identifiers $\mathbf{x}_1,\ldots,\mathbf{x}_n$ are the *capturing* identifiers of the hole. They are the ones that the suspended substitutions intend to replace when the hole is filled. For instance, filling $\mathbf{h}:\{\mathbf{x}:y\}$ with $\lambda y.(x\;\mathbf{x})$ initiates the substitution of y for $\mathbf{x}$ in $\lambda y.(x\;\mathbf{x})$, denoted as $[y/\mathbf{x}]\lambda y.(x\;\mathbf{x})$, thus yielding $\lambda y'.(x\;y)$. Notice that the λ-parameter y is renamed to a fresh y' to avoid inadvertent variable capture.

By associating a hole with the identifiers it intends to capture and by associating the capturing identifiers with their replacement terms, α-conversion is valid:

$$\begin{aligned} \lambda x.e \;\; &\rightarrow \;\; \lambda y.\langle y/x\rangle e \\ &\qquad \text{where } y \not\equiv x \text{ and } y \notin fv(e) \end{aligned} \tag{α}$$

The necessary α-substitution meta-operation $\langle y/x\rangle e$ is a conservative extension of that for the λ-calculus (assuming the variable conventions):

$$\begin{aligned} \langle y/x\rangle z &\equiv \begin{cases} y & \text{if } x \equiv z \\ z & \text{otherwise} \end{cases} \\ \langle y/x\rangle \lambda z.e &\equiv \lambda z.\langle y/x\rangle e \\ \langle y/x\rangle (e_1\;e_2) &\equiv \langle y/x\rangle e_1\;\langle y/x\rangle e_2 \\ \langle y/x\rangle \mathbf{z} &\equiv \mathbf{z} \\ \langle y/x\rangle \mathbf{h}:\{\mathbf{x}_1:e_1,\ldots,\mathbf{x}_n:e_n\} &\equiv \mathbf{h}:\{\mathbf{x}_1:\langle y/x\rangle e_1,\ldots,\mathbf{x}_n:\langle y/x\rangle e_n\} \end{aligned}$$

Thus, $\lambda xy.\mathbf{h}:\{\mathbf{x}:x\}$ is α-equivalent to $\lambda wz.\mathbf{h}:\{\mathbf{x}:w\}$

For the same reason, β-reduction is also valid:

$$(\lambda x.e)\;e' \;\; \rightarrow \;\; [e'/x]e \tag{β}$$

where β-substitution $[e'/x]e$ is conservatively extended as follows:

$$\begin{aligned} [e'/x]z &\equiv \begin{cases} e' & \text{if } x \equiv z \\ z & \text{otherwise} \end{cases} \\ [e'/x]\lambda z.e &\equiv \lambda z.[e'/x]e \end{aligned}$$

$$\begin{aligned}
[e'/x](e_1\ e_2) &\equiv [e'/x]e_1\ [e'/x]e_2\\
[e'/x]\mathbf{z} &\equiv \mathbf{z}\\
[e'/x]\mathbf{h}\!:\!\{\mathbf{x}_1\!:\!e_1,\ldots,\mathbf{x}_n\!:\!e_n\} &\equiv \mathbf{h}\!:\!\{\mathbf{x}_1\!:\![e'/x]e_1,\ldots,\mathbf{x}_n\!:\![e'/x]e_n\}
\end{aligned}$$

Thus, $(\lambda xy.\mathbf{h}\!:\!\{\mathbf{x}\!:\!(x\ y)\})\ (y\ y)$ one-step β-reduces to $\lambda y'.\mathbf{h}\!:\!\{\mathbf{x}\!:\!(y\ y\ y')\}$.

The hole filling operation $[e'/\mathbf{h}]e$ is defined inductively on the structure of e:

$$\begin{aligned}
[e'/\mathbf{h}]z &\equiv z\\
[e'/\mathbf{h}]\lambda z.e &\equiv \lambda z.[e'/\mathbf{h}]e\\
[e'/\mathbf{h}](e_1\ e_2) &\equiv [e'/\mathbf{h}]e_1\ [e'/\mathbf{h}]e_2\\
[e'/\mathbf{h}]\mathbf{z} &\equiv \mathbf{z}\\
[e'/\mathbf{h}]\mathbf{h}'\!:\!\{\mathbf{x}_1\!:\!e_1,\ldots,\mathbf{x}_n\!:\!e_n\} &\equiv \begin{cases}[[e'/\mathbf{h}]e_1/\mathbf{x}_1,\ldots,[e'/\mathbf{h}]e_n/\mathbf{x}_n]e' & \text{if } \mathbf{h}\equiv\mathbf{h}'\\ \mathbf{h}\!:\!\{\mathbf{x}_1\!:\![e'/\mathbf{h}]e_1,\ldots,\mathbf{x}_n\!:\![e'/\mathbf{h}]e_n\} & \text{otherwise}\end{cases}
\end{aligned}$$

There are two cases when such an $\mathbf{h}$-substitution is carried out on an annotated hole $\mathbf{h}'\!:\!\{\mathbf{x}_1\!:\!e_1,\ldots,\mathbf{x}_n\!:\!e_n\}$. In both cases, the substitution is first distributed to the replacement terms $e_1,\ldots,e_n$ of the capturing identifiers $\mathbf{x}_1,\ldots,\mathbf{x}_n$. If the name $\mathbf{h}'$ of the annotated hole is $\mathbf{h}$, the annotated hole is filled with e'. Hence, the suspended substitutions encoded in the annotation are activated upon the filler e'. When the holes do not match, the hole $\mathbf{h}'$ keeps its revised annotation.

The meta-operation $[e'_1/\mathbf{x}_1,\ldots,e'_n/\mathbf{x}_n]e$ simultaneously substitutes e'_i for the capturing identifier $\mathbf{x}_i$ in e. To simplify the presentation, we show the case for a single capturing identifier:

$$\begin{aligned}
[e'/\mathbf{x}]z &\equiv z\\
[e'/\mathbf{x}]\lambda z.e &\equiv \lambda z.[e'/\mathbf{x}]e\\
[e'/\mathbf{x}](e_1\ e_2) &\equiv [e'/\mathbf{x}]e_1\ [e'/\mathbf{x}]e_2\\
[e'/\mathbf{x}]\mathbf{z} &\equiv \begin{cases} e' & \text{if } \mathbf{x}\equiv\mathbf{z}\\ \mathbf{z} & \text{otherwise}\end{cases}\\
[e'/\mathbf{x}]\mathbf{h}\!:\!\{\mathbf{x}_1\!:\!e_1,\ldots,\mathbf{x}_n\!:\!e_n\} &\equiv \begin{cases}\mathbf{h}\!:\!\{\mathbf{x}_1\!:\![e'/\mathbf{x}]e_1,\ldots,\mathbf{x}_n\!:\![e'/\mathbf{x}]e_n\} & \text{if } \mathbf{x}\equiv\mathbf{x}_i\\ \mathbf{h}\!:\!\{\mathbf{x}_1\!:\![e'/\mathbf{x}]e_1,\ldots,\mathbf{x}_n\!:\![e'/\mathbf{x}]e_n,\mathbf{x}\!:\!e'\} & \text{otherwise}\end{cases}
\end{aligned}$$

In the clause for annotated holes, the $\mathbf{x}$-substitution is first distributed to the replacement terms $e_1, \ldots, e_n$. The identifier $\mathbf{x}$ then becomes a capturing identifier of the annotated hole unless the hole already has $\mathbf{x}$ as one of its capturing identifiers. In other words, the $\mathbf{x}$-substitution is suspended at the hole $\mathbf{h}$ so that it can be resumed when the hole is filled later.

There is no syntax in the binding structures for expressing hole filling. It remains strictly as a meta-theory operation. There is a good reason that hole filling is described at the meta-theory level only. Let us add new terms $\mathit{fill}_{\mathbf{h}}\ e_1\ e_2$ to the binding structures and define their reduction rule as follows:

$$\mathit{fill}_{\mathbf{h}}\ e_1\ e_2 \quad\rightarrow\quad [e_2/\mathbf{h}]e_1$$

That is, $\mathit{fill}_{\mathbf{h}}\ e_1\ e_2$ denotes the filling of the occurrences of the hole $\mathbf{h}$ in the term e_2 with the term e_1. Then, by contracting the redexes of the following term in different orders, we can show that the system is inconsistent:

$$(\lambda x.(\mathit{fill}_{\mathbf{h}}\ x\ \mathbf{h}\!:\!\{\mathbf{x}\!:\!y\}))\ (\mathbf{x}\ \mathbf{h}'\!:\!\{\mathbf{y}\!:\!y\})$$

In particular, when the fill-redex is contracted before the β-redex, we have:

$$\begin{array}{ll} \multicolumn{2}{l}{(\lambda x.\underline{(\mathit{fill}_{\mathbf{h}}\ x\ \mathbf{h}\!:\!\{\mathbf{x}\!:\!y\})})\ (\mathbf{x}\ \mathbf{h}'\!:\!\{\mathbf{y}\!:\!y\})} \\ \rightarrow & (\lambda x.\underline{[x/\mathbf{h}]\mathbf{h}\!:\!\{\mathbf{x}\!:\!y\}})\ (\mathbf{x}\ \mathbf{h}'\!:\!\{\mathbf{y}\!:\!y\}) \\ \equiv & (\lambda x.\underline{[y/\mathbf{x}]x})\ (\mathbf{x}\ \mathbf{h}'\!:\!\{\mathbf{y}\!:\!y\}) \\ \equiv & \underline{(\lambda x.x)\ (\mathbf{x}\ \mathbf{h}'\!:\!\{\mathbf{y}\!:\!y\})} \\ \rightarrow & \mathbf{x}\ \mathbf{h}'\!:\!\{\mathbf{y}\!:\!y\} \end{array}$$

But, when we swap the order in which the two redexes are contracted, we get a different result:

$$\begin{array}{ll} \multicolumn{2}{l}{\underline{(\lambda x.(\mathit{fill}_{\mathbf{h}}\ x\ \mathbf{h}\!:\!\{\mathbf{x}\!:\!y\}))\ (\mathbf{x}\ \mathbf{h}'\!:\!\{\mathbf{y}\!:\!y\})}} \\ \rightarrow & \underline{[\mathbf{x}\ \mathbf{h}'\!:\!\{\mathbf{y}\!:\!y\}/x](\mathit{fill}_{\mathbf{h}}\ x\ \mathbf{h}\!:\!\{\mathbf{x}\!:\!y\})} \\ \equiv & \underline{\mathit{fill}_{\mathbf{h}}\ (\mathbf{x}\ \mathbf{h}'\!:\!\{\mathbf{y}\!:\!y\})\ \mathbf{h}\!:\!\{\mathbf{x}\!:\!y\}} \end{array}$$

$$\begin{aligned}
&\equiv \underline{[\mathbf{x}\ \mathbf{h}' : \{\mathbf{y} : y\}/\mathbf{h}]\mathbf{h} : \{\mathbf{x} : y\}} \\
&\equiv \underline{[y/\mathbf{x}](\mathbf{x}\ \mathbf{h}' : \{\mathbf{y} : y\})} \\
&\equiv y\ \mathbf{h}' : \{\mathbf{y} : y, \mathbf{x} : y\}
\end{aligned}$$

The discrepancy is caused by the fact that the suspended substitution $[y/\mathbf{x}]$ associated with the hole $\mathbf{h}$ is activated as soon as the hole is filled, even though the filler x has yet to gather its to-be-captured free identifiers and annotated holes in the former case. Hence, to adapt binding structures to a programming language calculus, we must pin down what the to-be-captured identifiers and annotated holes of a filler term are; short of it, the system is not Church-Rosser.

In summary, to simulate the behavior of λ-contexts using the notion of annotated holes, we must devise a suitable $\boldsymbol{\lambda}\mathbf{C}$-encoding of annotations. Furthermore, we must have a representation schema for contexts that satisfies these three constraints:

- When a hole $\mathbf{h}$ of a context $\mathbf{C}_1$ is filled with another context $\mathbf{C}_2$, the annotation of the filled hole $\mathbf{h}$ must clearly specify the hole's capturing identifiers. Moreover, the filler context $\mathbf{C}_2$ must clearly specify its to-be-captured free identifiers and holes.

- Any to-be-captured free identifier of the filler context $\mathbf{C}_2$ that is not captured by the annotation of $\mathbf{h}$ must remain a to-be-captured free identifier.

- Similarly, the holes of $\mathbf{C}_2$ must be retained in the result of the hole filling operation. Moreover, the capturing identifiers of the filled hole $\mathbf{h}$ must be propagated to the holes of $\mathbf{C}_2$, provided that they are not shadowed by the holes' annotation.

4.3.2 Encoding Annotations

Let $\mathbf{n}_1, \ldots, \mathbf{n}_n$ be pairwise distinct identifier names or holes. We define an *annotation* $\{\mathbf{n}_1 : e_1, \ldots, \mathbf{n}_n : e_n\}^*$, abbreviated as $\{\bar{\mathbf{n}} : \bar{e}\}^*$, to be the following syntactic sugar:

$$\{\mathbf{n}_1 : e_1, \ldots, \mathbf{n}_n : e_n\}^* \ \equiv\ [\langle \mathbf{n}_1, \Phi\{\}.e_1\rangle, \ldots, \langle \mathbf{n}_n, \Phi\{\}.e_n\rangle]$$

It is the encoding of an environment associating the identifier $\mathbf{n}_i$ with the denotation $\Phi\{\}.e_i$ (cf. Section 3.8.2 on first-class environments). The arid annotation is the arid environment $[]$. The annotation linking operator is the environment importing operator *link* whose definition is repeated below:

$$\begin{aligned} \mathit{link}\,[]\; e' &= e' \\ \mathit{link}\,\langle \mathbf{x}, e\rangle :: s\; e' &= \mathbf{app}\; (\mathit{link}\, s\; (\mathit{lam}(\mathbf{x})\; e'))\; e \end{aligned}$$

The notation $\langle \mathbf{x}, e\rangle :: s$ denotes the sequence whose first element is $\langle \mathbf{x}, e\rangle$ and the remainder of the sequence is s. Hence, linking an annotation $\{\bar{\mathbf{n}}\!:\!\bar{e}\}^*$ to a free identifier abstraction $\Phi\rho.e$ exhibits the following behavior:

$$\mathit{link}\,\{\bar{\mathbf{n}}\!:\!\bar{e}\}^*\;\Phi\rho.e \;=\; \Phi\rho.[e_n/x_n]\cdots[e_1/x_1]e$$

where either $\mathbf{n}_i\!:\!x_i \in \rho$ or else x_i is fresh. That is, it substitutes e_i for free occurrences of x_i in e.

Define the annotation combining operator *combine* as follows:

$$\begin{aligned} \mathit{combine}\,[]\; s &= s \\ \mathit{combine}\,\langle \mathbf{x}', e'\rangle :: s'\; s &= \mathbf{if}\; (\mathit{member}\,\mathbf{x}'\; s)\; \mathbf{then}\; (\mathit{combine}\; s'\; s) \\ &\qquad \mathbf{else}\; (\mathit{cons}\;\langle \mathbf{x}', e'\rangle\; (\mathit{combine}\; s'\; s)) \end{aligned}$$

where the predicate *member* that tells whether $\mathbf{x}'$ is an identifier of the annotation s is defined inductively as follows:

$$\begin{aligned} \mathit{member}\,\mathbf{x}'\;[] &= F \\ \mathit{member}\,\mathbf{x}'\;\langle \mathbf{x}, e\rangle :: s &= \mathit{if}\;(\mathit{eq?}\;\mathbf{x}'\;\mathbf{x})\;\mathit{then}\;T\;\mathit{else}\;(\mathit{member}\,\mathbf{x}'\;s) \end{aligned}$$

Thus, *combine* $\{\bar{\mathbf{n}}\!:\!\bar{e}\}^*$ $\{\bar{\mathbf{n}}'\!:\!\bar{e}'\}^*$ combines the annotations $\{\bar{\mathbf{n}}\!:\!\bar{e}\}^*$ and $\{\bar{\mathbf{n}}'\!:\!\bar{e}'\}^*$, with the bindings of $\{\bar{\mathbf{n}}'\!:\!\bar{e}'\}^*$ preceding the bindings of $\{\bar{\mathbf{n}}\!:\!\bar{e}\}^*$ in the event of conflicts.

A degenerate case of annotations is $\{\mathbf{n}_1\!:\!x_1, \ldots, \mathbf{n}_n\!:\!x_n\}^*$ where $x_1, \ldots, x_n$ are pairwise distinct variables. In such a case, $\{\mathbf{n}_1\!:\!x_1, \ldots, \mathbf{n}_n\!:\!x_n\}$ qualifies as a free identifier abstraction parameter specification. Hence, we write ρ^* for $\{\bar{\mathbf{n}}\!:\!\bar{x}\}^*$ when ρ is $\{\bar{\mathbf{n}}\!:\!\bar{x}\}$.

4.3.3 Encoding Contexts

The second encoding of λ-contexts $\mathcal{R}_2$ is defined as follows:

$$\mathcal{R}_2(\mathbf{C}) \equiv \langle E_{\mathbf{C}}, I_{\mathbf{C}}, H_{\mathbf{C}} \rangle$$

The elements of the triple $\langle E_{\mathbf{C}}, I_{\mathbf{C}}, H_{\mathbf{C}} \rangle$ are defined as follows:

$$\begin{array}{rcl} E_{\mathbf{C}} & \equiv & \Phi\rho^{\mathbf{x}}_{\mathbf{c}}.\Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\mathcal{S}_{\{\}^*}(\mathbf{C}) & \equiv & \Phi\{\bar{\mathbf{x}}_{\mathbf{c}}:\bar{x}_{\mathbf{c}}\}.\Phi\{\bar{\mathbf{h}}_{\mathbf{c}}:\bar{h}_{\mathbf{c}}\}.\mathcal{S}_{\{\}^*}(\mathbf{C}) \\ I_{\mathbf{C}} & \equiv & \Phi\rho^{\mathbf{x}}_{\mathbf{c}}.\{\bar{\mathbf{x}}_{\mathbf{c}}:\bar{x}_{\mathbf{c}}\}^* & \equiv & \Phi\{\bar{\mathbf{x}}_{\mathbf{c}}:\bar{x}_{\mathbf{c}}\}.\{\bar{\mathbf{x}}_{\mathbf{c}}:\bar{x}_{\mathbf{c}}\}^* \\ H_{\mathbf{C}} & \equiv & \Phi\rho^{\mathbf{h}}_{\mathbf{c}}.\lambda g.\{\bar{\mathbf{h}}_{\mathbf{c}}:(g\ \bar{h}_{\mathbf{c}})\}^* & \equiv & \Phi\{\bar{\mathbf{h}}_{\mathbf{c}}:\bar{h}_{\mathbf{c}}\}.\lambda g.\{\bar{\mathbf{h}}_{\mathbf{c}}:(g\ \bar{h}_{\mathbf{c}})\}^* \end{array}$$

The first element $E_{\mathbf{C}}$ is the transformation $\mathcal{S}_{\{\}^*}(\mathbf{C})$ of $\mathbf{C}$ with the to-be-captured free identifiers *fi*$(\mathbf{C})$ and holes *oh*$(\mathbf{C})$ Φ-bound. The transformation $\mathcal{S}_{\rho^*}(\mathbf{C})$ is defined inductively on the structure of $\mathbf{C}$ as follows:

$$\begin{array}{rcl} \mathcal{S}_{\rho^*}(\mathbf{h}) & \equiv & q(\mathbf{h})\ \rho^* \\ \mathcal{S}_{\rho^*}(\mathbf{x}) & \equiv & q(\mathbf{x}) \\ \mathcal{S}_{\rho^*}(\boldsymbol{\lambda}\mathbf{x}.\mathbf{C}) & \equiv & \lambda q(\mathbf{x}).\mathcal{S}_{\rho_1^*}(\mathbf{C}) \\ & & \quad \text{where } \rho_1^* \text{ is } (\mathit{combine}\ \rho^*\ \{\mathbf{x}:q(\mathbf{x})\}^*) \\ \mathcal{S}_{\rho^*}(\mathbf{C}_1\ \mathbf{C}_2) & \equiv & \mathcal{S}_{\rho^*}(\mathbf{C}_1)\ \mathcal{S}_{\rho^*}(\mathbf{C}_2) \end{array}$$

That is, each λ-context $\mathbf{C}$, which can be either evolved or partially-evolved, is fully compiled into its image *im*$(\mathbf{C})$ where each hole $\mathbf{h}$ is encoded as the application $q(\mathbf{h})\ \rho^*$, which is our way of annotating $\mathbf{h}$ with its capturing identifiers encoded as the annotation ρ^*.

The second element

$$I_{\mathbf{C}} \equiv \Phi\{\bar{\mathbf{x}}_{\mathbf{c}}:\bar{x}_{\mathbf{c}}\}.\{\bar{\mathbf{x}}_{\mathbf{c}}:\bar{x}_{\mathbf{c}}\}^*$$

records the to-be-captured free identifiers *fi*$(\mathbf{C})$ of $\mathbf{C}$. When $\mathbf{C}$ is used to fill the hole $\mathbf{h}'$ of some context $\mathbf{C}'$, $[\mathbf{C}/\mathbf{h}']\mathbf{C}'$, the annotated hole $\mathbf{h}'$ may not capture all the free identifiers of $\mathbf{C}$. The second element $I_{\mathbf{C}}$ is then used to keep the free identifiers of $\mathbf{C}$ not captured by $\mathbf{h}'$ remain as to-be-captured free identifiers of $[\mathbf{C}/\mathbf{h}']\mathbf{C}'$.

The third element

$$H_{\mathbf{C}} \equiv \Phi\{\bar{\mathbf{h}}_{\mathbf{c}} : \bar{h}_{\mathbf{c}}\}.\lambda g.\{\bar{\mathbf{h}}_{\mathbf{c}} : (g\ \bar{h}_{\mathbf{c}})\}^*$$

serves a similar purpose as $I_{\mathbf{C}}$. It records the free holes $oh(\mathbf{C})$ of $\mathbf{C}$. When $\mathbf{C}$ is used to fill the hole $\mathbf{h}'$ of some context $\mathbf{C}'$, $[\mathbf{C}/\mathbf{h}']\mathbf{C}'$, $H_{\mathbf{C}}$ helps to propagate the capturing identifiers of the hole $\mathbf{h}'$ to the holes of $\mathbf{C}$. The notation $\lambda g.\{\bar{\mathbf{h}}_{\mathbf{c}} : (g\ \bar{h}_{\mathbf{c}})\}^*$ is an abbreviation of $\lambda g.\{\mathbf{h}_1 : (g\ h_1), \ldots, \mathbf{h}_n : (g\ h_n)\}^*$ with $h_i = q(\mathbf{h}_i)$.

As an example, let $\mathbf{C}_1$ be the λ-context:

$$\mathbf{C}_1 \quad \equiv \quad (\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}_1)\ (\mathbf{y}\ \mathbf{h}_2)$$

The components of the encoding $\langle E_{\mathbf{C}_1}, I_{\mathbf{C}_1}, H_{\mathbf{C}_1}\rangle$ of $\mathbf{C}_1$ are:

$$\begin{aligned}
E_{\mathbf{C}_1} &\equiv \Phi\{\mathbf{y} : y\}.\Phi\{\mathbf{h}_1 : h_1, \mathbf{h}_2 : h_2\}.((\lambda x.(h_1\ \{\mathbf{x} : x\}^*))\ (y\ (h_2\ \{\}^*))) \\
I_{\mathbf{C}_1} &\equiv \Phi\{\mathbf{y} : y\}.\{\mathbf{y} : y\}^* \\
H_{\mathbf{C}_1} &\equiv \Phi\{\mathbf{h}_1 : h_1, \mathbf{h}_2 : h_2\}.\lambda g.\{\mathbf{h}_1 : (g\ h_1), \mathbf{h}_2 : (g\ h_2)\}^*
\end{aligned}$$

The hole $\mathbf{h}_1$ is in the scope of the bound identifier $\mathbf{x}$; hence, the corresponding variable h_1 of $\mathbf{h}_1$ is "annotated" with the annotation $\{\mathbf{x} : x\}^*$. In contrast, the hole $\mathbf{h}_2$ is not in the scope of any bound identifiers; hence, it has an arid annotation $\{\}^*$. We should emphasize that in the encoding $E_{\mathbf{C}_1}$, the application λ-context $(\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}_1)\ (\mathbf{y}\ \mathbf{h}_2)$ is translated into a β-redex. Thus,

$$\begin{aligned}
E_{\mathbf{C}_1} &\equiv \Phi\{\mathbf{y} : y\}.\Phi\{\mathbf{h}_1 : h_1, \mathbf{h}_2 : h_2\}.\underline{((\lambda x.(h_1\ \{\mathbf{x} : x\}^*))\ (y\ (h_2\ \{\}^*)))} \\
&\to \Phi\{\mathbf{y} : y\}.\Phi\{\mathbf{h}_1 : h_1, \mathbf{h}_2 : h_2\}.(h_1\ \{\mathbf{x} : (y\ (h_2\ \{\}^*))\}^*)
\end{aligned}$$

Again, the representation $\mathcal{R}_2(\mathbf{C})$ of $\mathbf{C}$ is not unique. In particular, we can always add redundant parameters to $\rho^{\mathbf{x}}_{\mathbf{c}}$ and $\rho^{\mathbf{h}}_{\mathbf{c}}$ without affecting the behavior of the simulation. Hence, let $\rho^{\mathbf{x}}_{\mathbf{c}} \subseteq \rho_1$ and $\rho^{\mathbf{h}}_{\mathbf{c}} \subseteq \rho_2 \equiv \{\bar{\mathbf{h}} : \bar{h}\}$ and the components of a triple $\langle E, I, H\rangle$ be defined as follows:

$$\begin{aligned}
E &\equiv \Phi\rho_1.\Phi\rho_2.\mathcal{S}_{\{\}^*}(\mathbf{C}) \\
I &\equiv \Phi\rho_1.\rho_1^* \\
H &\equiv \Phi\rho_2.\lambda g.\{\bar{\mathbf{h}} : (g\ \bar{h})\}^*
\end{aligned}$$

Then, I and H cover the free identifiers and holes of E just like the way $I_{\mathbf{C}}$ and $H_{\mathbf{C}}$ cover $E_{\mathbf{C}}$. The triple $\langle E, I, H\rangle$ therefore qualifies as a representation of $\mathbf{C}$ as well.

4.3.4 Simulating Hole Filling

Filling the hole $\mathbf{h}$ of an (encoded) λ-context $\langle E_{\mathbf{C}}, I_{\mathbf{C}}, H_{\mathbf{C}}\rangle$ with another (encoded) λ-context $\langle E_{\mathbf{C}'}, I_{\mathbf{C}'}, H_{\mathbf{C}'}\rangle$ can be accomplished using the $\mathit{fill}_{\mathbf{h}}$ operator defined below:

$$\begin{aligned}&\mathit{fill}_{\mathbf{h}}\ \langle E_{\mathbf{C}'}, I_{\mathbf{C}'}, H_{\mathbf{C}'}\rangle\ \langle E_{\mathbf{C}}, I_{\mathbf{C}}, H_{\mathbf{C}}\rangle\\ &\quad = \ \langle (F_1\ \mathit{bind}_{\mathbf{h}}\ E_{\mathbf{C}'}\ I_{\mathbf{C}'}\ H_{\mathbf{C}'}\ E_{\mathbf{C}}), (F_2\ I_{\mathbf{C}'}\ I_{\mathbf{C}}), (F_3\ H_{\mathbf{C}'}\ H_{\mathbf{C}})\rangle\end{aligned}$$

where the combinators F_1, F_2, and F_3 are defined as follows:

$$\begin{aligned}F_1 &= \lambda fabcd.(\mathbf{app}\ (\mathbf{app}\ \Phi\{\}.(F_1'\ f\ a\ c)\ b)\ d)\\ F_1' &= \lambda fabcd.(f\ (\mathbf{app}\ \Phi\{\}.(F_1''\ a\ c)\ b)\ d)\\ F_1'' &= \lambda abcy.(\mathbf{load}\ (\mathit{link}\ (c\ (\lambda xy'.(x\ (\mathit{combine}\ y\ y'))))\\ &\qquad\qquad (\mathbf{load}\ (\mathit{link}\ (\mathit{combine}\ b\ y)\ a))))\\ F_2 &= \lambda ab.(\mathbf{app}\ (\mathbf{app}\ \Phi\{\}.\mathit{combine}\ a)\ b)\\ F_3 &= \lambda ab.(\mathbf{app}\ (\mathbf{app}\ \Phi\{\}.\lambda abg.(\mathit{combine}\ (a\ g)\ (b\ g))\ a)\ b)\end{aligned}$$

Let $\mathbf{C}_3$ be the result of filling the holes named $\mathbf{h}$ of the λ-context $\mathbf{C}_1$ with another λ-context $\mathbf{C}_2$, $\mathbf{C}_3 \equiv [\mathbf{C}_2/\mathbf{h}]\mathbf{C}_1$. Then, with the second encoding of λ-contexts $\mathcal{R}_2$ that represents each λ-context $\mathbf{C}$ as the triple $\langle E_{\mathbf{C}}, I_{\mathbf{C}}, H_{\mathbf{C}}\rangle$, hole filling can be accomplished as follows:

Theorem 4.3 *Let $\langle E_{\mathbf{C}_1}, I_{\mathbf{C}_1}, H_{\mathbf{C}_1}\rangle$ and $\langle E_{\mathbf{C}_2}, I_{\mathbf{C}_2}, H_{\mathbf{C}_2}\rangle$ be the encoding of $\mathbf{C}_1$ and $\mathbf{C}_2$, respectively. Then,*

$$\mathit{fill}_{\mathbf{h}}\ \langle E_{\mathbf{C}_2}, I_{\mathbf{C}_2}, H_{\mathbf{C}_2}\rangle\ \langle E_{\mathbf{C}_1}, I_{\mathbf{C}_1}, H_{\mathbf{C}_1}\rangle$$

is an encoding of $\mathbf{C}_3$.

Proof by example: A complete proof is quite involved; we demonstrate the inner workings of the filling operation through a concrete example, instead.

Let $\mathbf{C}_1$ and $\mathbf{C}_2$ be these λ-contexts:

$$\begin{aligned}\mathbf{C}_1 &\equiv (\boldsymbol{\lambda}\mathbf{x}.\mathbf{h}_1)\ (\mathbf{y}\ \mathbf{h}_2)\\ \mathbf{C}_2 &\equiv \mathbf{x}\ \mathbf{y}\ \boldsymbol{\lambda}\mathbf{y}.\mathbf{h}_2\end{aligned}$$

The encoding $\langle E_{\mathbf{C}_1}, I_{\mathbf{C}_1}, H_{\mathbf{C}_1}\rangle$ of $\mathbf{C}_1$ has these components:

$$\begin{aligned}E_{\mathbf{C}_1} &\equiv \Phi\{\mathbf{y}\!:\!y\}.\Phi\{\mathbf{h}_1\!:\!h_1, \mathbf{h}_2\!:\!h_2\}.((\lambda x.(h_1\ \{\mathbf{x}\!:\!x\}^*))\ (y\ (h_2\ \{\}^*)))\\ I_{\mathbf{C}_1} &\equiv \Phi\{\mathbf{y}\!:\!y\}.\{\mathbf{y}\!:\!y\}^*\\ H_{\mathbf{C}_1} &\equiv \Phi\{\mathbf{h}_1\!:\!h_1, \mathbf{h}_2\!:\!h_2\}.\lambda g.\{\mathbf{h}_1\!:\!(g\ h_1), \mathbf{h}_2\!:\!(g\ h_2)\}^*\end{aligned}$$

The components of the encoding $\langle E_{\mathbf{C}_2}, I_{\mathbf{C}_2}, H_{\mathbf{C}_2}\rangle$ of $\mathbf{C}_2$ are:

$$\begin{aligned}E_{\mathbf{C}_2} &\equiv \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.\Phi\{\mathbf{h}_2\!:\!h_2\}.(x\ y\ \lambda y.(h_2\ \{\mathbf{y}\!:\!y\}^*))\\ I_{\mathbf{C}_2} &\equiv \Phi\{\mathbf{y}\!:\!y, \mathbf{x}\!:\!x\}.\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}^*\\ H_{\mathbf{C}_2} &\equiv \Phi\{\mathbf{h}_2\!:\!h_2\}.\lambda g.\{\mathbf{h}_2\!:\!(g\ h_2)\}^*\end{aligned}$$

Let $\mathbf{C}_3$ be the result of filling the hole $\mathbf{h}_1$ of $\mathbf{C}_1$ with $\mathbf{C}_2$:

$$\begin{aligned}\mathbf{C}_3 &\equiv [\mathbf{C}_2/\mathbf{h}_1]\mathbf{C}_1\\ &\equiv (\boldsymbol{\lambda}\mathbf{x}.(\mathbf{x}\ \mathbf{y}\ \boldsymbol{\lambda}\mathbf{y}.\mathbf{h}_2))\ (\mathbf{y}\ \mathbf{h}_2)\end{aligned}$$

The components of the encoding $\langle E_{\mathbf{C}_3}, I_{\mathbf{C}_3}, H_{\mathbf{C}_3}\rangle$ of $\mathbf{C}_3$ are:

$$\begin{aligned}E_{\mathbf{C}_3} &\equiv \Phi\{\mathbf{y}\!:\!y\}.\Phi\{\mathbf{h}_2\!:\!h_2\}.((\lambda x.(x\ y\ \lambda y.(h_2\ \{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}^*)))\ (y\ (h_2\ \{\}^*)))\\ I_{\mathbf{C}_3} &\equiv \Phi\{\mathbf{y}\!:\!y\}.\{\mathbf{y}\!:\!y\}^*\\ H_{\mathbf{C}_3} &\equiv \Phi\{\mathbf{h}_2\!:\!h_2\}.\lambda g.\{\mathbf{h}_2\!:\!(g\ h_2)\}^*\end{aligned}$$

By the definition of $\mathit{fill}_{\mathbf{h}}$, we have

$$\mathit{fill}_{\mathbf{h}_1}\ \langle E_{\mathbf{C}_2}, I_{\mathbf{C}_2}, H_{\mathbf{C}_2}\rangle\ \langle E_{\mathbf{C}_1}, I_{\mathbf{C}_1}, H_{\mathbf{C}_1}\rangle \ = \ \langle E, I, H\rangle$$

where

$$E = \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.\Phi\{\mathbf{h}_1\!:\!h_1, \mathbf{h}_2\!:\!h_2\}.$$

$$\begin{aligned}
& [G/h_1]((\lambda x.(h_1\ \{\mathbf{x}\!:\!x\}^*))\ (y\ (h_2\ \{\}^*))) \\
&= \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.\Phi\{\mathbf{h}_1\!:\!h_1, \mathbf{h}_2\!:\!h_2\}.((\lambda x'.(G\ \{\mathbf{x}\!:\!x'\}^*))\ (y\ (h_2\ \{\}^*))) \qquad (4.5) \\
I &= \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.(\mathit{combine}\ \{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}^*\ \{\mathbf{y}\!:\!y\}^*) \\
&= \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}^* \\
H &= \Phi\{\mathbf{h}_1\!:\!h_1, \mathbf{h}_2\!:\!h_2\}.\lambda g.(\mathit{combine}\ \{\mathbf{h}_2\!:\!(g\ h_2)\}^*\ \{\mathbf{h}_1\!:\!(g\ h_1), \mathbf{h}_2\!:\!(g\ h_2)\}^*) \\
&= \Phi\{\mathbf{h}_1\!:\!h_1, \mathbf{h}_2\!:\!h_2\}.\lambda g.\{\mathbf{h}_1\!:\!(g\ h_1), \mathbf{h}_2\!:\!(g\ h_2)\}^*
\end{aligned}$$

with

$$\begin{aligned}
G &= \lambda w.(G_{\mathbf{h}}\ w\ (G_{\mathbf{x}}\ w\ E_{\mathbf{C}_2})) \\
G_{\mathbf{h}} &= \lambda wz.(\mathbf{load}\ (\mathit{link}\ \{\mathbf{h}_2\!:\!\lambda y'.(h_2\ (\mathit{combine}\ w\ y'))\}^*\ z)) \\
G_{\mathbf{x}} &= \lambda wz.(\mathbf{load}\ (\mathit{link}\ (\mathit{combine}\ \{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}^*\ w)\ z))
\end{aligned}$$

The subterm $G\ \{\mathbf{x}\!:\!x'\}^*$ of Equation 4.5 encodes the effect on the filler context $\mathbf{C}_2$ when it replaces the hole $\mathbf{h}_1$ of $\mathbf{C}_1$. Its simplification is

$$\begin{aligned}
& G\ \{\mathbf{x}\!:\!x'\}^* \\
&= G_{\mathbf{h}}\ \{\mathbf{x}\!:\!x'\}^*\ (G_{\mathbf{x}}\ \{\mathbf{x}\!:\!x'\}^*\ E_{\mathbf{C}_2}) \\
&= G_{\mathbf{h}}\ \{\mathbf{x}\!:\!x'\}^*\ (\mathbf{load}\ (\mathit{link}\ (\mathit{combine}\ \{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}^*\ \{\mathbf{x}\!:\!x'\}^*)\ E_{\mathbf{C}_2})) \\
&= G_{\mathbf{h}}\ \{\mathbf{x}\!:\!x'\}^*\ (\mathbf{load}\ (\mathit{link}\ \{\mathbf{x}\!:\!x', \mathbf{y}\!:\!y\}^*\ E_{\mathbf{C}_2})) \\
&= G_{\mathbf{h}}\ \{\mathbf{x}\!:\!x'\}^*\ \Phi\{\mathbf{h}_2\!:\!h_2\}.(x'\ y\ \lambda y.(h_2\ \{\mathbf{y}\!:\!y\}^*)) \qquad (4.6) \\
&= \mathbf{load}\ (\mathit{link}\ \{\mathbf{h}_2\!:\!\lambda y'.(h_2\ (\mathit{combine}\ \{\mathbf{x}\!:\!x'\}^*\ y'))\}^* \\
&\qquad\qquad \Phi\{\mathbf{h}_2\!:\!h_2\}.(x'\ y\ \lambda y.(h_2\ \{\mathbf{y}\!:\!y\}^*))) \\
&= x'\ y\ \lambda y.(h_2\ (\mathit{combine}\ \{\mathbf{x}\!:\!x'\}^*\ \{\mathbf{y}\!:\!y\}^*)) \\
&= x'\ y\ \lambda y.(h_2\ \{\mathbf{x}\!:\!x', \mathbf{y}\!:\!y\}^*) \qquad (4.7)
\end{aligned}$$

The hole $\mathbf{h}_1$ of $\mathbf{C}_1$ has only one capturing-identifier $\mathbf{x}$ whose associated variable is x'. The subterm $\Phi\{\mathbf{h}_2\!:\!h_2\}.(x'\ y\ \lambda y.(h_2\ \{\mathbf{y}\!:\!y\}^*))$ in Equation 4.6 is the result of linking free occurrences of the identifier $\mathbf{x}$ in $\mathbf{C}_2$ (encoded as $E_{\mathbf{C}_2}$) to the capturing-identifier of $\mathbf{h}_1$. The λ-context $\mathbf{C}_2$ has a hole $\mathbf{h}_2$ with a single capturing-identifier $\mathbf{y}$. In the resulting $\mathbf{C}_3$, the hole has two capturing-identifiers $\mathbf{x}$ and $\mathbf{y}$. The additional $\mathbf{x}$ comes from the capturing-identifier of $\mathbf{h}_1$. The propagation of the capturing-identifier of $\mathbf{h}_1$ to $\mathbf{h}_2$ is realized in Equation 4.7 with the annotated hole $h_2\ \{\mathbf{x}\!:\!x', \mathbf{y}\!:\!y\}^*$.

So, by replacing $x'\ y\ \lambda y.(h_2\ \{\mathbf{x}\!:\!x', \mathbf{y}\!:\!y\}^*)$ for $G\ \{\mathbf{x}\!:\!x'\}^*$ in Equation 4.5, we get

$$\begin{aligned} E &= \Phi\{\mathbf{x}\!:\!x, \mathbf{y}\!:\!y\}.\Phi\{\mathbf{h}_1\!:\!h_1, \mathbf{h}_2\!:\!h_2\}. \\ &\qquad\qquad ((\lambda x'.(x'\ y\ \lambda y.(h_2\ \{\mathbf{x}\!:\!x', \mathbf{y}\!:\!y\}^*)))\ (y\ (h_2\ \{\}^*))) \\ &\approx \Phi\{\mathbf{y}\!:\!y\}.\Phi\{\mathbf{h}_2\!:\!h_2\}.((\lambda x'.(x'\ y\ \lambda y.(h_2\ \{\mathbf{x}\!:\!x', \mathbf{y}\!:\!y\}^*)))\ (y\ (h_2\ \{\}^*))) \\ &\equiv E_{\mathbf{C}_3} \end{aligned}$$

thus concluding the demonstration. □

Chapter 5

Twice Context-Enriched Lambda Calculus

We have shown how our context-enriching schema conservatively extends the λ-calculus with incremental compilation and linking capabilities. Curious as we always are, it is fair to ask what effect the schema has on the enriched calculus $\boldsymbol{\lambda}\mathbf{C}$. We thus give a second demonstration of our context-enriching schema by applying it to $\boldsymbol{\lambda}\mathbf{C}$, yielding an extension of the λ-calculus that is enriched with the notion of contexts twice. Interestingly, we can show that the twice context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{CC}$ is a "fixpoint" of our schema. That is, applying the schema to $\boldsymbol{\lambda}\mathbf{CC}$ yields $\boldsymbol{\lambda}\mathbf{CC}$ itself. We have shown previously that $\boldsymbol{\lambda}\mathbf{C}$ is capable of compiling the λ-calculus. We demonstrate here that $\boldsymbol{\lambda}\mathbf{CC}$ is capable of compiling itself metacircularly. The second part of this chapter is devoted to the analysis and refinement of the new mechanisms introduced by the $\boldsymbol{\lambda}\mathbf{CC}$-calculus. The discussion focuses on the pursuit of simple, elegant, yet powerful language features. Their usefulness will become apparent in the forthcoming chapters.

Contexts:

$$\begin{aligned}
\mathbf{C} &::= \mathbf{h} \mid \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C} \mid \mathbf{C}\ \mathbf{C} \mid \boldsymbol{\Phi\rho}.\mathbf{C} \mid \boldsymbol{\delta} \\
\boldsymbol{\delta} &::= \mathbf{load} \mid \tilde{\mathrm{x}} \mid \mathbf{lam}_{\mathbf{X}} \mid \mathbf{app} \\
\boldsymbol{\rho} &::= \{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\} \\
&\qquad \mathbf{x}_1, \ldots, \mathbf{x}_n \text{ pairwise distinct,} \\
&\qquad \mathbf{y}_1, \ldots, \mathbf{y}_n \text{ pairwise distinct}
\end{aligned}$$

Evolved Contexts:

$$\mathbf{e} ::= \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{e} \mid \mathbf{e}\ \mathbf{e} \mid \boldsymbol{\Phi\rho}.\mathbf{e} \mid \boldsymbol{\delta}$$

Figure 5.1: Contexts of context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{C}$

5.1 Term Language

Our schema of enriching a λ-calculus with incremental program construction capabilities is founded on the notion of contexts. The contexts $\mathbf{C}$ of the $\boldsymbol{\lambda}\mathbf{C}$-calculus are shown in Figure 5.1. In addition to the λ-contexts, there are the free identifier abstraction contexts $\boldsymbol{\Phi\rho}.\mathbf{C}$ and the constant contexts $\boldsymbol{\delta}$.

The first step of our schema is to devise a compiled code representation for the evolved $\boldsymbol{\lambda}\mathbf{C}$-contexts $\mathbf{e}$. Again, an evolved $\boldsymbol{\lambda}\mathbf{C}$-context $\mathbf{e}$ can be represented as the free identifier abstraction (q is a one-to-one function that assigns a distinct variable name $q(\mathbf{x})$ to each identifier $\mathbf{x}$)

$$\Phi\{\mathbf{x}_1 : q(\mathbf{x}_1), \ldots, \mathbf{x}_n : q(\mathbf{x}_n)\}.im(\mathbf{e})$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are the free identifiers of $\mathbf{e}$ and the image $im(\mathbf{e})$ is defined inductively as follows:

$$\begin{aligned}
im(\mathbf{x}) &\equiv q(\mathbf{x}) \\
im(\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}) &\equiv \lambda q(\mathbf{x}).im(\mathbf{e})
\end{aligned}$$

$$\begin{aligned} im(\mathbf{e}_1\ \mathbf{e}_2) &\equiv im(\mathbf{e}_1)\ im(\mathbf{e}_2) \\ im(\boldsymbol{\Phi}\{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}.\mathbf{e}) &\equiv \Phi\{\mathbf{x}_1 : q(\mathbf{y}_1), \ldots, \mathbf{x}_n : q(\mathbf{y}_n)\}.im(\mathbf{e}) \\ im(\boldsymbol{\delta}) &\equiv \boldsymbol{\delta} \end{aligned}$$

The $\boldsymbol{\lambda}\mathbf{C}$-contexts $\boldsymbol{\delta}$ are constants. We can therefore express their compiled code as $\Phi\{\}.\boldsymbol{\delta}$. An evolved free identifier abstraction context $\boldsymbol{\Phi}\{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}.\mathbf{e}$ can be compiled into the following *nested* free identifier abstractions:

$$\Phi\{\mathbf{z}_1 : q(\mathbf{z}_1), \ldots, \mathbf{z}_m : q(\mathbf{z}_m)\}.\Phi\{\mathbf{x}_1 : q(\mathbf{y}_1), \ldots, \mathbf{x}_n : q(\mathbf{y}_n)\}.im(\mathbf{e})$$

where $\mathbf{z}_1, \ldots, \mathbf{z}_m$ are the free identifiers of $\boldsymbol{\Phi}\{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}.\mathbf{e}$. The machine code $im(\mathbf{e})$ is the image of the source code $\mathbf{e}$. Free occurrences of the identifiers $\mathbf{y}_1, \ldots, \mathbf{y}_n$ in the source code $\mathbf{e}$ are replaced by their respective placeholders $q(\mathbf{y}_1), \ldots, q(\mathbf{y}_n)$ in the machine code $im(\mathbf{e})$. Similarly, the free identifiers $\mathbf{z}_1, \ldots, \mathbf{z}_m$ are compiled into their respective placeholders $q(\mathbf{z}_1), \ldots, q(\mathbf{z}_m)$ in the machine code $im(\mathbf{e})$.

In addition to the two constructors $\mathbf{lam}_\mathbf{x}$ and $\mathbf{app}$ of $\boldsymbol{\lambda}\mathbf{C}$, we need a new category of constructors $\mathbf{phi}_{\boldsymbol{\rho}}$, one for each $\boldsymbol{\rho}$, to incrementally build the compiled code of $\boldsymbol{\lambda}\mathbf{C}$-contexts $\boldsymbol{\Phi\rho}.\mathbf{e}$ out of the compiled code of $\mathbf{e}$. Collectively, the syntax of the twice context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{CC}$ is presented in Figure 5.2. It extends the $\boldsymbol{\lambda}\mathbf{C}$-calculus with only one additional category of syntax, namely, the incremental free identifier abstraction compiled code construction operators $\mathbf{phi}_{\boldsymbol{\rho}}$. We abbreviate $\mathbf{phi}_{\{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}}$ as $\mathbf{phi}_{\{\bar{\mathbf{x}} : \bar{\mathbf{y}}\}}$ at times.

5.2 Calculus of Compiled Code

The α- and β-substitutions of $\boldsymbol{\lambda}\mathbf{C}$ can be easily extended to $\boldsymbol{\lambda}\mathbf{CC}$ since we have added only constants to $\boldsymbol{\lambda}\mathbf{C}$. The four reduction rules of $\boldsymbol{\lambda}\mathbf{C}$ (cf. Figure 3.2) also carry over to $\boldsymbol{\lambda}\mathbf{CC}$ without any changes. The additional operators $\mathbf{phi}_{\boldsymbol{\rho}}$ introduced by $\boldsymbol{\lambda}\mathbf{CC}$ are a means to construct *compiled* compiled code, so to speak. Their reduction rule is:

$$\mathbf{phi}_{\{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}}\ \Phi\rho.e \ \rightarrow\ \Phi\rho.\Phi\{\mathbf{x}_1 : y_1, \ldots, \mathbf{x}_n : y_n\}.e \qquad (\mathbf{phi}_{\boldsymbol{\rho}})$$

Syntactic Domains:

$$\begin{array}{rcll} x & \in & \mathit{Vars} & \text{(Variables)} \\ \mathbf{x}, \mathbf{y} & \in & \mathit{Idents} & \text{(Identifiers)} \\ \tilde{\mathbf{x}} & \in & \mathit{Unlnks} & \text{(Unlinked Identifier Indicators)} \end{array}$$

Abstract Syntax:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e\ e \mid \Phi\rho.e \mid \boldsymbol{\delta} \\ \boldsymbol{\delta} & ::= & \mathbf{load} \mid \tilde{\mathbf{x}} \mid \mathbf{lam}_{\mathbf{x}} \mid \mathbf{app} \mid \mathbf{phi}_{\boldsymbol{\rho}} \\ \rho & ::= & \{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\} \\ \boldsymbol{\rho} & ::= & \{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\} \end{array}$$

Figure 5.2: Term language of twice context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{CC}$

The variable y_i of each parameter $\mathbf{x}_i : y_i$ is the parameter variable of $\mathbf{y}_i$ specified by ρ; if $\mathbf{y}_i$ is not a parameter identifier of ρ, y_i is chosen to be a fresh variable name. The contraction thus relinks the free variables $y_1, \ldots, y_n$ of e to the newly constructed specification $\{\mathbf{x}_1 : y_1, \ldots, \mathbf{x}_n : y_n\}$. Again, this is possible since the parameter variables of ρ are quasi-statically scoped (cf. the **lam**-reduction rule of Section 3.3.4).

Metaphorically speaking, each operator $\mathbf{phi}_{\boldsymbol{\rho}}$ models the filling of the hole $\mathbf{h}$ of the context $\boldsymbol{\Phi\rho}.\mathbf{h}$ with the evolved context $\mathbf{e}$. It yields the compiled code $\Phi\rho.\Phi\rho'.e$ of $\boldsymbol{\Phi\rho}.\mathbf{e}$ from the compiled code $\Phi\rho.e$ of $\mathbf{e}$. To illustrate, let the compiled code of $\boldsymbol{\Phi}\{\mathbf{x}:\mathbf{x}\}.(\mathbf{x}\ (\mathbf{y}\ \mathbf{z}))$ be the free identifier abstraction $\Phi\{\mathbf{y}:y, \mathbf{z}:z\}.\Phi\{\mathbf{x}:x\}.(x\ (y\ z))$. Then,

$$\begin{array}{rl} \multicolumn{2}{l}{\mathbf{phi}_{\{\mathbf{w}:\mathbf{y}\}}\ \Phi\{\mathbf{y}:y, \mathbf{z}:z\}.\Phi\{\mathbf{x}:x\}.(x\ (y\ z))} \\ \to & \Phi\{\mathbf{y}:y, \mathbf{z}:z\}.\Phi\{\mathbf{w}:y\}.\Phi\{\mathbf{x}:x\}.(x\ (y\ z)) \\ \approx & \Phi\{\mathbf{z}:z\}.\Phi\{\mathbf{w}:y\}.\Phi\{\mathbf{x}:x\}.(x\ (y\ z)) \end{array}$$

The contractum is a compiled version of the $\boldsymbol{\lambda}\mathbf{C}$-context $\boldsymbol{\Phi}\{\mathbf{w}:\mathbf{y}\}.\boldsymbol{\Phi}\{\mathbf{x}:\mathbf{x}\}.(\mathbf{x}\ (\mathbf{y}\ \mathbf{z}))$, the result of filling $\boldsymbol{\Phi}\{\mathbf{w}:\mathbf{y}\}.\mathbf{h}$ with $\boldsymbol{\Phi}\{\mathbf{x}:\mathbf{x}\}.(\mathbf{x}\ (\mathbf{y}\ \mathbf{z}))$.

$$
\begin{array}{rcll}
(\lambda x.e)\ e' & \rightarrow & [e'/x]e & (\beta) \\
\textbf{load}\ \Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e & \rightarrow & [\tilde{\mathbf{x}}_n/x_n]\cdots[\tilde{\mathbf{x}}_1/x_1]e & (\textbf{load}) \\
\textbf{app}\ \Phi\rho_1.e_1\ \Phi\rho_2.e_2 & \rightarrow & \Phi\rho_1 \uplus \rho_2.(e_1\ e_2) & (\textbf{app}) \\
\textbf{lam}_{\mathbf{x}}\ \Phi\rho.e & \rightarrow & \Phi\rho.\lambda x.e & (\textbf{lam}) \\
 & & \quad \text{where } \mathbf{x} : x \in \rho & \\
 & & \quad \text{or else } x \text{ is fresh} & \\
\textbf{phi}_{\{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}}\ \Phi\rho.e & \rightarrow & \Phi\rho.\Phi\{\mathbf{x}_1 : y_1, \ldots, \mathbf{x}_n : y_n\}.e & (\textbf{phi}_{\boldsymbol{\rho}}) \\
 & & \quad \text{where } \mathbf{y}_i : y_i \in \rho & \\
 & & \quad \text{or else } y_i \text{ is fresh} &
\end{array}
$$

Figure 5.3: Reduction rules of twice context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{CC}$

The notion of reduction **cc** underlying the extended calculus $\boldsymbol{\lambda}\mathbf{CC}$ is the union of the five reduction relations collectively shown in Figure 5.3:

$$\mathbf{cc} \;=\; \beta \cup \mathbf{lam} \cup \mathbf{app} \cup \mathbf{load} \cup \mathbf{phi}_{\boldsymbol{\rho}}$$

We continue to use $\rightarrow$ to denote the one-step reduction relation induced by **cc** and $\twoheadrightarrow$ to denote the reflexive and transitive closure of $\rightarrow$. Again, the least equivalence relation generated by $\twoheadrightarrow$ is the equational theory $\boldsymbol{\lambda}\mathbf{CC}$ and equivalence under $\boldsymbol{\lambda}\mathbf{CC}$ is written as $e_1 = e_2$.

Intuitively, the additional compiled code operations of $\boldsymbol{\lambda}\mathbf{CC}$ are orthogonal to the compiled code operations of $\boldsymbol{\lambda}\mathbf{C}$ as well as the β-reduction. Hence, by the Hindley-Rosen Lemma, the notion of reduction **cc** is Church-Rosser:

Theorem 5.1 *The* **cc***-reduction relation* $\twoheadrightarrow$ *satisfies the diamond property.*

Moreover, free identifier abstractions indistinguishable by the compiled code operators of $\boldsymbol{\lambda}\mathbf{C}$ (cf. Theorem 3.5) are not distinguishable to the new operators $\mathbf{phi}_{\boldsymbol{\rho}}$ either:

Theorem 5.2 *Let $\Phi\rho_1.e \approx \Phi\rho_2.e$ and*

$$\begin{array}{lcl} \mathbf{phi}_{\boldsymbol{\rho}}\ \Phi\rho_1.e & \rightarrow & \Phi\rho_1.\Phi\rho_1'.e \\ \mathbf{phi}_{\boldsymbol{\rho}}\ \Phi\rho_2.e & \rightarrow & \Phi\rho_2.\Phi\rho_2'.e \end{array}$$

Then, $\Phi\rho_1'.e \approx \Phi\rho_2'.e$ and so $\Phi\rho_1.\Phi\rho_1'.e \approx \Phi\rho_2.\Phi\rho_2'.e$.

5.3 Thrice Context-Enriched Lambda Calculus

So far, using the context-enriching schema, we have derived from the λ-calculus the once context-enriched calculus $\boldsymbol{\lambda}$**C** and the twice context-enriched calculus $\boldsymbol{\lambda}$**CC**. It is quite natural to ponder what the thrice context-enriched calculus $\boldsymbol{\lambda}$**CCC** and so forth would be.

The contexts **C** of the twice context-enriched λ-calculus $\boldsymbol{\lambda}$**CC** are (cf. Figure 5.2 for the definition of $\boldsymbol{\lambda}$**CC**-terms):

$$\begin{array}{lcl} \mathbf{C} & ::= & \mathbf{h} \mid \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C} \mid \mathbf{C}\ \mathbf{C} \mid \boldsymbol{\Phi\rho}.\mathbf{C} \mid \boldsymbol{\delta} \\ \boldsymbol{\delta} & ::= & \mathbf{load} \mid \tilde{\mathbf{x}} \mid \mathbf{lam}_{\mathbf{X}} \mid \mathbf{app} \mid \mathbf{phi}_{\boldsymbol{\rho}} \end{array}$$

Compared to the $\boldsymbol{\lambda}$**C**-contexts of Figure 5.1, the only added contexts are the operators $\mathbf{phi}_{\boldsymbol{\rho}}$. But since $\mathbf{phi}_{\boldsymbol{\rho}}$ have no free identifiers, we can represent their compiled code simply as $\Phi\{\}.\mathbf{phi}_{\boldsymbol{\rho}}$. The thrice context-enriched λ-calculus therefore requires no new incremental compiled code constructors. It is thus exactly the same as the twice context-enriched λ-calculus $\boldsymbol{\lambda}$**CC**. In other words, our repeated application of the context-enriching schema to the λ-calculus has converged to the twice context-enriched λ-calculus $\boldsymbol{\lambda}$**CC**. That is, $\boldsymbol{\lambda}$**CC** is a "fixpoint" of our design methodology.

5.4 Metacircular Self-Compilation

As shown in Chapter 4, the context-enriched version of a calculus is capable of compiling the original calculus. Here, we demonstrate the compilation of $\boldsymbol{\lambda}$**CC** in the

$\boldsymbol{\lambda}$**CC**-calculus itself, thus yielding a self-compiler for $\boldsymbol{\lambda}$**CC** [8, 52]. The evolved $\boldsymbol{\lambda}$**CC**-contexts (source code) **e** can be represented in $\boldsymbol{\lambda}$**CC** as follows (recall that a program symbol **x** is represented as the pair $\langle \mathbf{lam_x}, \Phi\{\mathbf{x}:x\}.x\rangle$ whose components are denoted as $lam(\mathbf{x})$ and $phi(\mathbf{x})$):

$$\begin{array}{rcl} [\![\mathbf{x}]\!]_c & \equiv & \langle \mathbf{0}, \mathbf{x}\rangle \\ [\![\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}]\!]_c & \equiv & \langle \mathbf{1}, \langle \mathbf{x}, [\![\mathbf{e}]\!]_c\rangle\rangle \\ [\![\mathbf{e}_1\ \mathbf{e}_2]\!]_c & \equiv & \langle \mathbf{2}, \langle [\![\mathbf{e}_1]\!]_c, [\![\mathbf{e}_2]\!]_c\rangle\rangle \\ [\![\Phi\boldsymbol{\rho}.\mathbf{e}]\!]_c & \equiv & \langle \mathbf{3}, \langle \mathbf{phi}_{\boldsymbol{\rho}}, [\![\mathbf{e}]\!]_c\rangle\rangle \\ [\![\boldsymbol{\delta}]\!]_c & \equiv & \langle \mathbf{4}, \boldsymbol{\delta}\rangle \end{array} \tag{5.1}$$

The companion compilation function $\mathcal{C}$ of the above representation schema $[\![\ ]\!]_c$ is:

$$\begin{array}{rcl} \mathcal{C}\ \langle \mathbf{0}, e\rangle & = & phi(e) \\ \mathcal{C}\ \langle \mathbf{1}, \langle e_1, e_2\rangle\rangle & = & lam(e_1)\ (\mathcal{C}\ e_2) \\ \mathcal{C}\ \langle \mathbf{2}, \langle e_1, e_2\rangle\rangle & = & \mathbf{app}\ (\mathcal{C}\ e_1)\ (\mathcal{C}\ e_2) \\ \mathcal{C}\ \langle \mathbf{3}, \langle e_1, e_2\rangle\rangle & = & e_1\ (\mathcal{C}\ e_2) \\ \mathcal{C}\ \langle \mathbf{4}, e\rangle & = & \Phi\{\}.e \end{array}$$

An interpreter $\mathcal{M}$ for $\boldsymbol{\lambda}$**CC**-programs, which are evolved $\boldsymbol{\lambda}$**CC**-contexts without free identifiers, is then the composition of the compilation function $\mathcal{C}$ and the free identifier abstraction loading operator **load**:

$$\mathcal{M} \ \equiv\ \mathbf{load} \circ \mathcal{C}$$

The compiler $\mathcal{C}$ is both compositional and metacircular. It is *compositional* since the compilation of each ($[\![\ ]\!]_c$-encoded) composite evolved $\boldsymbol{\lambda}$**CC**-context is a function of the compilation of the context's ($[\![\ ]\!]_c$-encoded) components:

$$\begin{array}{rcl} \mathcal{C}\ [\![\boldsymbol{\lambda}\mathbf{x}.\mathbf{e}]\!]_c & = & \mathbf{lam_x}\ (\mathcal{C}\ [\![\mathbf{e}]\!]_c) \\ \mathcal{C}\ [\![\mathbf{e}_1\ \mathbf{e}_2]\!]_c & = & \mathbf{app}\ (\mathcal{C}\ [\![\mathbf{e}_1]\!]_c)\ (\mathcal{C}\ [\![\mathbf{e}_2]\!]_c) \\ \mathcal{C}\ [\![\Phi\boldsymbol{\rho}.\mathbf{e}]\!]_c & = & \mathbf{phi}_{\boldsymbol{\rho}}\ (\mathcal{C}\ [\![\mathbf{e}]\!]_c) \end{array}$$

The compiler $\mathcal{C}$ is *metacircular* since it translates each category of evolved $\boldsymbol{\lambda}$**CC**-contexts into the same category of $\boldsymbol{\lambda}$**CC**-terms [45]:

Theorem 5.3 *Let* $\mathbf{e}$ *be an evolved* $\boldsymbol{\lambda}$**CC**-*context and* $\mathbf{z}_1, \ldots, \mathbf{z}_m$ *be the applied identifiers of* $\mathbf{e}$*. Then,*

$$\mathcal{C}\ [\![\mathbf{e}]\!]_c \quad = \quad \Phi\{\mathbf{z}_1 : q(\mathbf{z}_1), \ldots, \mathbf{z}_m : q(\mathbf{z}_m)\}.im(\mathbf{e})$$

Proof: This theorem is a repeat of Theorem 4.1 for evolved $\boldsymbol{\lambda}$**CC**-contexts. The proof is a straightforward induction on the structure of $\mathbf{e}$. Here, we show only the case for free identifier abstraction contexts. Let $\{\bar{\mathbf{x}} : \bar{\mathbf{y}}\}$ and $\{\bar{\mathbf{x}} : q(\bar{\mathbf{y}})\}$ abbreviate $\{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}$ and $\{\mathbf{x}_1 : q(\mathbf{y}_1), \ldots, \mathbf{x}_n : q(\mathbf{y}_n)\}$, respectively. Then,

$$\begin{array}{rcll}
\mathcal{C}\ [\![\boldsymbol{\Phi}\{\bar{\mathbf{x}} : \bar{\mathbf{y}}\}.\mathbf{e}]\!]_c & \equiv & \mathcal{C}\ \langle \mathbf{3}, \langle \mathbf{phi}_{\{\bar{\mathbf{x}} : \bar{\mathbf{y}}\}}, [\![\mathbf{e}]\!]_c \rangle\rangle & \\
& = & \mathbf{phi}_{\{\bar{\mathbf{x}} : \bar{\mathbf{y}}\}}\ (\mathcal{C}\ [\![\mathbf{e}]\!]_c) & \\
& = & \mathbf{phi}_{\{\bar{\mathbf{x}} : \bar{\mathbf{y}}\}}\ \Phi\{\bar{\mathbf{z}} : q(\bar{\mathbf{z}})\}.im(\mathbf{e}) & \text{[induction hypothesis]} \\
& = & \Phi\{\bar{\mathbf{z}} : q(\bar{\mathbf{z}})\}.\Phi\{\bar{\mathbf{x}} : q(\bar{\mathbf{y}})\}.im(\mathbf{e}) &
\end{array}$$

□

To conclude, enriching a λ-calculus with the notion of contexts once gives us an extended calculus that is capable of constructing, compiling, and linking programs of the given calculus incrementally. Enriching the given calculus with the notion of contexts twice gives us an extended calculus that is expressive enough to incrementally construct, compile, and link programs of the extended calculus metacircularly. From now on, when we enrich a calculus with contexts, it will always be done twice.

In the rest of this chapter we focus on refining the compiled code operators $\mathbf{phi}_{\boldsymbol{\rho}}$ introduced by the $\boldsymbol{\lambda}$**CC**-calculus. In particular, the operators have a rather complex reduction rule. It is beneficial to break it down to more manageable parts. From the standpoint of language design, the advantages of doing so include a better understanding of the nature of the operators and perhaps the uncovering of other fundamental compiled code operations.

5.5 Renaming Free Identifiers

Let us repeat the reduction rule for the compiled code constructors $\mathbf{phi}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}$:

$$\mathbf{phi}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ \Phi\rho.e \ \rightarrow\ \Phi\rho.\Phi\{\bar{\mathbf{x}}:\bar{y}\}.e \qquad (\mathbf{phi}_{\boldsymbol{\rho}})$$
$$\mathbf{y}_i:y_i \in \rho \text{ or else } y_i \text{ is fresh}$$

In general, the operators $\mathbf{phi}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}$ involve pairs of not necessarily distinct identifiers $\mathbf{x}_i$ and $\mathbf{y}_i$. A degenerate case is when each $\mathbf{y}_i$ is the same identifier as $\mathbf{x}_i$. Compared to $\mathbf{phi}_{\boldsymbol{\rho}}$, the degenerate operators $\mathbf{phi}_{\{\bar{\mathbf{y}}\}}$ have a slightly simpler reduction rule:

$$\mathbf{phi}_{\{\bar{\mathbf{y}}\}}\ \Phi\rho.e \ \rightarrow\ \Phi\rho.\Phi\{\bar{\mathbf{y}}:\bar{y}\}.e \qquad (\mathbf{phi}_{\bar{\mathbf{x}}})$$
$$\mathbf{y}_i:y_i \in \rho \text{ or else } y_i \text{ is fresh}$$

It is the goal of this section to take the operators $\mathbf{phi}_{\{\bar{\mathbf{y}}\}}$ as given and add the necessary capabilities to recover the full functionality of $\mathbf{phi}_{\boldsymbol{\rho}}$.

What we need is a means to rename the identifiers $\bar{\mathbf{y}}$ of $\Phi\{\bar{\mathbf{y}}:\bar{y}\}.e$ to $\bar{\mathbf{x}}$. For that, we add a new class of operators $\mathbf{rename}_{\{\mathbf{y}_1:\mathbf{x}_1,\ldots,\mathbf{y}_n:\mathbf{x}_n\}}$, abbreviated as $\mathbf{rename}_{\{\bar{\mathbf{y}}:\bar{\mathbf{x}}\}}$, We can then define $\mathbf{phi}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}$ as derived forms:

$$\mathbf{phi}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}} \ \equiv\ \lambda x.(\mathbf{app}\ \Phi\{\}.\mathbf{rename}_{\{\bar{\mathbf{y}}:\bar{\mathbf{x}}\}}\ (\mathbf{phi}_{\{\bar{\mathbf{y}}\}}\ x)) \qquad (5.2)$$

To illustrate,

$$\begin{aligned}
&\mathbf{phi}_{\{\mathbf{w}:\mathbf{x},\,\mathbf{z}:\mathbf{y}\}}\ \Phi\{\mathbf{y}:y,\mathbf{z}:z\}.\lambda x.(y\ z)\\
&\rightarrow\ \mathbf{app}\ \Phi\{\}.\mathbf{rename}_{\{\mathbf{x}:\mathbf{w},\,\mathbf{y}:\mathbf{z}\}}\ \underline{(\mathbf{phi}_{\{\mathbf{x},\,\mathbf{y}\}}\ \Phi\{\mathbf{y}:y,\mathbf{z}:z\}.\lambda x.(y\ z))}\\
&\rightarrow\ \underline{\mathbf{app}\ \Phi\{\}.\mathbf{rename}_{\{\mathbf{x}:\mathbf{w},\,\mathbf{y}:\mathbf{z}\}}\ \Phi\{\mathbf{y}:y,\mathbf{z}:z\}.\Phi\{\mathbf{x}:w,\mathbf{y}:y\}.\lambda x.(y\ z)}\\
&\rightarrow\ \Phi\{\mathbf{y}:y,\mathbf{z}:z\}.\underline{(\mathbf{rename}_{\{\mathbf{x}:\mathbf{w},\,\mathbf{y}:\mathbf{z}\}}\ \Phi\{\mathbf{x}:w,\mathbf{y}:y\}.\lambda x.(y\ z))}\\
&\rightarrow\ \Phi\{\mathbf{y}:y,\mathbf{z}:z\}.\Phi\{\mathbf{w}:w,\mathbf{z}:y\}.\lambda x.(y\ z)
\end{aligned}$$

The last reduction step shows the effect of the renaming operator $\mathbf{rename}_{\{\mathbf{x}:\mathbf{w},\,\mathbf{y}:\mathbf{z}\}}$ on the free identifier abstraction $\Phi\{\mathbf{x}:w,\mathbf{y}:y\}.\lambda x.(y\ z)$. The parameter identifiers $\mathbf{x}$ and $\mathbf{y}$ are renamed to $\mathbf{w}$ and $\mathbf{z}$, respectively.

Metaphorically speaking, the new operators model the renaming of free identifiers of contexts. Their reduction rule might be:

$$\mathbf{rename}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ \Phi\{\bar{\mathbf{z}}:\bar{z}\}.e \quad\rightarrow\quad \Phi\{\bar{\mathbf{w}}:\bar{z}\}.e$$

where each $\mathbf{w}_i$ is some $\mathbf{y}_j$ when $\mathbf{z}_i$ is the same identifier as $\mathbf{x}_j$; otherwise, $\mathbf{w}_i$ is $\mathbf{z}_i$. That is, the contractum $\Phi\{\bar{\mathbf{w}}:\bar{z}\}.e$ is similar to the given $\Phi\{\bar{\mathbf{z}}:\bar{z}\}.e$ except that each $\mathbf{z}_i \equiv \mathbf{x}_j$ is renamed to $\mathbf{y}_j$.

There is just one problem—the resulting free identifier abstraction parameter specification $\{\bar{\mathbf{w}}:\bar{z}\}$ may not be well-formed. Indeed, there is no guarantee that the identifiers $\bar{\mathbf{w}}$ are pairwise distinct. The cause of the problem is that the pool from which the identifiers $\bar{\mathbf{w}}$ are drawn, which comprises the identifiers $\bar{\mathbf{y}}$ and $\bar{\mathbf{z}}$, does not necessarily form a set. That is, either $\bar{\mathbf{y}}$ are not pairwise distinct or $\bar{\mathbf{z}}$ are not distinct from $\bar{\mathbf{y}}$.

Adding the constraint that the replacement identifiers $\bar{\mathbf{y}}$ of a renaming operator $\mathbf{rename}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}$ must be pairwise distinct does not solve the problem. There is still the possibility that $\bar{\mathbf{y}}$ and $\bar{\mathbf{z}}$ may not be disjoint. An instinctive approach is to prohibit such a reduction whenever the resulting specification $\{\bar{\mathbf{w}}:\bar{z}\}$ is ill-formed. It is not compatible with our notion of free identifier abstraction indistinguishability, however. Two previously indistinguishable abstractions such as $\Phi\{\}.\lambda x.x$ and $\Phi\{\mathbf{x}:x,\mathbf{y}:y\}.\lambda x.x$ would be distinguishable by the renaming operator $\mathbf{rename}_{\{\mathbf{x}:\mathbf{y}\}}$:

$$\mathbf{rename}_{\{\mathbf{x}:\mathbf{y}\}}\ \Phi\{\}.\lambda x.x \quad\rightarrow\quad \Phi\{\}.\lambda x.x$$

but

$$\mathbf{rename}_{\{\mathbf{x}:\mathbf{y}\}}\ \Phi\{\mathbf{x}:x,\mathbf{y}:y\}.\lambda x.x \quad\not\rightarrow\quad \Phi\{\mathbf{y}:x,\mathbf{y}:y\}.\lambda x.x$$

since the term on the left hand side is not a redex.

A proper solution can be obtained via parameter renaming. When two identifiers $\mathbf{w}_i$ and $\mathbf{w}_j$ of $\{\bar{\mathbf{w}}:\bar{z}\}$ are the same, we can identify their corresponding parameter variables z_i and z_j by renaming them to the same variable. This is consistent with our intuition that both z_i and z_j in e denote the placeholder for the same free identifier and so they should be the same variable. Thus, the reduction rule for the identifier renaming operators $\mathbf{rename}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}$ is:

$$\mathbf{rename}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ \Phi\{\bar{\mathbf{z}}:\bar{z}\}.e \ \rightarrow\ \Phi wf\{\bar{\mathbf{w}}:\bar{w}\}.\langle\bar{w}/\bar{z}\rangle e \qquad \textbf{(ren)}$$

$$\text{where}\quad \mathbf{w}_i \ \equiv\ \begin{cases} \mathbf{y}_j & \text{if } \mathbf{z}_i \equiv \mathbf{x}_j \\ \mathbf{z}_i & \text{otherwise} \end{cases}$$

and $\bar{w}$ are not necessarily distinct
fresh variables with respect to e such
that $w_i \equiv w_j$ if and only if $\mathbf{w}_i \equiv \mathbf{w}_j$

The notation $\langle\bar{w}/\bar{z}\rangle e$ simultaneously renames the parameter variables $\bar{z}$ in e to $\bar{w}$, respectively. The new variables $\bar{w}$ may not be pairwise distinct. Whenever $\mathbf{w}_i$ and $\mathbf{w}_j$ are the same identifeir, so are their respective variables w_i and w_j. Consequently, the parameter specification $\{\bar{\mathbf{w}}:\bar{w}\}$ may not be well-formed. The notation $wf\{\bar{\mathbf{w}}:\bar{w}\}$ denotes the well-formed version of $\{\bar{\mathbf{w}}:\bar{w}\}$ by removing any duplicates. Formally,

$$wf\{\mathbf{w}_1:w_1,\ldots,\mathbf{w}_m:w_m\} \ \equiv\ \{\mathbf{w}_{a_1}:w_{a_1},\ldots\mathbf{w}_{a_k}:w_{a_k}\}$$

where $1 \leq a_i \leq m$ for all $1 \leq i \leq k$ and for each $1 \leq j \leq m$ there is exactly one a_i such that $\mathbf{w}_j \equiv \mathbf{w}_{a_i}$ and $w_j \equiv w_{a_i}$.

As an example,

$$\begin{aligned} &\mathbf{rename}_{\{\mathbf{x}:\mathbf{y},\,\mathbf{z}:\mathbf{x}\}}\ \Phi\{\mathbf{x}:x,\mathbf{y}:y,\mathbf{z}:z\}.(x\ y\ z) \\ \rightarrow\ &\Phi wf\{\mathbf{y}:w,\mathbf{y}:w,\mathbf{x}:v\}.\langle w/x,w/y,v/z\rangle(x\ y\ z) \\ \equiv\ &\Phi\{\mathbf{x}:v,\mathbf{y}:w\}.(w\ w\ v) \end{aligned} \tag{5.3}$$

Intuitively, the argument Φ-abstraction $\Phi\{\mathbf{x}:x,\mathbf{y}:y,\mathbf{z}:z\}.(x\ y\ z)$ represents the context $\mathbf{C}_1 \equiv \mathbf{x}\ \mathbf{y}\ \mathbf{z}$ and the contractum Φ-abstraction $\Phi\{\mathbf{x}:v,\mathbf{y}:w\}.(w\ w\ v)$ models the context $\mathbf{C}_2 \equiv \mathbf{y}\ \mathbf{y}\ \mathbf{x}$, the result of simultaneously renaming the identifiers $\mathbf{x}$ and $\mathbf{z}$ of $\mathbf{C}_1$ to $\mathbf{y}$ and $\mathbf{x}$, respectively.

This concludes the first part of the thesis. We have demonstrated the techniques involved in applying our context-enriching schema to the λ-calculus to yield a calculus that is capable of expressing the incremental construction of its own programs metacircularly. The transparency of our compiled code abstractions shows that it is straightforward for our design methodology to deal with programming languages with additional special forms that do not introduce variable linking relations. In the second part of our work, we focus on extending our schema to deal with more descriptive variable binding forms. In particular, we apply the techniques learned so far to more elaborate versions of the λ-calculus with fancier variable defining and referencing mechanisms. The resulting context-enriched λ-calculi have incremental program construction capabilities that are not only more expressive, but also more intuitive.

Chapter 6

Context-Enriched Calculus of Definitions

We have demonstrated in the previous chapters the basic techniques underlying our context-enriching schema. In this and the next chapters we apply them to more elaborate variable defining and referencing mechanisms. The results are twofold:

- We show that the schema can be easily extended to more descriptive versions of the λ-calculus that are not far removed from most high-level programming languages in practice.
- The context-enriched λ-calculi are expressive enough to capture the essence of advanced linking mechanisms fundamental to modules and objects in a very intuitive manner. Hence, the schema can serve as a simple means to enhance existing languages with modular and object-oriented programming capabilities.

Together, they further support our claim that enriching a programming language with the behavior of contexts is a useful language design methodology.

In this chapter we extend the λ-calculus with more intricate variable defining mechanisms and then apply our context-enriching schema to the extended calculus. It is motivated by our inability to give a clean description of first-class environments using the $\boldsymbol{\lambda}\mathbf{C}$-calculus; see Section 3.8.2. Here, by abstracting over some common programming idioms about variable definition, we arrive at an elegant and extensible alternative that is a module manipulation calculus in which linking is modeled by, as expected, variable capture.

6.1 Definitions

The variable defining mechanisms of concern are adapted from Plotkin's work in which he uses them to illustrate his structural approach to operational semantics [59]. The λ-calculus extended with definitions has the following abstract syntax:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e\ e \mid \mathit{let}\ d\ \mathit{in}\ e \\ d & ::= & \{x_1 = e, \ldots, x_n = e\} \\ & & x_1, \ldots, x_n \text{ pairwise distinct} \end{array} \qquad (\lambda_d)$$

A definition $\{x_1 = e_1, \ldots, x_n = e_n\}$ is a set of independent *bindings* $x_i = e_i$ that associate the *defining* variables $x_1, \ldots, x_n$ with their *denotation* terms $e_1, \ldots, e_n$, respectively. The semantics of definitions can be best understood through the following syntactic expansion:

$$\mathit{let}\ \{x_1 = e_1, \ldots, x_n = e_n\}\ \mathit{in}\ e \quad \equiv \quad \begin{cases} (\lambda x_1 \cdots x_n.e)\ e_1 \cdots e_n & \text{if } n > 0 \\ e & \text{otherwise} \end{cases}$$

Hence, definitions are merely a convenient notation for expressing some frequently used programming idioms.

There are compelling reasons to take definitions as core constructs of a programming language, however. They are more efficient to implement directly than their syntactically expanded counterparts [56]. Furthermore, in a statically-typed language, they are essential to polymorphic type inference because the scope of each denotation term is known statically [49]. Most importantly, they occur so often in programs that they deserve a special status. In our work, we take the best of both views. Semantically, we regard definitions as syntactic sugar to avoid introducing additional mechanisms to explain their computational behavior. Syntactically, we treat them as core constructs to demonstrate the incremental program construction capabilities induced by our context-enriching schema.

6.2 Context-Enriched Definition Calculus

Our context-enriching schema adds these mechanisms to a calculus to simulate the notion of contexts:

- compiled code abstractions to simulate evolved contexts,
- a compiled code loading operation to assimilate compiled code into machine code, and
- a compiled code construction operator for each category of composite contexts.

In enhancing the definition calculus λ_d with the behavior of contexts, the first task is therefore to define the notion of λ_d-contexts. One obvious possibility is shown in Figure 6.1. Another is to instantiate the **D**-part of **let D in C**, hence rendering **D**-contexts unnecessary:

$$\mathbf{C} \ ::= \ \mathbf{h} \mid \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C} \mid \mathbf{C}\ \mathbf{C} \mid \mathbf{let}\ \{\mathbf{x}_1 = \mathbf{C}, \ldots, \mathbf{x}_n = \mathbf{C}\}\ \mathbf{in}\ \mathbf{C}$$

We do not give preference to the latter because it would require a separate compiled code constructor for each **let**-context with a distinct set of identifiers $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. In contrast, the former needs only a single (polymorphic) **let**-context constructor. More importantly, by identifying **D** as a distinct category of contexts, we can replace them with holes. Indeed, a λ_d-context **let h in C** expresses

$$(\boldsymbol{\lambda}\_.\mathbf{C})\ \_$$

where the hole **h** actually consists of the two underscored parts. The latter is clearly not a legal λ-context, hence the increased expressiveness of λ_d-contexts. Moreover, we can add **D**-context manipulation operations in the future in a modular fashion, as we will see in the second half of this chapter.

Contexts:

$$\begin{aligned}\mathbf{C} &::= \mathbf{h} \mid \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C} \mid \mathbf{C}\;\mathbf{C} \mid \mathbf{let}\ \mathbf{D}\ \mathbf{in}\ \mathbf{C} \\ \mathbf{D} &::= \mathbf{h} \mid \{\mathbf{x}_1 = \mathbf{C}, \ldots, \mathbf{x}_n = \mathbf{C}\}\end{aligned}$$

Evolved Contexts:

$$\begin{aligned}\mathbf{e} &::= \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{e} \mid \mathbf{e}\;\mathbf{e} \mid \mathbf{let}\ \mathbf{d}\ \mathbf{in}\ \mathbf{e} \\ \mathbf{d} &::= \{\mathbf{x}_1 = \mathbf{e}, \ldots, \mathbf{x}_n = \mathbf{e}\}\end{aligned}$$

Figure 6.1: Contexts of definition calculus

There are now two categories of evolved λ_d-contexts, namely, the evolved term-contexts $\mathbf{e}$ and the evolved definition-contexts $\mathbf{d}$; see Figure 6.1. As usual, we continue to use free identifier abstractions $\Phi\rho.e$ to play the role of compiled evolved term-contexts $\mathbf{e}$. We also need a second category of abstractions to model the compiled code of evolved definition-contexts $\mathbf{d}$. One reason for the new compiled code abstractions dedicated to definitions is that the body of a free identifier abstraction $\Phi\rho.d$ would be a definition d, which is not a proper λ_d-term. Hence, the compiled code loading operation **load** $\Phi\rho.d$ would introduce definitions d as proper terms into the context-enriched calculus. Furthermore, a free identifier abstraction $\Phi\rho.d$ can only model the free identifiers of a $\mathbf{d}$-context but not its defining identifiers.

A new category of compiled code for the $\mathbf{d}$-contexts is essential. We thus introduce *definition abstractions* (Ψ*-abstractions*)

$$\Psi\langle\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_m : x_m\}, \{\mathbf{y}_1 : y_1, \ldots, \mathbf{y}_n : y_n\}\rangle.\{z_1 = e_1, \ldots, z_k = e_k\}$$

which we often abbreviate as $\Psi\langle\{\bar{\mathbf{x}} : \bar{x}\}, \{\bar{\mathbf{y}} : \bar{y}\}\rangle.\{\bar{z} = \bar{e}\}$ or simply as $\Psi\langle\varepsilon, \rho\rangle.d$. They are conceptual substitutes for definitions d as first-class citizens. The parameters $\{\bar{\mathbf{y}} : \bar{y}\}$ serve the same purpose as the ρ-parameters of a Φ-abstraction $\Phi\rho.e$. They specify the free identifiers $\bar{\mathbf{y}}$ whose corresponding placeholders in the machine code $\bar{e}$ are the variables $\bar{y}$. The parameters $\{\bar{\mathbf{x}} : \bar{x}\}$ identify the defining variables $\bar{z}$ of the

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x.e \mid e\ e \mid \Phi\rho.e \mid \Psi\langle\varepsilon,\rho\rangle.d \mid \boldsymbol{\delta} \\
\boldsymbol{\delta} & ::= & \mathbf{load} \mid \tilde{\mathbf{x}} \mid \mathbf{lam_X} \mid \mathbf{app} \mid \mathbf{let} \mid \mathbf{dfn_X} \\
d & ::= & \{x_1 = e, \ldots, x_n = e\} \\
 & & \quad x_1, \ldots, x_n \text{ pairwise distinct} \\
\varepsilon, \rho & ::= & \{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}
\end{array}
$$

Figure 6.2: Term language of context-enriched definition calculus $\boldsymbol{\lambda}\mathbf{D}$

definition $\{\bar{z} = \bar{e}\}$ externally as the identifiers $\bar{\mathbf{x}}$, which are the *defining* identifiers of the Ψ-abstraction.

Similar to Φ-abstractions, the ρ-parameters of $\Psi\langle\varepsilon,\rho\rangle.d$ need not coincide with the free variables of d. The ε-parameters must be a subset of the defining variables of d, however. That is, for any $\Psi\langle\{\bar{\mathbf{x}}:\bar{x}\}, \{\bar{\mathbf{y}}:\bar{y}\}\rangle.\{\bar{z} = \bar{e}\}$, $\{\bar{x}\} \subseteq \{\bar{z}\}$ must hold. If ε were allowed to specify more parameters than are defined by d, there would be defining identifiers with vacuous denotations. The correctness of code that depends on their existence would be jeopardized. On the other hand, if d defines more variables than are specified by ε, the bindings of the extraneous defining variables can always be considered hidden.

The syntax of the once context-enriched λ_d-calculus $\boldsymbol{\lambda}\mathbf{D}$ is summarized in Figure 6.2. In addition to Ψ-abstractions, $\boldsymbol{\lambda}\mathbf{D}$ requires two more compiled code operators than $\boldsymbol{\lambda}\mathbf{C}$. There is the binary operator **let** that constructs the compiled code of **let d in e** from the compiled code of **d** and **e**. There are also the unary operators $\mathbf{dfn_X}$, one for each identifier **x**, that build the compiled code of $\{\mathbf{x} = \mathbf{e}\}$ from the compiled code of **e**. (Compiled code construction of multiple binding definition-contexts $\{\bar{\mathbf{x}} = \bar{\mathbf{e}}\}$ is deferred to Section 6.5.1.) The operator **let** serves as a mechanism that converts Ψ-abstractions to Φ-abstractions. In contrast, $\mathbf{dfn_X}$ converts Φ-abstractions to Ψ-abstractions.

Contexts:

$$\begin{aligned} \mathbf{C} &::= \mathbf{h} \mid \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C} \mid \mathbf{C}\ \mathbf{C} \mid \mathbf{let\ D\ in\ C} \mid \boldsymbol{\Phi\rho}.\mathbf{C} \mid \boldsymbol{\Psi}\langle\varepsilon,\boldsymbol{\rho}\rangle.\mathbf{D} \mid \boldsymbol{\delta} \\ \mathbf{D} &::= \mathbf{h} \mid \{\mathbf{x}_1 = \mathbf{C},\ldots,\mathbf{x}_n = \mathbf{C}\} \end{aligned}$$

Evolved Contexts:

$$\begin{aligned} \mathbf{e} &::= \mathbf{x} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{e} \mid \mathbf{e}\ \mathbf{e} \mid \mathbf{let\ d\ in\ e} \mid \boldsymbol{\Phi\rho}.\mathbf{e} \mid \boldsymbol{\Psi}\langle\varepsilon,\boldsymbol{\rho}\rangle.\mathbf{d} \mid \boldsymbol{\delta} \\ \mathbf{d} &::= \{\mathbf{x}_1 = \mathbf{e},\ldots,\mathbf{x}_n = \mathbf{e}\} \end{aligned}$$

Figure 6.3: Contexts of **λD**-calculus

6.3 Twice Context-Enriched Definition Calculus

Following the development of the twice context-enriched λ-calculus **λCC**, we look for a fixpoint of our schema. Hence, rather than giving a full description of the once context-enriched definition calculus **λD**, we proceed directly to the twice context-enriched definition calculus **λDD**. The derivation of **λDD** parallels the development of **λCC**. The goal is to incorporate enough compilation mechanisms to express a metacircular self-compiler of **λDD**.

6.3.1 Term Language

The **λD**-contexts are shown in Figure 6.3. The syntax of the twice context-enriched definition calculus **λDD** is given in Figure 6.4.

The twice context-enriched calculus **λDD** conservatively extends **λD**. In addition to the compiled code operators of **λD**, we need mechanisms to simulate the filling of these two categories of **λD**-contexts: $\boldsymbol{\Phi\rho}.\mathbf{h}$ and $\boldsymbol{\Psi}\langle\varepsilon,\boldsymbol{\rho}\rangle.\mathbf{h}$. For the former, there are the operators $\mathbf{phi}_{\boldsymbol{\rho}}$ introduced by **λCC**; see Chapter 5. They build the compiled code of $\boldsymbol{\Phi\rho}.\mathbf{e}$ from the compiled code of $\mathbf{e}$. For the latter, we need a new category of operators $\mathbf{psi}_{\varepsilon,\boldsymbol{\rho}}$, one for each pair of ε and $\boldsymbol{\rho}$, to construct the compiled code of

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x.e \mid e\ e \mid \Phi\rho.e \mid \Psi\langle\varepsilon,\rho\rangle.d \mid \boldsymbol{\delta} \\
\boldsymbol{\delta} & ::= & \mathbf{load} \mid \tilde{\mathbf{x}} \mid \mathbf{lam_X} \mid \mathbf{app} \mid \mathbf{let} \mid \mathbf{dfn_X} \mid \mathbf{phi}_{\boldsymbol{\rho}} \mid \mathbf{psi}_{\boldsymbol{\rho}} \mid \mathbf{eps}_{\mathcal{E}} \\
d & ::= & \{x_1 = e, \ldots, x_n = e\} \\
\varepsilon, \rho & ::= & \{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\} \\
\boldsymbol{\varepsilon}, \boldsymbol{\rho} & ::= & \{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}
\end{array}
$$

Figure 6.4: Term language of twice context-enriched definition calculus **$\boldsymbol{\lambda}$DD**

$\boldsymbol{\Psi}\langle\varepsilon, \boldsymbol{\rho}\rangle.\mathbf{d}$ from the compiled code of $\mathbf{d}$. For reasons to be explained later, we further split the operators $\mathbf{psi}_{\mathcal{E},\boldsymbol{\rho}}$ into two categories of more primitive operation: $\mathbf{psi}_{\boldsymbol{\rho}}$ and $\mathbf{eps}_{\mathcal{E}}$. Conceptually, the degenerate operators $\mathbf{psi}_{\boldsymbol{\rho}}$ simulate the filling of the hole $\mathbf{h}_1$ of the context $\boldsymbol{\Psi}\langle\mathbf{h}_2, \boldsymbol{\rho}\rangle.\mathbf{h}_1$ with the context $\mathbf{d}$, yielding the partially complete context $\boldsymbol{\Psi}\langle\mathbf{h}_2, \boldsymbol{\rho}\rangle.\mathbf{d}$. There are then the new operators $\mathbf{eps}_{\mathcal{E}}$ to simulate the filling of the remaining hole $\mathbf{h}_2$ of $\boldsymbol{\Psi}\langle\mathbf{h}_2, \boldsymbol{\rho}\rangle.\mathbf{d}$ with ε.

The twice context-enriched definition caluclus **$\boldsymbol{\lambda}$DD** is a conservative extension of the **$\boldsymbol{\lambda}$CC**-calculus (cf. Figure 5.2). It adds to **$\boldsymbol{\lambda}$CC** these new compilation mechanisms: Ψ-abstractions $\Psi\langle\varepsilon, \rho\rangle.d$ and Ψ-abstraction operators **let**, $\mathbf{dfn_X}$, $\mathbf{psi}_{\boldsymbol{\rho}}$, and $\mathbf{eps}_{\mathcal{E}}$.

6.3.2 Alpha Convertibility

The first step toward defining the twice context-enriched definition calculus **$\boldsymbol{\lambda}$DD** is the formalization of its α-conversion rule. As before, a fundamental syntactic notion needed in formalizing the α-equivalence relation among **$\boldsymbol{\lambda}$DD**-terms is the set of free variables. Intuitively, an occurrence of a variable x is free in a term e if it does not refer to a λ-, Φ-, or Ψ-parameter. The set of free variables occurring in a term e is denoted as $fv(e)$. Its definition extends the counterpart of **$\boldsymbol{\lambda}$CC** with the following clause for the newly introduced definition abstractions:

$$fv(\Psi\langle\varepsilon,\{\bar{\mathbf{x}}:\bar{x}\}\rangle.d) \;=\; fv(d)\setminus\{\bar{x}\}$$

where the set of free variables occurring in a definition d, denoted as $fv(d)$, consists of the free variables occurring in the denotation terms:

$$fv(\{x_1=e_1,\ldots,x_n=e_n\}) \;=\; fv(e_1)\cup\cdots\cup fv(e_n)$$

The parameters of λ-, Φ-, and Ψ-abstractions, and the defining variables of Ψ-abstractions are α-convertible. Their reduction rules are:

$$\begin{array}{rcll}
\lambda x.e & \rightarrow & \lambda y.\langle y/x\rangle e & (\alpha\text{-}\lambda)\\
\Phi\{\bar{\mathbf{x}}:\bar{x}\}.e & \rightarrow & \Phi\{\bar{\mathbf{x}}:\langle y/x\rangle\bar{x}\}.\langle y/x\rangle e & (\alpha\text{-}\Phi)\\
\Psi\langle\varepsilon,\{\bar{\mathbf{x}}:\bar{x}\}\rangle.\{\bar{z}=\bar{e}\} & \rightarrow & \Psi\langle\varepsilon,\{\bar{\mathbf{x}}:\langle y/x\rangle\bar{x}\}\rangle.\{\bar{z}=\langle y/x\rangle\bar{e}\} & (\alpha\text{-}\Psi_\rho)\\
\Psi\langle\{\bar{\mathbf{x}}:\bar{x}\},\rho\rangle.\{\bar{z}=\bar{e}\} & \rightarrow & \Psi\langle\{\bar{\mathbf{x}}:\langle y/x\rangle\bar{x}\},\rho\rangle.\{\langle y/x\rangle\bar{z}=\bar{e}\} & (\alpha\text{-}\Psi_\varepsilon)\\
\Psi\langle\{\bar{\mathbf{x}}:\bar{x}\},\rho\rangle.\{\bar{z}=\bar{e}\} & \rightarrow & \Psi\langle\{\bar{\mathbf{x}}:\bar{x}\},\rho\rangle.\{\langle y/x\rangle\bar{z}=\bar{e}\}\quad\text{where } x\notin\{\bar{x}\} & (\alpha\text{-}\Psi_d)
\end{array}$$

In each of the five rules, the new variable y is not the same as the parameter variable or the defining variable it replaces, *i.e.*, $y\not\equiv x$. Furthermore, it is fresh in the following sense:

(α-λ) The variable y is not a free variable of e. The notation $\langle y/x\rangle e$ denotes the renaming of all free occurrences of x in e to y. Its definition is given below.

(α-Φ) The variable y is not a free variable of e. It is not one of the parameter variables $\bar{x}$ either. The notation $\langle y/x\rangle\bar{x}$ denotes the renaming of the binding occurrence of x in $\bar{x}$ to y.

(α-Ψ_ρ) The variable y is not a free variable of the denotation terms $\bar{e}$. It is not one of the parameter variables $\bar{x}$ either. The notation $\langle y/x\rangle\bar{e}$ denotes the renaming of all free occurrences of x in $\bar{e}$ to y.

(α-Ψ_ε) The variable y is not one of the defining variables $\bar{z}$.

(α-Ψ_d) The variable y is not one of the defining variables $\bar{z}$.

The notion of reduction α underlying the α-convertibility among **λDD**-terms is the union of the above five renaming rules:

$$\alpha \quad = \quad \alpha\text{-}\lambda \cup \alpha\text{-}\Phi \cup \alpha\text{-}\Psi_\rho \cup \alpha\text{-}\Psi_\varepsilon \cup \alpha\text{-}\Psi_d \tag{α}$$

The α-substitution meta-operation $\langle y/x\rangle e$ that replaces y for the free occurrences of x in e extends the same operation for the **λCC**-terms with the following clause:

$$\langle y/x\rangle\Psi\langle\varepsilon,\rho\rangle.\{\bar{z}=\bar{e}\} \quad \equiv \quad \Psi\langle\varepsilon,\rho\rangle.\{\bar{z}=\langle y/x\rangle\bar{e}\}$$

The parameter variables of ρ are assumed to be distinct from x and y. This is made possible by the hygiene variable convention.

6.3.3 Reduction Rules

The twice context-enriched definition calculus **λDD** has nine reduction rules. The four reduction rules for compiled code construction operators **load**, **lam$_{\mathbf{X}}$**, **app**, and **phi$_{\boldsymbol{\rho}}$** carry over from **λCC** without changes; see Section 5.2. They continue to work on Φ-abstractions only, however. The other five reduction rules are described below.

Function Invocation

The β-reduction rule that models function invocation is still of the form:

$$(\lambda x.e)\; e' \quad \rightarrow \quad [e'/x]e \tag{β}$$

The β-substitution meta-operation $[e'/x]e$ is extended with the following clause for definition abstractions:

$$[e'/x]\Psi\langle\varepsilon,\rho\rangle.\{\bar{z}=\bar{e}\} \quad \equiv \quad \Psi\langle\varepsilon,\rho\rangle.\{\bar{z}=[e'/x]\bar{e}\}$$

Again, we have used the variable hygiene convention to ensure that β-substitution does not cause inadvertent variable capture. Hence, x is not one of the parameter variables of ρ and none of the parameter variables of ρ are free in e'.

Constructing Compiled Let

The incremental compiled code construction operator **let** models the filling of the context **let** $\mathbf{h}_1$ **in** $\mathbf{h}_2$ with the contexts **d** and **e**. Its reduction rule is:

$$\mathbf{let}\ \Psi\langle\varepsilon_1, \rho_1\rangle.d\ \Phi\rho_2.e \quad \rightarrow \quad \Phi\rho_1 \uplus \rho_2.(let\ d\ in\ e) \qquad (\mathbf{let})$$
$$\text{where } \varepsilon_1 \uplus \rho_2$$

It yields the compiled version of **let d in e** expressed as $\Phi\rho_1 \uplus \rho_2.(let\ d\ in\ e)$ from the compiled code $\Psi\langle\varepsilon_1, \rho_1\rangle.d$ of **d** and the compiled code $\Phi\rho_2.e$ of **e**. Similar to the **app**-rule, the constraint $\rho_1 \uplus \rho_2$ ensures the well-formedness of the union of the two parameter specifications ρ_1 and ρ_2. It states that:

$$(\mathbf{x}\!:\!x_1 \in \rho_1\ \&\ \mathbf{x}\!:\!x_2 \in \rho_2)\ \text{ if and only if }\ x_1 \equiv x_2$$
$$\text{and}$$
$$(\mathbf{x}_1\!:\!x \in \rho_1\ \&\ \mathbf{x}_2\!:\!x \in \rho_2)\ \text{ if and only if }\ \mathbf{x}_1 \equiv \mathbf{x}_2$$

The other constraint $\varepsilon_1 \uplus \rho_2$ means

$$(\mathbf{x}\!:\!x_1 \in \varepsilon_1\ \&\ \mathbf{x}\!:\!x_2 \in \rho_2)\ \text{ if and only if }\ x_1 \equiv x_2$$
$$\text{and}$$
$$(\mathbf{x}_1\!:\!x \in \varepsilon_1\ \&\ \mathbf{x}_2\!:\!x \in \rho_2)\ \text{ if and only if }\ \mathbf{x}_1 \equiv \mathbf{x}_2$$

The defining variables of d specified in ε_1 capture the free variables of e specified in ρ_2 only when their identifiers match. Intuitively, it models the linking of the free identifiers of ρ_2 to the defining identifiers of ε_1.

We have learned from the reduction rule for the compiled code constructor **app** that the first constraint $\rho_1 \uplus \rho_2$ can be met by renaming the parameter variables of ρ_1 and ρ_2 (cf. Section 3.3). Likewise, the second constraint $\varepsilon_1 \uplus \rho_2$ can also be met by renaming the parameter variables of ε_1 and ρ_2. Furthermore, since the parameter specifications ε_1 and ρ_1 have no correlations, the two constraints $\rho_1 \uplus \rho_2$ and $\varepsilon_1 \uplus \rho_2$ are satisfiable at the same time by appropriate renaming of the parameter variables of ε_1, ρ_1, and ρ_2.

For instance, consider contracting the following **let**-redex:

$$\textbf{let}\ \Psi\langle\{\mathbf{x}\!:\!x\},\{\mathbf{x}\!:\!x',\mathbf{z}\!:\!z\}\rangle.\{x=x',y=z\}\ \Phi\{\mathbf{x}\!:\!x'',\mathbf{z}\!:\!z'\}.(x''\ z'\ y)$$

Here, ε_1 is $\{\mathbf{x}\!:\!x\}$, ρ_1 is $\{\mathbf{x}\!:\!x',\mathbf{z}\!:\!z\}$, and ρ_2 is $\{\mathbf{x}\!:\!x'',\mathbf{z}\!:\!z'\}$. To form the union $\rho_1 \uplus \rho_2$ of the two sets ρ_1 and ρ_2, we must identify x' with x'' and z with z'. Moreover, the constraint $\varepsilon_1 \uplus \rho_2$ requires us to identify x with x'' as well. Hence, we rename x, x', and x'' to the same fresh w; similarly, z and z' are renamed to the same fresh v. Other than that, according to the variable hygiene convention, no variable capture should occur; hence, the defining variable y of the Ψ-abstraction is renamed to a fresh y' so that it is different from the free variable y of the Φ-abstraction. We thus have

$$\begin{aligned}
&\textbf{let}\ \Psi\langle\{\mathbf{x}\!:\!x\},\{\mathbf{x}\!:\!x',\mathbf{z}\!:\!z\}\rangle.\{x=x',y=z\}\ \Phi\{\mathbf{x}\!:\!x'',\mathbf{z}\!:\!z'\}.(x''\ z'\ y)\\
&\quad\equiv\quad \textbf{let}\ \Psi\langle\{\mathbf{x}\!:\!w\},\{\mathbf{x}\!:\!w,\mathbf{z}\!:\!v\}\rangle.\{w=w,y'=v\}\ \Phi\{\mathbf{x}\!:\!w,\mathbf{z}\!:\!v\}.(w\ v\ y)\\
&\quad\rightarrow\quad \Phi\{\mathbf{x}\!:\!w,\mathbf{z}\!:\!v\}.(\mathit{let}\ \{w=w,y'=v\}\ \mathit{in}\ (w\ v\ y))
\end{aligned}$$

In terms of contexts, the first argument $\Psi\langle\{\mathbf{x}\!:\!x\},\{\mathbf{x}\!:\!x',\mathbf{z}\!:\!z\}\rangle.\{x=x',y=z\}$ is the compiled code of the context $\mathbf{d} \equiv \{\mathbf{x}=\mathbf{x}, y=\mathbf{z}\}$ and the second argument $\Phi\{\mathbf{x}\!:\!x'',\mathbf{z}\!:\!z'\}.(x''\ z'\ y)$ is the compiled version of the context $\mathbf{e} \equiv \mathbf{x}\ \mathbf{z}\ y$. The contractum $\Phi\{\mathbf{x}\!:\!w,\mathbf{z}\!:\!v\}.(\mathit{let}\ \{w=w,y'=v\}\ \mathit{in}\ (w\ v\ y))$ is the compiled code of the context **let** $\{\mathbf{x}=\mathbf{x}, y'=\mathbf{z}\}$ **in** $(\mathbf{x}\ \mathbf{z}\ y)$, the result of filling **let** $\mathbf{h}_1$ **in** $\mathbf{h}_2$ with $\mathbf{d}$ and $\mathbf{e}$.

Constructing Compiled Definition

The incremental compiled code constructors $\mathbf{dfn}_\mathbf{X}$, one for each identifier $\mathbf{x}$, model the filling of the context $\{\mathbf{x}=\mathbf{h}\}$ with the context $\mathbf{e}$. Their reduction rule is:

$$\mathbf{dfn}_\mathbf{X}\ \Phi\rho.e \quad\rightarrow\quad \Psi\langle\{\mathbf{x}\!:\!x\},\rho\rangle.\{x=e\} \qquad (\mathbf{dfn})$$

where the choice of the defining variable x is arbitrary. It constructs the compiled code of $\{\mathbf{x}=\mathbf{e}\}$ modeled as the definition abstraction $\Psi\langle\{\mathbf{x}\!:\!x\},\rho\rangle.\{x=e\}$ when given the compiled code $\Phi\rho.e$ of $\mathbf{e}$.

To illustrate, the following contraction represents the filling of $\{\mathbf{x} = \mathbf{h}\}$ with $\mathbf{x}\ \mathbf{z}$:

$$\begin{aligned}\mathbf{dfn}_{\mathbf{x}}\ \Phi\{\mathbf{x}{:}x, \mathbf{z}{:}z\}.(x\ z) &\rightarrow \Psi\langle\{\mathbf{x}{:}x\}, \{\mathbf{x}{:}x, \mathbf{z}{:}z\}\rangle.\{x = (x\ z)\} \\ &\equiv \Psi\langle\{\mathbf{x}{:}w\}, \{\mathbf{x}{:}x, \mathbf{z}{:}z\}\rangle.\{w = (x\ z)\}\end{aligned}$$

Constructing Compiled Definition Abstraction

The reason to include the operators $\mathbf{psi}_{\varepsilon,\boldsymbol{\rho}}$, one for each pair of ε and $\boldsymbol{\rho}$, resembles that for the operators $\mathbf{phi}_{\boldsymbol{\rho}}$ introduced by the twice context-enriched λ-calculus $\boldsymbol{\lambda}\mathbf{CC}$. They make $\boldsymbol{\lambda}\mathbf{DD}$ the fixpoint of the repeated applications of our context-enriching schema to the λ_d-calculus.

Each operator $\mathbf{psi}_{\varepsilon,\boldsymbol{\rho}}$ is intended to model the filling of the hole $\mathbf{h}$ of the context $\boldsymbol{\Psi}\langle\varepsilon,\boldsymbol{\rho}\rangle.\mathbf{h}$ with the context $\mathbf{d}$ represented by $\Psi\langle\varepsilon,\rho\rangle.d$. It yields a Φ-abstraction $\Phi\rho.\Psi\langle\varepsilon',\rho'\rangle.d$ representing the compiled code of the context $\boldsymbol{\Psi}\langle\varepsilon,\boldsymbol{\rho}\rangle.\mathbf{d}$. The reduction rule for $\mathbf{psi}_{\varepsilon,\boldsymbol{\rho}}$ is quite complicated:

$$\mathbf{psi}_{\{\bar{\mathbf{w}}:\bar{\mathbf{z}}\},\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ \Psi\langle\varepsilon,\rho\rangle.d \ \rightarrow\ \Phi\rho.\Psi\langle\{\bar{\mathbf{w}}{:}\bar{z}\},\{\bar{\mathbf{x}}{:}\bar{y}\}\rangle.d$$

The variable z_i of each $\mathbf{w}_i{:}z_i$ is the parameter variable of $\mathbf{z}_i$ in ε. The identifiers $\bar{\mathbf{z}}$ must therefore be specified in ε; otherwise, the term $\mathbf{psi}_{\{\bar{\mathbf{w}}:\bar{\mathbf{z}}\},\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ \Psi\langle\varepsilon,\rho\rangle.d$ is not a redex. The variable y_i of each $\mathbf{x}_i{:}y_i$ is the parameter variable of $\mathbf{y}_i$ in ρ, provided that $\mathbf{y}_i{:}y_i$ is an element of ρ; otherwise, y_i is a fresh variable. Intuitively, the behavior of the $\{\bar{\mathbf{x}}{:}\bar{\mathbf{y}}\}$-part is similar to a $\mathbf{phi}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}$ operation on Φ-abstractions. The behavior of the $\{\bar{\mathbf{w}}{:}\bar{\mathbf{z}}\}$-part is to hide away defining identifiers of ε not specified in $\bar{\mathbf{z}}$ and to rename the defining identifiers $\bar{\mathbf{z}}$ to $\bar{\mathbf{w}}$.

The operators $\mathbf{psi}_{\varepsilon,\boldsymbol{\rho}}$ not only have complex semantics, they are also too restrictive. It is necessary to fully specify the $\boldsymbol{\rho}$-part even if we are only interested in carrying out the ε-part, and vice versa. We therefore split the two functionalities into two independent operations.

For the ε-part, we introduce the operators $\mathbf{eps}_{\varepsilon}$ with the following reduction rule:

$$\mathbf{eps}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ \Psi\langle\varepsilon,\rho\rangle.d \ \rightarrow\ \Psi\langle\{\bar{\mathbf{x}}{:}\bar{y}\},\rho\rangle.d \qquad (\mathbf{eps}_{\varepsilon})$$

$$\text{where } \mathbf{y}_i{:}y_i \in \varepsilon$$

The variable y_i of each $\mathbf{x}_i : y_i$ is the parameter variable of $\mathbf{y}_i$ in ε. Thus, the identifiers $\bar{\mathbf{y}}$ must be specified in ε; otherwise, the term $\mathbf{eps}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ \Psi\langle\varepsilon,\rho\rangle.d$ is not a redex.

For the $\boldsymbol{\rho}$-part, we introduce the operators $\mathbf{psi}_{\boldsymbol{\rho}}$ with the following reduction rule:

$$\begin{array}{rcll} \mathbf{psi}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ \Psi\langle\varepsilon,\rho\rangle.d & \rightarrow & \Phi\rho.\Psi\langle\varepsilon,\{\bar{\mathbf{x}}:\bar{y}\}\rangle.d & (\mathbf{psi}_{\boldsymbol{\rho}}) \\ & & \text{where } \mathbf{y}_i : y_i \in \rho & \\ & & \text{or else } y_i \text{ is fresh} & \end{array}$$

The variable y_i of each $\mathbf{x}_i : y_i$ is the parameter variable of $\mathbf{y}_i$ in ρ, provided that $\mathbf{y}_i : y_i$ is an element of ρ; otherwise, y_i is a fresh variable.

Put together, the operators $\mathbf{psi}_{\varepsilon,\boldsymbol{\rho}}$ can be defined as follows:

$$\mathbf{psi}_{\varepsilon,\boldsymbol{\rho}} \equiv \mathbf{psi}_{\boldsymbol{\rho}} \circ \mathbf{eps}_{\varepsilon}$$

since we have

$$\begin{array}{rcl} \mathbf{psi}_{\{\bar{\mathbf{w}}:\bar{\mathbf{z}}\},\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ \Psi\langle\varepsilon,\rho\rangle.d & = & \mathbf{psi}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ (\mathbf{eps}_{\{\bar{\mathbf{w}}:\bar{\mathbf{z}}\}}\ \Psi\langle\varepsilon,\rho\rangle.d) \\ & \rightarrow & \mathbf{psi}_{\{\bar{\mathbf{x}}:\bar{\mathbf{y}}\}}\ \Psi\langle\{\bar{\mathbf{w}}:\bar{z}\},\rho\rangle.d \\ & \rightarrow & \Phi\rho.\Psi\langle\{\bar{\mathbf{w}}:\bar{z}\},\{\bar{\mathbf{x}}:\bar{y}\}\rangle.d \end{array}$$

6.3.4 Calculus of Compiled Code

The notion of reduction **dd** underlying the extended calculus $\boldsymbol{\lambda}$**DD** is the union of the nine reduction rules collectively displayed in Figure 6.5. We continue to use $\rightarrow$ to denote the one-step reduction relation induced by **dd** and $\twoheadrightarrow$ to denote the reflexive and transitive closure of $\rightarrow$. The least equivalence relation generated by $\twoheadrightarrow$ is the equational theory $\boldsymbol{\lambda}$**DD** and equivalence under $\boldsymbol{\lambda}$**DD** is written as $e_1 = e_2$.

The compiled code operators introduced by $\boldsymbol{\lambda}$**DD** operate orthogonally to one another and are independent of the Φ-abstraction operations of $\boldsymbol{\lambda}$**CC**. Hence, by the Hindley-Rosen Lemma, it is straightforward to show that the notion of reduction **dd** is Church-Rosser:

Theorem 6.1 *The* **dd***-reduction relation* $\twoheadrightarrow$ *satisfies the diamond property.*

$$
\begin{array}{rcll}
(\lambda x.e)\ e' & \rightarrow & [e'/x]e & (\beta) \\
\textbf{load}\ \Phi\{\mathbf{x}_1\!:\!x_1,\ldots,\mathbf{x}_n\!:\!x_n\}.e & \rightarrow & [\tilde{\mathbf{x}}_n/x_n]\cdots[\tilde{\mathbf{x}}_1/x_1]e & (\textbf{load}) \\
\textbf{app}\ \Phi\rho_1.e_1\ \Phi\rho_2.e_2 & \rightarrow & \Phi\rho_1 \uplus \rho_2.(e_1\ e_2) & (\textbf{app}) \\
\textbf{lam}_{\mathbf{x}}\ \Phi\rho.e & \rightarrow & \Phi\rho.\lambda x.e & (\textbf{lam}) \\
 & & \quad \text{where } \mathbf{x}\!:\!x \in \rho & \\
 & & \quad \text{or else } x \text{ is fresh} & \\
\textbf{phi}_{\{\mathbf{x}_1:\mathbf{y}_1,\ldots,\mathbf{x}_n:\mathbf{y}_n\}}\ \Phi\rho.e & \rightarrow & \Phi\rho.\Phi\{\mathbf{x}_1\!:\!y_1,\ldots,\mathbf{x}_n\!:\!y_n\}.e & (\textbf{phi}_{\boldsymbol{\rho}}) \\
 & & \quad \text{where } \mathbf{y}_i\!:\!y_i \in \rho & \\
 & & \quad \text{or else } y_i \text{ is fresh} & \\
\textbf{let}\ \Psi\langle\varepsilon_1,\rho_1\rangle.d\ \Phi\rho_2.e & \rightarrow & \Phi\rho_1 \uplus \rho_2.(let\ d\ in\ e) & (\textbf{let}) \\
 & & \quad \text{where } \varepsilon_1 \uplus \rho_2 & \\
\textbf{dfn}_{\mathbf{x}}\ \Phi\rho.e & \rightarrow & \Psi\langle\{\mathbf{x}\!:\!x\},\rho\rangle.\{x = e\} & (\textbf{dfn}) \\
 & & \quad \text{where } x \text{ is arbitrary} & \\
\textbf{psi}_{\{\mathbf{x}_1:\mathbf{y}_1,\ldots,\mathbf{x}_n:\mathbf{y}_n\}}\ \Psi\langle\varepsilon,\rho\rangle.d & \rightarrow & \Phi\rho.\Psi\langle\varepsilon,\{\mathbf{x}_1\!:\!y_1,\ldots,\mathbf{x}_n\!:\!y_n\}\rangle.d & (\textbf{psi}_{\boldsymbol{\rho}}) \\
 & & \quad \text{where } \mathbf{y}_i\!:\!y_i \in \rho & \\
 & & \quad \text{or else } y_i \text{ is fresh} & \\
\textbf{eps}_{\{\mathbf{x}_1:\mathbf{y}_1,\ldots,\mathbf{x}_n:\mathbf{y}_n\}}\ \Psi\langle\varepsilon,\rho\rangle.d & \rightarrow & \Psi\langle\{\mathbf{x}_1\!:\!y_1,\ldots,\mathbf{x}_n\!:\!y_n\},\rho\rangle.d & (\textbf{eps}_{\boldsymbol{\varepsilon}}) \\
 & & \quad \text{where } \mathbf{y}_i\!:\!y_i \in \varepsilon &
\end{array}
$$

Figure 6.5: Reduction rules of twice context-enriched definition calculus $\boldsymbol{\lambda}\mathbf{DD}$

The free identifier abstractions indistinguishable by the compiled code operators of $\boldsymbol{\lambda}$**CC** (cf. Theorems 3.5 and 5.2) remain indistinguishable by the new operators of $\boldsymbol{\lambda}$**DD**. Moreover, the same notion of indistinguishability applies to Ψ-abstractions as well. Formally, any two definition abstractions $\Psi\langle\varepsilon,\rho_1\rangle.d$ and $\Psi\langle\varepsilon,\rho_2\rangle.d$ that differ only in their parameter specifications ρ_1 and ρ_2 are indistinguishable to the compiled code operators of $\boldsymbol{\lambda}$**DD**, denoted as $\Psi\langle\varepsilon,\rho_1\rangle.d \approx \Psi\langle\varepsilon,\rho_2\rangle.d$, when the following condition holds:

$$\text{for each free variable } x \text{ of } d,\ \mathbf{x}\!:\!x \in \rho_1 \text{ if and only if } \mathbf{x}\!:\!x \in \rho_2 \qquad (\approx)$$

That is, the two parameter specifications ρ_1 and ρ_2 agree on the free variables of d.

6.3.5 Metacircular Self-Compilation

As shown in Chapter 5, the twice context-enriched version of the λ-calculus $\boldsymbol{\lambda}$**CC** is capable of compiling itself metacircularly. Here, we demonstrate the same behavior for $\boldsymbol{\lambda}$**DD**, but only for single binding definitions (the metacircular compilation can be easily generalized to multiple binding definitions after a mechanism for constructing such definitions is introduced in Section 6.5.1).

The abstract source code representation schema $[\![\ ]\!]_c$ for evolved $\boldsymbol{\lambda}$**DD**-contexts and its companion compilation function $\mathcal{C}$ extend their $\boldsymbol{\lambda}$**CC**-counterparts with the following clauses:

$$\begin{array}{rcl@{\qquad}rcl}
[\![\boldsymbol{\Psi}\langle\varepsilon,\boldsymbol{\rho}\rangle.\mathbf{d}]\!]_c & \equiv & \langle\mathbf{5},\langle\mathbf{psi}_{\boldsymbol{\varepsilon},\boldsymbol{\rho}},[\![\mathbf{d}]\!]_c\rangle\rangle & \mathcal{C}\ \langle\mathbf{5},\langle e_1,e_2\rangle\rangle & = & e_1\ (\mathcal{C}\ e_2) \\
[\![\{\mathbf{x}=\mathbf{e}\}]\!]_c & \equiv & \langle\mathbf{6},\langle\mathbf{dfn}_{\mathbf{x}},[\![\mathbf{e}]\!]_c\rangle\rangle & \mathcal{C}\ \langle\mathbf{6},\langle e_1,e_2\rangle\rangle & = & e_1\ (\mathcal{C}\ e_2) \\
[\![\mathbf{let\ d\ in\ e}]\!]_c & \equiv & \langle\mathbf{7},\langle[\![\mathbf{d}]\!]_c,[\![\mathbf{e}]\!]_c\rangle\rangle & \mathcal{C}\ \langle\mathbf{7},\langle e_1,e_2\rangle\rangle & = & \mathbf{let}\ (\mathcal{C}\ e_1)\ (\mathcal{C}\ e_2)
\end{array}$$

The compiler is *compositional* since the compilation of each ($[\![\ ]\!]_c$-encoded) composite evolved $\boldsymbol{\lambda}$**DD**-context is a function of the compilation of the context's ($[\![\ ]\!]_c$-encoded) components. It is *metacircular* since it translates each category of evolved $\boldsymbol{\lambda}$**DD**-contexts into the same category of $\boldsymbol{\lambda}$**DD**-terms.

To summarize, we have demonstrated that our context-enriching schema can be easily adapted to more sophisticated variable defining mechanisms. It is the case that as the number of variable defining constructs grows, more compiled code abstractions and compiled code operators are needed. The linking device at the heart of incremental program construction is still the same old variable capture, however.

6.4 Modules

As mentioned before, definition abstractions $\Psi\langle\varepsilon,\rho\rangle.d$ are substitutes for definitions d as first-class citizens. They are *modules* [55, 73] represented as a distinct category of compiled code. The defining variables of d are exported via the parameter specification ε; the parameters specified in ρ are the import variables of d. The compiled code operator **let** is the means to express module importation. The operators $\mathbf{dfn}_{\mathbf{X}}$ are module constructors. In the rest of this chapter we extend $\boldsymbol{\lambda}\mathbf{DD}$ with operations that combine and link modules to form new modules. These operations are explained in terms of context hole filling. The point to stress here is that the linking between import and export variables can be modeled strictly with variable capture.

Before presenting the module operators, we introduce some meta-operations on definitions and definition abstractions. Let $d \equiv \{x_1 = e_1, \ldots, x_m = e_m\}$ and $d' \equiv \{x'_1 = e'_1, \ldots, x'_n = e'_n\}$ be two definitions. We then define $d \oplus d'$ to be the disjoint union of d and d':

$$d \oplus d' \quad\equiv\quad \{x_1 = e_1, \ldots, x_m = e_m, x'_1 = e'_1, \ldots, x'_n = e'_n\}$$

provided that the defining variables of d are distinct from the defining variables of d', *i.e.*, $x_i \not\equiv x'_j$ for any i and j. Analogously, let ε and ε' be the parameter specifications $\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_m : x_m\}$ and $\{\mathbf{x}'_1 : x'_1, \ldots, \mathbf{x}'_n : x'_n\}$. We then define $\varepsilon \oplus \varepsilon'$ to be the disjoint union of ε and ε':

$$\varepsilon \oplus \varepsilon' \quad\equiv\quad \{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_m : x_m, \mathbf{x}'_1 : x'_1, \ldots, \mathbf{x}'_n : x'_n\}$$

provided that $\mathbf{x}_i \not\equiv \mathbf{x}'_j$ and $x_i \not\equiv x'_j$ for any i and j. The constraint is notated as $\varepsilon \cap \varepsilon' = \emptyset$.

6.5 Combining Modules

In this section we combine modules to form new modules. Each of the module combination operators introduced here is the context-enriched version of a corresponding definition combination construct whose semantics can be easily understood via syntactic expansion. Again, our emphasis is not on the semantic description of each definition combination construct; rather, it is on showing how straightforward it is to add a corresponding context-enriched module operator to the **$\boldsymbol{\lambda}$DD**-calculus.

For each definition combination construct that combines two definitions $\{\bar{x} = \bar{e}\}$ and $\{\bar{y} = \bar{e}'\}$ to form a new definition, we consider two fundamental issues:

override Are the bindings of $\{\bar{x} = \bar{e}\}$ included in the new definition? Similarly, are the bindings of the other constituent definition $\{\bar{y} = \bar{e}'\}$ part of the new definition? If both are included, the bindings of which constituent definition take precedence in case of conflicts?

linking In the new definition, what is the scope of the defining variables $\bar{x}$ and $\bar{y}$? Are the bindings of $\bar{x}$ and $\bar{y}$ visible to the denotation terms $\bar{e}$ and $\bar{e}'$?

We choose the following derived forms of definition along with their induced module combination operators to demonstrate our point that module linking can be modeled by coordinated renaming of import and export variables:

$$
\begin{array}{lll}
d ::= \cdots & \mathbf{d} ::= \cdots & \boldsymbol{\delta} ::= \cdots \\
\quad \mid \mathit{sim}\ d\ d & \quad \mid \mathbf{sim\ d\ d} & \quad \mid \mathbf{sim} \\
\quad \mid \mathit{priv}\ d\ d & \quad \mid \mathbf{priv\ d\ d} & \quad \mid \mathbf{priv} \\
\quad \mid \mathit{override}\ d\ d & \quad \mid \mathbf{override\ d\ d} & \quad \mid \mathbf{override} \\
\quad \mid \mathit{moverride}\ d\ d & \quad \mid \mathbf{moverride\ d\ d} & \quad \mid \mathbf{moverride}
\end{array}
$$

The left hand column shows the syntax of the definition combination constructs; the middle column consists of their respective evolved contexts; and the right hand column depicts the induced module operators. The first definition combination *sim d d* involves no override nor linking. The second combination *priv d d* entails linking but not override. The third construct *override d d* exhibits override but not linking. The last form *moverride d d* has both override and linking.

6.5.1 Simultaneous Definitions

The derived definition *sim* d_1 d_2, where the defining variables of d_1 and d_2 are disjoint, denotes the *simultaneous* combination of d_1 with d_2. It is the disjoint union of the bindings of d_1 and d_2:

$$\mathit{sim}\ d_1\ d_2 \quad \equiv \quad d_1 \oplus d_2$$

There is no linking nor override involved; neither d_1 nor d_2 is in the scope of the other.

The module combining operator that corresponds to the construction of simultaneous definitions is **sim**. It models the construction of the compiled code of the context **sim** $\mathbf{d}_1$ $\mathbf{d}_2$ from the compiled code of the contexts $\mathbf{d}_1$ and $\mathbf{d}_2$:

$$\begin{array}{rcl} \mathbf{sim}\ \Psi\langle\varepsilon_1, \rho_1\rangle.d_1\ \Psi\langle\varepsilon_2, \rho_2\rangle.d_2 & \longrightarrow & \Psi\langle\varepsilon_1 \oplus \varepsilon_2, \rho_1 \uplus \rho_2\rangle.(\mathit{sim}\ d_1\ d_2) \\ & & \text{where } \varepsilon_1 \cap \varepsilon_2 = \emptyset \end{array} \qquad (\mathbf{sim})$$

The condition $\varepsilon_1 \cap \varepsilon_2 = \emptyset$ expresses the constraints that the export identifiers of ε_1 must be distinct from those of ε_2 and that the variables of ε_1 are distinct from the variables of ε_2. Without them, neither the disjoint union $\varepsilon_1 \oplus \varepsilon_2$ nor the simultaneous definition *sim* d_1 d_2 would make sense. This rule, like all the other compiled code construction rules, relies on renaming the variables of ε_1, ρ_1, ε_2, and ρ_2 to meet the constraints as well as to ensure the absence of inadvertent variable capture.

As an example, let A_x and A_y be defined as follows:

$$\begin{array}{rcl} A_x & = & \Psi\langle\{\mathbf{xcor}\!:\!w\}, \{\}\rangle.\{w = 3\} \\ A_y & = & \Psi\langle\{\mathbf{ycor}\!:\!w\}, \{\}\rangle.\{w = 4\} \end{array}$$

Then,

$$\begin{array}{rcl} A & = & \mathbf{sim}\ A_x\ A_y \\ & \equiv & \mathbf{sim}\ \Psi\langle\{\mathbf{xcor}\!:\!x\}, \{\}\rangle.\{x = 3\}\ \Psi\langle\{\mathbf{ycor}\!:\!y\}, \{\}\rangle.\{y = 4\} \end{array}$$

$$
\begin{aligned}
&\rightarrow \quad \Psi\langle\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y\}, \{\}\rangle.(\mathit{sim}\ \{x = 3\}\ \{y = 4\}) \\
&= \quad \Psi\langle\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y\}, \{\}\rangle.\{x = 3, y = 4\}
\end{aligned}
$$

6.5.2 Private Definitions

In the derived definition *priv* d_1 d_2, the bindings of d_1 are *privately* available to the denotation terms of d_2:

$$\mathit{priv}\ d_1\ \{x_1 = e_1, \ldots, x_n = e_n\} \quad \equiv \quad \{x_1 = (\mathit{let}\ d_1\ \mathit{in}\ e_1), \ldots, x_n = (\mathit{let}\ d_1\ \mathit{in}\ e_n)\}$$

There is therefore the linking of the defining variables of d_1 with their free occurrences in the denotation terms $e_1, \ldots, e_n$. No override is involved since the bindings of d_1 are not carried over to the new definition.

The module constructor that corresponds to private definitions is **priv**. It models the construction of the compiled code of the context **priv** $\mathbf{d}_1$ $\mathbf{d}_2$ using the compiled code of the contexts $\mathbf{d}_1$ and $\mathbf{d}_2$:

$$\mathbf{priv}\ \Psi\langle\varepsilon_1, \rho_1\rangle.d_1\ \Psi\langle\varepsilon_2, \rho_2\rangle.d_2 \quad \rightarrow \quad \Psi\langle\varepsilon_2, \rho_1 \uplus \rho_2\rangle.(\mathit{priv}\ d_1\ d_2) \qquad (\mathbf{priv})$$
$$\text{where } \varepsilon_1 \uplus \rho_2$$

The constraint $\varepsilon_1 \uplus \rho_2$ expresses the linking of the import variables of ρ_2 to the export variables of ε_1 when they have identical identifiers. The new module has the same exports as ε_2 since the bindings of d_1 are private to d_2 only.

To illustrate, let A be the module defined in the last section and let B be the following module:

$$B \quad = \quad \Psi\langle\{\mathbf{dist}\!:\!d\}, \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y\}\rangle.\{d = \sqrt{x^2 + y^2}\}$$

Then,

$$
\begin{aligned}
C \quad &= \quad \mathbf{priv}\ A\ B \\
&\rightarrow \quad \Psi\langle\{\mathbf{dist}\!:\!d\}, \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y\}\rangle.(\mathit{priv}\ \{x = 3, y = 4\}\ \{d = \sqrt{x^2 + y^2}\}) \\
&\equiv \quad \Psi\langle\{\mathbf{dist}\!:\!d\}, \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y\}\rangle.\{d = \mathit{let}\ \{x = 3, y = 4\}\ \mathit{in}\ \sqrt{x^2 + y^2}\} \\
&= \quad \Psi\langle\{\mathbf{dist}\!:\!d\}, \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y\}\rangle.\{d = \sqrt{3^2 + 4^2}\}
\end{aligned}
$$

6.5.3 Overriding Definitions

The derived definition *override* d_1 d_2 denotes the combination of d_1 and d_2 such that the bindings of d_2 *override* the bindings of d_1 in case of conflicts:

$$\mathit{override}\ d_1\ d_2 \quad \equiv \quad (d_1 \setminus dv(d_2)) \oplus d_2$$

where $dv(d_2)$ denotes the set of defining variables of definition d_2 and $d_1 \setminus dv(d_2)$ removes from d_1 bindings whose defining variables are also specified by d_2. There is override, but no linking; the definitions are not in the scope of each other.

The corresponding module operator is **override**. It models the construction of the compiled code of the context **override d**$_1$ **d**$_2$ from the compiled code of the contexts **d**$_1$ and **d**$_2$:

$$\begin{array}{lr} \textbf{override}\ \Psi\langle\varepsilon_1,\rho_1\rangle.d_1\ \Psi\langle\varepsilon_2,\rho_2\rangle.d_2 & (\textbf{override}) \\ \qquad\qquad \rightarrow \quad \Psi\langle\varepsilon_1 \uplus \varepsilon_2, \rho_1 \uplus \rho_2\rangle.(\mathit{override}\ d_1\ d_2) & \end{array}$$

The constraint $\varepsilon_1 \uplus \varepsilon_2$ ensures that the export variables of ε_2 are identical to the export variables of ε_1 only when they have the same export identifiers.

For instance, let C be the module defined in the last section and let D be the module defined below:

$$D \quad = \quad \Psi\langle\{\mathbf{dist}\!:\!d\},\{\mathbf{dist}\!:\!d\}\rangle.\{d = d + 5\}$$

Then,

$$\begin{array}{ll} \multicolumn{2}{l}{\textbf{override}\ C\ D} \\ \rightarrow & \Psi\langle\{\mathbf{dist}\!:\!d\},\{\mathbf{dist}\!:\!d\}\rangle.(\mathit{override}\ \{d = \sqrt{3^2+4^2}\}\ \{d = d+5\}) \\ \equiv & \Psi\langle\{\mathbf{dist}\!:\!d\},\{\mathbf{dist}\!:\!d\}\rangle.\{d = d+5\} \end{array}$$

6.5.4 Mutually-Linked Overriding Definitions

The derived definition *moverride* d_1 d_2 is similar to *override* d_1 d_2 except that the definitions d_1 and d_2 are in the scope of each other. Hence, in addition to override, the two definitions are mutually linked. The syntactic expansion of *moverride* d_1 d_2 is a little bit more complicated:

$$\begin{aligned} & \mathit{moverride}\ \{x_1 = e_1, \ldots, x_m = e_m\}\ \{x'_1 = e'_1, \ldots, x'_n = e'_n\} \\ & \quad \equiv\ \mathit{override}\ \{x_1 = (\pi^m_1\ (\pi^2_1\ P)), \ldots, x_m = (\pi^m_m\ (\pi^2_1\ P))\} \\ & \qquad\qquad \{x'_1 = (\pi^n_1\ (\pi^2_2\ P)), \ldots, x'_n = (\pi^n_n\ (\pi^2_2\ P))\} \end{aligned}$$

where π^n_i is the ith selector of n-tuples and P is the pair defined recursively as follows:

$$\begin{aligned} P \ =\ & \langle \mathit{let}\ \{x'_1 = (\pi^n_1\ (\pi^2_2\ P)), \ldots, x'_n = (\pi^n_n\ (\pi^2_2\ P))\}\ \mathit{in}\ \langle e_1, \ldots, e_m\rangle, \\ & \ \mathit{let}\ \{x_1 = (\pi^m_1\ (\pi^2_1\ P)), \ldots, x_m = (\pi^m_m\ (\pi^2_1\ P))\}\ \mathit{in}\ \langle e'_1, \ldots, e'_n\rangle\rangle \end{aligned}$$

The operator **moverride** that constructs mutually linked overriding modules models the construction of the compiled code of **moverride** $\mathbf{d}_1$ $\mathbf{d}_2$ from the compiled code of $\mathbf{d}_1$ and $\mathbf{d}_2$ as follows:

$$\begin{aligned} & \mathbf{moverride}\ \Psi\langle\varepsilon_1, \rho_1\rangle.d_1\ \Psi\langle\varepsilon_2, \rho_2\rangle.d_2 \\ & \qquad \rightarrow\ \Psi\langle\varepsilon_1 \uplus \varepsilon_2, \rho_1 \uplus \rho_2\rangle.(\mathit{moverride}\ d_1\ d_2) \\ & \qquad\qquad \text{where } \varepsilon_1 \uplus \rho_2 \text{ and } \varepsilon_2 \uplus \rho_1 \end{aligned} \qquad (\mathbf{moverride})$$

In addition to the constraint $\varepsilon_1 \uplus \varepsilon_2$ of the **override**-rule, it requires that the import variables of ρ_1 be linked to the export variables of ε_2 when their identifiers match; hence, $\varepsilon_2 \uplus \rho_1$. Similarly, $\varepsilon_1 \uplus \rho_2$ means that the export variables of ε_1 are linked with the import variables of ρ_2. Again, all the constraints can be satisfied at once by the appropriate renaming of the parameter variables of ε_1, ρ_1 ε_2, and ρ_2.

To illustrate, let modules A and B be defined as follows:

$$\begin{aligned} A &= \Psi\langle\{\mathbf{odd}\!:\!f\}, \{\mathbf{even}\!:\!x\}\rangle.\{f = \lambda n.\mathit{if}\ n = 0\ \mathit{then}\ F\ \mathit{else}\ (x\ (n-1))\} \\ B &= \Psi\langle\{\mathbf{even}\!:\!g\}, \{\mathbf{odd}\!:\!y\}\rangle.\{g = \lambda n.\mathit{if}\ n = 0\ \mathit{then}\ T\ \mathit{else}\ (y\ (n-1))\} \end{aligned}$$

Then,

$$\begin{aligned} & \mathbf{moverride}\ A\ B \\ & \quad = \Psi\langle\{\mathbf{odd}\!:\!f, \mathbf{even}\!:\!g\}, \{\mathbf{odd}\!:\!f, \mathbf{even}\!:\!g\}\rangle. \end{aligned}$$

$$(\mathit{moverride}\ \{f = \lambda n.\mathit{if}\ n = 0\ \mathit{then}\ F\ \mathit{else}\ (g\ (n-1))\}$$
$$\{g = \lambda n.\mathit{if}\ n = 0\ \mathit{then}\ T\ \mathit{else}\ (f\ (n-1))\})$$

where the two functions f and g are mutually dependent on each other.

In summary, we have shown through the above exercises that it is easy to extend the **λDD**-calculus with additional module operations. The extensions themselves can be done in a modular fashion. More importantly, the complexities involved in the process stem mainly from the syntactic expansion of the derived definition constructs. Their induced module operators actually have relatively straightforward reduction rules that incur only simple identification of import and export variables. Additional examples of incremental module constructions can be found in Chapter 8.

Chapter 7

Context-Enriched Calculus of Relinkables

In the last chapter we have seen the enhancement of expressiveness achieved by enriching elaborate variable defining mechanisms with the notion of contexts. Here we explore the possibilities associated with the flip side of the coin. We show that expressiveness can also be enhanced by enriching intricate variable referencing mechanisms.

The fancier variable referencing mechanisms of interest are the relinkable variable references, which are simply called relinkables from here on. A *relinkable* is a variable reference that can be linked to different function parameters as its enclosing context grows. For instance, we can have a relinkable r such that in the following terms the same relinkable is bound by the different underlined λ-parameters:

$$\lambda \underline{x}.\lambda x.r$$

$$\lambda \underline{y}.\lambda x.\lambda x.r$$

$$\lambda x.\lambda \underline{x}.\lambda x.\lambda y.\lambda x.\lambda x.r$$

It refers to the second innermost λ-parameter x, the one underlined, in the first term. In the second term, the same relinkable r refers to the innermost λ-parameter y. Still, in the third term, it is bound by the fourth innermost λ-parameter x. The variable reference r is therefore said to be *relinked* according to its surrounding context.

Since relinkables can be redirected to mean different things in different contexts, they provide a very flexible means for modeling code reuse, a notion crucial to incremental program development. Indeed, the notion of relinkables covers many commonly found variable referencing mechanisms. For instance, a statically-scoped variable reference can be seen merely as a degenerate relinkable that cannot be relinked. A Common Lisp optional keyword parameter [66] is a relinkable that has been associated with a default denotation and can be relinked once to its optional denotation. A late binding C++ virtual reference [68] is a relinkable that can be relinked an arbitrary number of times.

Relinkables may seem counterintuitive to static scope. Indeed, as indicated above, relinkables exhibit behavior that strongly resembles dynamically-scoped variables since their linking relation does not appear to be fixed. In this chapter we show that the λ-calculus can be generalized to incorporate relinkables without jeopardizing static scope, however. Our calculus of relinkables is based on Berkling and Fehr's λ-calculus, which is reviewed in Section 7.1. In Section 7.2 we formalize the intuition behind relinkables into a λ-calculus. The context-enriched calculus of relinkables is presented in Section 7.3. The twice context-enriched calculus of relinkables is described in Section 7.4. It is then used in the next chapter to express the fundamental adaptive behavior inherently associated with object-oriented and interactive programming, two of the most prominent incremental programming paradigms in practice.

7.1 Berkling and Fehr's Lambda Calculus

The λ-calculus as we know it is not very well suited for mechanical implementation. The reason is that each β-substitution operation incurs potentially many α-conversion steps to avoid variable capture. Many have come up with variations of the λ-calculus to facilitate efficient implementation [1, 14, 41, 56, 72]. In particular, in his AUTOMATH project [15], de Bruijn uses a version of the λ-calculus in which each variable reference is replaced by its lexical address, which is the distance between the variable reference and its binding λ-parameter. An immediate consequence is that the calculus employs no variable names. There is therefore no need for α-conversion.

Motivated by the same efficiency concerns, Berkling and Fehr [9, 10] propose independently a λ-calculus that can be characterized as an integration of de Bruijn's nameless λ-calculus with the standard nameful version of the λ-calculus. The terms of their calculus have the following abstract syntax (using our notation):

$$\begin{array}{rcl} k & \in & \mathit{Dists} = \{1, 2, \ldots\} \\ e & ::= & x^k \mid \lambda x.e \mid e\ e \end{array} \qquad (\lambda_{bf})$$

Each variable reference x^k consists of a variable name x and a distance (lexical address) k. It refers to the kth nearest enclosing λ-parameter named x. For example, the reference x^2 in $\lambda \underline{x}.\lambda y.(y^1\ \lambda x.x^2)$ is bound by the underlined parameter, which is the second nearest enclosing x-parameter with respect to the variable reference x^2.

Conceptually, the variable references of λ_{bf} are lexical addresses partitioned according to their names. During β-substitution, their lexical addresses, but not their names, are adjusted to avoid inadvertent capture. Hence, no α-conversion is necessary. Below is a description of how this is accomplished.

The β-reduction rule for the λ_{bf}-calculus relies on two distance adjusting operations $\uparrow^n_x(e)$ and $\downarrow^n_x(e)$. Every free variable reference x^k in e with $k \geq n$ has its distance incremented by one under $\uparrow^n_x(e)$ and decremented by one under $\downarrow^n_x(e)$. These operations are needed to compensate for the addition or removal of an x-parameter from the enclosing context of e. Formally, the two operations $\uparrow^n_x(e)$, where $n \geq 1$, and $\downarrow^n_x(e)$, where $n \geq 2$, are defined inductively on the structure of e as follows:

$$\begin{array}{rcl} \uparrow^n_x(z^k) & \equiv & \begin{cases} z^{k+1} & \text{if } x \equiv z,\ k \geq n \\ z^k & \text{otherwise} \end{cases} \\ \uparrow^n_x(\lambda z.e) & \equiv & \begin{cases} \lambda z.\uparrow^{n+1}_x(e) & \text{if } x \equiv z \\ \lambda z.\uparrow^n_x(e) & \text{otherwise} \end{cases} \\ \uparrow^n_x(e_1\ e_2) & \equiv & \uparrow^n_x(e_1)\ \uparrow^n_x(e_2) \end{array} \qquad \begin{array}{rcl} \downarrow^n_x(z^k) & \equiv & \begin{cases} z^{k-1} & \text{if } x \equiv z,\ k \geq n \\ z^k & \text{otherwise} \end{cases} \\ \downarrow^n_x(\lambda z.e) & \equiv & \begin{cases} \lambda z.\downarrow^{n+1}_x(e) & \text{if } x \equiv z \\ \lambda z.\downarrow^n_x(e) & \text{otherwise} \end{cases} \\ \downarrow^n_x(e_1\ e_2) & \equiv & \downarrow^n_x(e_1)\ \downarrow^n_x(e_2) \end{array}$$

Intuitively, for a free variable reference occurring in the body e of a function $\lambda x.e$ to refer to the nth nearest λ-parameter named x enclosing the function, it must have a distance of $n+1$. Hence, the superscript n is incremented in the λ-abstraction clause when x is the name of the λ-parameter.

The β-reduction rule for the λ_{bf}-calculus is:

$$(\lambda x.e)\ e' \quad \rightarrow \quad \downarrow^2_x([\uparrow^1_x(e')/x^1]e) \qquad (\beta)$$

Every free variable reference in the function body e that refers to the parameter of the function $\lambda x.e$ is substituted with the term e', which is notated as $[e'/x^1]e$. Every free variable reference in e with the name x and a distance k greater than 1 must have its distance decremented by one, $\downarrow^2_x([e'/x^1]e)$, since one of its enclosing x-parameter is removed. This decrement operation should not apply to occurrences of the argument term e', however, since e' was not in the scope of the removed x-parameter. So, based on the fact that $\downarrow^2_x(\uparrow^1_x(e'))$ is identical to e', the effect of the decrement operation can be cancelled out if the distance of every free variable reference in e' with the name x is incremented before performing the substitution, hence the contractum $\downarrow^2_x([\uparrow^1_x(e')/x^1]e)$.

The β-substitution meta-operation $[e'/x^n]e$ has the following inductive definition:

$$\begin{aligned}
[e'/x^n]z^k &\equiv \begin{cases} e' & \text{if } x \equiv z \text{ and } k = n \\ z^k & \text{otherwise} \end{cases} \\
[e'/x^n]\lambda z.e &\equiv \begin{cases} \lambda z.[\uparrow^1_z(e')/x^{n+1}]e & \text{if } x \equiv z \\ \lambda z.[\uparrow^1_z(e')/x^n]e & \text{otherwise} \end{cases} \\
[e'/x^n](e_1\ e_2) &\equiv [e'/x^n]e_1\ [e'/x^n]e_2
\end{aligned}$$

Again, the most interesting clause is the one for λ-abstractions. The replacement term e' has the distance of its free variable references with the name z incremented by one, notated as $\uparrow^1_z(e')$, to avoid inadvertent capture since the substitution is in the scope of one more λ-parameter named z. When the name x of the to-be-replaced variable references x^n is the same as the parameter z of the λ-abstraction $\lambda z.e$, the distance of x^n is incremented by one to account for the fact that a free variable reference in $\lambda z.e$ with the name z must have a distance of $n+1$ in the function body e.

Notice that there is no need for parameter renaming in the above definition of β-substitution. In its place are distance adjustment operations. The advantage of doing so is a simpler machine implementation because the complicated task of choosing new variable names for α-conversion steps is replaced by simple integer arithmetic on lexical addresses.

We conclude this section by showing that the context-enriched version of λ_{bf} is more expressive than $\boldsymbol{\lambda}\mathbf{C}$, the context-enriched λ-calculus. Consider the following derivation:

$$\begin{aligned} \mathbf{lam_X}\ \underline{(\mathbf{lam_X}\ \Phi\{\mathbf{x}:x\}.x^2)} &= \underline{\mathbf{lam_X}\ \Phi\{\mathbf{x}:x\}.\lambda x.x^2} \\ &= \Phi\{\mathbf{x}:x\}.\lambda \underline{x}.\lambda x.x^2 \end{aligned}$$

The distance 2 of the variable reference x^2 provides a way for us to "ignore" the parameter of the first incrementally constructed λ-abstraction. Instead, x^2 refers to the parameter of the second incrementally constructed λ-abstraction, the one underlined. Such behavior is clearly not expressible in $\boldsymbol{\lambda}\mathbf{C}$. Still, enhancing λ_{bf} with contexts alone does not provide the adaptive behavior promised at the beginning of the chapter. We therefore introduce relinkables, which are a generalization of λ_{bf} variable references.

7.2 Relinkables

A relinkable is a sequence of λ_{bf} variable references $[x_1^{k_1}, x_2^{k_2}, \ldots]$, *e.g.*, $[x^2, y^3, z^1]$. Such a sequence can be conceptually infinite. (Of course, we must then find a finite syntactic representation for such sequences. Doing so would certainly limit the sequences that are expressible. Fortunately, many of the interesting applications of relinkables do have a finite representation; see Section 7.2.4.) Each constituent $x_i^{k_i}$ of a relinkable $[\ldots, x_i^{k_i}, \ldots, x_j^{k_j}, \ldots]$ *conditionally dominates* $x_j^{k_j}$, where $j > i$, in the sense that when $x_i^{k_i}$ is known to be bound by some λ-parameter, its link takes precedence over that of $x_j^{k_j}$. Hence, the denotation of $x_i^{k_i}$ is favored over the denotation of $x_j^{k_j}$. A relinkable

$[x_1^{k_1}, x_2^{k_2}, \ldots]$ is thus semantically equivalent to

$$\begin{array}{l} \textbf{if } x_1^{k_1} \text{ is bound } \textbf{then } x_1^{k_1} \\ \quad \textbf{else if } x_2^{k_2} \text{ is bound } \textbf{then } x_2^{k_2} \\ \qquad \ddots \\ \qquad\quad \textbf{else } [x_1^{k_1}, x_2^{k_2}, \ldots] \text{ is unbound} \end{array}$$

In other words, a relinkable is equivalent to its constituent variable reference $x_i^{k_i}$ with the smallest i such that $x_i^{k_i}$ is known to be linked. Such a constituent is called *dominant*.

Relinkables are so called since they can be relinked to different λ-parameters as their enclosing context grows. For instance, the same relinkable $[x^2, y^1, x^1, x^3]$ in the following successive terms changes its link to refer to outer λ-parameters (the dominant constituent and the parameter to which it refers are underlined):

$$\begin{aligned} \lambda\underline{x}.[x^2, y^1, \underline{x^1}, x^3] \\ \lambda\underline{y}.\lambda x.[x^2, \underline{y^1}, x^1, x^3] \\ \lambda\underline{x}.\lambda y.\lambda x.[\underline{x^2}, y^1, x^1, x^3] \\ \lambda x.\lambda\underline{x}.\lambda y.\lambda x.[\underline{x^2}, y^1, x^1, x^3] \end{aligned}$$

The semantics of the relinkable $[x^2, y^1, x^1, x^3]$ is therefore sensitive to its surrounding context.

7.2.1 Provisionally-Instantiated Relinkables

Relinkables pose a problem for β-reduction. Consider the β-redex

$$(\lambda\underline{x}.\lambda x.[\underline{x^2}, x^1])\ \lambda y.y$$

The to-be-replaced variable reference x^2 of $[x^2, x^1]$ is dominant; its denotation precedes that of the other constituents. We can therefore substitute $\lambda y.y$ for the relinkable $[x^2, x^1]$, yielding $\lambda x.\lambda y.y$. The situation gets murkier when we try to reduce the

β-redex

$$(\lambda \underline{x}.\lambda x.[x^3, \underline{x^2}, x^1])\ \lambda y.y$$

where the to-be-substituted x^2 is not guaranteed to be dominant. Since we do not know whether there is a denotation for x^3, it is wise for us to keep the denotation of x^2 around. We thus retain $\lambda y.y$ as a *provisional* denotation for the relinkable $[x^3, x^2, x^1]$. Clearly, the reduction should remove x^2 from $[x^3, x^2, x^1]$ since we already have a denotation for it. It should also remove x^1 since we already have a denotation for a more dominating x^2. Thus, only x^3 should remain in the result of the substitution. But since a binding occurrence of x has been removed by the β-reduction, the distance of x^3 should be decremented accordingly. We thus arrive at the relinkable $[x^2]$ and the result of the β-reduction is $\lambda x.[x^2] \sim \lambda y.y$, where $r \sim e$ is our notation for a *provisionally-instantiated* relinkable, r being the relinkable and e being its provisional denotation.

Intuitively, a provisionally-instantiated relinkable $[x_1^{k_1}, x_2^{k_2}, \ldots] \sim e$ is semantically equivalent to

$$\begin{array}{l} \textbf{if } x_1^{k_1} \text{ is bound } \textbf{then } x_1^{k_1} \\ \quad \textbf{else if } x_2^{k_2} \text{ is bound } \textbf{then } x_2^{k_2} \\ \qquad\qquad \ddots \\ \qquad\qquad\quad \textbf{else } e \end{array}$$

where e is interpreted as the "default" denotation of $[x_1^{k_1}, x_2^{k_2}, \ldots]$.

7.2.2 Variable Name Delimitation

Another problem concerning relinkables has to do with the scope of a variable name. It is particularly crucial to relinkables with an infinite number of constituents such as $[x^\infty, \ldots, x^1]$. The denotation of each of its constituents is destined to be provisional only. We would never know which is *the* denotation of the relinkable. We thus incorporate variable name delimitation terms $\nu x.e$ to confine the scope of the name

x to the term e. Any variable reference in e with the name x cannot have its binding λ-abstractor beyond the term $\nu x.e$. Thus, in the term

$$\lambda \underline{x}.\nu x.\lambda x.\lambda x.[x^3, y^1, x^2, x^1]$$

the reference x^3 does not refer to the underlined x even though its distance from x^3 is 3, discounting the delimiter νx. We can therefore rename the x's in the νx-delimited term to a fresh name, say w, thus yielding an α-equivalent term

$$\lambda \underline{x}.\nu w.\lambda w.\lambda w.[w^3, y^1, w^2, w^1]$$

It is now clear that the scope of the underlined x does not extend beyond the νv-delimiter.

Indeed, the relinkable $[x^3, y^1, x^2, x^1]$ in $\lambda \underline{x}.\nu x.\lambda x.\lambda x.[x^3, y^1, x^2, x^1]$ is semantically equivalent to $[y^1, x^2, x^1]$. We can remove its constituent variable reference x^3 since it is certain to be unlinked. In general, for each $\nu x.e$, we can *finalize* every relinkable in e with respect to the name x by removing its constituents of the form x^k that are not bound by λ-parameters within $\nu x.e$. Intuitively, the number n of binding occurrences of x (the number of λ-parameters named x) in between a relinkable r and the closest νx-delimiter is known statically. Any variable reference x^k in r with a distance k greater than n can be removed from r, since it is not bound by any λ-parameter within the boundary of $\nu x.e$. If the finalization of r yields the arid relinkable $[]$, r is unbound. Likewise, we can finalize every provisionally-instantiated relinkable $r \sim e'$ in the scope of a name-delimitation term $\nu x.e$. If the finalization of r yields $[]$, e' is the denotation of r; hence, we can replace $r \sim e'$ with the finalized version of e'. Otherwise, the result is a provisionally-instantiated relinkable associating the finalized r with the finalized e'. This can be summarized as the following reduction rule where $\Delta^1_x(e)$ is a notation for such a variable reference removal process:

$$\nu x.e \quad \rightarrow \quad \Delta^1_x(e) \qquad (\nu x)$$

7.2.3 Calculus of Relinkables

To summarize, we have generalized Berkling and Fehr's λ_{bf}-calculus to a new calculus with relinkables r, provisionally-instantiated relinkables $r \sim e$, and variable name delimitation terms $\nu x.e$:

$$\begin{array}{rcl} r & ::= & [x_1^{k_1}, x_2^{k_2}, \ldots] \\ e & ::= & r \mid \lambda x.e \mid e\ e \mid r \sim e \mid \nu x.e \end{array}$$

It is clear that λ_{bf} is a special case of our calculus of relinkables in which every relinkable is a singleton sequence and therefore there is no need for provisionally-instantiated relinkables nor name delimitations.

We must point out that the relinkables as presented above are still statically scoped in the sense that given any program, which is a term with every one of its names ν-delimited, we can statically determine the dominant constituent of each of its relinkables. Hence, every relinkable can be reduced to a singleton sequence consisting solely of its dominant constituent. Moreover, the link between the dominant constituent of each relinkable and its binding λ-parameter is consistently maintained by the β-reduction rule. The calculus of relinkables is therefore only as expressive as the λ-calculus. The new relinkable variable references themselves do not provide the flexibility alluded to at the beginning of this chapter. Their potential will be unleashed when they are enriched with contexts, however.

7.2.4 A Concrete Representation of Relinkables

Before continuing onto the enrichment of the calculus of relinkables with the notion of contexts, we give a concrete finite representation for relinkables. The very first limitation imposed by such a representation is that it cannot include all possible relinkables. Fortunately, it is general enough to express many interesting applications; see Chapter 8 for examples.

The finite representation uses the notation $x^{(u,l]}$, where $l, u \in \{1, 2, \ldots, \infty\}$ are two numbers such that $u > l$, to denote the possibly infinite series of variable references

$x^{u-1}, \ldots, x^l$ of the same name x. For instance, $x^{(4,1]}$ denotes the finite series x^3,x^2,x^1 and $x^{(\infty,1]}$ means the infinite series x^∞, …, x^1. A relinkable is then represented as a sequence of such series. Formally,

$$\begin{array}{rcl} l, u & \in & \{1, 2, \ldots, \infty\} \\ t & ::= & x^{(u,l]} \quad \text{where } u > l \\ r & ::= & [t_1, \ldots, t_n] \quad \text{where } n \geq 0 \end{array}$$

As an example, $[x^{(4,1]}, y^{(2,1]}, x^{(\infty,3]}]$ is a finite representation of the infinite relinkable $[x^3, x^2, x^1, y^1, x^\infty, \ldots, x^3]$.

We further require the following constraint to guarantee that the representation $[t_1, \ldots, t_n]$ of each relinkable, if there is one, is unique:

> For any two adjacent series $t_i \equiv x^{(u,l]}$ and $t_{i+1} \equiv x^{(u',l']}$ of the same variable name x, the interval $(u, l]$ is not downwardly adjacent to the interval $(u', l']$, *i.e.*, $l \neq u'$.

Without the constraint, the finite relinkable $[x^3, x^2, x^1]$ has these four possible different concrete representations:

$$\begin{array}{l} [x^{(4,3]}, x^{(3,2]}, x^{(2,1]}] \\ [x^{(4,2]}, x^{(2,1]}] \\ [x^{(4,3]}, x^{(3,1]}] \\ [x^{(4,1]}] \end{array}$$

With the constraint, only the last one $[x^{(4,1]}]$ is admissible.

7.3 Context-Enriched Calculus of Relinkables

Recall that our schema for incorporating the notion of contexts into a calculus is to extend the calculus with these mechanisms:

- compiled code abstractions to simulate evolved contexts,

Contexts:

$$\mathbf{r} ::= [\mathbf{x}_1^{k_1}, \mathbf{x}_2^{k_2}, \ldots]$$

$$\mathbf{C} ::= \mathbf{h} \mid \mathbf{r} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{C} \mid \mathbf{C}\;\mathbf{C} \mid \mathbf{r}\!\sim\!\mathbf{C} \mid \boldsymbol{\nu}\mathbf{x}.\mathbf{C}$$

Evolved Contexts:

$$\mathbf{e} ::= \mathbf{r} \mid \boldsymbol{\lambda}\mathbf{x}.\mathbf{e} \mid \mathbf{e}\;\mathbf{e} \mid \mathbf{r}\!\sim\!\mathbf{e} \mid \boldsymbol{\nu}\mathbf{x}.\mathbf{e}$$

Figure 7.1: Contexts of λ-calculus with relinkables

- a compiled code loading operation to assimilate compiled code into machine code, and

- a compiled code construction operator for each category of composite contexts.

The contexts of the λ-calculus with relinkables are defined in Figure 7.1. To enrich the calculus of relinkables, we extend it with:

- free identifier abstractions $\Phi\rho.e$ to model compiled code,

- a free identifier abstraction loading operator **load** and unlinked identifier indicators $\tilde{\mathbf{x}}$, and

- an incremental compiled code construction operator for each of the four composite contexts:

  - $\mathbf{lam}_\mathbf{X}$ for constructing compiled λ-abstractions,

  - **app** for constructing compiled applications,

  - **pro** for constructing compiled provisionally-instantiated relinkables, and

  - $\mathbf{del}_\mathbf{X}$ for constructing compiled name delimitations.

The operators $\mathbf{lam_X}$ and **app** are the familiar ones described in Chapter 3. The behavior of the operator **pro** is characterized by the following reduction rule:

$$\mathbf{pro}\ \Phi\rho_1.r\ \Phi\rho_2.e \quad\rightarrow\quad \Phi\rho_1\uplus\rho_2.(r\sim e)$$

It constructs the compiled code $\Phi\rho_1\uplus\rho_2.(r\sim e)$ of a provisionally-instantiated relinkable $\mathbf{r}\sim\mathbf{e}$ out of the compiled code $\Phi\rho_1.r$ of the relinkable **r** and the compiled code $\Phi\rho_2.e$ of the provisional denotation **e**. There is one problem with the above reduction rule, however. It requires the rule to "peek" into the body of the first free identifier abstraction argument $\Phi\rho_1.r$ to ensure that it is a relinkable.

An alternative is to introduce a specific category of compiled code abstraction $\Upsilon\rho.r$ whose body is always a relinkable:

$$\mathbf{pro}\ \Upsilon\rho_1.r\ \Phi\rho_2.e \quad\rightarrow\quad \Phi\rho_1\uplus\rho_2.(r\sim e)$$

The behavior of $\Upsilon\rho.r$ can be achieved without a dedicated category of compiled code, however. We can treat $\Upsilon\rho.r$ as the following syntactic sugar:

$$\Upsilon\rho.[x_1^{k_1}, x_2^{k_2}, \ldots,] \quad\equiv\quad \Phi\rho.\lambda y.[x_1^{k_1}, x_2^{k_2}, \ldots, y^1]$$

where y is a variable name distinct from the variable names of ρ and the names x_1, x_2, … of the relinkable $[x_1^{k_1}, x_2^{k_2}, \ldots,]$. We may then replace the operator **pro** with **app**:

$$\begin{aligned}\mathbf{app}\ \Upsilon\rho_1.[x_1^{k_1}, x_2^{k_2}, \ldots,]\ \Phi\rho_2.e \quad&\equiv\quad \mathbf{app}\ \Phi\rho_1.\lambda y.[x_1^{k_1}, x_2^{k_2}, \ldots, y^1]\ \Phi\rho_2.e\\ &\rightarrow\quad \Phi\rho_1\uplus\rho_2.((\lambda y.[x_1^{k_1}, x_2^{k_2}, \ldots, y^1])\ e)\\ &\rightarrow\quad \Phi\rho_1\uplus\rho_2.([x_1^{k_1}, x_2^{k_2}, \ldots]\sim e)\end{aligned}$$

Hence, we can do without the operator **pro** and its reduction rule.

We have argued previously that a free identifier abstraction $\Phi\rho.e$ models a piece of separately compiled but yet to be fully linked code and that the set of variables specified in ρ are the unlinked variables of the compiled code. Since our intention is to link these unlinked variables explicitly through incremental compiled code constructions, a Φ-abstraction $\Phi\rho.e$ should confine the variable names of ρ to e. In other words, the abstractor $\Phi\rho$ should also serve as a delimiter for the variable names of ρ. Thus, the name x of the variable x^3 in $\Phi\{\mathbf{x}\!:\!x\}.[x^3]$ is confined by the Φ-abstractor $\Phi\{\mathbf{x}\!:\!x\}$. It can therefore be renamed to some fresh y to yield the α-equivalent term $\Phi\{\mathbf{x}\!:\!y\}.[y^3]$.

Anomalies could occur without the above interpretation. Consider the term

$$\lambda\underline{x}.(y\ \Phi\{\mathbf{x}\!:\!x\}.[x^1])$$

Had we allowed the variable reference x^1 to "see" beyond the Φ-abstractor $\Phi\{\mathbf{x}\!:\!x\}$, it would be linked to the underlined λ-parameter. Suppose that the denotation of the free variable y is determined later to be $\mathbf{lam_X}$. We would then have

$$\begin{array}{rcl} [\mathbf{lam_X}/y]\lambda x.(y\ \Phi\{\mathbf{x}\!:\!x\}.[x^1]) & \equiv & \lambda x.(\mathbf{lam_X}\ \Phi\{\mathbf{x}\!:\!x\}.[x^1]) \\ & \to & \lambda x.\Phi\{\mathbf{x}\!:\!x\}.\lambda\underline{x}.[x^1] \end{array}$$

and the variable reference x^1 would be relinked to the underlined newly introduced λ-parameter. This kind of behavior clearly violates the spirit of static scope.

With Φ-abstractors also playing the role of variable name delimiters, the operation $\mathbf{load}\ \Phi\rho.e$ must activate the implicit name delimitation associated with every variable of ρ and thus finalize its references in e as discussed in Section 7.2.2. Thus, contracting the redex $\mathbf{load}\ \Phi\{\mathbf{x}\!:\!x\}.e$ should yield the term $\Delta^1_x(e)$ that is e but with every variable reference to the name x finalized, which is exactly the expected effect of $\nu x.e$. It is therefore unnecessary for us to include $\nu x.e$ as a distinct term in the context-enriched calculus. Removing $\nu x.e$ means that there is no need for the name delimitation term construction operators $\mathbf{del_X}$ either. The would-be reduction rule for $\mathbf{del_X}$ is

$$\mathbf{del_X}\ \Phi\rho.e \quad\to\quad \Phi\rho.\nu x.e$$

where either $\mathbf{x}\!:\!x \in \rho$ or else x is fresh. It can be simulated as follows (the operator $\mathbf{phi}_{\{\mathbf{X}\}}$ is defined in Chapter 5):

$$\mathbf{del_X} \quad\equiv\quad \lambda x.(\mathbf{app}\ \Phi\{\}.\mathbf{load}\ (\mathbf{phi}_{\{\mathbf{X}\}}\ x))$$

since

$$\begin{aligned}
\mathbf{del}_{\mathbf{x}}\ \Phi\rho.e &= \mathbf{app}\ \Phi\{\}.\mathbf{load}\ (\mathbf{phi}_{\{\mathbf{x}\}}\ \Phi\rho.e) \\
&= \mathbf{app}\ \Phi\{\}.\mathbf{load}\ \Phi\rho.\Phi\{\mathbf{x}\!:\!x\}.e \\
&\qquad \text{[where either } \mathbf{x}\!:\!x \in \rho \text{ or else } x \text{ is fresh]} \\
&= \Phi\rho.(\mathbf{load}\ \Phi\{\mathbf{x}\!:\!x\}.e) \\
&= \Phi\rho.\Delta^1_x(e)
\end{aligned}$$

To summarize, the syntax of the context-enriched calculus of relinkables $\boldsymbol{\lambda}\mathbf{R}$ is:

$$\begin{aligned}
e &::= r \mid r\sim e \mid \lambda x.e \mid e\ e \mid \Phi\rho.e \mid \boldsymbol{\delta} \\
\boldsymbol{\delta} &::= \mathbf{load} \mid \tilde{\mathbf{x}} \mid \mathbf{lam}_{\mathbf{x}} \mid \mathbf{app}
\end{aligned}$$

The new mechanisms introduced are exactly those added to the λ-calculus by $\boldsymbol{\lambda}\mathbf{C}$. The only major change is in the transformation of compiled code into machine code. The **load**-reduction rules used by $\boldsymbol{\lambda}\mathbf{C}$ and $\boldsymbol{\lambda}\mathbf{R}$ are:

$$\mathbf{load}\ \Phi\{\mathbf{x}_1\!:\!x_1,\ldots,\mathbf{x}_n\!:\!x_n\}.e \ \rightarrow\ [\tilde{\mathbf{x}}_n/x_n]\cdots[\tilde{\mathbf{x}}_1/x_1]e \qquad (\boldsymbol{\lambda}\mathbf{C})$$

$$\mathbf{load}\ \Phi\{\mathbf{x}_1\!:\!x_1,\ldots,\mathbf{x}_n\!:\!x_n\}.e \ \rightarrow\ \Delta^1_x(\cdots(\Delta^1_x(e))\cdots) \qquad (\boldsymbol{\lambda}\mathbf{R})$$

In $\boldsymbol{\lambda}\mathbf{C}$, the variable x_i of an unlinked free identifier $\mathbf{x}_i$ is assigned the indicator $\tilde{\mathbf{x}}_i$, *e.g.*,

$$\begin{aligned}
\mathbf{load}\ \Phi\{\mathbf{x}\!:\!x\}.x &= [\tilde{\mathbf{x}}/x]x \\
&= \tilde{\mathbf{x}}
\end{aligned}$$

The same behavior can be achieved in $\boldsymbol{\lambda}\mathbf{R}$ by manually associating the equivalent of the variable x with the unlinked identifier indicator $\tilde{\mathbf{x}}$ as its default denotation:

$$\begin{aligned}
\mathbf{load}\ \Phi\{\mathbf{x}\!:\!x\}.[x^1]\sim\tilde{\mathbf{x}} &= \Delta^1_x([x^1]\sim\tilde{\mathbf{x}}) \\
&= \tilde{\mathbf{x}}
\end{aligned}$$

The finalization of $[x^1]\sim\tilde{\mathbf{x}}$ with respect to the name x, denoted as $\Delta^1_x([x^1]\sim\tilde{\mathbf{x}})$, means that the only constituent x^1 of the relinkable $[x^1]$ is not to be bound; hence the finalization of $[x^1]$ is the arid sequence $[]$. The provisional denotation $\tilde{\mathbf{x}}$ therefore becomes the denotation of $[x^1]\sim\tilde{\mathbf{x}}$

Syntactic Domains:

$$
\begin{array}{rcll}
x & \in & \mathit{Vars} & \text{(Variable Names)} \\
k & \in & \mathit{Dists} & \text{(Distances)} \\
x^k & \in & \mathit{Refs} & \text{(Variable References)} \\
r & \in & \mathit{Relnks} \subset \mathit{Refs}^* & \text{(Relinkables)} \\
\mathbf{x} & \in & \mathit{Idents} & \text{(Identifiers)} \\
\tilde{\mathbf{x}} & \in & \mathit{Unlnks} & \text{(Unlinked Identifier Indicators)}
\end{array}
$$

Abstract Syntax:

$$
\begin{array}{rcl}
r & ::= & [x_1^{k_1}, x_2^{k_2}, \ldots] \\
e & ::= & r \mid r \sim e \mid \lambda x.e \mid e\ e \mid \Phi\rho.e \mid \boldsymbol{\delta} \\
\boldsymbol{\delta} & ::= & \mathbf{load} \mid \tilde{\mathbf{x}} \mid \mathbf{lam}_{\mathbf{x}} \mid \mathbf{app} \mid \mathbf{phi}_{\boldsymbol{\rho}} \\
\rho & ::= & \{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\} \\
\boldsymbol{\rho} & ::= & \{\mathbf{x}_1 : \mathbf{y}_1, \ldots, \mathbf{x}_n : \mathbf{y}_n\}
\end{array}
$$

Figure 7.2: Term language of twice context-enriched calculus of relinkables $\boldsymbol{\lambda}$**RR**

7.4 Twice Context-Enriched Calculus of Relinkables

Once again, we skip the presentation of the once context-enriched calculus $\boldsymbol{\lambda}$**R** and proceed directly to the formal description of the twice context-enriched calculus instead. The syntax of the twice context-enriched calculus of relinkables $\boldsymbol{\lambda}$**RR** is summarized in Figure 7.2. The new compiled code mechanism it adds to $\boldsymbol{\lambda}$**R** are the constructors $\mathbf{phi}_{\boldsymbol{\rho}}$ that build compiled Φ-abstraction code (cf. Chapter 5 for the development of $\boldsymbol{\lambda}$**CC** from $\boldsymbol{\lambda}$**C**). As usual, the description of $\boldsymbol{\lambda}$**RR** consists of the formal definition of the notion of free variables, the α-equivalence relation among $\boldsymbol{\lambda}$**RR**-terms, and the reduction rules underlying $\boldsymbol{\lambda}$**RR**.

In the following, we take the notational liberty of treating the representation of each relinkable r as if it were the (possibly infinite) sequence $[x_1^{k_1}, x_2^{k_2}, \ldots]$ that it denotes. We use the notation $\bar{x}^k$ to abbreviate a (possibly infinite) series of variable references $x_1^{k_1}, x_2^{k_2}, \ldots$. The notation $[\bar{x}^k]$ denotes a relinkable whose constituents are $x_1^{k_1}, x_2^{k_2}, \ldots$. We also write $f(\bar{x}^k)$, where f is one of the meta-operations to be defined, for the distribution of f over $\bar{x}^k$, *i.e.*, $f(x_1^{k_1}), f(x_2^{k_2}), \ldots$. Likewise, when $\bar{e}$ is a series of terms $e_1, \ldots, e_n$, $f(\bar{e})$ denotes $f(e_1), \ldots, f(e_n)$. Another notational convention we employ in this section is $x^k :: r$. It denotes a relinkable whose first constituent is x^k and the rest is the sequence r.

7.4.1 Free Variables and Free Variable Names

A variable reference x^k is *free* in a $\boldsymbol{\lambda}$**RR**-term e if it is linked to some λ- or Φ-parameter. The set of free variable references occurring in a term e is denoted as $fv(e)$. It is defined inductively on the structure of e as follows:

$$\begin{aligned}
fv([\bar{x}^k]) &= \{\bar{x}^k\} && (7.1)\\
fv(r \sim e) &= fv(r) \cup fv(e) && (7.2)\\
fv(\lambda x.e) &= \{y^k \in fv(e) \mid y \not\equiv x\} \cup \{x^{k-1} \mid x^k \in fv(e), k > 1\} && (7.3)\\
fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) &&\\
fv(\Phi\{\mathbf{\bar{x}}:\bar{x}\}.e) &= \{x^k \in fv(e) \mid x \notin \{\bar{x}\}\} && (7.4)\\
fv(\boldsymbol{\delta}) &= \emptyset &&
\end{aligned}$$

The free variables of a relinkable $[\bar{x}^k]$ are the variable references $\bar{x}^k$ (Clause 7.1). The free variables of a provisionally-instantiated relinkable $[\bar{x}^k] \sim e$ are the free variables of $[\bar{x}^k]$ and e combined (Clause 7.2). For a λ-abstraction , every free variable of the body term e with the same name x as the λ-parameter has its distance decremented by one to compensate for the binding occurrence introduced by the abstractor (Clause 7.3). In the Φ-abstraction clause (Clause 7.4), the ρ-parameters serve as name delimiters; hence, no variables bearing the same names are free beyond the abstraction.

The set of names of the free variable references occurring in a $\boldsymbol{\lambda}\mathbf{RR}$-term e is denoted as $fvn(e)$. Formally,

$$fvn(e) \quad = \quad \{x \mid x^k \in fv(e)\}$$

7.4.2 Distance Adjustments

The meta-operation $\uparrow^n_x(e)$, where $n \geq 1$, increments the distance of every free variable reference in e with name x and a distance not less than n. It is defined inductively on the structure of e as follows:

$$\begin{array}{rcl}
\uparrow^n_x([\bar{z}^k]) & \equiv & [\uparrow^n_x(\bar{z}^k)] \\
 & & \quad\text{where } \uparrow^n_x(z^k) \equiv \begin{cases} z^{k+1} & \text{if } x \equiv z \text{ and } k \geq n \\ z^k & \text{otherwise} \end{cases} \\
\uparrow^n_x(r \sim e) & \equiv & \uparrow^n_x(r) \sim \uparrow^n_x(e) \\
\uparrow^n_x(\lambda z.e) & \equiv & \begin{cases} \lambda z.\uparrow^{n+1}_x(e) & \text{if } x \equiv z \\ \lambda z.\uparrow^n_x(e) & \text{otherwise} \end{cases} \\
\uparrow^n_x(e_1\ e_2) & \equiv & \uparrow^n_x(e_1)\ \uparrow^n_x(e_2) \\
\uparrow^n_x(\Phi\{\bar{\mathbf{x}}:\bar{x}\}.e) & \equiv & \begin{cases} \Phi\{\bar{\mathbf{x}}:\bar{x}\}.e & \text{if } x \in \{\bar{x}\} \\ \Phi\{\bar{\mathbf{x}}:\bar{x}\}.\uparrow^n_x(e) & \text{otherwise} \end{cases} \\
\uparrow^n_x(\boldsymbol{\delta}) & \equiv & \boldsymbol{\delta}
\end{array}$$

As its counterpart for the λ_{bf}-calculus, the operation traverses down a term while keeping track of the number of λ-parameters named x encountered, which is n. It looks for variables of the form x^k and increases the distance k by one if k is not less than n. The traversal ends at a Φ-abstraction that has x as a parameter variable, since any reference to x in the abstraction's body is not free beyond the abstraction.

Analogously, the meta-operation $\downarrow^n_x(e)$, where $n \geq 2$, decrements the distance of every variable reference in e with the name x and a distance not less than n:

$$\begin{array}{rcl}
\downarrow^n_x([\bar{z}^k]) & \equiv & [\downarrow^n_x(\bar{z}^k)] \\
 & & \quad\text{where } \downarrow^n_x(z^k) \equiv \begin{cases} z^{k-1} & \text{if } x \equiv z \text{ and } k \geq n \\ z^k & \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{rcl}
\downarrow^n_x(r \sim e) & \equiv & \downarrow^n_x(r) \sim \downarrow^n_x(e) \\
\downarrow^n_x(\lambda z.e) & \equiv & \begin{cases} \lambda z.\downarrow^{n+1}_x(e) & \text{if } x \equiv z \\ \lambda z.\downarrow^n_x(e) & \text{otherwise} \end{cases} \\
\downarrow^n_x(e_1\ e_2) & \equiv & \downarrow^n_x(e_1)\ \downarrow^n_x(e_2) \\
\downarrow^n_x(\Phi\{\mathbf{\bar{x}}:\bar{x}\}.e) & \equiv & \begin{cases} \Phi\{\mathbf{\bar{x}}:\bar{x}\}.e & \text{if } x \in \{\bar{x}\} \\ \Phi\{\mathbf{\bar{x}}:\bar{x}\}.\downarrow^n_x(e) & \text{otherwise} \end{cases} \\
\downarrow^n_x(\boldsymbol{\delta}) & \equiv & \boldsymbol{\delta}
\end{array}$$

7.4.3 Alpha Convertibility

The renaming of a Φ-abstraction parameter variable is defined by the following reduction rule:

$$\Phi\{\mathbf{\bar{x}}:\bar{x}\}.e \ \rightarrow\ \Phi\{\mathbf{\bar{x}}:\langle y/x\rangle\bar{x}\}.\langle\!\langle y^1/x^1\rangle\!\rangle e \qquad (\alpha\text{-}\Phi)$$
$$\text{where } x \in \{\bar{x}\},\ y \notin \{\bar{x}\},\ y \notin fvn(e)$$

It replaces the name x of one of the parameter variables $\bar{x}$ with y, hence the notation $\langle y/x\rangle\bar{x}$. The new variable name y is neither one of $\bar{x}$ nor the name of any free variable occurring in the body e. The former ensures the well-formedness of $\{\mathbf{\bar{x}}:\langle y/x\rangle\bar{x}\}$. The latter avoids inadvertent capture.

The ρ-parameter variable renaming meta-operation $\langle\!\langle y^n/x^m\rangle\!\rangle e$, where $x \not\equiv y$, replaces free occurrences of x^m, x^{m+1}, x^{m+2}, ... in e with y^n, y^{n+1}, y^{n+2} ..., respectively. It is defined by induction on the structure of e as follows:

$$\begin{array}{rcl}
\langle\!\langle y^n/x^m\rangle\!\rangle[\bar{z}^k] & \equiv & [\langle\!\langle y^n/x^m\rangle\!\rangle\bar{z}^k] \\
 & & \text{where } \langle\!\langle y^n/x^m\rangle\!\rangle z^k \equiv \begin{cases} y^{n+k-m} & \text{if } x \equiv z \text{ and } k \geq m \\ z^k & \text{otherwise} \end{cases} \\
\langle\!\langle y^n/x^m\rangle\!\rangle(r \sim e) & \equiv & \langle\!\langle y^n/x^m\rangle\!\rangle r \sim \langle\!\langle y^n/x^m\rangle\!\rangle e \\
\langle\!\langle y^n/x^m\rangle\!\rangle\lambda z.e & \equiv & \lambda z.\langle\!\langle \uparrow^1_z(y^n)/\uparrow^1_z(x^m)\rangle\!\rangle e \\
\langle\!\langle y^n/x^m\rangle\!\rangle(e_1\ e_2) & \equiv & \langle\!\langle y^n/x^m\rangle\!\rangle e_1\ \langle\!\langle y^n/x^m\rangle\!\rangle e_2
\end{array}$$

$$
\begin{array}{rcl}
\langle\!\langle y^n/x^m \rangle\!\rangle \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e & \equiv & \begin{cases} \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e & \text{if } x \in \{\bar{x}\} \\ \Phi\{\bar{\mathbf{x}}\!:\!\langle z/y\rangle\bar{x}\}.\langle\!\langle y^n/x^m \rangle\!\rangle\langle\!\langle z^1/y^1 \rangle\!\rangle e & \text{if } x \notin \{\bar{x}\},\, y \in \{\bar{x}\} \\ \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.\langle\!\langle y^n/x^m \rangle\!\rangle e & \text{otherwise} \end{cases} \\
\langle\!\langle y^n/x^m \rangle\!\rangle \boldsymbol{\delta} & \equiv & \boldsymbol{\delta}
\end{array}
$$

In the second case of the Φ-abstraction clause, where $x \notin \{\bar{x}\}$ and $y \in \{\bar{x}\}$, the new variable name z to which y is renamed is fresh. It is not x, nor one of $\bar{x}$, nor a free variable name of e.

Although there is no more need for λ-parameter renaming in the definition of β-reduction, we are obliged to present such an α-conversion rule, however, since the calculus employs names:

$$\lambda x.e \;\;\rightarrow\;\; \lambda y.{\downarrow}^2_x(\langle y^1/x^1\rangle {\uparrow}^1_y(e)) \quad \text{where } x \not\equiv y \qquad (\alpha\text{-}\lambda)$$

The distance of every free variable y^k with $k \geq 1$ in e is incremented, ${\uparrow}^1_y(e)$, since it is in the scope of one more λ-parameter named y. The references to the original λ-parameter x in ${\uparrow}^1_y(e)$ are then renamed to y^1, which is denoted as $\langle y^1/x^1\rangle{\uparrow}^1_y(e)$. Finally, the distance of every free variable x^k with $k \geq 2$ in e is decremented because it is in the scope of one less λ-parameter named x. Now, there is no provision that the new name y must not be a free variable of e. Indeed, the only constraint on the choice of y is that it is not the same as x, the one to be replaced. Otherwise, the renaming would be unnecessary since we can show that ${\downarrow}^2_x(\langle x^1/x^1\rangle{\uparrow}^1_x(e))$ is identical to e. Intuitively, there are no free occurrences of x^1 after ${\uparrow}^1_x(e)$; hence the result of $\langle x^1/x^1\rangle{\uparrow}^1_x(e)$ is identical to ${\uparrow}^1_x(e)$. But then ${\downarrow}^2_x({\uparrow}^1_x(e))$ is identical to e.

The meta-operation $\langle y^n/x^m\rangle e$ that replaces each free variable x^m in e with y^n is defined inductively as follows:

$$
\begin{array}{rcl}
\langle y^n/x^m\rangle[\bar{z}^k] & \equiv & [\langle y^n/x^m\rangle\bar{z}^k] \\
 & & \qquad \text{where } \langle y^n/x^m\rangle z^k \equiv \begin{cases} y^n & \text{if } x \equiv z \text{ and } k = m \\ z^k & \text{otherwise} \end{cases} \\
\langle y^n/x^m\rangle(r \sim e) & \equiv & \langle y^n/x^m\rangle r \sim \langle y^n/x^m\rangle e \\
\langle y^n/x^m\rangle\lambda z.e & \equiv & \lambda z.\langle {\uparrow}^1_z(y^n)/{\uparrow}^1_z(x^m)\rangle e
\end{array}
$$

$$
\begin{array}{rcl}
\langle y^n/x^m\rangle(e_1\ e_2) & \equiv & \langle y^n/x^m\rangle e_1\ \langle y^n/x^m\rangle e_2 \\
\langle y^n/x^m\rangle\Phi\{\bar{\mathbf{x}}:\bar{x}\}.e & \equiv & \begin{cases} \Phi\{\bar{\mathbf{x}}:\bar{x}\}.e & \text{if } x \in \{\bar{x}\} \\ \Phi\{\bar{\mathbf{x}}:\langle z/y\rangle\bar{x}\}.\langle y^n/x^m\rangle\langle\!\langle z^1/y^1\rangle\!\rangle e & \text{if } x \notin \{\bar{x}\},\ y \in \{\bar{x}\} \\ \Phi\{\bar{\mathbf{x}}:\bar{x}\}.\langle y^n/x^m\rangle e & \text{otherwise} \end{cases} \\
\langle y^n/x^m\rangle\boldsymbol{\delta} & \equiv & \boldsymbol{\delta}
\end{array}
$$

In the second case of the Φ-abstraction clause, where $x \notin \{\bar{x}\}$ and $y \in \{\bar{x}\}$, the new variable name z to which y is renamed is fresh. It is not x, nor one of $\bar{x}$, nor a free variable name of e.

7.4.4 Reduction Rules

The three compiled code related reduction rules of **λCC**, namely, **lam**, **app**, and **phi**$_{\boldsymbol{\rho}}$, carry over unchanged to the twice context-enriched calculus of relinkables **λRR**. The other two rules β and **load** must be adapted to the new form of variable reference.

Function Invocation

The β-reduction rule of **λRR** is of the same form as the one for λ_{bf}:

$$(\lambda x.e)\ e' \ \ \rightarrow\ \ \downarrow^2_x([\uparrow^1_x(e')/x^1]e) \qquad (\beta)$$

Every free variable x^1 in e refers to the parameter x of the λ-abstraction to be applied. It should therefore be substituted with the term e', $[e'/x^1]e$. Every free variable x^k of a distance k greater than 1 in e must have its distance decremented by one, $\downarrow^2_x([e'/x^1]e)$, since one of its enclosing λ-abstractors is removed. This decrement operation should not apply to occurrences of e', however, since e' was not originally in the scope of the λ-abstraction to be applied. To compensate for the effect of the decrement operation, every free variable of the name x in e' is incremented before performing the substitution; hence, $\downarrow^2_x([\uparrow^1_x(e')/x^1]e)$.

Let $\sharp(x^n, r)$ denote the prefix of r up to, but not including, the constituent variable reference x^n:

$$\sharp(x^n, z^k :: r) \quad \equiv \quad \begin{cases} [] & \text{if } x \equiv z \text{ and } k = n \\ z^k :: \sharp(x^n, r) & \text{otherwise} \end{cases}$$

Then, the β-substitution meta-operation $[e'/x^n]e$ that provides every free variable x^n in e with the denotation e' is defined by induction on the structure of e as follows:

$$\begin{aligned}
[e'/x^n][\bar{z}^k] \quad &\equiv \quad \begin{cases} e' & \text{if } x^n \in \{\bar{z}^k\} \text{ and } \sharp(x^n, [\bar{z}^k]) \equiv [] \\ r \sim e' & \text{if } x^n \in \{\bar{z}^k\} \text{ and } \sharp(x^n, [\bar{z}^k]) \equiv r \not\equiv [] \\ [\bar{z}^k] & \text{otherwise} \end{cases} \\
[e'/x^n]([\bar{z}^k] \sim e) \quad &\equiv \quad \begin{cases} e' & \text{if } x^n \in \{\bar{z}^k\} \text{ and } \sharp(x^n, [\bar{z}^k]) \equiv [] \\ r \sim e' & \text{if } x^n \in \{\bar{z}^k\} \text{ and } \sharp(x^n, [\bar{z}^k]) \equiv r \not\equiv [] \\ [\bar{z}^k] \sim [e'/x^n]e & \text{otherwise} \end{cases} \\
[e'/x^n]\lambda z.e \quad &\equiv \quad \lambda z.[\uparrow^1_z(e')/\uparrow^1_z(x^n)]e \\
[e'/x^n](e_1\ e_2) \quad &\equiv \quad [e'/x^n]e_1\ [e'/x^n]e_2 \\
[e'/x^n]\Phi\rho.e \quad &\equiv \quad \Phi\rho.[e'/x^n]e \\
[e'/x^n]\boldsymbol{\delta} \quad &\equiv \quad \boldsymbol{\delta}
\end{aligned}$$

In the clause for provisionally-instantiated relinkables, when x^n is one of the variables in $[\bar{z}^k]$, the result of the substitution is either the denotation e', provided that x^n is the dominant constituent of $[\bar{z}^k]$, or a provisionally-instantiated relinkable $r \sim e'$ in which r consists of the variables of $[\bar{z}^k]$ that are more dominating than x^n. A similar reasoning applies to the clause for relinkables. We have used the variable conventions in the Φ-abstraction clause. In particular, we have assumed that x is not one of the variables of ρ and none of the variables of ρ occur free in e'.

Loading Compiled Code

The revised reduction rule for **load** is:

$$\textbf{load}\ \Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e \quad \rightarrow \quad \Delta^1_{x_1}(\cdots(\Delta^1_{x_n}(e))\cdots) \qquad (\textbf{load})$$

It loads the compiled code $\Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e$ by removing the parameter specification $\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}$ after *finalizing* the names $x_1, \ldots, x_n$ occurring in e.

Let $\hat{\Delta}^n_x(r)$ denote the finalization of a relinkable reference r with respect to the variable name x:

$$\begin{array}{rcl}
\hat{\Delta}^n_x([]) & \equiv & [] \\
\hat{\Delta}^n_x(z^k :: r) & \equiv & \begin{cases} \hat{\Delta}^n_x(r) & \text{if } x \equiv z \text{ and } k \geq n \\ z^k :: \hat{\Delta}^n_x(r) & \text{otherwise} \end{cases}
\end{array}$$

That is, it removes from r any constituent x^k with a distance $k \geq n$. Then, the finalization meta-operation $\Delta^n_x(e)$ that removes from e free variable references x^k with a distance k not less than n is defined by induction on the structure of e as follows:

$$\begin{array}{rcl}
\Delta^n_x(r) & \equiv & \hat{\Delta}^n_x(r) \\
\Delta^n_x(r \sim e) & \equiv & \begin{cases} \Delta^n_x(e) & \text{if } \hat{\Delta}^n_x(r) \equiv [] \\ \hat{\Delta}^n_x(r) \sim \Delta^n_x(e) & \text{otherwise} \end{cases} \\
\Delta^n_x(\lambda z.e) & \equiv & \begin{cases} \lambda z.\Delta^{n+1}_x(e) & \text{if } x \equiv z \\ \lambda z.\Delta^n_x(e) & \text{otherwise} \end{cases} \\
\Delta^n_x(e_1\ e_2) & \equiv & \Delta^n_x(e_1)\ \Delta^n_x(e_2) \\
\Delta^n_x(\Phi\{\bar{\mathbf{x}} : \bar{x}\}.e) & \equiv & \begin{cases} \Phi\{\bar{\mathbf{x}} : \bar{x}\}.e & \text{if } x \in \{\bar{x}\} \\ \Phi\{\bar{\mathbf{x}} : \bar{x}\}.\Delta^n_x(e) & \text{otherwise} \end{cases} \\
\Delta^n_x(\boldsymbol{\delta}) & \equiv & \boldsymbol{\delta}
\end{array}$$

The operation keeps a count n of the number of λ-bound x's as it traverses down a term until it encounters a relinkable variable r or a provisionally-instantiated relinkable $r \sim e$. At that point, it is clear that a variable x^k whose distance k is less than n is bound by one of the λ-parameters encountered in the traversal and thus has a denotation. Otherwise, the variable x^k does not have a denotation and is therefore discarded.

$$
\begin{array}{rcll}
(\lambda x.e)\ e' & \rightarrow & [e'/x]e & (\beta) \\
\mathbf{load}\ \Phi\{\mathbf{x}_1\!:\!x_1,\ldots,\mathbf{x}_n\!:\!x_n\}.e & \rightarrow & \Delta^1_{x_1}(\cdots(\Delta^1_{x_n}(e))\cdots) & (\mathbf{load}) \\
\mathbf{app}\ \Phi\rho_1.e_1\ \Phi\rho_2.e_2 & \rightarrow & \Phi\rho_1\uplus\rho_2.(e_1\ e_2) & (\mathbf{app}) \\
\mathbf{lam}_{\mathbf{x}}\ \Phi\rho.e & \rightarrow & \Phi\rho.\lambda x.e & (\mathbf{lam}) \\
 & & \quad \text{where } \mathbf{x}\!:\!x \in \rho & \\
 & & \quad \text{or else } x \text{ is fresh} & \\
\mathbf{phi}_{\{\mathbf{x}_1:\mathbf{y}_1,\ldots,\mathbf{x}_n:\mathbf{y}_n\}}\ \Phi\rho.e & \rightarrow & \Phi\rho.\Phi\{\mathbf{x}_1\!:\!y_1,\ldots,\mathbf{x}_n\!:\!y_n\}.e & (\mathbf{phi}_{\boldsymbol{\rho}}) \\
 & & \quad \text{where } \mathbf{y}_i\!:\!y_i \in \rho & \\
 & & \quad \text{or else } y_i \text{ is fresh} &
\end{array}
$$

Figure 7.3: Reduction rules of twice context-enriched calculus of relinkables $\boldsymbol{\lambda}\mathbf{RR}$

7.4.5 Calculus of Compiled Code

The notion of reduction **rr** underlying the twice context-enriched calculus of relinkables $\boldsymbol{\lambda}\mathbf{RR}$ is the union of the reduction rules collectively displayed in Figure 7.3:

$$\mathbf{rr} \quad = \quad \beta \cup \mathbf{load} \cup \mathbf{lam} \cup \mathbf{app} \cup \mathbf{phi}_{\boldsymbol{\rho}}$$

A term e_1 one-step **rr**-reduces to a term e_2, written as $e_1 \rightarrow e_2$, if e_2 is the result of replacing an **rr**-redex subterm of e_1 with its contractum. The reflexive and transitive closure of $\rightarrow$ is the **rr**-reduction relation $\twoheadrightarrow$. The least equivalence relation $=$ generated by $\twoheadrightarrow$ is the calculus of relinkable variables $\boldsymbol{\lambda}\mathbf{RR}$. Again, $\boldsymbol{\lambda}\mathbf{RR}$ preserves the Church-Rosser property as well as the indistinguishability relation $\approx$ among free identifier abstractions.

This ends the formal description of the twice context-enriched calculus of relinkables. In the next chapter we present two applications that rely heavily on relinkable variable references, namely, object-oriented programming and interactive program development. Before that, we explore the possibilities of a more flexible notion of compiled code optimization.

7.5 Optimizing Compiled Code

The only means we have so far for "executing" compiled code is the operator **load**. Its associated finalization functionality is quite eager:

$$\textbf{load}\ \Phi\{\bar{\mathbf{x}}:\bar{x}\}.e \quad\rightarrow\quad \Delta^{1}_{\bar{x}}(e)$$

All free references to the parameter variables $\bar{x}$ are removed from e, which is inadequate for some applications. For instance, it is often the case that we wish to optimize the body e of some compiled code $\Phi\rho.e$ to a certain kind of value v such as an abstraction without finalizing all of its relinkables and provisionally-instantiated relinkables (computation within the body of compiled code is classified as compiled code optimization). For example, we might want to optimize

$$A \quad=\quad \Phi\{\mathbf{x}:x\}.[x^2]\!\sim\!\lambda y.(y\ [x^1]\!\sim\!\mathbf{app})$$

to

$$B \quad=\quad \Phi\{\mathbf{x}:x\}.\lambda y.(y\ [x^1]\!\sim\!\mathbf{app})$$

Only the provisionally-instantiated relinkable $[x^2]\!\sim\!\lambda y.(y\ [x^1]\!\sim\!\mathbf{app})$ is finalized; the other provisionally-instantiated relinkable $[x^1]\!\sim\!\mathbf{app}$ is left untouched so that it can be relinked later. Such behavior is not expressible using **load**:

$$\textbf{load}\ A \quad=\quad \lambda y.(y\ \mathbf{app})$$

It would finalize both relinkables.

We therefore introduce in this section a special compiled code optimization mechanism for such occasions. Let us reclassify the **$\boldsymbol{\lambda}$RR**-terms by singling out terms that we consider as values:

$$e \quad ::= \quad v \mid r \mid r\!\sim\!e \mid e\ e$$

A value is anything but an application, a relinkable, or a provisionally-instantiated relinkable. An application may be a redex representing a computational step. It

$$\begin{array}{rcll}
\rhd\ \Phi\rho.v & \rightarrow & \Phi\rho.v & (\ \rhd_v) \\
\rhd\ \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.r & \rightarrow & \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.[] \quad \text{if } \Delta^{1}_{\bar{x}}(r) \equiv [] & (\ \rhd_r) \\
\rhd\ \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.(r \sim e) & \rightarrow & \rhd\ \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e \quad \text{if } \Delta^{1}_{\bar{x}}(r) \equiv [] & (\ \rhd_\sim) \\
\rhd\ \Phi\rho.(e_1\ e_2) & \rightarrow & \rhd\ (\mathbf{app}\ (\rhd\ \Phi\rho.e_1)\ \Phi\rho.e_2) & (\ \rhd_f) \\
 & & \quad \text{if } e_1\ e_2 \text{ is not a redex} & \\
 & & \quad \text{and } e_1 \text{ is not a value} & \\
\rhd\ \Phi\rho.(f\ \ e) & \rightarrow & \rhd\ (\mathbf{app}\ \Phi\rho.f\ (\rhd\ \Phi\rho.e)) & (\ \rhd_a) \\
 & & \quad \text{if } f\ e \text{ is not a redex} & \\
 & & \quad \text{and } e \text{ is not a value} &
\end{array}$$

Figure 7.4: Compiled code optimization rules

is therefore not considered a value. A relinkable can denote anything, including an application, and so can a provisionally-instantiated relinkable. They are thus not values either. We further classify value terms v into two categories: those that can serve as the operator of a redex, denoted as f, and those that cannot, denoted as a:

$$\begin{array}{rcl}
v & ::= & f \mid a \\
f & ::= & \lambda x.e \mid \mathbf{load} \mid \mathbf{lam}_{\mathbf{X}} \mid \mathbf{app} \mid \mathbf{phi}_{\boldsymbol{\rho}} \mid \mathbf{app}\ \Phi\rho.e \\
a & ::= & \Phi\rho.e \mid \tilde{\mathbf{x}}
\end{array}$$

The application term $\mathbf{app}\ \Phi\rho.e$ is considered equivalent to $\lambda x.(\mathbf{app}\ \Phi\rho.e\ x)$ and therefore qualifies as the operator of a redex.

With the above classification, we can characterize the optimization of compiled code $\Phi\rho.e$ to compiled value $\Phi\rho.v$ by introducing a new compiled code operator $\rhd$ with the reduction rules shown in Figure 7.4. Intuitively, $\rhd\ \Phi\rho.e$ attempts to reduce the body term e to a value, if there is one. In the process, relinkables are finalized on a by-need basis. The first three optimization rules handle the base cases:

$(\ \rhd_v)$ Optimization stops if the body term is already a value.

($\rhd_r$) When the body term is a relinkable, it is finalized with respect to the parameter variables of the compiled code to find out if the relinkable is unbound. If so, optimization stops at an unbound value; otherwise, the process hangs.

($\rhd_\sim$) Similarly, when the body term is a provisionally-instantiated relinkable, finalization is used to determine if the provisional denotation is the denotation of the relinkable.

When the body term is an application, the compiled code $\Phi\rho.(e_1\ e_2)$ can be deconstructed into the function part $\Phi\rho.e_1$ and the argument part $\Phi\rho.e_2$ from which the original compiled code can be easily reconstructed using the operator **app**:

$$\Phi\rho.(e_1\ e_2) \quad = \quad \mathbf{app}\ \Phi\rho.e_1\ \Phi\rho.e_2$$

If the application $e_1\ e_2$ is a redex, the redex should be kept intact so that it can be contracted. Otherwise, the application is deconstructed and the two parts are optimized, hoping that the reconstructed application will be a redex. They are modeled by the other two optimization rules:

($\rhd_f$) For the reconstructed application to be a redex, the function part must be optimized to a value that can serve as the operator of a redex.

($\rhd_a$) If the application is not a redex but its function part is already an operator, the argument part must be optimized to a value.

Again, the new reduction rules for compiled code optimization function orthogonally to the existing rule of the $\boldsymbol{\lambda}$**RR**-calculus. Their inclusion into $\boldsymbol{\lambda}$**RR** therefore does not upset the Church-Rosser or the free identifier indistinguishability property. Uses of such a compiled code optimization mechanism can be found in the following chapter.

Chapter 8

Incremental Programming

We employ the two context-enriched calculi **$\boldsymbol{\lambda}$RR** and **$\boldsymbol{\lambda}$DD** combined to model object-oriented and interactive programming paradigms. It is not our intention here to describe a complete programming system for each paradigm. The emphasis, instead, is on the use of modules and relinkables to express the incremental nature of their linking needs.

8.1 Object-Oriented Programming

We focus on using our context-enhanced calculi to express these three fundamental notions of object-oriented programming: object encapsulation, object inheritance, and late binding. Object encapsulation achieves information hiding and modularity. Objects interact with one another only through clearly specified import and export interfaces. Object inheritance implements code reuse and code organization. It is the mechanism by which new and enhanced objects can be defined in terms of existing objects. Late binding provides the necessary means for existing objects to adapt to their ever-changing surroundings.

We employ the following notational shorthands to enhance the readability of the examples. A simple variable reference x is an abbreviation of the one-element relinkable $[x^1]$. A relinkable $[x^\infty, \ldots, x^1]$ with an infinite number of constituents is notated as $\ddot{x}$. The notation $\ddot{x}[y]$ denotes the relinkable $[x^\infty, \ldots, x^1, y^1]$ where the name y is distinct from the name x. The least dominating variable reference y^1 is linked; the other constituents $x^\infty, \ldots, x^1$ are yet unbound.

8.1.1 Object Representation

An object is a module $\Psi\langle\varepsilon, \rho\rangle.d$. The identifiers specified by the export interface ε are the object's public attributes (an attribute can be either a method or an instance variable). Bindings of d not specified by ε are privately available to the object only. The import interface ρ specifies the object's dependency on other objects. As an example, the object *aCP* below is a two-dimensional Cartesian point whose coordinates are 3 and 4:

$$\begin{aligned} aCP \quad \equiv \quad & \Psi\langle\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d, \mathbf{closer}\!:\!c\}, \\ & \quad \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d\}\rangle. \\ & \quad (rec\ \{x = 3, y = 4, \\ & \qquad\quad d = \sqrt{\ddot{x}^2 + \ddot{y}^2}, \\ & \qquad\quad c = \lambda p.(\ddot{d} < \mathbf{dist}@p)\}) \end{aligned}$$

It exports four defining variables x, y, d, and c known externally as **xcor**, **ycor**, **dist**, and **closer**. The first two hold the coordinates. The third one denotes the Cartesian distance of the point from origin. The fourth attribute is a predicate measuring which point is closer to origin, the object *aCP* itself or the point object denoted by the parameter p. The notation **dist**@p selects the **dist**-attribute of the point object denoted by p. The object uses three late binding virtual references, namely, $\ddot{x}$, $\ddot{y}$, and $\ddot{d}$. They are known externally as the import attributes **xcor**, **ycor**, and **dist**.

The four bindings of *aCP* depend on one another. They are therefore expressed as a recursive definition:

$$rec\ \{x = \ldots, y = \ldots, d = \ldots, c = \ldots\}$$

The linking relations among the bindings can be more succintly expressed as follows:

$$\begin{aligned} aCP \quad \equiv \quad & \Psi\langle\{\mathbf{xcor}:x', \mathbf{ycor}:y', \mathbf{dist}:d', \mathbf{closer}:c'\}, \\ & \quad \{\mathbf{xcor}:x, \mathbf{ycor}:y, \mathbf{dist}:d\}\rangle. \\ & \quad (\mathit{rec}\ \{x' = 3, y' = 4, \\ & \qquad d' = \sqrt{\ddot{x}[x']^2 + \ddot{y}[y']^2}, \\ & \qquad c' = \lambda p.(\ddot{d}[d'] < \mathbf{dist}@p)\}) \end{aligned}$$

The virtual reference $\ddot{d}$ has been revised to depict its latest linking relation. The least dominating constituent d^1 of $\ddot{d} \equiv [d^\infty, \ldots, d^1]$ is bound by the object's defining variable d. It is the only constituent of $\ddot{d}$ that is linked. To highlight the linking relation, we therefore rename the defining variable d to a fresh name d'. Accordingly, the virtual reference $\ddot{d}$ becomes $[d^\infty, \ldots, d^1, d'^1]$, hence the abbreviation $\ddot{d}[d']$. In a similar fashion, the other two virtual references $\ddot{x}$ and $\ddot{y}$ become $\ddot{x}[x']$ and $\ddot{y}[y']$ to illustrate that they are linked to the defining variables x' and y'.

The above view of aCP stresses the linking effect among the object's attributes. Alternatively, we can illustrate its computational effect through β-substitutions, hence yielding the following simple definition representation of the same object aCP:

$$\begin{aligned} aCP \quad = \quad & \Psi\langle\{\mathbf{xcor}:x, \mathbf{ycor}:y, \mathbf{dist}:d, \mathbf{closer}:c\}, \\ & \quad \{\mathbf{xcor}:x, \mathbf{ycor}:y, \mathbf{dist}:d\}\rangle. \\ & \quad \{x = 3, y = 4, \\ & \quad\ d = \sqrt{(\ddot{x}\sim 3)^2 + (\ddot{y}\sim 4)^2}, \\ & \quad\ c = \lambda p.((\ddot{d}\sim\sqrt{(\ddot{x}\sim 3)^2 + (\ddot{y}\sim 4)^2}) < \mathbf{dist}@p)\} \end{aligned}$$

The denotation 3 of the defining variable x is substituted for the variable reference x^1 of the relinkable $\ddot{x} \equiv [x^\infty, \ldots, x^1]$ of $\sqrt{\ddot{x}^2 + \ddot{y}^2}$. Since x^1 is not the dominant constituent of $\ddot{x}$, 3 is only a provisional denotation. The substitution therefore results in a provisionally-instantiated relinkable $[x^{\infty-1}, \ldots, x^{2-1}]\sim 3$, or simply $\ddot{x}\sim 3$. Similarly, the relinkable $\ddot{y}$ of $\sqrt{\ddot{x}^2 + \ddot{y}^2}$ is replaced with the provisionally-instantiated relinkable

$\ddot{y}\!\sim\!4$. Likewise, the virtual reference $\ddot{d}$ of $\lambda p.(\ddot{d} < \textbf{dist}@p)$ is provisionally-instantiated to $\ddot{d}\!\sim\!\sqrt{(\ddot{x}\!\sim\!3)^2 + (\ddot{y}\!\sim\!4)^2}$. That is, the latest meaning of $\ddot{d}$ is the latest denotation of the defining variable d, which is $\sqrt{(\ddot{x}\!\sim\!3)^2 + (\ddot{y}\!\sim\!4)^2}$. All in all, the three virtual references $\ddot{x}$, $\ddot{y}$, and $\ddot{d}$ of *aCP* each not only gets an up-to-date denotation but also remains responsive to future overrides, as shown in the next section.

8.1.2 Object Inheritance

The essence of object-oriented programming is the ability to incrementally create new objects by modifying existing objects. The underlying mechanism is object inheritance, which changes the behavior of an existing object by extending the object with new attributes or by providing new denotations for the object's virtual references.

To illustrate, we modify the Cartesian point object *aCP* of the last section to obtain a Manhattan point object *aMP*. The only change is the distance measured from origin, which is encoded as the following module

$$\mathit{deltaM} \quad \equiv \quad \Psi\langle\{\textbf{dist}\!:\!d\}, \{\textbf{xcor}\!:\!x, \textbf{ycor}\!:\!y\}\rangle.\{d = \ddot{x} + \ddot{y}\}$$

The **dist**-attribute is now calculated as the sum of the coordinates. The new object *aMP* is then the following combination of the existing object *aCP* and the modifier *deltaM*:

$$\mathit{aMP} \quad = \quad \textbf{moverride}\ \mathit{aCP}\ \ \mathit{deltaM}$$

The module combination constructor **moverride** links the imports of *aCP* to the exports of *deltaM* and the imports of *deltaM* to the exports of *aCP*; moreover, the **dist**-binding of *deltaM* overrides the **dist**-binding of *aCP* (cf. Section 6.5.4). Hence,

$$\begin{aligned}
\mathit{aMP} \quad &= \quad \textbf{moverride}\ \mathit{aCP}\ \ \mathit{deltaM} \\
&= \quad \Psi\langle\{\textbf{xcor}\!:\!x, \textbf{ycor}\!:\!y, \textbf{dist}\!:\!d, \textbf{closer}\!:\!c\}, \\
&\qquad\quad \{\textbf{xcor}\!:\!x, \textbf{ycor}\!:\!y, \textbf{dist}\!:\!d\}\rangle. \\
&\qquad\quad (\mathit{moverride}\ \{x = 3, y = 4,
\end{aligned}$$

$$
\begin{aligned}
&\qquad d = \sqrt{(\ddot{x} \sim 3)^2 + (\ddot{y} \sim 4)^2},\\
&\qquad c = \lambda p.((\ddot{d} \sim \sqrt{(\ddot{x} \sim 3)^2 + (\ddot{y} \sim 4)^2}) < \mathbf{dist}@p)\}\\
&\qquad \{d = \ddot{x} + \ddot{y}\})\\
= \;& \Psi\langle\{\mathbf{xcor}:x, \mathbf{ycor}:y, \mathbf{dist}:d, \mathbf{closer}:c\},\\
&\quad \{\mathbf{xcor}:x, \mathbf{ycor}:y, \mathbf{dist}:d\}\rangle.\\
&\quad \{x = 3, y = 4,\\
&\quad\; d = (\ddot{x} \sim 3) + (\ddot{y} \sim 4),\\
&\quad\; c = \lambda p.((\ddot{d} \sim ((\ddot{x} \sim 3) + (\ddot{y} \sim 4))) < \mathbf{dist}@p)\}
\end{aligned}
$$

The virtual references $\ddot{x}$ and $\ddot{y}$ of the modifier $\{d = \ddot{x} + \ddot{y}\}$ get their denotation from *aCP*, resulting in $\{d = (\ddot{x} \sim 3) + (\ddot{y} \sim 4)\}$. The d-binding of *deltaM* overrides the one in *aCP*. It is also the denotation for the virtual reference $\ddot{d}$ mentioned in the **closer**-attribute of *aCP*. Thus, the provisionally-instantiated virtual reference $\ddot{d} \sim \sqrt{(\ddot{x} \sim 3)^2 + (\ddot{y} \sim 4)^2}$ is replaced by $\ddot{d} \sim ((\ddot{x} \sim 3) + (\ddot{y} \sim 4))$. From the standpoint of inheritance, the virtual references $\ddot{x}$ and $\ddot{y}$ of *deltaM* inherit their denotation from the object *aCP*. Furthermore, the virtual reference $\ddot{d}$ of *aCP* inherits the latest denotation provided by the modifier *deltaM*.

Alternatively, the linking relations of the new object *aMP* can be expressed as follows:

$$
\begin{aligned}
aMP \;=\;& \mathbf{moverride}\; aCP \;\; deltaM\\
=\;& \Psi\langle\{\mathbf{xcor}:x, \mathbf{ycor}:y, \mathbf{dist}:d, \mathbf{closer}:c\},\\
&\quad \{\mathbf{xcor}:x, \mathbf{ycor}:y, \mathbf{dist}:d\}\rangle.\\
&\quad (\mathit{moverride}\; (\mathit{rec}\; \{x = 3, y = 4,\\
&\qquad\qquad d = \sqrt{\ddot{x}^2 + \ddot{y}^2},\\
&\qquad\qquad c = \lambda p.(\ddot{d} < \mathbf{dist}@p)\})\\
&\qquad \{d = \ddot{x} + \ddot{y}\})\\
=\;& \Psi\langle\{\mathbf{xcor}:x', \mathbf{ycor}:y', \mathbf{dist}:d', \mathbf{closer}:c'\},
\end{aligned}
$$

$$\begin{aligned} &\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d\}\rangle. \\ &(\mathit{moverride}\ (\mathit{rec}\ \{x' = 3, y' = 4, \\ &\qquad\qquad z = \sqrt{\ddot{x}[x']^2 + \ddot{y}[y']^2}, \\ &\qquad\qquad c' = \lambda p.(\ddot{d}[d'] < \mathbf{dist}@p)\}) \\ &\qquad \{d' = \ddot{x}[x'] + \ddot{y}[y']\}) \end{aligned}$$

Notice that the overridden defining variable of the **dist**-attribute of *aCP* is given a fresh name z that is no longer specified in the export interface; hence it becomes a hidden (private) attribute of *aMP*. Meanwhile, the latest link of the virtual reference $\ddot{d}$ is updated to the defining variable d' supplied by the modifier *deltaM*, thus inheriting the new **dist**-attribute.

So, similar to C++ [26], but unlike Smalltalk [30], the objects *aCP* and *aMP* show that virtual references need not rely on some pseudo-variable such as **self**. Their behavior can be more closely modeled as relinkable variable references whose linking relations can be kept up to date in each incremental object construction stage via simple variable capture.

8.1.3 Object Self-Reference

We have shown above that intra-object virtual attribute references can be modeled with relinkables. In this section we tackle another form of self-reference needed in an object system, namely, the capability for an object to refer to itself.

Let the necessary modification for moving a two-dimensional point by the amount specified by the parameters x' and y' be defined as follows:

$$\begin{aligned} \mathit{deltaP} \;=\; & \lambda x'y'.\Psi\langle\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y\}, \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y\}\rangle. \\ & \{x = x + x', y = y + y'\} \end{aligned}$$

Then, the following module combination operation moves the Manhattan point object *aMP* from $(3, 4)$ by a vector of $(7, 9)$ to $(10, 13)$:

$$\mathbf{moverride}\ \mathit{aMP}\ (\mathit{deltaP}\ 7\ 9)$$

Ideally, a point object should include a **move**-attribute for moving itself around. Hence, a movable Manhattan point object *aMMP* should look something like:

$$\begin{aligned} aMMP &= \Psi\langle\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d, \mathbf{closer}\!:\!c, \mathbf{move}\!:\!m\}, \\ &\quad\ \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d\}\rangle. \\ &\quad\ \{x = 3, y = 4, d = \ldots, c = \ldots, \\ &\quad\ \ m = \lambda x'y'.(\mathbf{moverride}\ aMMP\ (deltaP\ x'\ y'))\} \end{aligned}$$

Notice that there is a self-reference to *aMMP* in the **move**-attribute. Consequently, to move *aMMP* by a vector of $(7, 9)$, we can simply select its **move**-attribute using **move**@*aMMP* and then apply it to 7 and 9:

$$\begin{aligned} bMMP &= \mathbf{move@}aMMP\ 7\ 9 \\ &= (\lambda x'y'.(\mathbf{moverride}\ aMMP\ (deltaP\ x'\ y')))\ 7\ 9 \\ &= \mathbf{moverride}\ aMMP\ (deltaP\ 7\ 9) \\ &= \Psi\langle\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d, \mathbf{closer}\!:\!c, \mathbf{move}\!:\!m\}, \\ &\quad\ \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d\}\rangle. \\ &\quad\ \{x = 10, y = 13, d = \ldots, c = \ldots, \\ &\quad\ \ m = \lambda x'y'.(\mathbf{moverride}\ aMMP\ (deltaP\ x'\ y'))\} \end{aligned}$$

There is just one problem with our definition of the **move**-attribute above. It does not respond to future modifications. In particular, the **move**-attribute of *bMMP* refers to *aMMP*. A future move from *bMMP* therefore starts at $(3, 4)$, not $(10, 13)$!

A correct definition of *aMMP* is to make the necessary self-reference a virtual reference so that its link changes accordingly as the point moves:

$$\begin{aligned} aMMP &= \Psi\langle\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d, \mathbf{closer}\!:\!c, \mathbf{move}\!:\!m\}, \\ &\quad\ \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d, \mathbf{self}\!:\!s\}\rangle. \\ &\quad\ (priv\ \{s' = aMMP\} \\ &\qquad\quad \{x = 3, y = 4, d = \ldots, c = \ldots, \\ &\qquad\quad\ \ m = \lambda x'y'.(fix\mathbf{self}\ (\mathbf{moverride}\ \ddot{s}[s']\ (deltaP\ x'\ y')))\}) \end{aligned}$$

The relinking of the **self**-attribute is accomplished using the following operator

$$\textit{fix}\mathbf{self} \;\equiv\; \lambda x.(\textit{fix}\ \lambda y.(\mathbf{priv}\ \Psi\langle\{\mathbf{self}\!:s\},\{\}\rangle.\{s=y\}\ x))$$

The behavior of *fix***self** when it is applied to an object $\Psi\langle\varepsilon,\rho\rangle.d$ with a virtual **self**-attribute is as follows:

$$\begin{aligned}
S &= \textit{fix}\mathbf{self}\ \Psi\langle\varepsilon,\rho\rangle.d\\
&= \textit{fix}\ \lambda y.(\mathbf{priv}\ \Psi\langle\{\mathbf{self}\!:s\},\{\}\rangle.\{s=y\}\ \Psi\langle\varepsilon,\rho\rangle.d)\\
&= \textit{fix}\ \lambda y.\Psi\langle\varepsilon,\rho\rangle.(\textit{priv}\ \{s'=y\}\ d)\\
&\qquad [\mathbf{self}\!:s'\in\rho \text{ or else } s' \text{ is fresh to } d]\\
&= \Psi\langle\varepsilon,\rho\rangle.(\textit{priv}\ \{s'=S\}\ d)
\end{aligned}$$

Thus, any virtual reference to **self** in the definition d denotes the object S.

Now, the new object *bMMP* refers to itself instead of *aMMP*:

$$\begin{aligned}
\textit{bMMP} &= \mathbf{move}@\textit{aMMP}\ 7\ 9\\
&= \textit{fix}\mathbf{self}\ (\mathbf{moverride}\ \textit{aMMP}\ (\textit{deltaP}\ 7\ 9))\\
&= \textit{fix}\mathbf{self}\ \Psi\langle\{\mathbf{xcor}\!:x,\mathbf{ycor}\!:y,\mathbf{dist}\!:d,\mathbf{closer}\!:c,\mathbf{move}\!:m\},\\
&\qquad\qquad \{\mathbf{xcor}\!:x,\mathbf{ycor}\!:y,\mathbf{dist}\!:d,\mathbf{self}\!:s\}\rangle.\\
&\qquad\quad (\textit{priv}\ \{s'=\textit{aMMP}\}\\
&\qquad\qquad\quad \{x=10,y=13,d=\ldots,c=\ldots,\\
&\qquad\qquad\quad\ m=\lambda x'y'.(\textit{fix}\mathbf{self}\ (\mathbf{moverride}\ \ddot{s}[s']\ (\textit{deltaP}\ x'\ y')))\})\\
&= \Psi\langle\{\mathbf{xcor}\!:x,\mathbf{ycor}\!:y,\mathbf{dist}\!:d,\mathbf{closer}\!:c,\mathbf{move}\!:m\},\\
&\qquad \{\mathbf{xcor}\!:x,\mathbf{ycor}\!:y,\mathbf{dist}\!:d,\mathbf{self}\!:s\}\rangle.\\
&\qquad (\textit{priv}\ \{s''=\textit{bMMP}\}\\
&\qquad\quad (\textit{priv}\ \{s'=\textit{aMMP}\}\\
&\qquad\qquad \{x=10,y=13,d=\ldots,c=\ldots,\\
&\qquad\qquad\ m=\lambda x'y'.(\textit{fix}\mathbf{self}\ (\mathbf{moverride}\ \ddot{s}[s'']\ (\textit{deltaP}\ x'\ y')))\}))\\
&= \Psi\langle\{\mathbf{xcor}\!:x,\mathbf{ycor}\!:y,\mathbf{dist}\!:d,\mathbf{closer}\!:c,\mathbf{move}\!:m\},
\end{aligned}$$

$$\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d, \mathbf{self}\!:\!s\}\rangle.$$
$$\{x = 10, y = 13, d = \ldots, c = \ldots,$$
$$m = \lambda x' y'.(\mathit{fix}\,\mathbf{self}\ (\mathbf{moverride}\ (\ddot{s} \sim bMMP)\ (\mathit{deltaP}\ x'\ y')))\}$$

The virtual reference $\ddot{s}[s']$ bound to the private defining variable s' of $aMMP$ has been relinked to the private defining variable s'' of $bMMP$, hence $\ddot{s}[s'']$, or equivalently $\ddot{s} \sim bMMP$.

8.1.4 Attribute Selection and Sealing

The notation $\mathbf{x}@e$ selects the denotation of the $\mathbf{x}$-attribute from the object denoted by the term e. Formally,

$$\mathbf{x}@e \quad \equiv \quad \mathbf{load}\ (\mathbf{let}\ e\ \Phi\{\mathbf{x}\!:\!x\}.x)$$

For example,

$$\begin{aligned}
\mathbf{dist}@aMP &= \mathbf{load}\ (\mathbf{let}\ aMP\ \Phi\{\mathbf{dist}\!:\!d\}.d) \\
&= \mathbf{load}\ \Phi\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d\}. \\
&\qquad\qquad (\mathit{let}\ \{\ldots, d = (\ddot{x} \sim 3) + (\ddot{y} \sim 4), \ldots\}\ \mathit{in}\ d) \\
&= \mathbf{load}\ \Phi\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d\}.((\ddot{x} \sim 3) + (\ddot{y} \sim 4)) \\
&= \Delta^1_x(\Delta^1_y(\Delta^1_d((\ddot{x} \sim 3) + (\ddot{y} \sim 4)))) \qquad (8.1) \\
&= 3 + 4 \\
&= 7
\end{aligned}$$

Equation 8.1 finalizes the ρ-parameters x and y mentioned in $(\ddot{x} \sim 3) + (\ddot{y} \sim 4)$, thus making the provisional denotations 3 and 4 the final values of the relinkables $\ddot{x}$ and $\ddot{y}$.

In essence, to accommodate future changes, an object is kept "open" via its virtual references. During the selection of an attribute from an object, the open references are "sealed" off to determine their final values. This sealing process can also be performed at the object level using the following derived object operator:

$$seal \;\equiv\; \mathbf{load} \circ \mathbf{psi}_{\{\}}$$

It finalizes all the import attributes of an object. The module operator $\mathbf{psi}_{\{\}}$ that converts Ψ-abstraction to Φ-abstraction is defined in Section 6.3.3.

For instance, the following operation seals the entire object *aMP* by cementing the links of virtual references to **xcor**, **ycor**, and **dist**:

$$\begin{aligned}
seal\ aMP \;&=\; \mathbf{load}\ (\mathbf{psi}_{\{\}}\ aMP) \\
&=\; \mathbf{load}\ \Phi\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d\}. \\
&\qquad\qquad \Psi\langle\{\mathbf{xcor}\!:\!x', \mathbf{ycor}\!:\!y', \mathbf{dist}\!:\!d', \mathbf{closer}\!:\!c'\}, \{\}\rangle. \\
&\qquad\qquad\quad (moverride\ (rec\ \{x' = 3, y' = 4, \\
&\qquad\qquad\qquad\qquad z = \sqrt{\ddot{x}[x']^2 + \ddot{y}[y']^2}, \\
&\qquad\qquad\qquad\qquad c' = \lambda p.(\ddot{d}[d'] < \mathbf{dist}@p)\}) \\
&\qquad\qquad\qquad \{d' = \ddot{x}[x'] + \ddot{y}[y']\}) \\
&=\; \Psi\langle\{\mathbf{xcor}\!:\!x', \mathbf{ycor}\!:\!y', \mathbf{dist}\!:\!d', \mathbf{closer}\!:\!c'\}, \{\}\rangle. \\
&\qquad (moverride\ (rec\ \{x' = 3, y' = 4, \\
&\qquad\qquad\qquad z = \sqrt{x'^2 + y'^2}, \\
&\qquad\qquad\qquad c' = \lambda p.(d' < \mathbf{dist}@p)\}) \\
&\qquad\quad \{d' = x' + y'\})
\end{aligned}$$

As we can see in the resulting object, the virtual references $\ddot{x}[x']$, $\ddot{y}[y']$, and $\ddot{d}[d']$ are replaced by their respective latest linked constituents x', y', and d'. The sealed object is therefore not sensitive to future changes to the import attributes **xcor**, **ycor**, and **dist**. Indeed, they are no longer part of the import specification of the sealed object.

The attribute sealing operator defined above works from the outside of an object. Such a mechanism can be used to package up an object when its development has been completed. We can also control the sealing process from within an object. The necessary mechanisms are relinkables that can only be relinked a finite number of times. For instance, the following is a Cartesian point object bCP whose **closer**-attribute is sensitive to one revision of **dist** only:

$$\begin{aligned} bCP \;=\; & \Psi\langle\{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d, \mathbf{closer}\!:\!c\}, \\ & \quad \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d\}\rangle. \\ & \quad (rec\ \{x = 3, y = 4, \\ & \qquad\quad d = \sqrt{\ddot{x}^2 + \ddot{y}^2}, \\ & \qquad\quad c = \lambda p.([d^2, d^1] < \mathbf{dist}@p)\}) \end{aligned}$$

The constituent variable reference d^1 of $[d^2, d^1]$ is linked to the defining variable d of bCP. The variable reference d^2 is the only other constituent left unbound. So,

$$\begin{aligned} bMP \;\equiv\; & \mathbf{moverride}\ bCP\ \ deltaM \\ =\; & \Psi\langle\{\mathbf{xcor}\!:\!x', \mathbf{ycor}\!:\!y', \mathbf{dist}\!:\!d', \mathbf{closer}\!:\!c'\}, \\ & \quad \{\mathbf{xcor}\!:\!x, \mathbf{ycor}\!:\!y, \mathbf{dist}\!:\!d\}\rangle. \\ & \quad (moverride\ (rec\ \{x' = 3, y' = 4, \\ & \qquad\qquad z = \sqrt{\ddot{x}[x']^2 + \ddot{y}[y']^2}, \\ & \qquad\qquad c' = \lambda p.(d' < \mathbf{dist}@p)\}) \\ & \qquad\quad \{d' = \ddot{x}[x'] + \ddot{y}[y']\}) \end{aligned}$$

The dominant variable reference d^2 of $[d^2, d^1]$ is now permanently linked to the defining variable d' of the modifier bMP. Consequently, any future changes to the **dist**-attribute of bMP does not have a rippling effect on the **closer**-attribute.

8.1.5 Discussions

To summarize, objects are modules with relinkables. Modules provide the information hiding facilities needed for an object system [64, 65]. Incremental module combination operations such as **moverride** are the means for reusing existing objects to form new objects. This style of inheritance is often referred to as mixin-based [11].

The feature unique to our object system is the incorporation of relinkables to model late binding virtual references. Recall that a relinkable $\ddot{x} \equiv [x^\infty, \ldots, x^1]$ is semantically equivalent to

```
if $x^\infty$ is bound then $x^\infty$
    $\ddots$
        else if $x^1$ is bound then $x^1$
            else $[x^\infty, \ldots, x^1]$ is unbound
```

It is reminiscent of a dispatch according to the boundness of the constituent variable references $x^\infty, \ldots, x^1$. Using the same metaphor, a relinkable $\ddot{x}[y] \equiv [x^\infty, \ldots, x^1, y^1]$ whose least dominating constituent y^1 is known to be bound is therefore semantically equivalent to the following dispatch:

```
if $x^\infty$ is bound then $x^\infty$
    $\ddots$
        else if $x^1$ is bound then $x^1$
            else $y^1$
```

Hence, a relinkable such as $\ddot{x}$ is a good candidate for modeling the late binding behavior required of a virtual variable reference.

In practice, a virtual table is associated with each object to help resolve the object's virtual variable references. Conceptually, when the linking relation of a virtual variable is altered during the construction of a new object, the variable's slot in the virtual table is updated to reflect the necessary change. A prominent example using such a technique to implement late binding virtual references efficiently is C++ [26]. In our system, a slot is associated with each individual virtual variable reference. That is, $\ddot{x}[y]$ models a table entry for the virtual reference $\ddot{x}$ whose current contents is the location of the variable y. Consequently, if the denotation of the variable y is e, then the denotation of the virtual reference $\ddot{x}$ is also e, which is expressed in our system as a provisionally-instantiated relinkable $\ddot{x} \sim e$.

Our semantics of inheritance is the same fixpoint semantics of Kamin [37], Reddy [60], and Cook and Palsberg [22]. In their descriptions of objects, late binding intra-object virtual attribute references are modeled as run-time variable lookups from the environment denoted by some pseudo-variable **self**. Moreover, the same pseudo-variable **self** is overloaded to explain object self-reference as well. In our system, such run-time lookup operations are replaced with compile-time linking operations and the unnecessary overloading is removed from **self**. Clearly, we gain significant advantages in efficiency, as discussed above. But more importantly, we have shown that the essence of object-oriented programming can be explained directly in terms of compilation and linking, instead of indirectly via complicated computational steps. Furthermore, through the sealing of objects and attributes, we have shown that our relinkables can provide a degree of flexibility that has not been fully realized in other object systems. To summarize, modeling virtual references as relinkables is desirable because of the simplicity, clarity, and flexibility they provide.

8.2 Interactive Programming

Lisp [66] and its dialects such as Scheme [4, 21] employ an interactive evaluator as a means for incremental program development. Conceptually, an interactive evaluator uses an ever-growing interactive environment to keep track of the results produced by previously evaluated **define**-expressions. Each **define**-expression submitted to the evaluator is evaluated in the scope of the current interactive environment to yield a binding. The binding is then used to extend or override the existing interactive environment to yield a new environment for the evaluation of the next expression. The interactive environment thus constitutes an ever-expanding program in which mutually dependent bindings are added in an incremental fashion.

We assume that when the interactive evaluator is ready to accept the next expression, it issues a prompt (>) preceded by a number that is added for reference purposes. Here is a sample interactive programming session using our notation:

```
1>   (define foo λn.(if (bar n) then 23 else 45))
2>   (define bar λn.T)
3>   (foo 3)
     23
4>   (define bar λn.F)
5>   (foo 3)
     45
```

Below is a description of what happens to each of the expressions (the expressions issued to the evaluator are typeset in boldface to highlight the fact that they are source code to the evaluator):

1> The interactive environment is extended with a function **foo** that depends on the function **bar** to decide between returning 23 or 45.

2> The interactive environment is extended with the definition of another function **bar** that always returns the boolean value T .

3> The function **foo** is invoked. Since the latest denotation of **bar** in the interactive environment returns T, the function invocation (**foo 3**) yields 23, which the evaluator prints below the prompt.

4> The function **bar** is redefined to always return the boolean value false instead. The new definition becomes the latest denotation of **bar** in the interactive environment.

5> The redefinition of **bar** is seen by **foo**. Hence, invoking **foo** the second time yields a different result of 45.

The interactive session above demonstrates the fact that the function **foo** is responsive to the changes to the definition of **bar**. Traditionally, the adaptive behavior of the reference **bar** used in **foo** is explained in terms of side effects [21]. That is, during (or before) the evaluation of the definition of the function **foo**, a cell is allocated in the interactive environment for the function **bar**. When **bar** is defined or redefined, its latest denotation is deposited into the cell. Hence, each time the function **foo** is invoked, it has access to the up-to-date denotation of **bar**.

Here, we show that the adaptive behavior associated with interactive programming can also be modeled using relinkables. Indeed, the style of interactive programming shown above closely resembles the style of object-oriented programming described in the last section. An interactive environment is an object that exports the previously evaluated **define**-expressions. A **define**-expression is evaluated in the scope of the current interactive environment to yield an object modifier that the evaluator then uses to transform the current interactive environment to a new environment object. An input expression that is not a **define**-expression is evaluated in the scope of the current interactive environment to a value that is then displayed by the evaluator. We elaborate on the evaluation of these two categories of expression below.

8.2.1 Evaluating Define Expressions

To evaluate an input expression (**define x e**) in an interactive environment γ to yield a new interactive environment γ', the input **e** is first translated into the compiled code

$$A \quad = \quad \Phi\{\bar{\mathbf{x}} : \bar{x}\}.e$$

where $\bar{\mathbf{x}}$ are the free identifiers of **e**, $\bar{x} \equiv q(\bar{\mathbf{x}})$ are the unique variable names assigned to $\bar{\mathbf{x}}$ by some one-to-one function q, and e is the image of **e** except that each occurrence of the free identifier $\mathbf{x}_i$ in **e** is compiled into a virtual reference $\ddot{x}_i$ in e. The new interactive environment γ' is then constructed as follows:

$$\gamma' \quad = \quad G\ \gamma\ \mathit{delta}\gamma$$

where the object constructor G and the object modifier $\mathit{delta}\gamma$ are defined as

$$\begin{aligned} G &= \lambda xy.(\mathbf{override}\ (\mathbf{priv}\ y\ x)\ y), \\ \mathit{delta}\gamma &= \mathbf{dfn}_{\mathbf{x}}\ (\mathbf{app}\ \Phi\{\}.\mathit{fix}\ (\mathbf{lam}_{\mathbf{x}}\ (\triangleright\ (\mathbf{let}\ \gamma\ A)))) \end{aligned}$$

Let the interactive environment γ be the object $\Psi\langle\varepsilon, \rho\rangle.d$. Then, the module importation operation

$$\begin{aligned}\textbf{let}\ \gamma\ A &= \textbf{let}\ \Psi\langle\varepsilon,\rho\rangle.d\ \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e \\ &= \Phi\rho\uplus\{\bar{\mathbf{x}}\!:\!\bar{x}\}.(\mathit{let}\ d\ \mathit{in}\ e) \\ &= \Phi\rho'.e'\end{aligned}$$

links the free identifiers of the compiled code A of $\mathbf{e}$ with the bindings of γ. The resulting compiled code $\Phi\rho'.e'$ is then passed to the evaluation function $\rhd$, which models the evaluation strategy employed by Lisp and Scheme. It optimizes the body term e' to a value term v:

$$\begin{aligned}\rhd\ (\textbf{let}\ \gamma\ A) &= \rhd\ \Phi\rho'.e' \\ &= \Phi\rho'.v\end{aligned}$$

Notice that the virtual references in the value term v are still sensitive to future updates. The compiled value $\Phi\rho'.v$ is then made recursively available to itself:

$$\begin{aligned}&\textbf{app}\ \Phi\{\}.\mathit{fix}\ (\textbf{lam}_{\mathbf{X}}\ (\rhd\ (\textbf{let}\ \gamma\ A))) \\ &= \textbf{app}\ \Phi\{\}.\mathit{fix}\ (\textbf{lam}_{\mathbf{X}}\ \Phi\rho'.v) \\ &= \textbf{app}\ \Phi\{\}.\mathit{fix}\ \Phi\rho'.\lambda x.v \\ &\qquad [\mathbf{x}\!:\!x \in \rho' \text{ or else } x \text{ is fresh to } v] \\ &= \Phi\rho'.(\mathit{fix}\ \lambda x.v)\end{aligned}$$

Next, the recursively defined compiled value $\Phi\rho'.(\mathit{fix}\ \lambda x.v)$ is converted into a module $\mathit{delta}\gamma$ to be used as an environment modifier:

$$\begin{aligned}\mathit{delta}\gamma &= \textbf{dfn}_{\mathbf{X}}\ (\textbf{app}\ \Phi\{\}.\mathit{fix}\ (\textbf{lam}_{\mathbf{X}}\ (\rhd\ (\textbf{let}\ \gamma\ A)))) \\ &= \textbf{dfn}_{\mathbf{X}}\ \Phi\rho'.(\mathit{fix}\ \lambda x.v) \\ &= \Psi\langle\{\mathbf{x}\!:\!x\}, \rho'\rangle.\{x = (\mathit{fix}\ \lambda x.v)\}\end{aligned}$$

Finally, using G, we can construct the new interactive environment γ' by modifying the current environment object γ as follows:

$$\gamma' = G\ \gamma\ \mathit{delta}\gamma$$

$$\begin{aligned}
&= \textbf{override}\ (\textbf{priv}\ \mathit{delta}\gamma\ \gamma)\ \mathit{delta}\gamma \\
&= \Psi\langle \varepsilon \uplus \{\mathbf{x}\!:\!x\}, \rho \uplus \rho' \rangle.(\mathit{override}\ (\mathit{priv}\ \{x = (\mathit{fix}\ \lambda x.v)\}\ d)\ \{x = (\mathit{fix}\ \lambda x.v)\}) \\
&= \Psi\langle \varepsilon \uplus \{\mathbf{x}\!:\!x\}, \rho \uplus \rho' \rangle.(\mathit{override}\ [(\mathit{fix}\ \lambda x.v)/x]d\ \{x = (\mathit{fix}\ \lambda x.v)\})
\end{aligned}$$

So, the new **x**-binding provided by the modifier $\mathit{delta}\gamma$ overrides any existing **x**-binding of γ. Moreover, virtual **x**-references in γ are relinked to the new **x**-binding.

To illustrate, let the interactive environment after evaluating the first expression

1> (**define foo** $\boldsymbol{\lambda}$**n**.(**if** (**bar n**) **then 23 else 45**))

be the object

$$\gamma \ = \ \Psi\langle\{\mathbf{foo}\!:\!f\},\{\mathbf{bar}\!:\!x\}\rangle.\{f = \lambda n.(\mathit{if}\ (\ddot{x}\ n)\ \mathit{then}\ 23\ \mathit{else}\ 45)\}$$

It exports the function **foo** and has a virtual **bar**-reference. The compiled code of the denotation term of the second expression

2> (**define bar** $\boldsymbol{\lambda}$**n**.**T**)

is $\Phi\{\}.\lambda n.T$. The resulting environment modifier $\mathit{delta}\gamma$ is therefore

$$\begin{aligned}
\mathit{delta}\gamma &= \mathbf{dfn_{bar}}\ (\mathbf{app}\ \Phi\{\}.\mathit{fix}\ (\mathbf{lam_{bar}}\ (\triangleright\ \underline{(\mathbf{let}\ \gamma\ \Phi\{\}.\lambda n.T)}))) \\
&= \mathbf{dfn_{bar}}\ (\mathbf{app}\ \Phi\{\}.\mathit{fix}\ (\mathbf{lam_{bar}}\ \underline{(\triangleright\ \Phi\{\}.\lambda n.T)})) \\
&= \mathbf{dfn_{bar}}\ \underline{(\mathbf{app}\ \Phi\{\}.\mathit{fix}\ (\mathbf{lam_{bar}}\ \Phi\{\}.\lambda n.T))} \\
&= \mathbf{dfn_{bar}}\ \Phi\{\}.\underline{(\mathit{fix}\ \lambda x.\lambda n.T)} \\
&= \underline{\mathbf{dfn_{bar}}\ \Phi\{\}.\lambda n.T} \\
&= \Psi\langle\{\mathbf{bar}\!:\!x\},\{\}\rangle.\{x = \lambda n.T\}
\end{aligned}$$

Thus, the interactive environment γ' after the evaluation of the second expression is:

$$\begin{aligned}
\gamma' &= G\ \gamma\ \mathit{delta}\gamma \\
&= \textbf{override}\ (\textbf{priv}\ \mathit{delta}\gamma\ \gamma)\ \mathit{delta}\gamma \\
&= \Psi\langle\{\mathbf{bar}\!:\!x, \mathbf{foo}\!:\!f\},\{\mathbf{bar}\!:\!x\}\rangle.
\end{aligned}$$

$$
\begin{aligned}
& (\mathit{override}\ \underline{\mathit{priv}\ \{x = \lambda n.T\}\ \{f = \lambda n.(\mathit{if}\ (\ddot{x}\ n)\ \mathit{then}\ 23\ \mathit{else}\ 45)\}} \\
& \qquad \{x = \lambda n.T\}) \\
= {} & \Psi\langle\{\mathbf{bar}\!:\!x, \mathbf{foo}\!:\!f\}, \{\mathbf{bar}\!:\!x\}\rangle. \\
& (\underline{\mathit{override}\ \{f = \lambda n.(\mathit{if}\ ((\ddot{x} \sim \lambda n.T)\ n)\ \mathit{then}\ 23\ \mathit{else}\ 45)\}} \\
& \qquad \underline{\{x = \lambda n.T\}}) \\
= {} & \Psi\langle\{\mathbf{bar}\!:\!x, \mathbf{foo}\!:\!f\}, \{\mathbf{bar}\!:\!x\}\rangle. \\
& \{f = \lambda n.(\mathit{if}\ ((\ddot{x} \sim \lambda n.T)\ n)\ \mathit{then}\ 23\ \mathit{else}\ 45), \\
& \ x = \lambda n.T\}
\end{aligned}
$$

It exports both **bar** and **foo**. Furthermore, the free reference to **bar** in **foo** has the latest denotation of **bar** as its provisional meaning.

8.2.2 Evaluating Non-Define Expressions

The evaluation of an input expression **e** that is not a **define**-expression has a much simpler process. As before, the input **e** is first translated into the compiled code

$$A \quad = \quad \Phi\{\bar{\mathbf{x}}\!:\!\bar{x}\}.e$$

The compiled code is then linked with the current environment γ and the resulting compiled code is evaluated to yield the value of the expression **e** under γ:

$$\mathbf{load}\ (\mathbf{let}\ \gamma\ A)$$

Hence, evaluating the third expression

3> (**foo 3**)

under γ' results in:

$$
\begin{aligned}
& \mathbf{load}\ \underline{(\mathbf{let}\ \gamma'\ \Phi\{\mathbf{foo}\!:\!f\}.(\ddot{f}\ 3))} \\
= {} & \underline{\mathbf{load}\ \Phi\{\mathbf{foo}\!:\!f, \mathbf{bar}\!:\!x\}.((\ddot{f} \sim \lambda n.(\mathit{if}\ ((\ddot{x} \sim \lambda n.T)\ n)\ \mathit{then}\ 23\ \mathit{else}\ 45))\ 3)} \\
= {} & \Delta^1_x(\underline{\Delta^1_f((\ddot{f} \sim \lambda n.(\mathit{if}\ ((\ddot{x} \sim \lambda n.T)\ n)\ \mathit{then}\ 23\ \mathit{else}\ 45))\ 3)})
\end{aligned}
$$

$$
\begin{aligned}
&= \underline{\Delta^1_{\ddot{x}}((\lambda n.(\mathit{if}\ ((\ddot{x} \sim \lambda n.T)\ n)\ \mathit{then}\ 23\ \mathit{else}\ 45))\ 3)} \\
&= \underline{(\lambda n.(\mathit{if}\ ((\lambda n.T)\ n)\ \mathit{then}\ 23\ \mathit{else}\ 45))\ 3} \\
&= 23
\end{aligned}
$$

In conclusion, disregarding input/output aspects, we have shown that interactive programming in the style advocated by Lisp is a form of object-oriented programming. It is modular programming with the addition of the incremental linking of late binding variable references.

Chapter 9

Finale

After showing the practicality of our incremental program construction capability enhancing schema, it is now time to summarize the results of this research, to compare with other work, and to project directions for future work.

9.1 Results

Our goal is to integrate incremental program construction capabilities into programming systems. At issue are linking mechanisms flexible enough to compensate for the limitations imposed by statically-scoped variables and yet well-behaved enough to uphold all the nice properties of static scope. The basis of our work is the notion of contexts, particularly the name capturing feature of context hole filling. By perceiving fully-evolved contexts (proper parse trees) as compiled code, partially-evolved contexts (improper parse trees with non-terminal leaves) as compilation operators, and context hole filling as composition or application of such compilation operators, our approach culminates into a schema for conservatively extending programming languages with mechanisms capable of modeling the incremental construction, linking, and loading of compiled program components.

We apply our context-enriching schema to the pure λ-calculus to demonstrate its basic mechanics. In addition to the machine code modeled by λ-terms, the schema introduces free identifier abstractions to model the compiled code of fully-evolved λ-contexts. Also introduced by our schema are operators for constructing the compiled code of composite fully-evolved λ-contexts from the compiled code of their immediate sub-contexts. Unlike ordinary λ-abstractions, the new compiled code abstractions have quasi-statically-scoped variable references serving as temporary placeholders for yet to be linked free variables. Such free variable references are subject to capture by statically-scoped λ-parameters in the construction of new compiled code.

The context-enriched pure λ-calculus $\boldsymbol{\lambda}$**CC** is expressive enough to encode symbol-related incremental programming mechanisms such as first-class environments. A compositional and metacircular compiler for $\boldsymbol{\lambda}$**CC**-programs is $\boldsymbol{\lambda}$**CC**-definable. If dealing with source code is undesirable, a linker for integrating compiled $\boldsymbol{\lambda}$**CC**-programs is also $\boldsymbol{\lambda}$**CC**-definable.

To further illustrate the power of our schema, we extend the pure λ-calculus with notations for expressing some frequently used programming idioms about variable definition. The semantics of the new variable defining notations can be explained in terms of simple syntactic expansions. So, there is minimal gain in expressiveness by introducing these additional notations into the pure λ-calculus. Their potentials are unleashed once they are enriched with the incremental program construction capabilities induced by the notion of contexts, however.

The abstractions introduced by our schema to model the compiled code of fully-evolved variable definition contexts are first-class modules that import through their free variables and export via their defining variables. The incremental compiled code construction operators derived from our context-enriching schema provide the capabilities to import from modules, to construct modules from scratch, and to combine and link existing modules to form new modules. The context-enriched variable definition calculus $\boldsymbol{\lambda}$**DD** is a conservative extension of the context-enriched pure λ-calculus $\boldsymbol{\lambda}$**CC**. Whereas $\boldsymbol{\lambda}$**CC**, with its basic incremental compiled code constructors, facilitates incremental program construction in the small, $\boldsymbol{\lambda}$**DD**, with its additional module mechanisms, is also well-suited for incremental program construction in the large.

The extension of $\boldsymbol{\lambda}$**CC** to $\boldsymbol{\lambda}$**DD** demonstrates one important aspect of our context-enriching schema, namely, it is a modular language design methodology. New compilation mechanisms are introduced by the $\boldsymbol{\lambda}$**DD**-calculus to deal with the new variable defining notations added to the pure λ-calculus; the semantics of the existing compilation mechanisms of $\boldsymbol{\lambda}$**CC** is unchanged. This is the case because our schema clearly distinguishes compiled code from machine code. Moreover, the compiled code constructors use only α-conversion for linking purposes, no computational β-steps are necessary. As a result, the operators specify how programs are constructed without involving any implementation dependent tricks. It is thus possible to add new incremental compilation operators into our context-enriched calculi in a modular fashion.

As yet another demonstration of our schema, we apply it to an extension of the pure λ-calculus with relinkable variable referencing devices. A relinkable variable reference is a variable reference that can be bound by different defining variables as its surrounding context grows. It thus possesses the potential to act as a late binding variable reference. At the machine code level, relinkable variable references behave no differently than statically-scoped variables since their context has already been fixed. Adding them therefore does not greatly enhance the expressiveness of the pure λ-calculus. In the context-enriched calculus of relinkable variable references $\boldsymbol{\lambda}$**RR**, evolving contexts are a meaningful and programmable notion. Relinkable variable references occurring in compiled code abstractions therefore mimic late binding free variables whose linking relation is subject to change as their context evolves.

The context-enriched λ-calculi $\boldsymbol{\lambda}$**DD** and $\boldsymbol{\lambda}$**RR** are combined to express two of the most common forms of incremental programming in practice, namely, object-oriented programming and interactive programming. Objects are modules. Object encapsulation is module encapsulation. Object inheritance is module combination. Late binding virtual references are relinkable variable references. Their linking relation is resolved directly without the help of some pseudo-variable such as **self**.

The environment used by an interactive evaluator is a module. It exports previously evaluated **define**-expressions and imports free variables mentioned in the evaluated **define**-expressions. The free variables are relinkable variable references. They provide the needed sensibility for existing **define**-expressions to respond to future **define**-expressions. Evaluating a new **define**-expression overrides or extends the existing interactive environment to yield a new environment.

In summary, programming language design is about the abstraction of recurring programming idioms. The expressiveness of a programming language is tied to the idioms it can fluently express. In this sense, our context-enriched λ-calculi are highly desirable:

- They are expressive enough to cover the basic notions underlying many prominent incremental programming paradigms.
- Their linking mechanisms are explained in terms of code copying and variable renaming, thus corresponding nicely to the two source code editing functions most often used if incremental program construction were done by hand.
- Their incremental program construction mechanisms are orthogonal to the computational devices employed by λ-calculi.
- They are derived from a modular language design methodology.

9.2 Related Work

We relate our context-enriched calculi to other extensions of the λ-calculus with name-based programming mechanisms. None of them explicitly supports incremental code construction. To simplify the discussion, we take the liberty of altering their syntax in the style of our context calculi.

9.2.1 Denotational Semantics of Lambda Calculus

The direct style denotational semantics of the λ-calculus [67] shown in Figure 9.1 constitutes a view of incremental program construction involving computational β-steps and environment lookups. An environment r maps the syntactic representation

$$\begin{array}{rcl} \mathcal{E}\ [\![x]\!] & = & \lambda r.(r\ [\![x]\!]) \\ \mathcal{E}\ [\![\lambda x.e]\!] & = & (\mathcal{L}\ [\![x]\!])\ (\mathcal{E}\ [\![e]\!]) \\ \mathcal{E}\ [\![e_1\ e_2]\!] & = & \mathcal{A}\ (\mathcal{E}\ [\![e_1]\!])\ (\mathcal{E}\ [\![e_2]\!]) \end{array} \qquad \text{with} \qquad \begin{array}{rcl} \mathcal{L} & = & \lambda s.\lambda f.\lambda r.\lambda x.(f\ r[s \mapsto x]) \\ \mathcal{A} & = & \lambda f_1.\lambda f_2.\lambda r.((f_1\ r)\ (f_2\ r)) \end{array}$$

Figure 9.1: Denotational semantics of λ-calculus

$[\![x]\!]$ of variables x to their denotation. The notation $r[[\![x]\!] \mapsto e]$ denotes an extension of the environment r such that an environment lookup $r[[\![x]\!] \mapsto e]\ [\![y]\!]$ is e if x and y are the same variable, but is otherwise the result of looking up $[\![y]\!]$ in the environment r, $r\ [\![y]\!]$. The meaning $\mathcal{E}\ [\![e]\!]$ of (the syntax $[\![e]\!]$ of) a λ-term e is usually defined in an abstract mathematical model of the λ-calculus known as domains [67]. Here, in contrast to common practice, we elect to remain metacircular; that is, $\mathcal{E}\ [\![e]\!]$ is merely another λ-term. We can then prove the following in the λ-calculus with a structural induction on e:

Theorem 9.1 *Let $x_1, \ldots, x_n$ be the free variables of e. Then,*

$$\mathcal{E}\ [\![e]\!] \twoheadrightarrow \lambda r.[(r\ [\![x_n]\!])/x_n] \cdots [(r\ [\![x_1]\!])/x_1]e$$

That is, the meaning of the syntactic representation $[\![e]\!]$ of e is the λ-term e parameterized over an environment r and that every occurrence of each free variable x_i in e is replaced by an environment lookup $r\ [\![x_i]\!]$.

A λ-term of the form $\lambda r.[(r\ [\![x_n]\!])/x_n] \cdots [(r\ [\![x_1]\!])/x_1]e$ is reminiscent of a free identifier abstraction $\Phi\{\mathbf{x}_1 \!:\! x_1, \ldots, \mathbf{x}_n \!:\! x_n\}.e$ when the syntactic representation $[\![x_i]\!]$ of variable x_i is viewed as a realization of identifier $\mathbf{x}_i$. Once we recognize the connection, the deliberately filtered out meaning functions $\mathcal{A}$ and $(\mathcal{L}\ [\![x]\!])$ of Figure 9.1 are the compiled code construction operators **app** and $\mathbf{lam}_\mathbf{X}$ of our context-enriched λ-calculi.

The advantages of defining incremental program construction based on the notion of environment lookup are:

- There is no need for the special free identifier abstraction mechanisms. Reasoning can be done purely in the λ-calculus.
- It is straightforward to encode a prototype implementation with a suitable representation of environments.

The advantages of thinking incremental program construction in terms of free identifier abstractions are:

- There are no administrative β-reductions involved in the construction and application of environments to contend with. Instead, we deal with variable renaming, which is much simpler. More importantly, the latter is conceptually closer to what happens in a linker.
- By recognizing free identifier abstractions as a distinct data type and studying their behavior as such, it is easier to incorporate more free identifier abstraction operators in the future in a manner that would have been less than intuitive with the environment-based semantics.
- We can capture the essence of free variables without worrying about implementation details. Reasoning with free identifier abstractions is more natural than reasoning with environments precisely because they separate specification issues from implementation issues concerning free variables, a fundamental requirement for any useful abstraction mechanism.

9.2.2 Lambda Calculus with Names

Dami's λ-calculus with names λN [24] extends the λ-calculus with keyword parameters to facilitate program extensibility. The syntax of λN using our notation is:

$$\begin{array}{rcl} e & ::= & x \mid \lambda\rho.e \mid e(\mathbf{x} \mathord{\Rightarrow} e) \mid e! \mid \mathbf{err} \\ \rho & ::= & \{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\} \end{array}$$

A λN-abstraction $\lambda\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e$ has each parameter x_i known externally via the name (keyword or identifier) $\mathbf{x}_i$. The λN-calculus breaks the β-reduction rule into two independent reduction rules called bind and close. A *bind* reduction is

$$(\lambda\rho.e_1)(\mathbf{x} \Rightarrow e_2) \quad \rightarrow \quad \lambda\rho.[e_2/x]e_1$$

where either $\mathbf{x} : x \in \rho$ or else x is fresh. It binds the $\mathbf{x}$-parameter of the abstraction $\lambda\rho.e_1$ to the denotation e_2 without invoking the abstraction. A *close* reduction is

$$(\lambda\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e)! \quad \rightarrow \quad [\mathbf{err}/x_1] \cdots [\mathbf{err}/x_n]e$$

It corresponds to the invocation of the abstraction $\lambda\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e$. Any unbound parameter is given the error value $\mathbf{err}$.

The λN-calculus closely resembles our $\boldsymbol{\lambda}\mathbf{C}$-calculus. Indeed, we can easily embed λN into $\boldsymbol{\lambda}\mathbf{C}$ using the translation $\mathcal{C}[\![\ ]\!]$ defined inductively as follows:

$$\begin{aligned}
\mathcal{C}[\![x]\!] &= x \\
\mathcal{C}[\![\lambda\rho.e]\!] &= \Phi\rho.\mathcal{C}[\![e]\!] \\
\mathcal{C}[\![e_1(\mathbf{x} \Rightarrow e_2)]\!] &= \mathbf{app}\ (\mathbf{lam}_{\mathbf{x}}\ \mathcal{C}[\![e_1]\!])\ \Phi\{\}.\mathcal{C}[\![e_2]\!] \\
\mathcal{C}[\![e!]\!] &= \mathbf{load}\ \mathcal{C}[\![e]\!]
\end{aligned}$$

A λN-abstraction $\lambda\rho.e$ is the free identifier abstraction $\Phi\rho.\mathcal{C}[\![e]\!]$. A close operation $(\lambda\rho.e)!$ loads the free identifier abstraction $\Phi\rho.\mathcal{C}[\![e]\!]$ of $\lambda\rho.e$, provided that we identify unlinked identifier indicators $\tilde{\mathbf{x}}$ with the error value $\mathbf{err}$. A bind operation $(\lambda\rho.e_1)(\mathbf{x} \Rightarrow e_2)$ is semantically equivalent to the compiled β-redex $\Phi\rho.((\lambda x.e_1)\ e_2)$ where either $\mathbf{x} : x \in \rho$ or else x is fresh.

Although the two calculi have similar abstraction mechanism, there is a fundamental difference in the way names are utilized. In the λN-calculus, names are used in parameter passing steps. In contrast, in our context-enriched calculi, names are used in incremental program construction steps. The difference has a profound effect on the philosophical view of software composition. In the view of Dami's λN-calculus, as well as other extensions of the λ-calculus with name-based programming mechanisms such as records [19], label-selective λ-calculus [6], transparent data parameters

[36, 40], and quasi-static procedures [42], composing programs is about passing named parameters among program components, a notion tied to the computational behavior of programs. With our context-enriched calculi, program composition is about linking program components together, a notion that is completely independent of the computational behavior of programs.

9.2.3 Jigsaw

Jigsaw [12, 13] is a framework for designing modular programming languages based on the idea that inheritance is an essential linguistic mechanism for module manipulation. A module is an abstract class [46, 68]. It is a mutually recursive scope consisting of definitions and declarations. Definitions bind names to values. Declarations specify pure virtual attribute references. Inheritance covers the notions of code reuse and late binding. Module operators reuse preexisting modules to produce new modules. Module combination operators replace declarations in one module with definitions from other modules, thus achieving late binding.

Jigsaw shares our view that modules and objects are closely related incremental programming concepts that should be supported in a single system. The main difference lies in the way virtual attribute references are resolved. Jigsaw's semantics is based upon a denotational model of inheritance [22, 60] where modules are modeled as record generators, which are functions from records to records, and attribute references are translated into record field selections. Module manipulation operators are then defined as operations on such record generators. These record generator operations are responsible for building records incrementally. They do not resolve virtual attribute references. The linking relation of virtual references are resolved only when modules are instantiated, since the complete picture of a record is only known at that time. In contrast, our system as presented in Chapter 8 truthfully captures the incremental nature of module construction. The linking relation of every attribute reference, whether virtual or not, of every module is determined as soon as the module is defined. If necessary, such linking relation can be adjusted to accommodate the effect of future changes. Our description of incremental programming is more attractive since it has a cleaner semantics for modules and objects. Moreover, the semantic description very closely resembles actual implementations employed by object-oriented languages such as C++ [26].

9.3 Future Work

There are many issues concerning our context-enriching schema left to be explored. First and foremost is the need of an effective implementation for our context-enriched calculi. Efficient implementation should not be the deciding factor of the practicality of our context-enriched calculi, but a stronger case can be presented if we have one. Second, so far we have only shown examples of enriching untyped programming languages. We would like to broaden the applicability of our schema to typed λ-calculi as well. Third, a nagging problem about our free identifier abstractions is the need of a special notion of equivalence ($\approx$) to garbage collect their redundant parameters. It is one of our top priorities to develop a new representation for free identifier abstractions that would automatically rid them of redundant parameters.

9.3.1 Effective Implementation

Devising an efficient implementation for the incremental compilation mechanisms of our context-enriched λ-calculi is a challenging proposition. Intuitively, the compiled code construction operations induced by our schema are run-time operations. It is therefore conceivable that they require run-time code generation [20, 43]. We are inclined to believe that some form of run-time code generation is inevitable when efficiency is a relevant issue.

A general rule of thumb about compilation technologies is that the more contextual information we have about a program, the more static analysis we can perform and therefore the more efficient the compiled program is. This is unfortunately not the case with our view of incremental program construction. For instance, it is easy to determine the binding parameter of a statically-scoped variable reference when we know enough about the context in which it is used. Consequently, such a variable reference can be translated into a simple lexical address relative to the closure representation of some λ-abstraction [17, 27, 62]. Adapting such an implementation technique to our free identifier abstractions becomes a non-trivial task since the context of the free identifier references has yet to be built. It is therefore impossible to translate them into some fixed lexical addresses beforehand. Particularly, what lexical address should we assign for the variable reference x in $\Phi\{\mathbf{x}:x\}.x$ with respect to the closure representation of $\Phi\{\mathbf{x}:x\}.x$? Whatever the lexical address is, it is likely to change in the future. To illustrate, in the process of

$$\mathbf{app}\ \Phi\{\mathbf{x}:x\}.x\ \Phi\{\mathbf{y}:y\}.y \quad \rightarrow \quad \Phi\{\mathbf{x}:x, \mathbf{y}:y\}.(x\ y)$$

either of the two variable references x or y in the resulting term must have its lexical address altered to refer to its new position in the parameter specification $\{\mathbf{x}:x, \mathbf{y}:y\}$. On the other hand, the problem vanishes when lexical addresses are assigned only when a free identifier abstraction is about to be loaded because the complete contextual information about any free identifier reference is fixed at that time.

9.3.2 Enriching Typed Lambda Calculi

Another rich avenue of future research is to adapt our incremental program construction capability enhancing schema to typed λ-calculi. It means that our incremental compiled code construction operations must also perform type checking (or type inferencing) duties. Since there is a diverse array of type systems [18, 50], it is a major undertaking to study the various typing disciplines they impose on our incremental compiled code constructors.

An extensive exploration of the typing of our context-enriched calculi is at the top of our agenda. There is an abundance of results on types, and subtypes in particular, that we can draw upon. Furthermore, since some of the crucial notions underlying object-oriented programming are shared by our context-enriched calculi, it is tempting to investigate if our incremental program construction mechanisms simplify or complicate typing issues of object-oriented programming languages [2, 32, 51, 57].

9.3.3 Garbage Collecting Redundant Parameters

In the context-enriched λ-calculi we rely on the meta-notion of indistinguishability ($\approx$) to remove redundant parameters from free identifier abstractions. We are wondering if such redundant parameters can be removed without resorting to indistinguishability.

A positive answer seems reachable. The reason we need indistinguishability is that the reduction of the body term e of a free identifier abstraction $\Phi\rho.e$ might remove references to some parameters specified in ρ, thus making them redundant. One way to automatically garbage collect such redundant parameters is to do without the ρ-specification altogether and use a new category of free identifier references $\mathbf{x}^k$ that are bound by the kth nearest enclosing Φ-abstraction. Formally, we are expressing

$$\Phi\{\mathbf{x}_1 : x_1, \ldots, \mathbf{x}_n : x_n\}.e$$

as

$$\Phi.\langle \mathbf{x}_1^1 / x_1 \rangle \cdots \langle \mathbf{x}_n^1 / x_n \rangle e$$

by renaming every occurrence of x_i in e with $\mathbf{x}_i^1$. For instance,

$$\Phi\{\mathbf{x} : x, \mathbf{y} : y\}.\Phi\{\mathbf{x} : x', \mathbf{z} : z\}.((\lambda z.(x'\ y))\ x)$$

becomes

$$\Phi.\Phi.\underline{((\lambda z.(\mathbf{x}^1\ \mathbf{y}^2))\ \mathbf{x}^2)}$$

Notice that the redundant parameter $\mathbf{z} : z$ has disappeared. Furthermore, when we contract the underlined β-redex to yield

$$\Phi.\Phi.(\mathbf{x}^1\ \mathbf{y}^2)$$

the redundant parameter $\mathbf{x} : x$ is eliminated automatically.

With the new syntactic notation $\Phi.e$ for free identifier abstractions, β-reduction as well as incremental compiled code constructors such as $\mathbf{lam}_{\mathbf{X}}$ and **app** must be adapted accordingly. It would be interesting to see how complicated such adjustments turn out if they are feasible at all.

9.4 Concluding Remarks

Th contribution of this thesis to the study of programming languages is a modular language design methodology for enhancing programming languages with incremental program construction capabilities. Such capabilities are given an intuitively elegant and yet sensible syntactic description. Furthermore, they can be added to a programming language regardless of its computational idiosyncrasies. Last but not least, this thesis illustrates that many incremental programming paradigms are indeed programming with the notion of contexts. Our work answers only some basic questions about contextual programming. There are a series of open problems waiting to be explored. We expect further experimentation with the notion of contextual programming to shed more light on the nature of incremental program development.

Bibliography

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, pages 296–320. Springer-Verlag, 1994.

[3] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, 1996.

[4] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[5] Norman Adams and Jonathan Rees. Object-oriented programming in Scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 277–288, 1988.

[6] Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus: Syntax and confluence. In *Proceedings of the 13th International Conference on Foundations of Software Technologies and Theoretical Computer Science*, pages 24–40, Springer-Verlag, 1993.

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

[8] Henk Barendregt. Self-interpretation in lambda calculus. *Journal of Functional Programming*, 1(2):229–234, 1991.

[9] Klaus J. Berkling and Elfriede Fehr. A consistent extension of the lambda-calculus as a base for functional programming languages. *Information and Control*, 55:89–101, 1982.

[10] Klaus J. Berkling and Elfriede Fehr. A modification of the λ-calculus as a base for functional programming languages. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, pages 35–47, Lecture Notes in Computer Science 140. Springer-Verlag, 1982.

[11] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 303–311, 1990.

[12] Gilad Bracha and Gary Lindstrom. Modularity meets Inheritance. In *Proceedings of the International Conference on Computer Languages*, pages 282–290, 1992.

[13] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.

[14] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[15] N. G. de Bruijn. A survey of the project AUTOMATH. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 579–606, Academic Press, 1980.

[16] M.W. Bunder. An extension of Klop's counterexample to the Church-Rosser property to lambda-calculus with other ordered pair combinators. *Theoretical Computer Science*, 39:337–342, 1985.

[17] Luca Cardelli. Compiling a functional language. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 208–217, 1984.

[18] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Survey*, 17(4):471–522, 1985.

[19] Luca Cardelli and John C. Mitchell. Operations on records. In [32].

[20] Craig Chambers, David Ungar, and Eugene Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. *Lisp and Symbolic Computation*, 4(3):243–281, 1991.

[21] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, 4(3):1–55, 1991.

[22] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 433–443, 1989.

[23] William Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Brown University, 1989.

[24] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches.* PhD thesis, University of Geneva, 1994.

[25] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pages 297–302, 1984.

[26] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[27] Marc Feeley and Guy Lapalme. Closure generation based on viewing lambda as epsilon plus compile. *Journal of Computer Languages*, 17(4):251–267, 1992.

[28] Matthias Felleisen and Daniel P. Friedman. A closer look at export and import statements. *Journal of Computer Languages*, 11(1):29–37, 1986.

[29] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

[30] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley, 1983.

[31] James Gosling and Henry McGilton. The Java language environment: A white paper. Sun Microsystems, 1995.

[32] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[33] Chris Hankin. *Lambda Calculi: A Guide For Computer Scientists*. Oxford University Press, 1994.

[34] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[35] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[36] Stanley Jefferson, Shinn-Der Lee, and Daniel P. Friedman. A syntactic theory of transparent parameterization. In *Proceedings of the 3rd European Symposium on Programming*, pages 211–226, Springer-Verlag, 1990.

[37] Samuel N. Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[38] Stephen C. Kleene. λ-Definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.

[39] J.W. Klop. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127, Mathematisch Centrum, Amsterdam, 1984.

[40] John Lamping. A unified system of parameterization for programming languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 316–326, 1988.

[41] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[42] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 479–492, 1993.

[43] Mark Leone and Peter Lee. Lightweight run-time code generation. In *Proceedings of the ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1994.

[44] John McCarthy. Towards a mathematical science of computation. In *Proceedings of IFIP Congress 63*, pages 21–28. North-Holland, 1963.

[45] John McCarthy *et al. LISP 1.5 Programmer's Manual.* MIT Press, 1965.

[46] Bertrand Meyer. *Eiffel the Language.* Prentice Hall, 1992.

[47] James S. Miller and Guillermo J. Rozas. Free variables and first-class environments. *Lisp and Symbolic Computation*, 4(2):107–141, 1991.

[48] Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[49] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[50] John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Volume B, pages 365–458. MIT Press, 1990.

[51] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *Proceedings of the 8th IEEE Symposium on Logic In Computer Science*, 1993.

[52] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. Technical Report, DIKU, University of Copenhagen, 1994.

[53] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignments, and the lambda calculus. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 43–56, 1993.

[54] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[55] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, 1972.

[56] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[57] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994.

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

[59] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[60] Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

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

[62] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 150–161, 1994.

[63] Bruce Shriver and Peter Wegner. *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[64] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 38–45, 1986.

[65] Alan Snyder. Inheritance and the development of encapsulation software components. In [63].

[66] Guy L. Steele Jr. *Common Lisp: The Language, Second Edition.* Digital Press, 1990.

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

[68] Bjarne Stroustrup. *The C++ Programming Language*, second edition. Addison Wesley, 1991

[69] Masako Takahashi. Parallel reductions in λ-calculus. *Journal of Symbolic Computation*, 7:113–123, 1989.

[70] Carolyn Talcott. Binding structures. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 427–448. Academic Press, 1991.

[71] Carolyn Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112:99–143, 1993.

[72] David A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[73] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.

[74] Mario Wolczko. Semantics of Smalltalk-80. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 108–120, Lecture Notes in Computer Science 276. Springer-Verlag, 1987.

Curriculum Vita

Shinn-Der Lee received his bachelor's degree from National Taiwan University, Taiwan, in 1982. In 1984, he earned his master's degree in Computer Science from National Taiwan University. His doctoral research has been conducted under the supervision of Prof. Daniel P. Friedman. During its course, he has published four papers and given presentations of his work at conferences in the United States. His graduate studies at Indiana University, Bloomington, have been supported by university funded associate instructorships and NSF funded research assistantships. He is a member of the Association for Computing Machinery since 1989.