The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

REASONING ABOUT HIGHER-ORDER FUNCTIONS AND PARAMETERS

A Thesis in

Computer Science and Engineering

by

Joseph Adam Fischbach

© 2005 Joseph Adam Fischbach

Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

August 2005
The thesis of Joseph Adam Fischbach was reviewed and approved* by the following:

John Hannan  
Associate Professor of Computer Science and Engineering  
Thesis Adviser  
Chair of Committee

Ali Hurson  
Professor of Computer Science and Engineering

Mahmut Kandemir  
Associate Professor of Computer Science and Engineering

Stephen Simpson  
Professor of Mathematics

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

The compilation of functional programming languages such as Standard ML and Haskell rely on a number of program transformations/optimizations in order to generate efficient code. In this thesis, I present formal, type-based specifications of three such transformations: useless-variable elimination, lambda-lifting and unCurrying. These three transformations have in common the fact that they all involve the analysis and manipulation of function parameters. Useless-variable elimination identifies function parameters whose corresponding arguments contribute nothing to the observable behavior of the program, and eliminates those arguments. Lambda-lifting introduces additional parameters to functions in order to eliminate occurrences of non-local, non-global variables. UnCurrying transforms Curried functions and applications so that functions are applied to multiple arguments in a single application.

The conciseness and simplicity of type-based analyses provide a convenient platform for reasoning about program transformation. Using straightforward induction techniques, we can prove properties such as type preservation and operational correctness. We can guarantee the correctness of specific algorithms by proving via induction that these algorithms are sound with respect to the formal specification. I present a number of these proofs in this thesis.
# Table of Contents

List of Figures ........................................... vii

Acknowledgments ........................................... ix

Chapter 1. Introduction .................................... 1

Chapter 2. A Simple Functional Language ................... 5
  2.1 Syntax ........................................... 5
  2.2 Type System ....................................... 7
  2.3 Operational Semantics ................................. 9

Chapter 3. Useless-Variable Elimination ..................... 15
  3.1 Formal Specification of UVE .......................... 16
    3.1.1 Language Extensions ............................ 17
    3.1.2 Annotated Types and Subtypes ................. 21
    3.1.3 Inference Rules ................................ 24
    3.1.4 Examples of UVE ............................... 31
  3.2 Algorithm ......................................... 39
    3.2.1 Analysis and Annotation ....................... 41
    3.2.2 Constraint Solving ............................. 47
    3.2.3 Translation .................................... 51
    3.2.4 Correctness ................................... 52
3.3 Related Work ........................................... 54

Chapter 4. Lambda Lifting .................................... 57
  4.1 Introduction ........................................... 57
  4.2 Formal Specification of Parameter Lifting .......... 59
    4.2.1 Singleton Types .................................. 60
    4.2.2 Inference Rules .................................. 61
  4.3 Examples of Parameter-Lifting ....................... 67
    4.3.1 Example 1 ...................................... 68
    4.3.2 Example 2 ...................................... 72
  4.4 Algorithm Correctness ................................ 74
    4.4.1 Two-Phase Specification ......................... 76
    4.4.2 Algorithm ..................................... 85
    4.4.3 Correctness ................................... 91

Chapter 5. UnCurrying .................................... 93
  5.1 Target Language ...................................... 96
    5.1.1 Parameter Lists ................................ 96
    5.1.2 Partial Application .............................. 99
    5.1.3 Dummy Constants ................................ 103
  5.2 Type Annotations .................................... 106
    5.2.1 UnCurry Annotations .............................. 107
    5.2.2 Parameter-Type Lists ............................ 109
  5.3 UnCurry Transformation .............................. 110
5.3.1 Variable ........................................ 112
5.3.2 Abstraction ................................... 114
5.3.3 Letrec ......................................... 116
5.3.4 Application Rules .............................. 118
5.3.5 Useless-Variable Rules ....................... 123
5.3.6 Function Specialization ....................... 128
5.4 Correctness ...................................... 133
  5.4.1 Type Soundness .............................. 133
  5.4.2 Operational Correctness ..................... 138

Chapter 6. Concluding Remarks ....................... 143
  6.1 Extensions to Real Languages .................. 145
  6.2 Algorithms for Lambda-Lifting and unCurrying 145
  6.3 Higher-Order to First-Order Transformation ..... 146

Appendix A. Useless-Variable Elimination Proof of Correctness .......... 151

Appendix B. Lambda Lifting Algorithm Proof of Correctness ............... 162

Appendix C. UnCurrying Proof of Type Soundness ....................... 167

Appendix D. UnCurrying Proof of Operational Correctness ................ 186

References .......................................... 208
# List of Figures

2.1 Type System ........................................... 8
2.2 Operational Semantics ................................. 10
2.3 Example derivation .................................. 11
2.4 Simple recursion example ............................ 11
3.1 Extensions to source language ....................... 17
3.2 UVE with Subtypes .................................... 25
3.3 Derivation for Example 1 ............................ 32
3.4 Incorrect derivation for Example 2 ................. 33
3.5 Correct derivation for Example 2 ................... 34
3.6 Derivation of Example 3 ............................. 35
3.7 Derivation of Example 4 - first application ....... 36
3.8 Derivation of Example 4 - second application .... 37
3.9 Analysis and annotation phase ..................... 41
3.10 Solving constraints ................................. 48
3.11 The translate function .............................. 51
4.1 Parameter Lifting .................................... 62
4.2 Example derivation using (lift-app) ............... 70
4.3 Example derivation using (lift-abs) ............... 70
4.4 Parameter Lifting Annotation Phase ............... 77
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Parameter Lifting Translation Phase</td>
<td>77</td>
</tr>
<tr>
<td>4.6</td>
<td>Analysis example</td>
<td>83</td>
</tr>
<tr>
<td>4.7</td>
<td>Translation example</td>
<td>84</td>
</tr>
<tr>
<td>4.8</td>
<td>Example declaration derivation</td>
<td>85</td>
</tr>
<tr>
<td>4.9</td>
<td>Example application derivation</td>
<td>86</td>
</tr>
<tr>
<td>4.10</td>
<td>Parameter-lifting algorithm</td>
<td>87</td>
</tr>
<tr>
<td>4.11</td>
<td>Constraint solver</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>Target Language Static Semantics</td>
<td>104</td>
</tr>
<tr>
<td>5.2</td>
<td>Target Language Dynamic Semantics</td>
<td>105</td>
</tr>
<tr>
<td>5.3</td>
<td>UnCurry Inference Rules (Part 1)</td>
<td>113</td>
</tr>
<tr>
<td>5.4</td>
<td>UnCurrying abstraction example</td>
<td>116</td>
</tr>
<tr>
<td>5.5</td>
<td>UnCurry Inference Rules (Part 2)</td>
<td>119</td>
</tr>
<tr>
<td>5.6</td>
<td>UnCurry Inference Rules (Part 3)</td>
<td>125</td>
</tr>
</tbody>
</table>
Acknowledgments

I would like to thank the members of my Ph.D. committee, as well as Dale Miller and Catuscia Palamidessi, for their valuable time and efforts on my behalf. Thanks also to Matt (or is it Hank?) Davis for providing an invaluable ear and sense of humor during my time at Penn State.

Most importantly, I must acknowledge and offer my profound thanks to my advisor, John Hannan. Without his guidance and infinite patience, I would never have successfully navigated my way through graduate school to the completion and successful defense of this thesis.
Chapter 1

Introduction

Functional programming languages such as Standard ML and Haskell provide the programmer with a layer of abstraction not found in imperative languages such as C and C++. This allows programmers to express solutions in a more natural, mathematical style, rather than as a series of repeated commands that change a machine's state. To facilitate the use of recursive functions, Standard ML and Haskell treat functions as first-class values, meaning they can be considered data. Functions can be passed as arguments to other functions and returned as results. This added abstraction from the machine is meant to make programming easier and more natural. However this abstraction places a greater burden on the compiler. A functional program must go through a series of automatic transformations during the compilation process before the compiler can generate efficient machine code.

In this thesis, I present formal, type-based specifications for three of these automatic transformations: useless-variable elimination, lambda-lifting and unCurrying. These transformations are concerned with the analysis and manipulation of function parameters and arguments.

Useless-variable elimination analyzes each function in order to determine if any of the function's parameters contribute nothing to the function's result. Any such parameter is considered useless, and any argument that corresponds to that parameter is
eliminated, provided that the evaluation of the argument does not result in any side-effects. The purpose of the transformation is to remove unnecessary code from the program without affecting the program's observable behavior.

In addition to presenting the formal specification, I also describe a type-based algorithm for useless-variable elimination that is derived from the specification. In order to demonstrate the convenience of a formal specification, I show that the algorithm is type safe and operationally correct merely by proving that it is sound with respect to the specification.

Lambda-lifting eliminates non-local, non-global variables from function definitions so that these definitions can be "lifted" to the global level. This automatically eliminates the nesting of functions, resulting in a program that is in a simpler form, one that is required by other transformations. Unlike useless-variable elimination, lambda-lifting does not actually remove variables from functions, but rather introduces additional function parameters. These parameters provide local bindings for a function's (formerly) non-local, non-global variables.

For useless-variable elimination, I describe and prove correct an original algorithm derived from the specification. For lambda-lifting, I demonstrate how the formal specification can be used to prove the correctness of an existing algorithm. I present a type-based version of Johnsson's lambda-lifting algorithm and show that it is type-safe and operationally correct by proving that it is sound with respect to the specification.

Finally, unCurrying reduces the number of unnecessary function calls in a program. In Curried form, functions are applied to each of their arguments in individual
applications (i.e. function calls). UnCurrying groups parameters and arguments together so that a function can be applied to many, if not all, of its arguments in a single application.

Rather than describe a particular unCurry algorithm, I prove that the formal specification itself is both type safe and operationally correct by relating the specification to the static and dynamic semantics, respectively, of the object language.

The two primary motivating concepts in this thesis are the use of types to direct analyses and the use of formal specifications to describe program transformations. By using types to encode information from our analyses and guide the transformations, we are able to capture otherwise complex operations in relatively simple ways. Types not only allow us to craft concise, elegant descriptions of program transformation, they allow us to craft formal descriptions.

A formal specification of a program transformation provides a platform from which to study the transformations. For example, using established proof techniques such as well-founded induction, we can formally prove that a transformation is type safe and operationally correct, meaning it preserves the typeability and observable behavior of a program. A transformation that is type safe assures the compiler writer that the transformation can be used along with other type-based transformations. Certainly operational correctness is a vital property for any transformation. Once we have shown that a formal specification is correct, any algorithm implementing that transformation can be proved correct simply by showing that it conforms to the specification. A compiler built from correct algorithms assures the programmer that, though the compiler may transform the program in a number of ways, it will not alter the program's behavior.
The formal specifications described in this thesis are given as inference rules that define judgments of the form $\Gamma \vdash e : \tau \Rightarrow m$. This judgment essentially says that program $e$ transforms into program $m$. The type $\tau$ is used to direct the transformation. Each specification actually defines a relation between source and target programs. These relations are not one-to-one correspondences. Some source program may relate to several different transformed programs (and vice versa). For example, as explained in Chapter 4, there are many ways to lambda-lift a program and a formal specification can describe all of them. An algorithm, on the other hand, would only describe one particular form of lambda-lifting.

The remainder of this thesis is organized as follows. Chapter 2 introduces a core functional language based on the \( \lambda \)-calculus. The specifications described in this thesis transform programs written in this language. The formal, type-based specifications for the useless-variable elimination, lambda-lifting and unCurrying program transformations are described in Chapters 3-5, respectively. Chapter 6 contains some concluding remarks. The proofs of the correctness theorems are provided in the appendices.
Chapter 2

A Simple Functional Language

The source language assumed by the three program transformations presented in this thesis is a strongly typed core functional language based on the λ-calculus with a call-by-value operational semantics. In this chapter, I present the syntax of expressions and types as well as the static and dynamic semantics of the language. This chapter also serves as an introduction to the style of inference rules that I use here and in subsequent chapters to define the language semantics and program transformations.

2.1 Syntax

The syntax of the language is based on the λ-calculus and is comprised primarily of function abstraction and application. I will introduce other expressions in additional chapters as they relate to the particular transformation being discussed. The syntax of expressions is as follows:

\[ e ::= x \mid \lambda x.e \mid e_1 \circ e_2 \mid \mu e. \lambda x.e \mid \text{letrec } \vec{f}_i = \lambda x_i.e_i \text{ in } e \]

We use \( x \) and \( f \) to denote program variables. Typically, \( f \) is used to denote a function name. The line over \( f_i = \lambda x_i.e_i \) is a notational convenience representing

\[ f_1 = \lambda e_1; f_2 = \lambda x_2.e_2; \cdots; f_n = \lambda x_n.e_n \]
for some natural number $n > 0$. Any number of functions can be defined in a single
\[ \tau ::= \iota | \tau \to \tau \]

Since our language is strongly typed, I include the syntax of types as well:

I use $\iota$ to range over basic types (e.g. int, real, bool, etc.), and the $\to$ denotes a
function type. For example, int $\to$ bool is the type of a function that expects an integer
argument and returns a boolean result.

The $\mathbin{\@}$ symbol explicitly represents function application. In order to apply the
identity function $\lambda x.x$ to an argument $e$, we would write $(\lambda x.x) \mathbin{\@} e$. Functions in the
language are Curried, meaning they are applied to one argument at a time. This follows
the convention of functional languages such as Standard ML. Assuming we defined an
operator $+$ for addition, we could write the function that computes the sum of two
integers as: $\lambda x.\lambda y.x + y$. However, because the function is Curried, it must be given its
two arguments separately, requiring two applications:

\[ (\lambda x.\lambda y.x + y) \mathbin{\@} e_1 \mathbin{\@} e_2. \]

The leftmost application results in the function $\lambda y.x + y$ where $x$ has been bound to the
value of $e_1$. Essentially, this sum function is a function of one argument that returns
another function of one argument. The second application then binds the parameter $y$
to the value of the argument $e_2$ and results in the computed sum of $x$ and $y$. 
Recursion is represented in one of two ways in our language: using \( \mu \) or letrec. We use \( \mu \) in our useless-variable elimination transformation and letrec for lambda-lifting and unCurrying. The reasons for this will be made clear in the subsequent chapters.

The \( \mu \) operator can be used to define non-mutually recursive functions. For example, the recursive function that simply calls itself with the same argument can be written as \( \mu f. \lambda x.(f \, @ \, x) \). In order to apply this function to some argument \( e \), we simply write \( (\mu f. \lambda x.(f \, @ \, x)) \, @ \, e \).

I use letrec to define mutually recursive functions (without the \( \mu \) operator). The expression

\[
\text{letrec } f = \lambda x. (g \, @ \, x); \\
g = \lambda y. (f \, @ \, y)
\]

defines the functions \( f \) and \( g \) that recursively call each other, passing along the same argument. The body of the letrec provides the initial call to \( f \) with the argument \( e \).

2.2 Type System

The static semantics for the language is given in Figure 2.1 using inference rules. There is a rule for each form of expression in the language defining how to infer that expression’s type. The set of rules enables us to construct derivations for the judgment \( \Gamma \, \vdash \, e : \tau \), which is read: “Given the context \( \Gamma \), expression \( e \) has type \( \tau \).” The context \( \Gamma \) defines a mapping of variables to their types.

For each inference rule, if we can derive the statements above the line, then we can derive the statement below the line. For example, the rule for application says that, if we
\[
\begin{align*}
\Gamma(x) &= \tau \\
\Gamma \vdash x : \tau \\
\Gamma \vdash e_1 : \tau_1 \to \tau \\
\Gamma \vdash e_2 : \tau_1 \\
\Gamma \vdash e_1 \otimes e_2 : \tau \\
\Gamma \vdash \lambda x.e : \tau_1 \to \tau \\
\Gamma \vdash \lambda x.e : \tau \\
\Gamma \vdash \mu f.\lambda x.e : \tau \\
\end{align*}
\]

\[
\Gamma^k = \Gamma\{f_i : \tau_i\} \quad \Gamma^k \vdash \lambda x_1.e_1 : \tau_1 \cdots \Gamma^k \vdash \lambda x_n.e_n : \tau_n \\
\Gamma \vdash \text{lletrec } f_i = \lambda x_i.e_i \text{ in } e : \tau \\
i = [1..n]
\]

Fig. 2.1. Type System

can prove $\Gamma \vdash e_1 : \tau_1 \to \tau$ and we can prove $\Gamma \vdash e_2 : \tau_1$, then we know $\Gamma \vdash e_1 \otimes e_2 : \tau$.

In other words, given context $\Gamma$, if $e_1$ has the function type $\tau_1 \to \tau$ and $e_2$ has type $\tau_1$, then the application $e_1 \otimes e_2$ must have the type $\tau$.

We infer the type of a variable simply by looking it up in the given context. If $\Gamma$ includes the binding $x : \tau$, then $\Gamma(x) = \tau$. If there is no binding for $x$ in $\Gamma$, then $\Gamma(x)$ is undefined and the type of the variable cannot be inferred.

In the abstraction rule, $\Gamma\{x : \tau_1\}$ extends the context $\Gamma$ with the new binding of variable $x$ to type $\tau_1$. Hence the rule says that if we assume $x$ has type $\tau_1$ and can prove that the expression $e$ has type $\tau$, then the abstraction $\lambda x.e$ must have type $\tau_1 \to \tau$. The rule for $\mu$ also extends the context. Here we assume the variable $f$, which represents the recursive function, has the same type as the function.

Since functions declared in a \text{lletrec} can be mutually recursive, we must define a new context $\Gamma^k$ to be an extension of $\Gamma$ containing the bindings for each new function variable introduced in the \text{lletrec} to their respective types. This new context must be used to infer the types of the functions $\lambda x_i.e_i$ as well as the expression $e$. 
It may seem odd that we must assume the types of the function variables in \( \Gamma^* \) in order to infer the types of the functions themselves. However, this is necessary since any of the function variables \( f_i \) can appear in any of the expressions \( e_i \), thus their types must be in the context. It should be noted that the inference rules in Figure 2.1 do not define an algorithm for type inference. Rather, the inference rules define the relationships among the types of expressions. The rule for letrec specifies that the types of the function variables in \( \Gamma^* \) must be the same as the types of the functions defined in the letrec. The rule does not specify how an algorithm would compute those types. This concept will be revisited throughout this thesis as we present inference rules that define the relationships between source and target expressions for each of our program transformations. The inference rules define the transformations, they do not directly provide algorithms for computing the transformations.

2.3 Operational Semantics

The big-step, call-by-value operational semantics for the language is presented in Figure 2.2. The operational semantics defines the relationship between expressions and their values. The only values in our simple language are functions.\(^1\) If we were to include constants such as integers or characters, they would be considered values as well.

The judgment \( \rho \triangleright e \leftrightarrow v \) is read: “Given an environment \( \rho \), the expression \( e \) has the value \( v \).” The environment \( \rho \) defines a mapping of variable expressions to their values. For example, the rule for inferring the value of a variable simply requires the lookup of that variable in the environment. If \( \rho(x) = v \), then \( \rho \triangleright x \leftrightarrow v \).

\(^1\)In Figure 2.2, values are actually function closures.
\[
\begin{align*}
\rho(x) &= v & \frac{\rho \triangleright x \rightarrow v}{\rho \triangleright \lambda x.e \rightarrow [\rho, \lambda x.e]} \\
\rho \triangleright e_1 &\rightarrow [\rho', \lambda x.e] & \rho \triangleright e_2 &\rightarrow v_2 & \rho' \{x \mapsto v_2\} \triangleright e \rightarrow v \\
\rho \triangleright e_1 \& e_2 &\rightarrow v & \rho \triangleright \mu f.\lambda x.e \rightarrow [\rho, \lambda x.e[\mu f.\lambda x.e/f]] \\
\rho \triangleright e_i[\text{letrec } f_j = e_j \text{ in } f_k/f_k] &\rightarrow v_i & \rho \{f_j \mapsto v_i\} \triangleright e \rightarrow v & \rho \triangleright \text{letrec } f_i = e_i \text{ in } e \rightarrow v \quad i, j, k = [1..n]
\end{align*}
\]

Fig. 2.2. Operational Semantics

Since we use explicit environments in Figure 2.2, the value of a function is actually a function closure. The body of a function must always be evaluated using the same environment with which it is defined. This is necessary since functions may contain free variables. Whenever a function is applied to its argument, the values of all of these free variables must be available. To make this possible, we use closures, which package the function along with its environment. In the rule for abstraction, the value of the function \( \lambda x.e \) in environment \( \rho \) is the closure \([\rho, \lambda x.e]\).

We can see how closures are used in our operational semantics by examining the rule for application. If \( e_1 \) is applied to \( e_2 \), then \( e_1 \) must be a function, therefore its value is a closure. Since we are using a call-by-value semantics, the value of \( e_2 \) must also be determined. The environment \( \rho' \) from the function closure is extended by including the mapping of the function parameter \( x \) to the value \( v_2 \). The body of the function from the function closure is evaluated using this extended environment.
\[
\begin{align*}
\Gamma \vdash \lambda x.x & : [\rho, \lambda x.x] \\
\Gamma \vdash 0 & : [\rho, 0] \\
\Gamma \vdash \lambda x.0 & : [\rho, \lambda x.0] \\
\end{align*}
\]

\[\rho \vdash \lambda x.x \rightarrow [\rho, \lambda x.x] \quad \rho \vdash 0 \rightarrow 0 \quad \rho \vdash \lambda x.0 \rightarrow 0\]

Fig. 2.3. Example derivation

\[\rho \vdash \mu f. \lambda x.(f \circ x) \rightarrow [\rho, \lambda x.(\mu f. \lambda x.(f \circ x)) \circ x] \quad (\rho \vdash 0 \rightarrow 0) \vdash (\mu f. \lambda x.(f \circ x)) \circ x \rightarrow ? \]

\[\rho \vdash (\mu f. \lambda x.(f \circ x)) \circ 0 \rightarrow ?\]

Fig. 2.4. Simple recursion example

As an example of a derivation using the rules in Figure 2.2, I illustrate the evaluation of the application \((\lambda x.x) \circ 0\) in Figure 2.3. This derivation assumes a constant rule for evaluating the constant expression \(0\) to the value \(0\). At the bottom of the derivation, we use the application rule to evaluate \((\lambda x.x) \circ 0\). Above the line for the application rule, we use the abstraction rule, constant rule, and variable rule to evaluate \(\lambda x.x\), \(0\), and \(x\) (the body of the function), respectively. The derivation serves as a proof that the expression \((\lambda x.x) \circ 0\) has the value \(0\).

Since \(\mu\) defines a function, albeit a recursive one, its value is also a closure. The notation \(e[\mu f. \lambda x.e/f]\) represents the substitution of \(\mu f. \lambda x.e\) for every occurrence of the variable \(f\) in \(e\). In Figure 2.4, the function \(\mu f. \lambda x.(f \circ x)\) calls itself recursively, passing itself the same argument \(x\). Since this function clearly never terminates, no derivation actually exists, meaning \((\mu f. \lambda x.(f \circ x))\) has no value.
The rule for letrec must derive the values of mutually recursive functions. The substitution $e_i[\text{letrec } f_j = e_j \text{in } f_k/f_x]$ in the antecedent of the rule replaces occurrences of function names within the expressions $e_i$ with letrec expressions containing definitions of the functions defined by the letrec in the consequent of the rule. Since any function can call any other function defined in the same letrec, this substitution essentially “packages” each function name along with the definitions of the functions defined in the same letrec that the named function can call. This allows us to evaluate the expressions $e_i$ using the given environment $\rho$, which facilitates the proof of operational correctness in Chapter 5. The expression $e$ is evaluated using the given environment extended to including mappings of the function names to their values.

The following examples involves a letrec expression that defines two mutually recursive functions $f$ and $g$. Since each function simply calls the other, passing along the same argument, the letrec never terminates. In attempting to build a derivation, we would first use the letrec rule:

$$
\begin{array}{c}
\Pi_1 \\
\Pi_2 \\
\Pi_3 \\
\hline
\rho \vdash \text{letrec } f = \lambda x.g @ x; g = \lambda y.f @ y \text{ in } f @ 0 \rightarrow ?
\end{array}
$$

The derivations $\Pi_1$ and $\Pi_2$ are instances of the abstraction rule. In both cases, the function names are replaced with letrec expressions containing the definitions of $f$ and $g$. The first derivation evaluates the abstraction bound to $f$:

$$
\begin{array}{c}
\Pi_1 \vdash x.(\text{letrec } f = \lambda x.g @ x; g = \lambda y.f @ y \text{ in } g) @ x \rightarrow \\
[\rho, \lambda x.(\text{letrec } f = \lambda x.g @ x; g = \lambda y.f @ y \text{ in } g) @ x]
\end{array}
$$
Similarly, the second derivation evaluates the abstraction bound to $g$:

\[
\Pi_2 :: \\
\rho \triangleright \lambda y.(\letrec f = \lambda x.g \in x; g = \lambda y.f \in y \in f) \in y \mapsto \\
[\rho, \lambda y.(\letrec f = \lambda x.g \in x; g = \lambda y.f \in y \in f) \in y]
\]

In $\Pi_3$, we attempt to derive a value for $f \in 0$ using the application rule but, as the function bound to $f$ diverges, no such value exists. The environment $\rho' = \rho[f \mapsto [\rho, \lambda x.(\letrec f = \lambda x.g \in x; g = \lambda y.f \in y \in g) \in x] \{ g \mapsto [\rho, \lambda y.(\letrec f = \lambda x.g \in x; g = \lambda y.f \in y \in f) \in y] \}].$

\[
\rho'(f) = [\rho, \lambda x.(\letrec f = \lambda x.g \in x; g = \lambda y.f \in y \in g) \in x] \\
\rho' \triangleright f \mapsto [\rho, \lambda x.(\letrec f = \lambda x.g \in x; g = \lambda y.f \in y \in g) \in x] \\
\rho' \triangleright 0 \mapsto 0 \quad \Pi_4
\]

$\Pi_3 :: \rho' \triangleright f \in 0 \mapsto \_ ?$

In $\Pi_4$, which is another instance of the application rule, we attempt to derive a value for the application of function $g$ to the argument $x$.

\[
\Pi_4 :: \\
\rho' \triangleright (\letrec f = \lambda x.g \in x; g = \lambda y.f \in y \in g) \in x \mapsto \_ ?
\]

The derivation $\Pi_5$ uses the letrec rule, which evaluates the letrec expression to a closure.

\[
\Pi_5 :: \\
\rho[0 \mapsto 0] \triangleright \letrec f = \lambda x.g \in x; g = \lambda y.f \in y \in g \mapsto \\
[\rho[0 \mapsto 0], \lambda y.(\letrec f = \lambda x.g \in x; g = \lambda y.f \in y \in f) \in y]$
\]
In $\Pi_6$, we attempt to evaluate the body of the function in the closure using the environment from the closure extended to include the mapping of the function’s parameter $y$ to the value of the argument.

$$\Pi_6 :: \rho\{x \mapsto 0\}\{y \mapsto 0\} \triangleright (\text{letrec } f = \lambda x.g \odot x; g = \lambda y.f \odot y \text{ in } f) \odot y \mapsto ?$$

Observe that we are right back where we started, trying to apply the function $f$ to the argument 0. Since the application diverges, the derivations $\Pi_6, \Pi_4$ and $\Pi_3$ do not exist.
Chapter 3

Useless-Variable Elimination

Useless-variable elimination (UVE) analyzes a given program to determine which variables, or, more specifically, which function parameters, are useless. A variable is useless if its value does not contribute to the observable behavior of a program. By observable behavior, we mean the value returned as the result of evaluating a program as well as any side-effects that may occur during the program’s evaluation. In the expression

$$(\lambda x.\lambda y.x) \circ \circ 1 \circ 2,$$

the parameter $y$ is a useless variable. Because $y$ is a useless variable, the integer expression 2 is a useless operand. The goal of the UVE transformation presented in this chapter is to eliminate such useless operands by replacing them with dummy variables. In the simple example given here, replacing the expression 2 with a dummy variable does not actually accomplish very much. However in more complex examples, useless operands can be computationally expensive expressions. Eliminating such expressions would clearly reduce the runtime of many programs.

Why would useless variables and operands appear in an expression to begin with? Typically we would not expect a program written by a human to contain useless code. However a program written by a machine might. Other compiler optimizations may transform a program in such a way that renders code useless or, as is the case with
lambda lifting and unCurrying,\(^1\) actually introduce parameters that are useless. The purpose of UVE is to “clean up” after these optimizations by detecting and removing useless code.

In this chapter, I formally describe a useless-variable elimination transformation and subsequently present an algorithm that detects and eliminates useless code from an input program. The work presented here is joint work (with John Hannan) and has previously been published as a conference paper [9]. A full account of the work, including proofs of correctness, is available as a technical report [10].

### 3.1 Formal Specification of UVE

We define our useless-variable elimination transformation using inference rules similar to those used in the previous chapter to define the static and dynamic semantics of our core functional programming language.

Since our transformation is type-based, we encode information regarding the usefulness of expressions within their types. In particular, the type of a function will identify the usefulness of that function’s parameter. Since type information is vital to the transformation, type inference is a pivotal part of the process. Before an expression can be translated, its type must be inferred and encoded with the appropriate information. An algorithm must perform these two operations (type inference and program translation) separately. However since we are defining the UVE transformation in this section, and not presenting a specific algorithm that computes that transformation, we can define both operations simultaneously in one rather simple set of inference rules.

\(^1\)Both lambda lifting and unCurrying are discussed in detail in subsequent chapters.
\[
\begin{align*}
\Gamma \vdash \mathit{e}_1 : \text{int} & \quad \Gamma \vdash \mathit{e}_2 : \text{int} \\
\Gamma \vdash \mathit{e}_1 + \mathit{e}_2 : \text{int} \\
\rho \triangleright \mathit{e} & \rightarrow \mathit{v} \\
\rho \triangleright \mathit{e}_1 & \rightarrow \mathit{v}_1 \\
\rho \triangleright \mathit{e}_2 & \rightarrow \mathit{v}_2 \\
\mathit{v} = \mathit{v}_1 + \mathit{v}_2 \\
\rho \triangleright \mathit{e}_1 + \mathit{e}_2 & \rightarrow \mathit{v}
\end{align*}
\]

Fig. 3.1. Extensions to source language

### 3.1.1 Language Extensions

Some modifications and extensions to the core functional language described in Chapter 2 are necessary in order to present our work in UVE. In fact, we work with two languages in this chapter: a source language and a target language. A program encoded in the source language can be transformed via UVE into a program encoded in the target language.

The source language for our transformation is the same as the language presented in the previous chapter extended to include natural numbers and the addition operation. The typing rules and operational semantics for these two new expressions are presented in Figure 3.1. The \( n \) ranges over the natural numbers. To keep things simple, and to correspond with other recent work in UVE [6, 17, 28], we use the \( \mu \) operator for recursion rather than the mutually recursive letrec.

The target language, discussed in detail below, differs from the source language in that it includes dummy variables (that take the place of useless operands) and two forms of application: one for applying a function to a needed operand and one for applying a function to a useless operand.
Effects

As we will see, just because our transformation identifies a particular function’s parameter as useless does not necessarily mean that all operands to which that function is applied are also useless. Eliminating an operand solely based on the use of the corresponding parameter can drastically alter the behavior of a program.

For example, consider the expression $(\mu. \lambda x. f \circ x) \circ e$. Clearly this application never results in a value since the recursion does not terminate. Suppose that this application is the only expression in a program that does not terminate. Suppose also that this application is an operand that is identified as useless (because the function applied to this operand has a useless parameter). If we eliminate this application, we have drastically altered the behavior of the program. The original program does not terminate because of this application, whereas the translated program, in which the application has been eliminated, does terminate.

As a second example, consider the addition operation. Addition may result in overflow. We will assume that overflow is automatically detected and results in the premature termination of the program. If an operand is detected as useless but includes an addition operation, removing the operand may affect the behavior of a program. If the addition resulted in overflow then the program would terminate prematurely. However if we eliminate the operand then the program may terminate naturally.

In the first example, careless UVE would cause a nonterminating program to terminate. In the second example, careless UVE would allow a program that terminates because of a runtime error to terminate naturally. Since a program transformation as part of the compilation process should never, under any circumstances, alter the behavior of
a program, neither of the above transformations are acceptable. Thus before eliminating
an operand, our UVE transformation must not only detect the operand as useless, but
must also guarantee that the operand does not potentially affect the observable behavior
of the program. In our language, the only expressions with potential side-effects are
recursion and addition.

Analyzing a program to detect such effect-causing expressions is a separate prob-
lem (see [27] for an example of an effect analysis), and not one we address directly.
Instead, we assume effect information has been computed prior to UVE and is accessible
via the noeffect function.

Target Language

The goal of UVE is to identify function parameters that are useless. If a function’s
parameter is useless, that suggests the operand to which the function is applied is also
useless (unless the operand has an effect). Our UVE system transforms a source program
by replacing each useless operand with a dummy variable. Furthermore, we introduce
the concept of useless applications. Specifically, if a function with a useless parameter is
applied to a useless operand then the application itself is considered useless.

The transformation’s target language is the same as the source language, except
there are two forms of application and we have introduced dummy variables:

\[ e ::= n \mid x \mid d^T \mid \lambda x.e \mid e \oplus_n e \mid e \oplus_u e \mid e + e \mid \mu f.\lambda x.e. \]
To maintain the typeability of our programs, a dummy variable \( d^\tau \) must be tagged with the type of the operand that it replaces. An application is tagged with either \( n \) or \( u \) if that application is considered needed (i.e. normal application) or useless, respectively.

The typing rules for applications are the same as the application rule in Figure 2.1. The tag on the application has no bearing on its type. The typing rule for dummy variables is as follows:

\[
\Gamma \vdash d^\tau : \tau 
\]

The type of the dummy variable is the type that it is tagged with.

The operational semantics for the two applications illustrates the difference between needed and useless applications. For a needed application, the operational semantics is the same as that for a regular application in Figure 2.2. The operational semantics for a useless application ignores the operand completely and simply evaluates the body of the applied function:

\[
\rho \triangleright e_1 \leftrightarrow [\rho', \lambda x.e] \quad \rho' \triangleright e \leftrightarrow v \\
\rho \triangleright e_1 \circledast u \ e_2 \leftrightarrow v 
\]

This rule assumes that, since the application is tagged as useless, the operand can safely be ignored. It is the responsibility of the UVE transformation presented in Section 3.1.3 to guarantee that this is the case.

We do not introduce an operational semantics rule for dummy variables since their evaluation is meaningless. The UVE transformation only inserts dummy variables in the target expression as operands of useless applications. Consequently, the value of a dummy variable (which is undefined) never needs to be known.
Section 3.1.3 explains how the UVE transformation handles dummy variables and useless applications, and examples of expressions containing dummy variables and useless applications are given in Section 3.1.4.

3.1.2 Annotated Types and Subtypes

We encode the use information of function parameters by annotating function types with either n or u, representing needed and useless parameters respectively. The types are annotated as follows, where a ranges over type annotations:

\[ \tau ::= \text{int} \mid \tau \to_a \tau. \]

Consider the following function: \( \lambda x. x + 2 \). Clearly the value of the parameter \( x \) is needed in order for this function to compute its result. Therefore the annotated type of the function must be \( \text{int} \to_n \text{int} \). The \( n \) indicates that the function's parameter is needed, thus any operand this function is applied to is also needed.

On the other hand, consider the function \( \lambda x. 1 \). This function returns the same value regardless of its parameter. Thus the value of the function's parameter is irrelevant and the parameter is considered useless. To reflect this observation, the function's type is annotated with \( u \): \( \text{int} \to_u \text{int} \).

Finally, consider the K-combinator: \( \lambda x. \lambda y. x \). This is a function of two parameters that returns the value of the first. The first parameter is required in order for the function to return a result, but the second parameter is not. Thus the \( x \) is considered needed and the \( y \) is considered useless. The annotated type of the K-combinator is: \( \tau_1 \to_n \tau_2 \to_u \tau_1 \).
The $n$ on the first arrow indicates that the first parameter is needed whereas the $u$ on
the second arrow indicates that the second parameter is useless.

The analysis presented in the next section is conservative. It does not detect
as useless every parameter in an input term that is actually useless. If a parameter is
detected as needed, that means only that the parameter may be needed. If, however, a
parameter is detected as useless, then that parameter is definitely useless.

Consider the K-combinator. Although it can clearly be seen by examining the
body of the function that the second parameter is useless, it may be the case that the
UVE analysis must consider the second parameter to be needed. This is possible, for
example, if the K-combinator is applied to expressions containing side effects. Thus the
type $\tau_1 \rightarrow_n \tau_2 \rightarrow_n \tau_1$ is also a valid type for the K-combinator. Note however that the
type $\tau_1 \rightarrow_u \tau_2 \rightarrow_u \tau_1$ is not a valid type. Considering a needed parameter to be useless
can have devastating consequences when translating the input term: We might delete
code that is necessary for the computation of the term's result.

To account for the fact that useless parameters may be considered needed, and
to allow our analysis more flexibility, we introduce the following subtyping relations:

**Definition 1 (UVE Subtyping of Annotations).** For annotations $a$ and $a'$, the
relation $a \leq_a a'$ is defined by the following rules:

\[
\begin{align*}
\text{u} & \triangleq_a n \\
\text{a} & \leq_a a
\end{align*}
\]
Definition 2 (UVE Subtyping of Annotated Types). For annotated types $\tau$ and $\tau'$, the relation $\tau \leq \tau'$ is defined by the following rules:

\[
\begin{align*}
\text{int} \leq \text{int} & \quad \frac{\tau_1' \leq \tau_1 \quad \tau_2' \leq \tau_2 \quad a \leq a'}{(\tau_1 \rightarrow_a \tau_2) \leq (\tau_1' \rightarrow_a \tau_2')} \\
\end{align*}
\]

If $\tau_1 \leq \tau_2$, then any term of type $\tau_1$ can also have type $\tau_2$, but the reverse is not true. The first definition above tells us that $u \leq_a n$, which means that any $u$ annotation may be replaced with an $n$ annotation (i.e. any parameter considered useless may also be considered needed). The simple function $\lambda z.1$ has type $\tau \rightarrow_u \text{int}$ since its parameter is clearly useless. Notice that $\tau \rightarrow_u \text{int} \leq \tau \rightarrow_n \text{int}$ and $\tau \rightarrow_n \text{int}$ is also a valid type for $\lambda z.1$. Now consider the function $\lambda x.x + 1$. Its type must be $\text{int} \rightarrow_n \text{int}$ since its parameter is obviously needed. The only type greater than or equal to $\text{int} \rightarrow_n \text{int}$ according to the subtyping relation is $\text{int} \rightarrow_n \text{int}$.

Observe that the subtyping relation is contravariant. In other words, if $(\tau_1 \rightarrow_a \tau_2) \leq (\tau_1' \rightarrow_a \tau_2')$ then $\tau_1' \leq \tau_1$. Suppose function $\lambda x.x \@ e$ has type $(\text{int} \rightarrow_n \text{int}) \rightarrow_n \text{int}$. Since $x$ is a function with a needed parameter, the UVE analysis determines that $e$ is a needed operand and translates the above expression into $\lambda x.x \@_n e$. The subtyping relation tells us that the function may also have type $(\text{int} \rightarrow_u \text{int}) \rightarrow_n \text{int}$. This means that $x$ can be bound to a function whose parameter is useless. For example, there is nothing wrong in applying $\lambda x.x \@_n e$ to $\lambda z.1$.

On the other hand, suppose the UVE analysis determines that $\lambda x.x \@ e_1$ has type $(\text{int} \rightarrow_u \text{int}) \rightarrow_n \text{int}$. Since $x$ is a function with a useless parameter, the UVE analysis will determine that $e_1$ is a useless operand and replace it with a dummy variable. The
function $\lambda x.x @ e_1$ will become $\lambda x.x @_u d^{\text{int}}$. Since $(\text{int} \to u \text{ int}) \to_n \text{ int} \not\preceq (\text{int} \to_n \text{ int}) \to_n \text{ int}$, $x$ cannot be bound to a function with a needed parameter. What happens, for example, if we apply $\lambda x.x @_u d^{\text{int}}$ to $\lambda y.y$? The result of the application is $d^{\text{int}}$, a meaningless dummy variable, when the result should be the value of $e_1$.

### 3.1.3 Inference Rules

Inference rules allow us to formally specify all of the valid program transformations allowed by our UVE system in a concise and simple manner. The rules do not define a one-to-one correspondence between source and target terms. The generality of the specification means that there are several different ways to translate certain source terms, yielding many different target terms. The simplicity of our system is due, in part, to this generality. By defining what UVE means (i.e. what transformations are possible) rather than defining a method for computing a particular transformation, we are able to avoid algorithmic details that would unnecessarily complicate the specification.

The inference rules provide an inductive definition of the judgment $\Gamma \parallel e : \tau \Rightarrow e'$. The $e$ and $e'$ represent the source and target terms, respectively. The type $\tau$ is the annotated type of the source and target terms. The context $\Gamma$ is similar to the type context in Figure 2.1 except that it maps variables to annotated types. Essentially, the judgment can be read as, “Given a type context $\Gamma$, the term $e$ has annotated type $\tau$ and can be translated via UVE to the term $e'$.” Our UVE specification defines a relation between source and target terms. Figure 3.2 shows the complete set of rules for UVE but I will discuss each rule individually below.
\[
\begin{align*}
\Gamma \vdash n : \text{int} & \Rightarrow n & \Gamma(x) = \tau \\
\Gamma \vdash x : \tau & \Rightarrow x \\
\Gamma \{ x : \tau_1 \} \vdash e : \tau_2 & \Rightarrow e' & \text{x \not\in FV(e')} \\
\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 & \Rightarrow \lambda x. e' \\
\Gamma \{ f : \tau \} \vdash \lambda x. e : \tau & \Rightarrow \lambda x. e' \\
\Gamma \vdash \mu f. \lambda x. e : \tau & \Rightarrow \mu f. \lambda x. e' \\
\Gamma \vdash e_1 : \tau_2 & \Rightarrow e_1' & \Gamma \vdash e_2 : \tau_2' \Rightarrow e_2' & \tau_2' \leq \tau_2 \\
\Gamma \vdash (e_1 @ e_2) : \tau & \Rightarrow (e_1' @ e_2') \\
\Gamma \vdash e_1 : \text{int} & \Rightarrow e_1' & \Gamma \vdash e_2 : \text{int} & \Rightarrow e_2' \\
\Gamma \vdash (e_1 + e_2) : \text{int} & \Rightarrow (e_1' + e_2') \\
\end{align*}
\]

Fig. 3.2. UVE with Subtypes
Constants and Variables

\[ \Gamma \vdash n : \text{int} \Rightarrow n \quad \Gamma \vdash x : \tau \Rightarrow x \]

The simplest rules apply to integer constants and variables. The rule for integers simply states that, regardless of the type context, the integer \( n \) has type \( \text{int} \) and translates to the integer \( n \). This makes sense as there are no parameters in a constant expression to eliminate. The rule for variables is almost as trivial. The variable \( x \) translates to itself, and its type is derived simply by looking up the variable in the type context \( \Gamma \). If the variable \( x \) does not appear in the context, then its translation is undefined.

Recursion and Addition

\[
\Gamma \{ f : \tau \} \vdash \lambda x.e : \tau \Rightarrow \lambda x.e' \\
\Gamma \vdash \mu f.\lambda x.e : \tau \Rightarrow \mu f.\lambda x.e' \\
\Gamma \vdash e_1 : \text{int} \Rightarrow e'_1 \\
\Gamma \vdash e_2 : \text{int} \Rightarrow e'_2 \\
\Gamma \vdash (e_1 + e_2) : \text{int} \Rightarrow (e'_1 + e'_2)
\]

The rules for recursion and addition are also fairly straightforward as neither rule has any constraints pertaining specifically to UVE. Basically, a recursion or addition source expression is translated by translating the source subexpressions. The source expression \((e_1 + e_2)\) has type \( \text{int} \) and translates to the target expression \((e'_1 + e'_2)\). The subexpressions \(e'_1\) and \(e'_2\) are the translated forms of \( e_1 \) and \( e_2 \).

Similarly, the source expression \( \mu f.\lambda x.e \) has type \( \tau \) and translates to the target expression \( \mu f.\lambda x.e' \), where \( \lambda x.e' \) is the translated form of \( \lambda x.e \). The type \( \tau \) is the
type of the abstraction $\lambda x. e$, derived by the judgment $\Gamma \{ f : \tau \} \vdash \lambda x. e : \tau \Rightarrow \lambda x.e'$ in the antecedent of the recursion rule. Note that this rule is similar to the typing rule for recursion in Figure 2.1. The only differences here are that $\tau$ is an annotated type identifying useless parameters, and the source term is related not only to its type, but also to its translated form.

**Abstraction**

$$
\frac{\Gamma \{ x : \tau_1 \} \vdash e : \tau_2 \Rightarrow e' \quad x \notin \text{FV}(e')}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow_{u} \tau_2 \Rightarrow \lambda x.e'}
$$

$$
\frac{\Gamma \{ x : \tau_1 \} \vdash e : \tau_2 \Rightarrow e'}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow_{n} \tau_2 \Rightarrow \lambda x.e'}
$$

The actual business of detecting and eliminating useless variables is confined to the rules for abstraction and application. For an abstraction, the analysis must specify under what conditions a parameter is useless. Likewise, for an application, the analysis must specify under what conditions an operand is useless. Remember that the point of UVE is to detect as many useless parameters as possible and to determine which corresponding operands can be replaced by dummy variables, thus eliminating useless code.

Unlike the other expressions, abstraction and application both require two rules. In each case, one rule applies for parameters or operands that are useless and the other applies for parameters or operands that are needed. We’ll look at abstraction first. The rule for an abstraction with a useless parameter must answer the question: What, exactly, does it mean for a parameter to be useless? The answer is surprisingly simple,
and is represented in the abstraction rule by the constraint \( x \notin FV(e') \), where \( FV(e') \) is defined as follows:

**Definition 3.** For a target expression \( e \), the set \( FV(e) \) is the set of expression variables free in \( e \):

\[
\begin{align*}
FV(n) &= \emptyset \quad FV(x) = \{x\} \quad FV(d^\top) = \emptyset \\
FV(\lambda x.e) &= FV(e) - \{x\} \quad FV(e_1 \oplus_a e_2) = FV(e_1) \cup FV(e_2) \\
FV(\mu f.\lambda x.e) &= FV(\lambda x.e) - \{f\} \quad FV(e_1 + e_2) = FV(e_1) \cup FV(e_2)
\end{align*}
\]

The abstraction rule for useless parameters states that the source expression \( \lambda x.e \) translates to the target expression \( \lambda x.e' \), where \( e' \) is the translated form of \( e \). The type of the abstraction can be annotated with \( u \), thus declaring the parameter \( x \) to be useless, provided that the constraint \( x \notin FV(e') \) holds. If the constraint is true, then the parameter does not appear in the translated body of the abstraction and is useless. If the parameter does appear in \( e' \), then it must be assumed that the parameter’s value will be needed in order to compute the value of the abstraction. In such a case, the parameter cannot be considered useless.

The second rule for abstraction applies when the abstraction’s parameter is considered needed. This rule simply states that an abstraction \( \lambda x.e \) is translated to \( \lambda x.e' \), where \( e' \) is the translated form of \( e \) (just as in the other abstraction rule). The type of the abstraction is annotated with \( n \) regardless of whether the parameter appears in \( e' \) or not. This rule is actually applicable for any abstraction since it contains no constraint
specific to UVE. Recall from Section 3.1.2 that any useless parameter can be considered needed.

Application

\[
\Gamma \vdash e_1 : \tau_2 \Rightarrow u \Rightarrow e'_1 \quad \text{noeffect}(e_2) \\
\Gamma \vdash (e_1 @ e_2) : \tau \Rightarrow (e'_1 @ u d^{r_2})
\]

\[
\Gamma \vdash e_1 : \tau_2 \Rightarrow n \Rightarrow e'_1 \quad \Gamma \vdash e_2 : \tau'_2 \Rightarrow e'_2 \quad \tau'_2 \leq \tau_2 \\
\Gamma \vdash (e_1 @ e_2) : \tau \Rightarrow (e'_1 @ n e'_2)
\]

The useless application rule specifies the circumstances under which an operand may be replaced by a dummy variable. In order for an operand to be useless, the type of the operator must be annotated with \( u \). However knowing that the operator's parameter is useless is not sufficient enough to determine whether the operator is useless. As discussed in Section 3.1.1, we must take into account the possibility that a side effect may occur in the operand. The predicate \( \text{noeffect}(e) \) holds if the expression \( e \) is guaranteed to be free of side effects as determined by a pre-computed effect analysis.

If the operator of an application \( (e_1) \) has a useless parameter and the operand \( (e_2) \) is guaranteed to be effect-free, then the application \( (e_1 @ e_2) \) can be translated to \( (e'_1 @ u d^{r_2}) \), where \( e'_1 \) is the translated form of \( e_1 \). Note that since \( e_2 \) is deemed useless, it is not translated. In the translated application, the operand is replaced with a dummy variable annotated with the operand's type. The application itself is annotated with \( u \), identifying the entire expression as a useless application. Recall that the operational
semantics ignores the operand in a useless application and simply evaluates the body of the operator.

The second application rule applies when the constraints of the first rule do not hold. In this case, the annotation on the type of the operator must be $n$. Note that there is no reference to effects in this rule. This is because any application can be considered needed regardless of whether the operand has a side-effect or not. The application $(e_1 \odot e_2)$ is translated to $(e_1' \odot_n e_2')$, where $e_1'$ and $e_2'$ are the translated forms of the operator and operand, respectively. The application is annotated with $n$, identifying the entire expression as a normal (needed) application.

The second application rule makes use of the subtyping relation defined earlier. This ordering on types allows for greater precision in our analysis (i.e. allows more uselessly variables to be identified) by not allowing unnecessarily strict type constraints to force $u$ annotations to become $n$ annotations. We present a specific example illustrating the benefit of subtyping in our system in the following section.

There is no rule for translating dummy variables because dummy variables do not appear in the source language.

The formal specification of UVE in Figure 3.2 is correct with respect to both the static and dynamic semantics of the source and target languages. This means that if some expression $e$ translates to $e'$, then both $e$ and $e'$ have the same type as well as the same operational behavior. Details concerning the specification’s correctness, including the formal statement of the operational correctness theorem and the corresponding induction proof, can be found in the technical report [10].
3.1.4 Examples of UVE

In this section I discuss five example programs, each one contrived to illustrate a certain aspect of our transformation. In each case I present the source expression and the UVE translated target expression. In some of the cases I also provide a partial derivation of the transformation using the inference rules described in the previous section.

For clarity, I use simple let expressions in the following examples. The expression let $f = \lambda x.e_1$ in $e_2$ is syntactic sugar for the application $(\lambda f.e_2) \odot (\lambda x.e_1)$. This let expression should not be confused with the letrec expression described in Chapter 2.

Example 1

$$(\lambda x.0) \odot e$$

We start with a very simple example that illustrates the detection of a useless parameter and the elimination of a useless operand. In the application above, the parameter $x$ is clearly useless since it does not appear in the body of the abstraction. Thus the type of the abstraction is $\tau \rightarrow_{u} \text{int}$. Assuming the expression $e$ has no effect, the translated form of the source expression is a useless application in which $e$ has been replaced with a dummy variable tagged with type $\tau$ (the type of $e$):

$$(\lambda x.0) \odot_{u} d^{\tau}$$
\[ \Gamma[x: \tau] \vdash 0 : \text{int} \Rightarrow 0 \quad x \notin FV(0) \]
\[ \Gamma \vdash \lambda x.0 : \tau \Rightarrow^u \text{int} \Rightarrow \lambda x.0 \quad \text{noeffect}(e) \]
\[ \Gamma \vdash (\lambda x.0) @ e : \text{int} \Rightarrow (\lambda x.0) @^d \tau \]

Fig. 3.3. Derivation for Example 1

The derivation for this simple transformation is presented in Figure 3.3. The derivation is constructed using the rules in Figure 3.2.

**Example 2**

let \( f = \lambda x.0 \)

in \( f @ (e_1+e_2) \)

This example illustrates how the presence of a side-effect can restrict the detection of useless parameters. The function variable \( f \) is bound by the let expression to the same abstraction used in the previous example. In Example 1, the parameter \( x \) was detected as useless, so it is not unreasonable to expect that the parameter will also be useless in this example. After all, \( x \) does not appear in the body of the abstraction. So we would like the type of \( f \) to be \( \text{int} \rightarrow^u \text{int} \). Figure 3.4 gives a derivation for the application \( f @ (e_1+e_2) \) where \( \Gamma \) is a type context that includes the binding \( \{ f : \text{int} \rightarrow^u \text{int} \} \).

The problem with this derivation is that the antecedent of the useless application rule requires that the expression \((e_1+e_2)\) have no effect. However, suppose the pre-computed effect analysis determined that the expression \((e_1+e_2)\) could cause an
overflow error. In this case the expression has a potential side-effect and the predicate `noeffect(e_1+e_2)` is false. This means that the derivation in Figure 3.4 is incorrect and the transformed application cannot be \( f \circ_{u} d^{\text{int}} \).

Since we must use the `needed` application rule from Figure 3.2, the type of \( f \) has to be annotated with \( n \). The correct derivation for this example is in Figure 3.5. Here, \( \Gamma' \) is a type context that includes the binding \( \{ f : \text{int} \rightarrow_{n} \text{int} \} \). The presence of a side-effect in this example forces the parameter \( x \) to be considered `needed` even though it does not appear in the body of the abstraction. Furthermore, we are unable to eliminate the operand and must translate the above example to:

\[
\text{let } f = \lambda x.0 \\
\text{in } f \circ_{n} (e'_1+e'_2)
\]
\[
\Gamma'(f) = \text{int} \rightarrow_n \text{int} \quad : \quad \vdash \Gamma' \succ e_1 : \text{int} \Rightarrow e'_1 \quad \vdash \Gamma' \succ e_2 : \text{int} \Rightarrow e'_2 \\
\Gamma' \succ f : \text{int} \rightarrow_n \text{int} \Rightarrow f \\
\Gamma' \succ \text{int} \leq \text{int} \\
\Gamma' \succ f \circ (e_1 + e_2) : \text{int} \Rightarrow f \circ_n (e'_1 + e'_2)
\]

Fig. 3.5. Correct derivation for Example 2

**Example 3**

\[
\text{let } f = \lambda x.e_1 \\
g = \lambda y.f \circ y \\
in g \circ e_2
\]

In this example we assume that \( x \) is a useless parameter and show how our UVE system then concludes that the expression \( e_2 \) is a useless operand. The annotated type of the function \( f \) is \( \tau_1 \rightarrow_u \tau_2 \). This means that, since \( f \) has a useless parameter, any operand that \( f \) is applied to is a useless operand (provided the operand has no side-effect). In the above example, \( f \) is applied to \( y \) in the body of the function \( g \). Since \( y \) is a variable, it clearly has no side-effect. Thus \( y \) is a useless operand.

The function \( g \) can be transformed into the following:

\[
g = \lambda y.f \circ_u d^{\tau_1}.
\]
\[
\begin{align*}
\Gamma \{ f : \tau_1 \rightarrow_u \tau_2 \} \{ y : \tau_1 \} (f) &= \tau_1 \rightarrow_u \tau_2 \\
\Gamma \{ f : \tau_1 \rightarrow_u \tau_2 \} \{ y : \tau_1 \} \triangleright f : \tau_1 \rightarrow_u \tau_2 \Rightarrow f & \text{ noeffect}(y) \\
\Gamma \{ f : \tau_1 \rightarrow_u \tau_2 \} \{ y : \tau_1 \} \triangleright f \circ y : \tau_1 \rightarrow_u \tau_2 \Rightarrow f \circ y & \text{ noeffect}(y) \quad y \notin FV(f \circ u d^{\tau_1}) \\
\Gamma \{ f : \tau_1 \rightarrow_u \tau_2 \} \triangleright \lambda y. f \circ y & \triangleq \lambda y. f \circ y \rightarrow_u \tau_2 \Rightarrow \lambda y. f \circ y & \text{ noeffect}(y) \quad y \notin FV(f \circ u d^{\tau_1})
\end{align*}
\]

Fig. 3.6. Derivation of Example 3

Although the parameter \( y \) appears in the body of the abstraction \( \lambda y. f \circ y \), it does not appear in the body of the translated abstraction. So \( y \) is a useless parameter.

The derivation in Figure 3.6 illustrates how the useless abstraction rule from Figure 3.2 detects \( y \) as useless when translating the abstraction \( \lambda y. f \circ y \).

Since the type of \( g \) is found to be \( \tau_1 \rightarrow_u \tau_2 \), the operand \( e_2 \) is useless if it has no side-effect. The entire example can be translated to the expression below (where \( e_1' \) is the translated form of \( e_1 \)).

\[
\begin{align*}
\text{let } f &= \lambda x. e_1' \\
g &= \lambda y. f \circ y \rightarrow_u \tau_2 \Rightarrow \lambda y. f \circ y \\
in g \circ u d^{\tau_1}
\end{align*}
\]

Example 4

\[
\begin{align*}
\text{let } f &= \lambda x. e_1 \\
g &= \lambda h. h \circ e_2 \\
in (g \circ f) + (f \circ e_3)
\end{align*}
\]
\[
\begin{align*}
\Gamma(g) &= (\tau \rightarrow_{n} \text{int}) \rightarrow_{n} \text{int} & \Gamma(f) &= \tau \rightarrow_{u} \text{int} \\
\Gamma \triangleright g : (\tau \rightarrow_{n} \text{int}) \rightarrow_{n} \text{int} & \Rightarrow g & \Gamma \triangleright f : \tau \rightarrow_{u} \text{int} & \Rightarrow f \\
(\tau \rightarrow_{u} \text{int}) & \leq (\tau \rightarrow_{n} \text{int}) & \Gamma \triangleright (g @ f) : \text{int} & \Rightarrow (g @ f)
\end{align*}
\]

Fig. 3.7. Derivation of Example 4 - first application

This example illustrates the benefit of subtyping in our UVE system. We make the following assumptions about this example:

- In the function \( f \), the parameter \( x \) is useless.
- The expression \( e_2 \) has a side-effect.
- The expression \( e_3 \) does not have a side-effect.
- The expression \( e_1 \) has type \( \text{int} \) and the expressions \( e_2 \) and \( e_3 \) both have type \( \tau \).

Since \( x \) is useless, the type of \( f \) should be \( \tau \rightarrow_{u} \text{int} \). In the function \( g \), the parameter \( h \) should have type \( \tau \rightarrow_{n} \text{int} \) since \( e_2 \) has an effect. Without subtyping this presents a significant problem. Because \( h \) is bound to \( f \) (by the application \( g @ f \)), \( h \) and \( f \) would have to have the same type. This would force the type of \( f \) to be \( \tau \rightarrow_{n} \text{int} \), failing to detect \( x \) as useless. The immediate consequence of this is that the effect-free expression \( e_3 \) is not detected as a useless operand. Thus, despite the fact that \( f \) does not actually need the value of its parameter, \( e_3 \) will be evaluated, resulting in a completely useless computation.

We avoid this problem by including the subtyping relation explained in Section 3.1.2. The proper derivation for the translation of the application \((g @ f)\) is
\[
\Gamma(f) = \tau \rightarrow^u \text{int} \\
\begin{align*}
\Gamma \vdash f & : \tau \rightarrow^u \text{int} \Rightarrow f \text{ noeffect}(e_3) \\
\Gamma \vdash (f \ @ e_3) & : \text{int} \Rightarrow f \ @^u d^T \end{align*}
\]

Fig. 3.8. Derivation of Example 4 - second application

detailed in Figure 3.7. The context \(\Gamma\) includes the bindings \(\{ f : \tau \rightarrow^u \text{int} \}\) and \(\{ g : (\tau \rightarrow^n \text{int}) \rightarrow^n \text{int} \}\). Because of the subtyping relation, \(f\) can have the type \(\tau \rightarrow^u \text{int}\) despite the fact that the type of \(g\)'s parameter is \(\tau \rightarrow^n \text{int}\). As is shown below, \(\tau \rightarrow^u \text{int}\) is a subtype of \(\tau \rightarrow^n \text{int}\):

\[
\frac{\tau \leq \tau \quad \text{int} \leq \text{int} \quad u \leq_a n}{(\tau \rightarrow^u \text{int}) \leq (\tau \rightarrow^n \text{int})}.
\]

In the application \((f \ @ e_3)\), since \(f\) has type \(\tau \rightarrow^u \text{int}\) and \(e_3\) is effect free, the application can be translated to \((f \ @^u d^T)\), properly detecting \(e_3\) as a useless operand. The derivation for this translation is presented in Figure 3.8.

The entire let expression can be translated as follows (where \(e_1'\) and \(e_2'\) are the translated forms of \(e_1\) and \(e_2\), respectively):

\[
\text{let } f = \lambda x.e_1' \\
g = \lambda h.h \ @^n e_2' \\
in (g \ @^n f) + (f \ @^u d^T).
\]
Example 5

\[ f = \lambda x. \lambda y. \lambda z. y \]

in \( f @ e_1 @ e_2 @ e_3 \)

There is often more than one way to translate a given source term. Consider the example above. Assuming the expressions \( e_1, e_2 \) and \( e_3 \) are effect-free, the function \( f \) has two useless parameters: \( x \) and \( z \). This suggests that the operands \( e_1 \) and \( e_3 \) are useless and can be replaced by dummy variables. Although this is true, the two operands do not have to be considered useless. Recall that any useless parameter can also be considered needed. The applications in the example above can be translated, using the rules in Figure 3.2, to any of the following expressions (where \( e'_1, e'_2 \) and \( e'_3 \) are the translated forms of \( e_1, e_2 \) and \( e_3 \), and \( \tau_1 \) and \( \tau_3 \) are the annotated types of \( e_1 \) and \( e_3 \)): 

- \( f @ u d_1 @ n e'_2 @ u d_3 \)
- \( f @ u d_1 @ n e'_2 @ n e'_3 \)
- \( f @ n e'_1 @ n e'_2 @ u d_3 \)
- \( f @ n e'_1 @ n e'_2 @ n e'_3 \)

The first target expression is typically what we would want. It detects and eliminates the most useless code. However the other three transformations are also valid in our system. Note that the last target expression is exactly what we started with (except for the translated subterms). The UVE system described in Section 3.1.3 defines what
transformations are possible. It is the purpose of an algorithm to actually compute a specific transformation.

3.2 Algorithm

I present here an algorithm, based on the rules in Figure 3.2, that detects and removes as many useless operands as possible. In other words, the algorithm infers annotations wherever possible. This is done by using the needed parameter abstraction rule only when constraints prevent the algorithm from using the useless parameter abstraction rule.

The UVE algorithm is divided into four phases: type inference, analysis and annotation, constraint solving, and translation. The first phase is traditional type inference and so we do not explicitly define it here. The analysis and annotation phase expects a well-typed input expression, where the expression and all subexpressions are explicitly tagged with their inferred types. For example, the expression

\[ ((\lambda f.(\lambda x.((f \, @ \, x)+(f \, @ \, x)))) \, @ \, (\lambda y.y)) \]

is explicitly tagged via type inference as

\[ ((\lambda f^{\text{int} \rightarrow \text{int}}.(\lambda x^{\text{int}}.((f^{\text{int} \rightarrow \text{int}} \, @ \, x^{\text{int}})^{\text{int} \rightarrow \text{int}}) + \text{int}^{\text{int} \rightarrow \text{int} \rightarrow \text{int}})) \, @ \, (\lambda y^{\text{int} \rightarrow \text{int}}.y^{\text{int} \rightarrow \text{int}})^{\text{int} \rightarrow \text{int} \rightarrow \text{int}}) \]
The inferred type must contain annotations on function arrows. Since it is up to the UVE algorithm to determine the values of the annotations, type inference merely assigns annotation variables to function arrows. We extend the grammar of annotations to include such annotation variables:

\[ a ::= u \mid n \mid \gamma \]

The expression above, when annotation variables are included, is

\[
((\lambda f^{\text{int} \to \gamma_1^{\text{int}}}.(\lambda x^{\text{int}}.((f^{\text{int} \to \gamma_1^{\text{int}}} @ x)^{\text{int}})^{\text{int}})^+ +
(f^{\text{int} \to \gamma_1^{\text{int}}} @ x)^{\text{int}})^{\text{int} \to \gamma_2^{\text{int}}})(\text{int} \to \gamma_1^{\text{int}}) \to \gamma_3^{\text{int} \to \gamma_2^{\text{int}}}
@ (\lambda y^{\text{int}}.y)^{\text{int} \to \gamma_4^{\text{int}}})^{\text{int} \to \gamma_2^{\text{int}}}
\]

Notice that parameter \( f \) will be bound to the function \( \lambda y.y \), yet the types of \( f \) and \( \lambda y.y \) have different annotations. This is due to the subtyping described earlier. The UVE algorithm will enforce the constraint \( \gamma_4 \leq \gamma_1 \).

Given such an expression tagged with annotated types, the analysis and annotation phase, followed by constraint solving and translation, produces a translated form of the input expression in which applications are annotated with \( n \) or \( u \) and useless operands are replaced with dummy variables. These three phases of the UVE algorithm are described in detail below.
\[ U(x^\gamma) = (\{ x^\emptyset \}, \{ e \}, x) \]

\[ U(n^{\text{int}}) = (\{ e \}, \{ n \}) \]

\[ U((e_1^{\tau_1 \rightarrow \gamma \tau_2} \circ e_2^{\tau_1})_{\tau_2}) = \]

let \( (\Theta_1, \Phi_1, e'_1) = U(e_1^{\tau_1}) \)

\( (\Theta_2, \Phi_2, e'_2) = U(e_2^{\tau_1}) \)

\( \Phi_3 = \Phi_2 \cup \text{getorder}(e_2^{\tau_1}, \tau_1) \)

\( \Phi_4 = \text{if noeffect}(e_2) \text{ then } \{ \} \)

else \( \{ \langle \gamma = n \rangle \emptyset \} \)

in \( (\Theta_1 \cup \gamma \Theta_2, \Phi_1 \cup \gamma \Phi_3 \cup \Phi_4, e'_1 \circ \gamma, e'_2) \)

\[ U((\lambda x^{\tau_1} e^{\tau_2})^{\tau_1 \rightarrow \gamma \tau_2}) = \]

let \( (\Theta, \Phi, e') = U(e^{\tau_2}) \)

in \( \Theta \setminus \{ x \in \Theta : \gamma = n \emptyset \} \cup \Phi, \lambda x. e' \)

\[ U((\mu f^{\tau}. e^{\tau})^{\tau}) = \]

let \( (\Theta, \Phi, e') = U(e^{\tau}) \)

in \( \Theta \setminus \{ f \}, \Phi, \mu f. e' \)

\[ U((e_1^{\text{int}} + e_2^{\text{int}})^{\text{int}}) = \]

let \( (\Theta_1, \Phi_1, e'_1) = U(e_1^{\text{int}}) \)

\( (\Theta_2, \Phi_2, e'_2) = U(e_2^{\text{int}}) \)

in \( (\Theta_1 \cup \Theta_2, \Phi_1 \cup \Phi_2, e'_1 + e'_2) \)

---

**Fig. 3.9.** Analysis and annotation phase

### 3.2.1 Analysis and Annotation

The algorithm for the analysis and annotation phase, which is inspired by Hannan and Hicks' algorithm for higher-order unCurrying [14] (which, in turn, was inspired by Milner's algorithm for polymorphic type inference [19]), is presented in Figure 3.9. Given a well-typed input term \( e^{\tau} \), the algorithm \( U \) returns the triple \( (\Theta, \Phi, e') \). The first element of the triple is the set of free variables in \( e \) that are considered needed.

The second element is a set of constraints on annotations. This is the set that will be solved by the next phase of the UVE algorithm. Finally, the third element of the triple is the expression \( e' \), which is an annotated form of \( e \). The only difference between \( e \) and \( e' \) is that, in \( e' \), each application is annotated with an annotation variable and type.
information has been removed.\textsuperscript{2} No code is actually removed in this phase. In fact, until the constraints in $\Phi$ are solved, the values of the annotation variables in $e'$ are unknown. We examine each case of algorithm $\mathcal{U}$ in detail below.

\textbf{Constants and Variables}

Constant integers and variables, not surprisingly, present the simplest cases. The integer case, in particular, is quite trivial. When given the expression $n^{\text{int}}$, algorithm $\mathcal{U}$ simply returns the triple ($\{\}$, $\{\}$, $n$). Both the sets $\Theta$ and $\Phi$ are empty since a constant clearly contains no variables, nor does it impose any constraint on annotations. Since there are no applications in a constant, the annotated form of $n^{\text{int}}$ is $n$.

The case for a variable is nearly as trivial. As with a constant, a variable imposes no constraints on annotations, hence $\Phi$ is empty. Similarly, as with a constant, a variable contains no applications, so the annotated form of $x^\tau$ is $x$. However, unlike with a constant, the set $\Theta$ is not empty. Recall that this set contains the free variables in the input expression that are considered needed. If the input expression is $x^\tau$, then clearly the set of free-variables considered needed in this expression is $\{x\}$. The set $\Theta$ in this case is actually $\{x^{\emptyset}\}$. The purpose of the superscript $\emptyset$ will be made clear when discussing the case for application below.

\textsuperscript{2}Removing type information is not actually necessary. Later phases of the compiler might actually need the type information intact. We remove the types here only to simplify the presentation.
Recursion and Addition

The recursion rule in Figure 3.2 suggests that translating a recursive expression $\mu f. \lambda x. e$ is merely a matter of translating the subexpression $\lambda x. e$. This is true for the algorithm as well. In Figure 3.9 a recursive expression is analyzed and annotated by analyzing and annotating the subexpression. Given the expression $(\mu f^\tau. e^\tau)^\tau$, algorithm $\mathcal{U}$ recursively calls itself on the subexpression $e^\tau$, resulting in the triple $(\Theta, \Phi, e')$. Algorithm $\mathcal{U}$ then returns the triple $(\Theta\setminus f, \Phi, \mu f, e')$. The operation $\Theta\setminus f$ is the set $\Theta$ with all occurrences of $f$ removed and is formally defined below.

**Definition 4.** For a variable set $\Theta$ and expression variable $x$, the set $\Theta\setminus x$ is defined as:

\[
\emptyset \setminus x = \emptyset \\
(x^\Delta \cup \Theta) \setminus x = \Theta \setminus x \\
(y^\Delta \cup \Theta) \setminus x = \{y^\Delta \} \cup \Theta \setminus x, \text{where } x \neq y
\]

Given an addition expression $(e_1^{\text{int}} + e_2^{\text{int}})^{\text{int}}$, algorithm $\mathcal{U}$ recursively calls itself on the subterms $e_1^{\text{int}}$ and $e_2^{\text{int}}$, resulting in the triples $(\Theta_1, \Phi_1, e_1')$ and $(\Theta_2, \Phi_2, e_2')$. The triple returned by $\mathcal{U}$ for the addition expression is then derived by computing the union of the two sets of free variables considered needed and by computing the union of the two sets of constraints: $(\Theta_1 \cup \Theta_2, \Phi_1 \cup \Phi_2, e_1' + e_2')$.

Abstraction

The inference rules in Figure 3.2 define a general transformation, so there can be two different rules for abstraction. A deterministic algorithm must have only one case
for each form of an expression. It is here that specific decisions must be made regarding how an input term will be translated. Since the algorithm should detect as much useless code as the rules in Figure 3.2 allow, the abstraction case of algorithm $U$ should only force a variable to be needed if the condition $x \notin FV(e')$ does not hold.

As in the previously discussed cases, given the expression $(\lambda x^{\gamma_1} . e^{\gamma_2})^{\gamma_1} \rightarrow^{\gamma_2}$, algorithm $U$ recursively calls itself on the subexpression $e^{\gamma_2}$, resulting in the triple $(\Theta, \Phi, e')$. The abstraction case then introduces the constraint $\langle x \in \Theta \cup \gamma = n \rangle$. This constraint says that, if the parameter $x$ is a free variable considered needed in expression $e'$, then the parameter must be needed. The abstraction case imposes no additional constraint on the annotation $\gamma$. Thus the triple returned by $U$ for the abstraction is $(\Theta \setminus x, \langle x \in \Theta \cup \gamma = n \rangle^{\emptyset} \cup \Phi, \lambda x.e')$. Algorithm $U$ cannot determine whether $x$ is in $\Theta$ because, until constraints are solved in the next phase of the UVE algorithm, the exact contents of $\Theta$ are unknown.

**Application**

Although there are two application rules in Figure 3.2, there is only one application case in algorithm $U$. Given the application $(e^{\gamma_1 \rightarrow^{\gamma_2} \gamma_2} @ e_2')^{\gamma_2}$, algorithm $U$ recursively calls itself on the subterms of the application resulting in the triples $(\Theta_1, \Phi_1, e_1')$ and $(\Theta_2, \Phi_2, e_2')$. However $U$ does not simply compute the union of the sets and return the triple $(\Theta_1 \cup \Theta_2, \Phi_1 \cup \Phi_2, e_1' \oplus \gamma e_2')$. Why not? The expression $e_2'$ might be useless.

If $e_2'$ is useless, it will be removed by a later phase of the UVE algorithm, and so should not impose any constraints on annotation variables. Basically, if $e_2'$ is useless, the sets $\Theta_2$ and $\Phi_2$ should be ignored. So if $e_2'$ is useless, algorithm $U$ should return the
triple \((\Theta_1, \Phi_1, e'_1 \oplus \gamma e'_2)\). On the other hand, if \(e'_2\) is needed, then algorithm \(U\) should return the triple \((\Theta_1 \cup \Theta_2, \Phi_1 \cup \Phi_2, e'_1 \oplus \gamma e'_2)\). The problem is that, until constraints are solved, there is no way to determine whether \(e'_2\) is useless or not.

Since \(U\) cannot determine whether to compute the union of \(\Theta_1\) and \(\Theta_2\), \(U\) computes the *conditional* union of \(\Theta_1\) and \(\Theta_2\). This conditional union is represented by the operation \(\Theta_1 \cup \gamma \Theta_2\), where the annotation \(\gamma\) represents the need of the operator’s parameter. Every variable in a \(\Theta\) set is superscripted with a set of annotations. The operation \(\gamma \Theta_2\) adds the annotation \(\gamma\) to each set superscripting a variable in \(\Theta_2\). For example, if \(\Theta_2 = \{x^0, y^{\gamma'}\}\) then \(\gamma \Theta_2 = \{x^\gamma, y^{\gamma\gamma'}\}\).

**Definition 5.** For variable set \(\Theta\) and annotation \(\gamma\), the set \(\gamma \Theta\) is defined as:

\[
\gamma^\emptyset = \emptyset \\
\gamma(\{x^\Delta\} \cup \Theta) = \{x^{\{\gamma\} \cup \Delta}\} \cup \gamma \Theta
\]

The sets of annotations superscripted to variables in \(\Theta\) are used by the constraint solver to determine which variables are actually considered members of \(\Theta\). If the constraint solver determines that any annotation in a variable’s annotation set is \(u\), then that variable should not be in the set.

Suppose the set \(\Theta_1\) is \(\{z^0\}\) and \(\Theta_2\) is as above. The conditional union \(\Theta_1 \cup \gamma \Theta_2\) is the set \(\{z^0, x^\gamma, y^{\gamma\gamma'}\}\). Now suppose the constraint solver determines that \(\gamma\) is \(u\) and \(\gamma'\) is \(n\). The conditional union is now \(\{z^0, x^u, y^{u,n}\}\). Since \(\gamma\) is \(u\), the set \(\Theta_2\) should not be included in the union. By removing variables with \(u\) annotations in their sets,

\[3\text{Recall that the variable rule returns the set } \{x^0\} \]
we end up with \{z\}, which is exactly what we want. If the constraint solver determines that \( \gamma \) is \( n \), then \( \Theta_2 \) should be included in the union. In this case, none of the variables have \( u \) in their sets, so the union gives is \( \{ z, x, y \} \).

Now consider the following expression:

\[
h^{\text{int} \to \gamma_1 \text{int}} @ ((f^{\text{int} \to \gamma_2 \text{int}} @ x^{\text{int}}) + (g^{\text{int} \to \gamma_3 \text{int}} @ x^{\text{int}}))
\]

The set \( \Theta \) for this expression is:

\[
\{h^\emptyset, f^{\{\gamma_1\}}, x^{\{\gamma_1, \gamma_2\}}, g^{\{\gamma_1\}}, x^{\{\gamma_1, \gamma_3\}}\}
\]

The inclusion of the variables \( f, g, \) and \( x \) in the set \( \Theta \) for this expression depend upon the value of the annotation \( \gamma_1 \), which represents the need of \( h \)'s parameter. Thus, in the set \( \Theta \), the annotation \( \gamma_1 \) is added to the annotation sets of the variables \( f, g, \) and \( x \). Notice that there are two occurrences of \( x \) in \( \Theta \). This is due to the fact that there are two occurrences of \( x \) in the expression. Each \( x \) is superscripted with a different set of annotations because it is possible for one occurrence of \( x \) to be needed while the other is useless. For example, the type of \( g \) might be \( \text{int} \to \gamma_u \text{int} \). Since \( \gamma_3 \) is \( u \) in this case, the second occurrence of \( x \) is useless and should not be included in \( \Theta \). But, if both \( \gamma_1 \) and \( \gamma_2 \) turn out to be \( n \), then the first occurrence of \( x \) should be in \( \Theta \).

The same concept applies to the constraint set \( \Phi \). Constraints in \( \Phi \) are also superscripted with annotation sets. Any constraint with a \( u \) in its annotation set is a meaningless constraint and should not be solved.
Two forms of constraints on annotations are introduced by \( \mathcal{U} \)'s application rule: one for subtyping, and the other for effects. The constraint set generated by the function \( \text{getord}(\tau_1', \tau_1) \) enforces the subtyping relation \( \tau_1' \leq \tau_1 \).

**Definition 6.** For annotated types \( \tau_1 \) and \( \tau_2 \), the constraint set \( \text{getord}(\tau_1, \tau_2) \) is defined as:

\[
\text{getord}(\text{int, int}) = \emptyset \\
\text{getord}(\tau_1 \rightarrow \gamma_1, \tau_2 \rightarrow \gamma_2) = \{\langle \gamma_1 \leq \gamma_2 \rangle^0\} \cup \text{getord}(\tau_2, \tau_1) \cup \text{getord}(\tau_1', \tau_2')
\]

If the operand \( e_2 \) has an effect, then the constraint \( \langle \gamma = \eta \rangle^0 \) is added to the constraint set returned by \( \mathcal{U}((e_1 \rightarrow \gamma_1 \gamma_2 \oplus e_2)^{\tau_1}) \). Recall that, if the operand has a side effect, then the operator’s parameter must be needed.

### 3.2.2 Constraint Solving

Once the analysis and annotation phase concludes, the set of constraints \( \Phi \) computed by algorithm \( \mathcal{U} \) must be solved. The function \( \text{solve}(\Phi) \), defined in Figure 3.10, returns a substitution \( \delta \) that maps all annotation variables to either \( \eta \) or \( \omega \) according to the constraints in \( \Phi \).

The substitution \( \delta \) is a composition of annotation variable-to-annotation mappings. We use the notation \( \{\gamma \mapsto \eta\} \) to represent the mapping of annotation variable \( \gamma \) to annotation \( \eta \). Such a mapping can be considered a substitution function that, when applied to an annotated expression, returns that expression with all annotations variables replaced with the appropriate annotations. For example, applying
1. solve\( (x \in \Theta \supset \gamma = n)^{\{n\} \cup \Delta \cup \Phi} \) = solve\( (x \in \Theta \supset \gamma = n)^{\Delta \cup \Phi} \)

2. solve\( (x \in \{x,n\} \cup \Theta \supset \gamma = n)^{\{n\} \cup \Delta \cup \Phi} \) = solve\( (x \in \{x\} \cup \Theta \supset \gamma = n)^{\{n\} \cup \Delta \cup \Phi} \)

3. solve\( (x \in \{x\} \cup \Theta \supset \gamma = n)^{\{x\} \cup \Delta \cup \Phi} \) = solve\( (\gamma = n)^{\{x\} \cup \Delta \cup \Phi} \)

4. solve\( (\gamma = n)^{\{n\} \cup \Delta \cup \Phi} \) = solve\( (\gamma = n)^{\Delta \cup \Phi} \)

5. solve\( (\gamma = n)^{\{n\} \cup \Delta \cup \Phi} = \) solve\( (\Phi[n/\gamma]) \circ \{ \gamma \mapsto n \} \)

6. solve\( (n = n)^{\{n\} \cup \Delta \cup \Phi} = \) solve\( (\Phi) \)

7. solve\( (\gamma_1 \leq \gamma_2)^{\{n\} \cup \Delta \cup \Phi} = \) solve\( (\gamma_1 \leq \gamma_2)^{\Delta \cup \Phi} \)

8. solve\( (n \leq \gamma)^{\{n\} \cup \Delta \cup \Phi} = \) solve\( (\Phi[n/\gamma]) \circ \{ \gamma \mapsto n \} \)

9. solve\( (\gamma \leq n)^{\{n\} \cup \Delta \cup \Phi} = \) solve\( (\Phi) \)

10. solve\( (n \leq n)^{\{n\} \cup \Delta \cup \Phi} = \) solve\( (\Phi) \)

11. otherwise, solve\( (\Phi) = \delta_{\Phi} \)

Fig. 3.10. Solving constraints

the substitution \( \{ \gamma_2 = u \} \circ \{ \gamma_1 = n \} \) (where \( \circ \) denotes composition) to the expression \( (\lambda x.\lambda y. x) \circ_{\gamma_1} 1 \circ_{\gamma_2} 2 \) results in the expression \( (\lambda x.\lambda y. x) \circ_{\gamma_1} 1 \circ_{u} 2 \).

The special substitution \( \delta_{\Phi} \), used in Rule 11 of Figure 3.10, maps all annotation variables to \( u \). Note that applying the substitution \( \delta_{\Phi} \circ \{ \gamma = n \} \) to the expression \( f \circ_{n} x \) results in the expression \( f \circ_{u} x \) even though \( \delta_{\Phi} \) maps \( \gamma \) to \( u \).

There are three forms of annotation constraints introduced by \( U \):

- \( \langle x \in \Theta \supset \gamma = n \rangle^{\Delta} \)

- \( \langle \gamma = n \rangle^{\Delta} \)

- \( \langle \gamma_1 \leq \gamma_2 \rangle^{\Delta} \)

Rules 1-3 in Figure 3.10 deal with the first form of constraint, rules 4-6 deal with the second form, and rules 7-10 handle the third. Rule 11 applies when there are no more
constraints that can be solved. Since our UVE algorithm is meant to detect as much useless code as possible, the substitution returned by solve(Φ) only maps annotation variables to n when the constraints in Φ force it to. Rule 11 guarantees that all annotation variables that are not so constrained are mapped to u.

Before the constraint solver can actually solve a particular constraint, it must determine whether the constraint is relevant. If any annotation in a constraint’s annotation set turns out to be u, then that constraint is irrelevant and should be ignored. Therefore, the only constraints that should be solved are those whose annotation sets are empty or contain only n annotations. Rules 1, 4, and 7 remove n annotations from annotation sets so that all constraints that can be solved will eventually have empty annotation sets.

When a constraint’s annotation set is empty, the constraint solver can attempt to solve it. The first form of constraint forces the annotation γ to n, but only if the specified variable x is a member of the set Θ. Just checking for membership in Θ is insufficient to determine whether γ should be n. The variable x could be in Θ as a result of a conditional union. So the constraint solver must determine whether a variable is actually a member of Θ in the same way that it determines whether a constraint is relevant. Rule 2 removes n annotations from the variable’s annotation set. If the annotation set becomes empty, rule 3 solves the constraint by removing the condition x ∈ Θ, leaving the simple constraint 〈γ = n〉^Θ.

Rule 5 solves the constraint 〈γ = n〉^Θ by first mapping all instances of γ to n in the remainder of the constraint set. The constraint solver then solves the remainder of the constraint set and returns that substitution composed with the substitution mapping γ to n. Rule 6 simply eliminates a trivial constraint.
Rule 8 solves the constraint \( \langle n \leq \gamma \rangle^0 \) in the same fashion as rule 5, by mapping the annotation \( \gamma \) to \( n \). Rules 9 and 10 eliminate trivial constraints.

Rules 5 and 8 introduce mappings into the substitution returned by the constraint solver that force certain annotation variables to be substituted with \( n \). Rule 11 ensures that all other annotation variables (once all solvable constraints have been solved) are substituted with \( u \).

Consider the following expression:

\[
((\lambda x^{\text{int}}.((\lambda y^{\text{int}}.x^{\text{int}})^{\text{int}}\rightarrow\gamma_2^{\text{int}} @ x^{\text{int}})^{\text{int}}\rightarrow\gamma_1^{\text{int}} @ 1^{\text{int}})^{\text{int}})
\]

Algorithm \( \mathcal{U} \) returns the following triple:

- \( \Theta = \emptyset \)
- \( \Phi = \{ \langle x \in \{ x^0, x^2 \} \supset \gamma_1 = n \rangle^0, \langle y \in \{ x^0 \} \supset \gamma_2 = n \rangle^0 \} \)
- Annotated expression = \( ((\lambda x.((\lambda y.x) @ \gamma_2 x)) @ \gamma_1 1) \)

The constraint solver can apply rule 3 on the first constraint resulting in the constraint set:

\[
\{ \langle \gamma_1 = n \rangle^0, \langle y \in \{ x^0 \} \supset \gamma_2 = n \rangle^0 \}
\]

Rule 5 can then be applied to the first constraint, introducing the mapping \{ \( \gamma_1 \mapsto n \) \} and removing the first constraint from the constraint set. The second, remaining, constraint cannot be solved by any of the rules 1-10. Therefore, rule 11 returns the substitution \( \delta_u \), which, in this case, is \{ \( \gamma_1 = u \) \} \circ \{ \( \gamma_2 = u \) \}. The final substitution returned by the
\[
\begin{align*}
\text{translate}(n) &= n \\
\text{translate}(x) &= x \\
\text{translate}(\lambda x . e) &= \lambda x . \text{translate}(e) \\
\text{translate}(e_1 \oplus_\Phi e_2) &= \text{translate}(e_1) \oplus_\Phi \text{translate}(e_2) \\
\text{translate}(e_1 \oplus_\mathcal{U} e_2) &= \text{translate}(e_1) \oplus_\mathcal{U} \text{translate}(e_2) \\
\text{translate}(\mu f . x) &= \mu f . \text{translate}(e) \\
\text{translate}(e_1 + e_2) &= \text{translate}(e_1) + \text{translate}(e_2)
\end{align*}
\]

Fig. 3.11. The translate function.

The constraint solver when applied to $\Phi$ is:

$$\{\gamma_1 = u\} \circ \{\gamma_2 = u\} \circ \{\gamma_1 = n\}.$$

Applying this substitution to the annotated expression gives us the expression

$$((\lambda x . ((\lambda y . x) \oplus_\Phi x)) \oplus_\Phi 1),$$

which correctly identifies $y$ as a useless parameter.

3.2.3 Translation

Once the substitution returned by the constraint solver is applied to the annotated expression returned by algorithm $\mathcal{U}$, we have an expression in which all applications are annotated with either $u$ or $n$. The only remaining operation that needs to be performed by the UVE algorithm is the final translation: replacing useless parameters with dummy variables. This translation is defined in Figure 3.11.
The entire UVE algorithm can be stated by the following let expression, where $e^\tau$ is a well-typed source expression:

$$ UVE(e^\tau) = \text{let } (\Theta, \Phi, e') = U(e^\tau) \text{ in } \delta = \text{solve}(\Phi) \text{ in translate}(\delta e'). $$

### 3.2.4 Correctness

We would like to verify that the UVE algorithm is correct. In particular, we would like to show that the algorithm does not alter the behavior of the source expression. A compiler writer could then include UVE as an optimization phase in a compiler, knowing that an input program will have the same runtime behavior as the useless-variable eliminated form of the program.

Furthermore, since the transformation is designed for a typed language, we would also like to show that the UVE algorithm preserves the type of the input expression. This guarantees that any subsequent optimization phase of a compiler will receive a well-typed program as input provided that the UVE phase receives a well-typed program as input.

The formal specification of UVE defined in Section 3.1.3 is both operationally correct and type preserving [10]. Therefore, if I can show that every transformation allowed by the UVE algorithm is also allowed by the formal specification, then we know that the algorithm is operationally correct and type preserving. In other words, I can prove the correctness of the algorithm by proving that the algorithm is sound with respect to the inference rules in Figure 3.2.
The following definitions are required for the statement of the theorem below. First I define \(|e^T|\), which is the expression \(e\) with all explicit type tags removed. This is required since, in the rules in Figure 3.2, expressions are not tagged with their types.

DEFINITION 7. If \(e^T\) is a well-typed expression in which all subexpressions are explicitly tagged with their types, then \(|e^T|\) is defined as follows:

\[
\begin{align*}
|n^{\text{int}}| &= n \\
|x^T| &= x \\
|((\lambda x^{T_1}.e^{T_2})^{T_1\to T_2}| &= \lambda x.|e^{T_2}| \\
|((e_1^{T_2\to T_2} @ e_2^{T_1})^{T_2}| &= |e_1^{T_2\to T_2}| @ |e_2^{T_1}| \\
|((\mu f^T.e^T)^T| &= \mu f.|e^T| \\
|((e_1^{\text{int}}+e_2^{\text{int}})^{\text{int}}| &= |e_1^{\text{int}}|+|e_2^{\text{int}}|
\end{align*}
\]

So that I can state a relation between the set \(\Theta\) computed by algorithm \(U\) and the set \(FV(e)\) from Figure 3.2 (for some expression \(e\)), I define the function \(\text{simplify}(\Theta)\), which returns all of the variables in \(\Theta\) whose suffix sets do not contain the \(u\) annotation (i.e. all of the variables that should be considered actual members of the set \(\Theta\)).

DEFINITION 8. The function \(\text{simplify}(\Theta)\) returns the set of variables \(x_i\) such that \(x_i^\Delta \in \Theta\) and \(\Delta\) contains no occurrences of \(u\):

\[
\text{simplify}(\{x^\Delta\} \cup \Theta) = \begin{cases} 
\text{simplify}(\Theta) & u \in \Delta \\
\{x\} \cup \text{simplify}(\Theta) & \text{otherwise}
\end{cases}
\]

\(\text{simplify}(\emptyset) = \emptyset\)
Finally I can state the correctness theorem for the UVE algorithm. The theorem states that if the UVE algorithm translates \( e \) to \( e' \), then such a transformation is allowed by the rules in Figure 3.2.

**Theorem 3.1.** If \( U(e^\tau) = (\Theta, \Phi, e') \), \( \Phi \subseteq \Phi' \), solve(\( \Phi' \)) = \( \delta \), and translate(\( \delta e' \)) = \( e'' \) for any well-typed, annotated term \( e^\tau \) and constraint set \( \Phi' \), then

1. \( FV(e'') = \text{simplify}(\delta\Theta) \)
2. \( \Gamma \vdash |e^\tau| : \delta\tau \Rightarrow e'' \), for all \( \Gamma \) such that \( FV(e^\tau) \subseteq \text{dom}(\Gamma) \) and \( \Gamma(x) = \delta\tau' \) for all \( x' \) free in \( e^\tau \).

Part 1 of the theorem states that the UVE algorithm computes the set of free variables in the translated term, which is required in Part 2 to prove that the algorithm imposes the same constraints on annotations as the inference system in Section 3.1.3. The proof of both parts are by induction on the structure of \( e^\tau \) and can be found in Appendix A.

### 3.3 Related Work

Kobayashi [17, 18] uses a different approach to type-based useless-variable elimination. Rather than annotating types to represent the need of parameters and operands, his analysis replaces the types of useless expressions with the type of unit. In the expression \( (\lambda x.1 \circ e) \), the types of the operator and operand are unit \( \rightarrow \) int and unit, respectively. These types tell us that the abstraction's parameter as well as the expression \( e \) are useless. Kobayashi's transformation replaces any expression of type unit with the unit constant (\( ) \). A separate transformation (not explicitly defined) then removes all
unit constants from the expression. As a result, the above application becomes the integer expression 1. Such aggressive elimination of expressions is not always safe. In fact, Kobayashi’s transformation does not preserve non-termination or other side-effects. He does, however, suggest that a pre-computed effect analysis would remedy this problem. He also suggests that replacing some useless operands with dummy constants instead of () increases the amount of useless code that can be detected by his transformation. Our transformation uses both of these methods.\footnote{Although we replace all useless operands with dummy variables.}

Wand and Siveroni [28] define useless-variable elimination for an untyped functional language. In a typed program, if functions $f$ and $g$ can be called from the same call site then $f$ and $g$ will have the same type (or types related via subtyping). A UVE transformation must transform both functions in a consistent manner in order to maintain their types. Such types are not available in the language considered in [28]. Instead, Wand and Siveroni rely on dependencies derived from a control-flow analysis to guarantee consistent transformations. For each call site in a program, control-flow analysis computes the set of functions that can be called at that call site. A dependency analysis then determines which parameters each function actually needs to compute its result. Using the control-flow information, the dependency analysis guarantees that a function’s parameter is considered useless only if parameters in the same position are considered useless by all other functions that can be called at the same call sites as the function being analyzed. Although Wand and Siveroni prove that their analysis is correct, like Kobayashi their transformation does not preserve non-termination and other side-effects.
Control-flow analysis is unnecessarily complicated and computationally expensive for a problem such as useless-variable elimination. Essentially, as argued by Kobayashi in [18], a control-flow analysis computes too much information. For some function application $e_1 \circ e_2$, it is not necessary to know exactly which functions $e_1$ can evaluate to when the consistency of types guarantees that all such functions will have the same type as $e_1$ (relative to subtyping). Transforming an untyped program does not necessarily preclude the use of types in directing the transformation. For example, Hanan and Hicks [14] were able to modify their type-based higher-order unCurrying analysis for an untyped language.

Our UVE analysis uses simple types to detect useless parameters and operands. However we can extend the type system in order to increase the precision of the analysis. Consider the following example:

\[
\text{let } f = \lambda h. \lambda z. h \circ \text{zin} \ (f \circ (\lambda x.3) \circ e) + (f \circ (\lambda x.x) \circ 7)
\]

Using simple types, even with our subtyping relation, our specification cannot detect the expression $e$ as useless. Different methods have been proposed to deal with this problem: dependent types [9], conjunctive types [5] and let-polymorphism [17]. All three of these methods allow the expression $e$ to be detected as useless and replaced by a dummy variable by allowing the function $f$ to have different types.
Chapter 4

Lambda Lifting

4.1 Introduction

Lambda-lifting is a program transformation that eliminates the block structure of a functional program by “lifting” all functions to the global level. The transformation was originally developed independently by Johnsson [16] and Hughes [15] to aid in the compilation of lazy functional languages. The graph reduction method used at the time to compile call-by-name functional languages requires that the input program consist of a series of global function definitions followed by a main expression. Since the compiler writers wanted to allow programmers the convenience of writing programs with nested functions, compilers for lazy languages such as Haskell first lambda-lifted a program and then compiled the resulting transformed program using graph reduction. More recently, lambda-lifting has been used as a preliminary phase of partial evaluation [2].

Lambda-lifting consists of two separate phases: parameter-lifting and block-floating. The parameter-lifting phase closes all of the functions in a given program by introducing additional parameters to functions containing free variables. For example, the function \( \lambda x.x + y \) is parameter-lifted by introducing \( y \) as a new parameter: \( \lambda y.\lambda x.x + y \). The function must then be applied to the variable \( y \) whenever it is called. Once all functions have been closed in this fashion, the block-floating phase moves all function definitions to the global level. Since the block-floating phase of lambda-lifting, once functions are
closed, is nearly trivial, I focus exclusively on parameter-lifting in the remainder of this chapter.

Lambda-lifting has been described in terms of specific algorithms (e.g. [16, 15, 22, 23]). These algorithms necessarily describe exactly what parameters to introduce and when to apply parameter-lifted functions to their new arguments. As I illustrate later in this chapter, the higher-order nature of functional programming languages allows for a great deal of flexibility in parameter-lifting. None of the existing descriptions of lambda-lifting take advantage of these flexibilities.

The goal in this chapter is to formally describe the parameter-lifting transformation using a set of declarative inference rules that capture not only the parameter-lifting method originally described by Johnsson, but also other methods of parameter-lifting not described elsewhere. Furthermore I demonstrate how this formal description can be used to prove that a parameter-lifting algorithm is correct.

The flexibility of the inference rules allows programs to be partially parameter-lifted. A partial lambda-lifting strategy, combined with closure conversion, is used in [4] to compile programs in the functional language Scheme. Block-floating can also be employed in a partial fashion. Peyton-Jones, Partain and Santos detail the benefits of a form of partial block-floating in [24].

Danvy and Schultz [7] introduce a transformation called lambda-dropping that restores the block structure of a lambda-lifted program. Like lambda-lifting, this transformation consists of two phases: block-sinking and parameter-dropping. Since our specification describes parameter-lifting as a relation between programs and parameter-lifted forms of programs, it can also be considered a description of parameter-dropping.
In the remainder of this chapter, I introduce our formal description of parameter-lifting and then take advantage of this specification to prove the correctness of the parameter-lifting phase of Johnsson's lambda-lifting algorithm. The work presented in this chapter was completed in conjunction with John Hannan and is published as a conference paper [8] and journal article [11].

4.2 Formal Specification of Parameter Lifting

As we did with useless-variable elimination, we define a formal specification of the parameter-lifting transformation using inference rules. These rules relate expressions in our language to parameter-lifted forms of those expressions. An expression may have several parameter-lifted forms. We are not interested here in prescribing a particular parameter-lifting strategy, but rather in describing what parameter-lifting strategies are possible within the language's type system.

Since parameter-lifting does not introduce any new forms of expression, the source and target languages for the transformation are the same. We use the language described in Chapter 2, using letrec instead of μ to define recursive functions.

The parameter-lifting transformation I describe in this section is type-based, requiring us to augment the type system presented in Chapter 2 so that the types of expressions convey information pertaining to the parameter-lifting of functions. To this end, we make use of singleton types [1].
4.2.1 Singleton Types

Consider the function $\lambda x.x \@ y$. Since $y$ is a free variable in this expression, we can parameter-lift the function by introducing a binding for $y$, resulting in the new function $\lambda y.\lambda x.x \@ y$. Assuming the types of $x$ and $y$ are $\tau_1 \rightarrow \tau_2$ and $\tau_1$, respectively, then the type of the first function is $(\tau_1 \rightarrow \tau_2) \rightarrow \tau_2$. What is the type of the parameter-lifted function? Since our parameter-lifting specification is type-based, the type of the parameter-lifted function must identify the fact that $y$ has been lifted. We might consider using an annotation on the function arrow, like the annotations we used for useless-variable elimination. For example, we could decide that the type of the lifted function should be $\tau_1 \rightarrow_l (\tau_1 \rightarrow \tau_2) \rightarrow \tau_2$. The $l$ on the function arrow would indicate that the first parameter is a lifted parameter. However this does not tell us what the parameter’s name is. The type must convey the parameter’s name so that we can apply the lifted function to the appropriate variable. A simple annotation does not convey enough information. Instead, we use singleton types.

A singleton type is a type that is inhabited by one expression. The singleton type $\{e\}_{\tau}$ is only inhabited by the expression $e$ of type $\tau$. Using singleton types to convey information about parameter-lifting, the type of the lifted function above is $\{y\}_{\tau_1} \rightarrow (\tau_1 \rightarrow \tau_2) \rightarrow \tau_2$. The singleton type $\{y\}_{\tau_1}$ tells us not only that the first parameter has been lifted, but that the lifted parameter’s name is $y$. Since the only expression that inhabits the type $\{y\}_{\tau_1}$ is the variable $y$, whenever the lifted function is called, its first argument must be $y$. 
We extend the types introduced in Chapter 2 to include singleton types:

\[ \tau ::= \tau_1 \mid \tau \rightarrow \tau_1 \mid \{ x \}_\tau \rightarrow \tau \]

We restrict singleton types in our system so that they only appear to the left of function arrows, and are only inhabited by variables.

### 4.2.2 Inference Rules

The inference rules in Figure 4.1 define our parameter-lifting transformation. Using these rules, we can derive judgments of the form \( \Gamma \vdash e : \tau \Rightarrow e' \). The expression \( e \) is the source term and \( e' \) is a parameter-lifted form of \( e \). The type \( \tau \) is the type of \( e' \), possibly containing singleton types. The type context \( \Gamma \) maps variables to their types. The rules assume all variable names in \( e \) are unique.

Note that, because of the possible presence of singleton types, \( \tau \) may not be a valid type for the expression \( e \). For example, if \( e \) is \( \lambda x.x \circ y \) and \( e' \) is \( \lambda y.\lambda x.x \circ y \), then the type \( \tau \) is \( \{ y \}_{\tau_1} \rightarrow (\tau_1 \rightarrow \tau_2) \rightarrow \tau_2 \). Clearly, this is not a valid type for \( e \).

The rules in Figure 4.1 define a relation among expressions and their parameter-lifted forms. They also define a relation among expressions and their parameter-dropped forms. Since \( e' \) is a parameter-lifted form of \( e \), we can say that \( e \) is a parameter-dropped form of \( e' \).

I describe each of the rules in Figure 4.1 in detail in the remainder of this section.
\[
\begin{align*}
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \Rightarrow x} \quad \text{(var)}
\end{align*}
\]

\[
\frac{\Gamma\{y{:}\tau_1\} \vdash e : \tau \Rightarrow e' \quad y \notin FV(\tau_1 \rightarrow \tau)}{\Gamma \vdash \lambda y.e : \tau_1 \rightarrow \tau \Rightarrow \lambda y.e'} \quad \text{(abs)}
\]

\[
\frac{\Gamma(x) = \tau_1 \quad \Gamma \vdash \lambda y.e : \tau \Rightarrow \lambda z.e'}{\Gamma \vdash \lambda y.e : \{x\}_{\tau_1} \rightarrow \tau \Rightarrow \lambda x.\lambda z.e'} \quad \text{(lift-abs)}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \Rightarrow e'_1 \quad \Gamma \vdash e_2 : \tau_1 \Rightarrow e'_2}{\Gamma \vdash e_1 @ e_2 : \tau \Rightarrow e'_1 @ e'_2} \quad \text{(app)}
\]

\[
\frac{\Gamma \vdash e : \{x\}_{\tau_1} \rightarrow \tau \Rightarrow e' \quad \Gamma(x) = \tau_1}{\Gamma \vdash e : \tau \Rightarrow e' @ x} \quad \text{(lift-app)}
\]

\[
\frac{\Gamma^* = \Gamma\{f_i : \tau_i\}}{\Gamma \vdash \text{letrec } f_i = e_i \text{ in } e : \tau \Rightarrow \text{letrec } f_i = e'_i \text{ in } e'} \quad \text{if } i \in \{1..n\} \quad \text{(letrec)}
\]

Fig. 4.1. Parameter Lifting
Variables and Application

\[
\begin{align*}
\Gamma(x) &= \tau \\
\Gamma \triangleright x : \tau &\Rightarrow x \quad \text{(var)}
\end{align*}
\]

\[
\begin{align*}
\Gamma \triangleright e_1 : \tau_1 &\Rightarrow e'_1 \\
\Gamma \triangleright e_2 : \tau_1 &\Rightarrow e'_2 \\
\Gamma \triangleright e_1 \circ e_2 : \tau &\Rightarrow e'_1 \circ e'_2 \quad \text{(app)}
\end{align*}
\]

The variable and application rules are straightforward and contain no constraints specific to parameter-lifting. In fact, the variable rule is the same as that used for useless-variable elimination. The parameter-lifted form of a variable is simply the variable. The type of the variable must be looked up in the given context.

In order to parameter-lift the application \( e_1 \circ e_2 \), we must simply parameter-lift the subterms. So, if \( e'_1 \) and \( e'_2 \) are the parameter-lifted forms of \( e_1 \) and \( e_2 \), respectively, then the parameter-lifted form of the application \( e_1 \circ e_2 \) is \( e'_1 \circ e'_2 \).

Parameter Lifting

\[
\begin{align*}
\Gamma(x) &= \tau_1 \\
\Gamma \triangleright \lambda y.e : \tau &\Rightarrow \lambda z.e' \\
\Gamma \triangleright \lambda y.e : \{ x \}_{\tau_1} &\Rightarrow \lambda x.\lambda z.e' \quad \text{(lift-abs)}
\end{align*}
\]

\[
\begin{align*}
\Gamma \triangleright e : \{ x \}_{\tau_1} &\Rightarrow e' \\
\Gamma(x) &= \tau_1 \\
\Gamma \triangleright e : \tau &\Rightarrow e' \circ x \quad \text{(lift-app)}
\end{align*}
\]

The rules (lift-abs) and (lift-app) define the actual parameter-lifting operations. The (lift-abs) rule allows any variable currently in scope to be lifted in the abstraction. The rule also introduces the appropriate singleton type to identify the lifted parameter.

The abstraction \( \lambda y.e \) is translated in the antecedent of the rule to the parameter-lifted expression \( \lambda z.e' \). We use \( z \) as the parameter in the translated expression to account
for the fact that additional parameters may be lifted in the abstraction and the leftmost
\( \lambda \) might no longer bind \( y \). For example, assuming \( w \) is a variable in \( \Gamma \), the abstraction
\( \lambda y . e \) can be translated to \( \lambda w \cdot \lambda y . e'' \).

In the consequent of the rule, \( \lambda y . e \) is translated to \( \lambda x . \lambda z . e' \). The variable \( x \) has
been introduced as a lifted parameter. The only constraint concerning lifted parameters
is that they must be in scope (i.e., must be in the domain of \( \Gamma \)). The variable \( x \) may
or may not appear free in the abstraction \( \lambda y . e \). As far as our specification is concerned,
whether \( x \) appears in \( e \) is irrelevant. Any variable in scope may be lifted.

Deciding exactly which variables to lift is an algorithmic detail. For example,
Johnsson's lambda-lifting algorithm lifts exactly those variables (excluding function
names) that are free in the body of the given function. We wish our lambda-lifting
specification to be as general as possible. While our system does capture the transfor-
mation described by Johnsson (for a typed language), our system also allows several
other lambda-lifting strategies.

When a parameter is lifted in a function, a variable that is free in the function is
bound locally by introducing a new \( \lambda \) (as in the (lift-abs) rule). This operation is only
valid if the new abstraction is then applied to the variable that was free in the original
function. If \( y \) is lifted in the function \( \lambda x . x \odot y \), then the resulting function \( \lambda y . \lambda x . x \odot y \)
must be applied to \( y \) before it is applied to the argument that should be bound to \( x \).
The (lift-app) rule enforces this requirement.

If there is some expression \( e' \) that has type \( \{ x \}_{\tau_1} \rightarrow \tau \), then \( e' \) evaluates to a
function with a lifted parameter. The \( x \) in the singleton type reveals that \( x \) is the name
of the variable that was bound locally in the function. Thus, in the consequent of the
(lift-app) rule, the source expression \( e \) is translated to the new application \( (e' @ x) \), where \( e' \) is the parameter-lifted form of \( e \). The variable \( x \) must be in the type context. This guarantees that a lifted function is applied to its new argument while the argument is in scope. Since variable names are unique, there is no danger of applying a lifted function to the wrong variable.

If variable names were not unique then the following incorrect transformation would be derivable (assuming our language included a generic let expression):

\[
\begin{align*}
\text{let } x &= e_1 & \text{let } x &= e_1 \\
\text{in letrec } f &= \lambda y.x @ y & \Rightarrow & \text{in letrec } f = \lambda x.\lambda y.x @ y \\
& \text{in let } x = e_2 & \text{in let } x = e_2 \\
& \text{in } f @ e_3 & \text{in } f @ x @ e_3
\end{align*}
\]

The expression to the right of the arrow is a parameter-lifted form of the expression to the left. The function \( f \) is closed by introducing a binding for the free variable \( x \). Then \( f \) is applied to \( x \) before it is applied to \( e_3 \). The problem is that \( f \) is applied to the wrong \( x \). The \( x \) in the body of \( f \) is referring to the value of \( e_1 \), not the value of \( e_2 \).

\[\text{[1]}\text{Here and in later examples, lifted parameters and inserted operands are shown in bold type.}\]
The proper way to parameter-lift the above expression is to first $\alpha$-convert the expression so that variable names are unique. The following is a correct transformation:

\[
\begin{align*}
\text{let } \pi x \equiv e_1 & \implies \text{let } \pi x \equiv e_1 \\
\text{in } \text{letrec } f = \lambda y. x \odot y & \implies \text{in letrec } f = \lambda x. \lambda y. x \odot y \\
\text{in let } z = e_2 & \implies \text{in let } z = e_2 \\
\text{in } f \odot e_3 & \implies \text{in } f \odot x \odot e_3
\end{align*}
\]

The (lift-app) rule follows the same principle of generality as its companion, the (lift-abs) rule. The rule does not require $e$ to be an abstraction, it only requires that the type of $e$ has a function type with a singleton type to the left of the arrow. We present examples in Section 4.3 that illustrate ways in which the generality of these two rules allows for a number of different parameter-lifting strategies.

**Abstraction and Recursion**

\[
\begin{align*}
\frac{\Gamma; \tau_1 \vdash e : \tau \Rightarrow e' \quad y \notin FV(\tau_1 \rightarrow \tau)}{\Gamma \vdash \lambda y. e : \tau_1 \rightarrow \tau \Rightarrow \lambda y. e'} & \quad (\text{abs}) \\
\frac{\Gamma^* = \Gamma \{ \underline{f_1 : \tau_i} \}}{\Gamma^* \vdash e_i : \tau_i \Rightarrow e'_i} & \quad (\text{letrec})
\end{align*}
\]

The abstraction and recursion rules are similar to the typing rules in Chapter 2, except an additional constraint is added concerning lifted parameters.
The constraint imposed by the abstraction rule, \( y \notin FV(\tau) \), guarantees that lifted variables do not escape their scope. We define \( FV(\tau_1 \rightarrow \tau) \) as follows:

**Definition 9.** For a type \( \tau \), the set \( FV(\tau) \) is defined as:

\[
FV(\iota) = \emptyset \\
FV(\{x\}_\tau) = \{x\} \\
FV(\tau_1 \rightarrow \tau_2) = FV(\tau_1) \cup FV(\tau_2)
\]

We do not want \( y \) to appear in a singleton type in \( \tau_1 \rightarrow \tau \) because \( y \) does not exist outside the abstraction.

Likewise, the constraint, \( f_i \notin FV(\tau) \), in the recursion rule guarantees that none of the function names \( f_1 \cdots f_n \) escape their scope.

The specification of parameter-lifting described in this section is correct with respect to both the static and dynamic semantics of our language. Refer to [11] for the formal statement of correctness theorems and the corresponding induction proofs.

### 4.3 Examples of Parameter-Lifting

In this section, I discuss two examples that illustrate certain properties of our parameter-lifting specification. For the examples in this section, I assume that our language includes an addition operator.
4.3.1 Example 1

The first example involves mutual recursion. The expression below declares two functions, \( f \) and \( g \), that call each other. Each function contains a different free variable and we would like to parameter-lift the expression so that each function is closed.

\[
\text{letrec } f = \lambda x.g @ (x + y) \\
g = \lambda w.f @ (w + z)
\]

in \( c \)

Since \( y \) is free in function \( f \) and \( z \) is free in function \( g \), we can lift \( y \) in \( f \) and \( z \) in \( g \), resulting in the expression below (where \( c' \) is the parameter-lifted form of \( c \)).

\[
\text{letrec } f = \lambda y.\lambda x.g @ z @ (x + y) \\
g = \lambda z.\lambda w.f @ y @ (w + z)
\]

in \( c' \)

Since both \( f \) and \( g \) have new parameters, any calls to \( f \) and \( g \) must have new arguments. In particular, \( g \) is now applied to \( z \) and \( f \) is now applied to \( y \).

While the above expression is a valid parameter-lifted form of the original expression according to the rules in Figure 4.1, the two functions are not closed. The free variables \( z \) and \( y \) have been added to functions \( f \) and \( g \), respectively. We can
parameter-lift the expression above so that both $f$ and $g$ contain no free variables:

$$\text{letrec } f = \lambda z. \lambda y. \lambda x. g \ @ y \ @ z @ (x + y)$$

$$g = \lambda y. \lambda z. \lambda w. f \ @ z \ @ y @ (w + z)$$

in $e''$

Figures 4.2 and 4.3 show portions of the derivation used to transform the first
letrec expression into the third. We assume a type context $\Gamma$ containing the mappings
\{y : \text{int}\} and \{z : \text{int}\}. The context $\Gamma^*$ derived by the letrec rule from Figure 4.1 extends
$\Gamma$ to include the types of $f$ and $g$. Recall that the types in the inference rules refer to
the target expression, not the source expression. Since $z$ and $y$ are lifted parameters of
$f$ in the target expression, the type of $f$ is:

$$\{z\}_\text{int} \rightarrow \{y\}_\text{int} \rightarrow \text{int} \rightarrow \tau.$$  

Likewise, the type of $g$ in the target expression is:

$$\{y\}_\text{int} \rightarrow \{z\}_\text{int} \rightarrow \text{int} \rightarrow \tau.$$  

The derivation in Figure 4.2 illustrates how the body of function $f$ is transformed
from the application $g @ (x + y)$ into the application $g @ y @ z @ (x + y)$. Since the type
of $g$ in the context contains two singleton types, we use the (lift-app) rule to transform
$g$ into $g @ y$, followed by another instance of the (lift-app) rule to transform $g @ y$ into
\[
\begin{align*}
\Gamma^* \{ x : \text{int} \} (g) &= \{ y \}_{\text{int}} \rightarrow \{ z \}_{\text{int}} \rightarrow \text{int} \rightarrow \tau \\
\Gamma^* \{ x : \text{int} \} \triangleright g : \{ y \}_{\text{int}} \rightarrow \{ z \}_{\text{int}} \rightarrow \text{int} \rightarrow \tau \Rightarrow g \\
\Gamma^* \{ x : \text{int} \} \triangleright g : \{ z \}_{\text{int}} \rightarrow \text{int} \rightarrow \tau \Rightarrow g \otimes y \\
\Gamma^* \{ x : \text{int} \} \triangleright g : \text{int} \rightarrow \tau \Rightarrow g \otimes y \otimes z &\Rightarrow (x + y)
\end{align*}
\]

\[\vdash\]
\[
\begin{align*}
\Gamma^* \{ x : \text{int} \} \triangleright g \otimes (x + y) : \tau \Rightarrow g \otimes y \otimes z \otimes (x + y) \\
\Gamma^* \triangleright \lambda x.g \otimes (x + y) : \text{int} \rightarrow \tau \Rightarrow \lambda x.g \otimes y \otimes z \otimes (x + y) \\
\Gamma^* \triangleright \lambda x.g \otimes (x + y) : \{ y \}_{\text{int}} \rightarrow \text{int} \rightarrow \tau \Rightarrow \lambda y.\lambda x.g \otimes y \otimes z \otimes (x + y) \\
\Gamma^* \triangleright \lambda x.g \otimes (x + y) : \{ z \}_{\text{int}} \rightarrow \{ y \}_{\text{int}} \rightarrow \text{int} \rightarrow \tau \Rightarrow \lambda z.\lambda y.\lambda x.g \otimes y \otimes z \otimes (x + y)
\end{align*}
\]

Fig. 4.2. Example derivation using (lift-app)

\[\vdash\]
\[
\begin{align*}
\Gamma^* \{ x : \text{int} \} \triangleright g \otimes (x + y) : \tau \Rightarrow g \otimes y \otimes z \otimes (x + y) \\
\Gamma^* \triangleright \lambda x.g \otimes (x + y) : \text{int} \rightarrow \tau \Rightarrow \lambda x.g \otimes y \otimes z \otimes (x + y) \\
\Gamma^* \triangleright \lambda x.g \otimes (x + y) : \{ y \}_{\text{int}} \rightarrow \text{int} \rightarrow \tau \Rightarrow \lambda y.\lambda x.g \otimes y \otimes z \otimes (x + y) \\
\Gamma^* \triangleright \lambda x.g \otimes (x + y) : \{ z \}_{\text{int}} \rightarrow \{ y \}_{\text{int}} \rightarrow \text{int} \rightarrow \tau \Rightarrow \lambda z.\lambda y.\lambda x.g \otimes y \otimes z \otimes (x + y)
\end{align*}
\]

Fig. 4.3. Example derivation using (lift-abs)
$g @ y @ z$. Finally, the normal application rule from Figure 4.1 transforms $g @ (x + y)$ into $g @ y @ z @ (x + y)$.

In Figure 4.3, we transform the abstraction bound to $f$ by the letrec from $\lambda x.g @ (x + y)$ into the abstraction $\lambda z.\lambda y.\lambda x.g @ y @ z @ (x + y)$. We use the (abs) rule from Figure 4.1 (along with the derivation in Figure 4.2) to transform $\lambda x.g @ (x + y)$ into $\lambda x.g @ y @ z @ (x + y)$. Below that in the derivation, two instances of the (lift-abs) rule results in the final parameter-lifted abstraction that contains no free variables. The constraints from the (lift-abs) and (abs) rules are not shown in the figure, but we can easily verify that both $y$ and $z$ have type $\text{int}$ in $\Gamma^*$. It also must be the case that neither $y$ nor $z$ appear in the type $\tau$.

The order in which parameters are lifted does not matter. Our use of singleton types guarantees that lifted functions are applied to the proper arguments in the proper order. So, if $f$ and $g$ needed to have the same type (e.g. they are both passed as a parameter to the same function in $e''$), we could lift $y$ followed by $z$ (or vice versa) for both functions:

$$\text{letrec } f = \lambda y.\lambda z.\lambda x.g @ y @ z @ (x + y)$$
$$g = \lambda y.\lambda z.\lambda w.f @ y @ z @ (w + z)$$

in $e''$

In this expression, both $f$ and $g$ have type $\{y\}_\text{int} \rightarrow \{z\}_\text{int} \rightarrow \text{int} \rightarrow \tau$. 
4.3.2 Example 2

The following example illustrates some of the flexibility of our parameter-lifting specification. The letrec expression below defines three functions: $f$, $g$ and $h$.

$$\text{letrec } f = \lambda y. y + x$$
$$g = \lambda k. k + k$$
$$h = \lambda w. \lambda z. w \otimes z$$
$$\text{in } (h \otimes f \otimes e_1) + (h \otimes g \otimes e_2)$$

We would like to parameter-lift $f$ because the function contains the free variable $x$. The rules in Figure 4.1 provide many ways in which to do this while still maintaining the typeability of the target expression.

A simple option would be to lift $x$ and then apply $f$ directly to the lifted parameter:

$$\text{letrec } f = \lambda x. \lambda y. y + x$$
$$g = \lambda k. k + k$$
$$h = \lambda w. \lambda z. w \otimes z$$
$$\text{in } (h \otimes f \otimes x) \otimes e'_1) + (h \otimes g \otimes e'_2)$$

A drawback to this method is that we have introduced an unnecessary partial application. In the original expression, $f$ is applied to its argument when the body of $h$ is evaluated. In the transformed expression, $f$ is applied to its first argument before being passed to $h$. Then $f$ is applied to its second argument when the body of $h$ is evaluated.
The partial application in the expression above is unnecessary because we can avoid it by applying $f$ to both of its arguments at the same time in the body of $h$:

\[
\text{letrec } f = \lambda x. \lambda y. y + x \\
g = \lambda k. k + k \\
h = \lambda w. \lambda z. w \circ x \circ z \\
in (h \circ f \circ e''_1) + (h \circ g \circ e''_2)
\]

Of course, this is not quite a correct transformation because the expression is not typeable. Since both $f$ and $g$ are passed as parameters to $h$, they must have the same type.

By lifting $x$ in function $f$, we have changed $f$’s type from $\text{int} \rightarrow \text{int}$ to $\{x\}_{\text{int}} \rightarrow \text{int} \rightarrow \text{int}$.

In order to force the types of $f$ and $g$ to match, we must lift $x$ in $g$. The fact that $x$ does not appear in $g$ is irrelevant. The (lift-abs) rule in Figure 4.1 allows any variable currently in scope to be lifted in any function. Thus we can correctly transform the original expression as follows:

\[
\text{letrec } f = \lambda x. \lambda y. y + x \\
g = \lambda x. \lambda k. k + k \\
h = \lambda w. \lambda z. w \circ x \circ z \\
in (h \circ f \circ e''_1) + (h \circ g \circ e''_2)
\]
Since we have introduced a free variable in the body of $h$, we can further transform the expression so that all three functions declared by the \texttt{letrec} are closed:

\[
\begin{align*}
\text{letrec } f &= \lambda x.\lambda y. y + x \\
g &= \lambda x.\lambda k. k + k \\
h &= \lambda x.\lambda w.\lambda z. w @ x @ z
\end{align*}
\]

\[
in (h @ x @ f @ e''') + (h @ x @ g @ e''')
\]

This is an example of “higher-order” lambda-lifting: the transformation can take advantage of the higher-order nature of the program (by recognizing that $f$ is passed as a parameter to $g$) to avoid introducing the partial application. Existing descriptions of lambda-lifting only allow for the first transformation (where $f$ is applied directly to $x$).

### 4.4 Algorithm Correctness

The lambda-lifting algorithm originally described by Johnsson in [16] transforms a functional program with \textit{local} function definitions into a program consisting of a set of \textit{global} function definitions along with a main expression to be evaluated. For example, Johnsson’s algorithm transforms the program

\[
\begin{align*}
\text{let } y &= e_1 \\
in \text{letrec } f &= \lambda x. f @ (y + x) \\
in \text{letrec } g &= \lambda w. f @ w \\
in g @ e_2
\end{align*}
\]
into the “flattened” program

\[ f = \lambda y. \lambda x. f \, \text{at} \, y \, \text{at} \, (y + x); \]
\[ g = \lambda y. \lambda w. f \, \text{at} \, y \, \text{at} \, w; \]
\[ \text{let } y = e'_1 \text{ in } g \, \text{at} \, y \, \text{at} \, e'_2. \]

If we omit the block-floating step that flattens the program, Johnsson’s algorithm gives us the following parameter-lifted program:

\[ \text{let } y = e'_1 \]
\[ \text{in letrec } f = \lambda y. \lambda x. f \, \text{at} \, y \, \text{at} \, (y + x) \]
\[ \text{in letrec } g = \lambda y. \lambda w. f \, \text{at} \, y \, \text{at} \, w \]
\[ \text{in } g \, \text{at} \, y \, \text{at} \, e'_2. \]

For some function \( h \), the parameter-lifting phase of Johnsson’s algorithm introduces a new parameter

- for each free variable that appears in \( h \), as well as
- for each free variable that appears in functions called by \( h \).

In the example above, a new parameter is introduced for \( f \) because the variable \( y \) appears free in the body of \( f \). A new parameter is also introduced for \( g \) because \( g \) calls \( f \).

Each occurrence of a function name is then applied to all of that function’s new parameters. In the example above, both instances of \( f \) are replaced with \( f \, \text{at} \, y \), and the instance of \( g \) is replaced with \( g \, \text{at} \, y \).
Function names are considered constants by Johnson since all function names will be made global by block-floating. Hence, a parameter is never introduced for a function name even if that name appears free in the body of a function. For the definition of the function $g$, a new parameter is not introduced for $f$ even though $f$ appears free in the body of $g$.

The algorithm described by Johnson is designed for an untyped, call-by-name functional language. In this section, I present a formulation of the parameter-lifting phase of Johnson’s algorithm for our typed, call-by-value language and, using the specification in Figure 4.1, prove that the algorithm is correct.

4.4.1 Two-Phase Specification

I first give our formulation of Johnson’s algorithm as two sets of inference rules. The rules in Figures 4.4 and 4.5 avoid the generalities of the parameter-lifting specification in Figure 4.1 by

- restricting the lifting of parameters to letrec expressions rather than $\lambda$-abstractions in general,
- specifying the set of variables that must be lifted, and
- requiring that function names (rather than any expression with a singleton type) be applied directly to their lifted parameters.

These sets of rules provide us with a formal description of a particular parameter-lifting strategy. We can more easily derive an actual algorithm from these new rules than from the more generic description of parameter-lifting in Figure 4.1.
\[
\Lambda(x) = (\tau, \emptyset) \\
\Lambda \triangleright x : (\tau, \{x\}) \Rightarrow x^0 \quad \text{(param)} \\
\Lambda(f) = (\tau, \emptyset) \\
\Lambda \triangleright f : (\tau, \emptyset) \Rightarrow f^\emptyset \quad \text{(fun-name)} \\
\Lambda \triangleright y : (\tau_1, \emptyset) \Rightarrow e : (\tau, \emptyset) \Rightarrow m \\
\Lambda \triangleright \lambda y.e : (\tau_1 \rightarrow \tau, \emptyset - \{y\}) \Rightarrow \lambda y.m \quad \text{(abs)} \\
\Lambda \triangleright e_1 : (\tau_1 \rightarrow \tau, \emptyset_1) \Rightarrow m_1 \\
\Lambda \triangleright e_2 : (\tau_1, \emptyset_2) \Rightarrow m_2 \\
\Lambda \triangleright e_1 \circ e_2 : (\tau, \emptyset_1 \cup \emptyset_2) \Rightarrow m_1 \circ m_2 \quad \text{(app)} \\
\Lambda^* = \Lambda\{f_i : (\tau_i, \emptyset_i)\} \\
\Lambda^* \triangleright \lambda y_i.e_i : (\tau_i, \emptyset_i) \Rightarrow \lambda y_i.m_i \quad \Lambda^* \triangleright e : (\tau, \emptyset) \Rightarrow m \quad i \in \{1..n\} \\
\Lambda \triangleright \text{letrec } f_i = \lambda y_i.e_i \text{ in } e : (\tau, \emptyset \cup \emptyset_i \cup \emptyset) \Rightarrow \text{letrec } f_i^{\emptyset_i} = \emptyset_i \lambda y_i.m_i \text{ in } m \quad \text{(letrec)}
\]

Fig. 4.4. Parameter Lifting Annotation Phase

\[
\frac{}{y^0 \Rightarrow_{t} y} \quad \text{(var)} \quad \frac{y^s \Rightarrow_{t} e}{y^{x::s} \Rightarrow_{t} e \circ x} \quad \text{(var-app)} \\
\frac{m \Rightarrow_{t} e}{\lambda y.m \Rightarrow_{t} \lambda y.e} \quad \text{(abs)} \quad \frac{m_1 \Rightarrow_{t} e_1 \quad m_2 \Rightarrow_{t} e_2}{m_1 \circ m_2 \Rightarrow_{t} e_1 \circ e_2} \quad \text{(app)} \\
\frac{m \Rightarrow_{t} e \quad g_i = \emptyset_i \lambda y_i.m_i \Rightarrow_{t} g_i = e_i \quad i \in \{1..n\} \quad \text{letrec}}{\text{letrec } g_i = \emptyset_i \lambda y_i.m_i \text{ in } m \Rightarrow_{t} \text{letrec } g_i = e_i \text{ in } e \quad \text{(letrec)}} \\
\frac{\lambda y.m \Rightarrow_{t} e}{g = \emptyset \lambda y.m \Rightarrow_{t} g = e} \quad \text{(decl)} \quad \frac{g = \emptyset \lambda y.m \Rightarrow_{t} g = e}{g = x::s \lambda y.m \Rightarrow_{t} g = e} \quad \text{(decl-lift)}
\]

Fig. 4.5. Parameter Lifting Translation Phase
This new specification is separated into two phases: analysis (Figure 4.4) and translation (Figure 4.5). The analysis phase determines exactly which parameters will be lifted and annotates the expression accordingly. The translation phase interprets these annotations and introduces the new parameters and arguments into the target term.

In Figure 4.4, the judgment $\Lambda \triangleright e : (\tau, \theta) \Rightarrow m$ associates a source expression, $e$, with its annotated form, $m$. The set $\theta$ is the set of variables free in $e$. This set is used by the translation to determine which new parameters and arguments should appear in the parameter-lifted expression. If $e$ is a function defined by a letrec, then $\theta$ contains exactly those variables that will be lifted and introduced as new parameters to that function. Since the transformation uses $\theta$ to determine which variables to lift, and since it applies variable names directly to their lifted parameters, there is no need for singleton types. So the type $\tau$ is a simple type containing no annotations or singleton types.

The context $\Lambda$ maps each variable to its type as well as an ordered set of variables. If the variable is the name of a function, this ordered set is the set of lifted parameters for that function. Otherwise, the ordered set is empty. For example, if $f$ is declared as the function $\lambda x . x @ y @ z$ by a letrec expression, then $\Lambda(f) = (\tau, \{ y, z \})$, where $\tau$ is the type of $f$. The variables $y$ and $z$ appear in the set in $\Lambda(f)$ because they occur free in the function bound to $f$ and will be introduced as new parameters by the translation. This set must be ordered so that the translation introduces parameters and their corresponding arguments in the proper order.

The translation phase, Figure 4.5, relates an annotated expression with its parameter-lifted form. In the judgment $m \Rightarrow_t e$, $e$ is the parameter-lifted form of $m$. Since $m$ is annotated, the translation does not need to be given a context.
Variables

\[
\Lambda(x) = (\tau, \theta) \quad (\text{param}) \\
\Lambda \triangleright x : (\tau, \{x\}) \Rightarrow x^\theta \\
\Lambda(f) = (\tau, \theta) \quad (\text{fun-name}) \\
\Lambda \triangleright f : (\tau, \theta) \Rightarrow f^\theta
\]

Because the transformation differentiates between function names (i.e. variables introduced by letrec) and other variables (i.e. those introduced by \(\lambda\)-abstraction), there are two variable rules: (fun-name) and (param). The (fun-name) rule translates a function name by annotating it with \(\theta\), the ordered set in \(\Lambda(f)\). Since \(\theta\) includes all variables that occur free in the function bound to \(f\), \(\theta\) is the set of new parameters that the translation will introduce. This means that \(\theta\) is also the set of new arguments that \(f\) must be applied to in the parameter-lifted expression.

Since \(f\) will be applied to the variables in \(\theta\) in the parameter-lifted expression, we consider \(\theta\) to be the variables “free” in the expression \(f\). Thus, since the type of \(f\) is \(\tau\), the pair \((\tau, \theta)\) appears in the consequent of the rule. Note that \(f\) itself is not a member of \(\theta\). This is due to the fact that function names are considered constants (presuming parameter-lifted functions will be made global by a subsequent block floating transformation) and are never introduced as new parameters or arguments.

Recall that this transformation only parameter-lifts functions declared by letrec expressions. Thus, in the (param) rule, variables introduced by a \(\lambda\)-abstraction are annotated with the empty set. This is to indicate to the translation that the variable should not be applied to any new arguments. The pair \((\tau, \{x\})\) appears in the consequent of the rule since \(\tau\) is the type of \(x\) and the only variable that occurs free in the expression \(x^\theta\).
$x$ is $x$ itself. Because of the higher-order nature of the language, $x$ may be bound to a function. However $x$ is not the name of a function (i.e. $x$ is not defined by letrec).

\[
\begin{align*}
\frac{\emptyset}{y \Rightarrow^* y} \quad (\text{var}) \quad \frac{\emptyset}{y^s \Rightarrow^* e \Rightarrow^* e @ x} \quad (\text{var-app})
\end{align*}
\]

To translate a variable,\(^2\) the (var) and (var-app) rules from Figure 4.5 introduce an argument in the parameter-lifted expression for each variable in the annotating set. If the set is empty, the (var) rule simply translates the variable to itself, removing the annotation. In the consequent of the (var-app) rule, we use the notation :: in the representation of the annotation. Since the annotation is an ordered set, the representation $x :: s$ identifies $x$ as the first element in the set and $s$ as the remainder of the set. For example, the expression $f^x :: y :: z :: \emptyset$ is translated into the application $f @ z @ y @ x$.

**Abstraction and Application**

\[
\frac{\Lambda \{ y : \tau \} \Rightarrow^* e : (\tau, \theta) \Rightarrow m}{\Lambda \Rightarrow \lambda y : (\tau_1 \rightarrow \tau, \theta \setminus \{ y \}) \Rightarrow \lambda y : m} \quad (\text{abs})
\]

\[
\frac{\Lambda \Rightarrow e_1 : (\tau_1 \rightarrow \tau, \theta) \Rightarrow m_1 \quad \Lambda \Rightarrow e_2 : (\tau_1, \theta_1) \Rightarrow m_2}{\Lambda \Rightarrow e_1 @ e_2 : (\tau, \theta_1 \cup \theta_2) \Rightarrow m_1 @ m_2} \quad (\text{app})
\]

Since parameters will only be lifted from functions declared by a letrec, the (abs) rule does not introduce any annotations. Analyzing both abstractions and applications is simply a matter of analyzing the subterms. The $y \notin FV(\tau)$ constraint from the

\(^2\)The translation phase does not differentiate between function names and other variables.
abstraction rule in Figure 4.1 does not apply here since we are not using singleton types with this system.

\[
\frac{m \Rightarrow_t e}{\lambda y. m \Rightarrow_t \lambda y. e} \quad \text{(abs)} \quad \frac{m_1 \Rightarrow_t e_1 \quad m_2 \Rightarrow_t e_2}{m_1 \otimes m_2 \Rightarrow_t e_1 \otimes e_2} \quad \text{(app)}
\]

The translation rules for abstraction and application are simple. To translate such expressions, translate the subexpressions.

**Recursion**

\[
\Lambda^* = \Lambda \{ f_i : (\tau, \theta_i) \} \\
\Lambda^* \triangleright \lambda y_i.e_i : (\tau, \theta_i) \Rightarrow \lambda y_i.m_i \quad \Lambda^* \triangleright e : (\tau, \theta) \Rightarrow m \quad i \in \{ 1 \ldots n \} \\
\Lambda \triangleright \text{letrec } f_i = \lambda y_i.e_i \text{ in } e : (\tau \cup \theta_i \cup \theta) \Rightarrow \text{letrec } f_i = \theta_i \lambda y_i.m_i \text{ in } m
\]

The only functions that can be parameter-lifted with this system are functions declared using the letrec expression. The (letrec) rule annotates each function declared by the given letrec with the ordered set of that function’s lifted parameters. In the antecedent of the rule, the abstractions are analyzed to derive the sets of lifted parameters. In the consequent of the rule, each function declaration in the translated letrec is annotated with the appropriate set. These sets must be ordered sets so that the translation phase can introduce the parameters in the proper order.
\[
m \Rightarrow_t e \quad g_i = \theta^i \lambda y_i.m_i \Rightarrow_t g_i = e_i \quad \text{for } i \in \{1..n\}
\]

(letrec)

\[
\text{letrec } g_1 = \theta^1 \lambda y_1.m_1 \ldots g_n = \theta^n \lambda y_n.m_n \text{ in } m \Rightarrow_t \text{letrec } g_1 \ldots g_n = e_n \text{ in } e
\]

(decl)

\[
\frac{\lambda y.m \Rightarrow_t e}{g = \emptyset \lambda y.m \Rightarrow_t g = e}
\]

(decl-lift)

\[
\frac{g = \emptyset \lambda x \lambda y.m \Rightarrow_t g = e}{g = \emptyset \lambda y.m \Rightarrow_t g = e}
\]

The letrec is translated by translating the body of the letrec, \(m\), as well as each declaration, \(g_i = \theta^i \lambda y_i.m_i\). When the annotation on a declaration is the empty set, then the (decl) rule translates the declaration by translating the abstraction. If the annotation is a non-empty set, then the (decl-lift) rule translates the declaration by translating the same declaration with \(x\) as an additional parameter and without \(x\) in the annotating set.

This two-phase specification is sound with respect to the parameter-lifting specification in Figure 4.1. See [11] for the formal statement of the soundness theorem and the corresponding proof. Since we know the rules in Figure 4.1 are correct with respect to the static and dynamic semantics of our language, this soundness results shows that the two-phase specification is also correct.

**Example Abstraction Derivation**

Consider the following letrec expression declaring the mutually recursive functions \(f\) and \(g\).

\[
\text{letrec } f = \lambda x.g \circ (x + y)
\]

\[
g = \lambda z.f \circ (z + z)
\]

in \(e\).
We can use the rules in Figure 4.4 to transform this expression into the annotated expression below, where $e'$ is an annotated form of $e$:

\[
\begin{align*}
\text{letrec } f &= \{y\} \lambda x.g \{y\} @ (x^{\theta} + y^{\theta}) \\
g &= \{y\} \lambda z.f \{y\} @ (z^{\theta} + z^{\theta})
\end{align*}
\]

in $e'$.

The variable $y$ is parameter-lifted in $f$ because it occurs free in $f$'s abstraction. The variable $y$ is also parameter-lifted in $g$ because $g$ calls $f$. Thus both declarations, as well as both function calls, are annotated with the set $\{y\}$. Since $x$, $y$ and $z$ are not function names, each occurrence of these variables is annotated with the empty set.

The derivation used to annotate $f$'s abstraction is given in Figure 4.6. We assume for this example that the addition operator is included in our language. The type context $\Lambda$ maps the free variable $y$ to the pair (int, $\emptyset$) but does not contain mappings for $f$, $g$ or
\[
\begin{align*}
g^0 & \Rightarrow_t g \\
x^0 & \Rightarrow_t x \\
y^0 & \Rightarrow_t y \\
g \{y\} & \Rightarrow_t g @ y \\
x^0 + y^0 & \Rightarrow_t x + y \\
g \{y\} @ (x^0 + y^0) & \Rightarrow_t g @ y @ (x + y) \\
\lambda x. g \{y\} @ (x^0 + y^0) & \Rightarrow_t \lambda x. g @ y @ (x + y)
\end{align*}
\]

Fig. 4.7. Translation example

\(x\). The type context derived by the letrec rule, \(\Lambda^*\), is

\[\Lambda\{f : (\text{int} \rightarrow \tau, \{y\})\}\{g : (\text{int} \rightarrow \tau, \{y\})\}.\]

The context \(\Lambda'\) is the context \(\Lambda^*\) extended to include the mapping \(\{x : (\text{int}, \emptyset)\}\), which is introduced by the (abs) rule.

We complete the parameter-lifting process by using the rules in Figure 4.5 to transform the annotated letrec above to the parameter-lifted expression below, where \(e''\) is a translated form of \(e'\):

\[
\begin{align*}
\text{letrec } f & = \lambda y. \lambda x. g @ y @ (x + y) \\
g & = \lambda y. \lambda z. f @ y @ (z + z) \\
in e''.
\end{align*}
\]

Based on the annotations, each function now includes a binding for \(y\) and each occurrence of \(f\) and \(g\) is now applied to \(y\). The derivation used to translate \(f\)'s annotated abstraction is given in Figure 4.7.
\[ \lambda z. \lambda y. \lambda x. \lambda w. m_1 \Rightarrow \lambda z. \lambda y. \lambda x. \lambda w. e_1 \]

\[ h = 0 \lambda z. \lambda y. \lambda x. \lambda w. m_1 \Rightarrow \lambda z. \lambda y. \lambda x. \lambda w. e_1 \]

\[ h = \{z\} \lambda y. \lambda x. \lambda w. m_1 \Rightarrow \lambda z. \lambda y. \lambda x. \lambda w. e_1 \]

\[ h = \{y, z\} \lambda x. \lambda w. m_1 \Rightarrow \lambda z. \lambda y. \lambda x. \lambda w. e_1 \]

\[ h = \{x, y, z\} \lambda w. m_1 \Rightarrow \lambda z. \lambda y. \lambda x. \lambda w. e_1 \]

**Fig. 4.8. Example declaration derivation**

**Example Declaration and Application Derivation**

Figures 4.8 and 4.9 illustrate derivations, using the translation rules in Figure 4.5, involving annotation sets containing more than one variable. Figure 4.8 shows the derivation transforming the annotated declaration \( h = \{x, y, z\} \lambda w. m_1 \) into the parameter-lifted declaration \( h = \lambda z. \lambda y. \lambda x. \lambda w. e_1 \). The derivation in Figure 4.9 transforms the annotated function application \( h[x, y, z] \circ m_2 \) into the expression \( h \circ z \circ y \circ x \circ e_2 \), in which \( h \) is applied to all of its lifted parameters in the proper order.

**4.4.2 Algorithm**

The formulation of Johnsson's algorithm described in this section is derived directly from the two-phase parameter-lifting specification. Figure 4.10 defines the functions \( \mathcal{P} \) and \( \mathcal{L} \) and translate, which correspond to the analysis and translation phases, respectively.
\[
\begin{align*}
\frac{h^0}{h} & \Rightarrow_t h \\
\frac{h\{z\}}{h \circ z} & \Rightarrow_t h @ z \\
\frac{h\{y,z\}}{h \circ z \circ y} & \Rightarrow_t h @ z @ y \\
\frac{h\{x,y,z\}}{h \circ z \circ y \circ x} & \Rightarrow_t h @ z @ y @ x @ e_2 \\
\end{align*}
\]

Fig. 4.9. Example application derivation

The function \(\mathcal{P}\mathcal{L}\) takes a context, \(\Delta\), and an expression, \(e\), and returns a set of variables, \(\theta\), and an annotated expression, \(m\). The set \(\theta\), as in Figure 4.4, represents the set of parameters to be lifted in the current expression. The context \(\Delta\) corresponds to the context in Figure 4.4, except that no type information is included. The rules in Figure 4.4 do not use types to convey information specific to parameter-lifting, so if we assume the input expression to \(\mathcal{P}\mathcal{L}\) is well-typed, our algorithm can ignore types altogether. As a result, the context \(\Delta\) maps variables to their sets of lifted parameters, but not to their types.

Each case of \(\mathcal{P}\mathcal{L}\) corresponds directly to a rule in Figure 4.4, so there is little to explain here. The only new complication arises in the case for letrec expressions. The inference rule for letrec extends the given context to include the mappings of the function names to their types and sets of lifted parameters. An algorithm must specify how these sets of lifted parameters are computed.

In the letrec case, the algorithm must extend the context before recursively calling \(\mathcal{P}\mathcal{L}\) on the subexpressions. However before the recursive calls to \(\mathcal{P}\mathcal{L}\) compute the sets
\[ P \mathcal{L}(\Delta, x) = \text{let} \]
\[ \theta = \Delta(x) \]
\[ \text{in} \ (\{x\}, x^\theta) \]

\[ P \mathcal{L}(\Delta, f) = \text{let} \]
\[ \theta = \Delta(f) \]
\[ \text{in} \ (\theta, f^\theta) \]

\[ P \mathcal{L}(\Delta, \lambda y.e) = \text{let} \]
\[ (\theta, m) = P \mathcal{L}(\Delta\{y : \emptyset\}, e) \]
\[ \text{in} \ (\theta - \{y\}, \lambda y.m) \]

\[ P \mathcal{L}(\Delta, e_1 \circ e_2) = \text{let} \]
\[ (\theta_1, m_1) = P \mathcal{L}(\Delta, e_1) \]
\[ (\theta_2, m_2) = P \mathcal{L}(\Delta, e_2) \]
\[ \text{in} \ (\theta_1 \cup \theta_2, m_1 \circ m_2) \]

\[ P \mathcal{L}(\Delta, \text{letrec } g_i = \lambda y_i.e_i \text{ in } e) = \text{let} \]
\[ \Delta' = \Delta\{g_1 : \theta_{g_1}, \ldots, g_n : \theta_{g_n}\} \]
\[ (\theta_1, m_1) = P \mathcal{L}(\Delta', \lambda y_1.e_1) \]
\[ \vdots \]
\[ (\theta_n, m_n) = P \mathcal{L}(\Delta', \lambda y_n.e_n) \]
\[ \varepsilon = \text{solve}((\{g_1 = \theta_1\setminus\theta_{g_1}, \ldots, g_n = \theta_n\setminus\theta_{g_n}\})) \]
\[ (\theta, m) = P \mathcal{L}(\varepsilon\Delta', e) \]
\[ \text{in} \ (\varepsilon(e_{g_1}) \cup \ldots \cup \varepsilon(e_{g_n}) \cup \theta, \]
\[ \text{letrec } g_1 = \varepsilon(e_{g_1}) \lambda y_1.e m_1 \ldots g_n = \varepsilon(e_{g_n}) \lambda y_n.e m_n \text{ in } m) \]

\text{translate}(g^{x \mapsto s}) = \text{translate}(g^s) \circ x \]
\text{translate}(y^\emptyset) = y \]

\text{translate(letrec } g_1 = \theta_1 \lambda y_1.m_1 \ldots g_n = \theta_n \lambda y_n.m_n \text{ in } m) = \]
\[ \text{letrec translate}(g_1 = \theta_1 \lambda y_1.m_1) \ldots \text{translate}(g_n = \theta_n \lambda y_n.m_n) \text{ in translate}(m) \]

\text{translate}(g = \lambda y.m) = \text{translate}(g =^s \lambda x.\lambda y.m) \]
\text{translate}(g =^\emptyset \lambda y.m) = (g = \text{translate}(\lambda y.m))

\text{translate}(\lambda y.m) = \lambda y.\text{translate}(m) \]
\text{translate}(m_1 \circ m_2) = \text{translate}(m_1) \circ \text{translate}(m_2)

Fig. 4.10. Parameter-lifting algorithm
of lifted parameters, the algorithm has no way to determine exactly how to extend the context. We resolve this issue by introducing set variables. We use the notation $\theta_g$ to represent the set variable for function name $g$. Instead of extending the context by mapping the function names to their sets of lifted parameters, the algorithm extends the context by mapping the function names to these set variables.

The recursive call to $PL$ for each abstraction declared by the letrec returns the set of parameters to be lifted in that function. This set may actually be a union of sets, including set variables. Consider the following example:

$$\text{letrec } f = \lambda x.g @ (x + y)$$
$$g = \lambda w.f @ (w + z)$$

in $e$.

The letrec case extends the given context with the mappings $\{f : \theta_f\}$ and $\{g : \theta_g\}$. Using this extended context, the recursive calls to $PL$ give us the following results:

$$PL(\Delta', \lambda x.g @ (x + y)) = (\{y\} \cup \theta_g, \lambda x.g^{\theta_g} @ (x^0 + y^0))$$
$$PL(\Delta', \lambda w.f @ (w + z)) = (\{z\} \cup \theta_f, \lambda w.f^{\theta_f} @ (w^0 + z^0))$$

Before the algorithm can translate the annotated expression, it must replace the set variables $\theta_g$ and $\theta_f$ with actual sets.

The letrec case introduces constraints equating the set variables with the sets, or unions of sets, returned by the recursive calls to $PL$. In the example above, the algorithm would introduce the constraints $\theta_f = \{y\} \cup \theta_g$ and $\theta_g = \{z\} \cup \theta_f$. These constraints are
solve(\{\theta_{g_i} = \theta_j \cup \theta\} \cup \{\theta_{g_i} = \theta_j\} \cup \Phi) = \\
solve(\{\theta_{g_i} = \theta_j \setminus \theta_{g_i} \cup \theta\} \cup \{\theta_{g_j} = \theta_j\} \cup \Phi) \\
solve(\Phi) = \Phi, if no \theta_{g_i} appears on the RHS of a constraint in \Phi.

\text{Fig. 4.11. Constraint solver}

then passed to a constraint solver, defined in Figure 4.11, that returns a substitution mapping the set variables to actual sets of lifted parameters. This substitution is applied to the set and annotated expression returned by the letrec case.

To guarantee that the constraint solver will terminate, instances of set variables are removed from the right-hand side of constraints in which they appear on the left-hand side. For example, all instances of \theta_f must be removed from the RHS of the constraint \theta_f = \theta, represented by the operation \theta \setminus \theta_f.

\text{Definition 10. For any set of parameters } \theta \text{ and set variable } \theta_f, \text{ the set } \theta \setminus \theta_f \text{ is defined as:}

\begin{align*}
\{x_1, ..., x_n\} \setminus \theta_f &= \{x_1, ..., x_n\} \\
(\theta \cup \theta_f) \setminus \theta_f &= \theta \setminus \theta_f \\
(\theta \cup \theta_g) \setminus \theta_f &= (\theta \setminus \theta_f) \cup \theta_g, f \neq g
\end{align*}
To illustrate the constraint solver, we apply solve to the constraints generated for the example above:

\[
\text{solve}([\theta_f = \{y\} \cup \theta_g] \cup \{\theta_g = \{z\} \cup \theta_f\}) = \]

\[
\text{solve}([\theta_f = \{y\} \cup (\{z\} \cup \theta_f) \setminus \theta_f] \cup \{\theta_g = \{z\} \cup \theta_f\}) = \]

\[
\text{solve}([\theta_f = \{y, z\}] \cup \{\theta_g = \{z\} \cup \{y, z\} \setminus \theta_g\}) = \]

\[
\text{solve}([\theta_f = \{y, z\}] \cup \{\theta_g = \{y, z\}\})
\]

Since there are no more set variables on the RHS of any constraint, the constraint solver returns the substitution mapping both \(\theta_f\) and \(\theta_g\) to the set \(\{y, z\}\). When this substitution is applied to the annotated expression, we have

\[
\text{letrec } f = \{y, z\} \lambda x. \text{g}(y, z) \odot (x^\emptyset + y^\emptyset) \\
g = \{y, z\} \lambda w. \text{f}(y, z) \odot (w^\emptyset + z^\emptyset)
\]

in \(m\).

The translate function takes an annotated expression and returns a parameter-lifted form of the expression. When translate is applied to the annotated expression above, the result is the letrec below in which both \(y\) and \(z\) are lifted in each function:

\[
\text{letrec } f = \lambda z. \lambda y. \lambda x. \text{g}(z \odot y \odot (x + y)) \\
g = \lambda z. \lambda y. \lambda w. \text{f}(z \odot y \odot (w + z))
\]

in \(e'\)
4.4.3 Correctness

I prove the correctness of this formulation of Johnsson's parameter-lifting algorithm by showing that the algorithm is sound with respect to the two-phase specification in Figures 4.4 and 4.5.

In order to show a correspondence between the algorithm and the inference rules, I must define a correspondence between different kinds of contexts. The inference rules use a context mapping variables to types and sets of parameters. The algorithm uses a context mapping variables to sets of parameters. Since we assume that the input expression to the algorithm is well-typed, we can make use of the type context, \( \Gamma \), from Figure 2.1 in the following definition.

**Definition 11.** Let \( \Gamma \) be a context mapping variables to types and \( \Delta \) be a context mapping variables to sets of variables, such that \( \text{dom}(\Gamma) = \text{dom}(\Delta) \). Then \( \Gamma \circ \Delta \) is the context such that, for all \( \nu \in \text{dom}(\Gamma) \):

\[
(\Gamma \circ \Delta)(\nu) = (\Gamma(\nu), \Delta(\nu))
\]

The context \( \Gamma \circ \Delta \) is the context used in the rules of Figure 4.4.

A context is well-formed, or closed, if all variables appearing in annotating sets within the context are in the domain of the context. Furthermore the annotating set of any variable that is not a function name must be empty.
DEFINITION 12. A context $\Delta$ is closed, written $\text{Closed}(\Delta)$, iff for all $\nu \in \text{dom}(\Delta)$,

1. if $\Delta(f) = \theta$ then $\theta \subseteq \text{dom}(\Delta)$, and

2. if $\Delta(x) = \theta$ then $\theta = \emptyset$.

I also demonstrate the correctness of the constraint solver. The following lemma guarantees not only that the solver always produces a substitution when given a finite set of constraints, but that it always produces a correct substitution.

LEMMA 4.1. For any finite set of constraints $\Phi = \{\theta_{g_i} = \theta_{\setminus g_i}\}$:

1. solve($\Phi$) halts, and

2. if solve($\Phi$) = $\epsilon$, then $\epsilon \theta_{g_i} = \epsilon \theta_{\setminus g_i}$ for all $\theta_{g_i}$ in $\Phi$.

The proofs of both parts of the lemma can be found in Appendix B.

The soundness theorem for the algorithm states that the functions $\mathcal{PL}$ and translate are sound with respect to the inference rules in Figures 4.4 and 4.5, respectively.

THEOREM 4.1 (ALGORITHM SOUNDNESS).

1. If $\text{Closed}(\Delta)$, $\mathcal{PL}(\Delta, e) = (\theta, m)$, and $\Gamma \triangleright e : \tau$ then $(\Gamma \ast \Delta) \triangleright e : (\tau, \theta) \Rightarrow m$.

2. If $\text{translate}(m) = e$ then $m \Rightarrow_t e$.

The proof of both parts of the theorem can be found in Appendix B.

Since the algorithm is sound with respect to the two-phase specification, it is sound with respect to the specification in Figure 4.1. Since we know that the specification in Figure 4.1 is correct with respect to the operational semantics, we know that the algorithm is correct as well.
Chapter 5

Uncurrying

Higher-order functional languages, such as Standard ML, allow functions to be Curried: a function is applied to each of its arguments separately. This feature allows functions to be partially applied. For example, if we have the function $f = \lambda x. \lambda y. x + y$, the partial application $f @ 1$ results in a function that increments its argument by one.

While the Currying of functions allows the programmer a great deal of flexibility when working with higher-order functions, it is not well-suited for an implementation. Applying the function above to both of its arguments requires two separate function calls. Since function calls require a certain amount of overhead, it is best to avoid them whenever practically possible. If the function above is applied to both of its arguments at the same time, only one function call is necessary.

We can potentially reduce the number of function calls in a program with Curried functions by employing an unCurry analysis. We unCurry a program by identifying where function arguments and parameters can be grouped together and passed to the function in a single function call. The function above can be unCurried to $\lambda [x, y]. x + y$ if $f$ is always applied to both of its arguments at the same time. An application of $f$ would then have the form $f @ [e_1, e_2]$ thus avoiding a second, unnecessary application for the second argument.
Hannan and Hicks introduce a formal, type-based specification of higher-order un-Currying in [14]. Their transformation is able to identify opportunities for unCurrying in the presence of higher-order functions. In the expression \((\lambda k. k @ e_1 @ e_2) @ (\lambda x. \lambda y. x)\), the first abstraction represents a higher-order function since it receives a function as an argument. Higher-order unCurrying is able to determine that

1. \(k\) is bound to a function of two parameters, and

2. \(\lambda x. \lambda y. x\) is applied to both of its arguments in consecutive applications.

Thus the expression can be unCurried to \((\lambda [k]. k @ [e_1, e_2]) @ (\lambda [x, y]. x)\).

Higher-order unCurrying cannot, however, completely unCurry every function. It is constrained by the reality that functions in a Curried higher-order language are not always applied to all of their arguments in consecutive applications. In the expression

\[
\text{letrec } h = \lambda f, f @ e_2 \\

k = \lambda x. \lambda y. x \\
in h @ (k @ e_1),
\]

the function \(k\) is applied to its first argument, passed to another function, and then applied to its second argument. Since \(k\) is applied to its two arguments at separate locations in the program, the higher-order unCurrying of [14] cannot unCurry \(k\).

Hannan and Hicks later suggested an extension to their work on higher-order unCurrying: an expression may be further unCurried by replacing partial applications with closure-like tuples. In the example above, the partial application \((k @ e_1)\) would be replaced with the tuple \((k, e_1)\), avoiding an unnecessary function call. Using this
method, functions are not applied to their arguments in an unCurried expression until 
all of their arguments are available. In this example, the function \( k \) is applied only in 
the body of \( h \), when its second argument is available:

\[
\text{letrec } h = \lambda[f].\#1(f) \circ [\#2(f), e_2] \\
k = \lambda[x, y].x \\
\text{ in } h \circ (k, e_1).
\]

Delaying the application of \( k \) until both of its arguments are available allows us to 
unCurry \( k \). The expression may be further optimized via arity raising [13], which splits 
the tuple \((k, e_1)\) into individual arguments to function \( h \):

\[
\text{letrec } h = \lambda[f_1, f_2].f_1 \circ [f_2, e_2] \\
k = \lambda[x, y].x \\
\text{ in } h \circ [k, e_1].
\]

In this chapter, I present a formal, type-based specification of this extended un-
Currying and prove that this specification defines a program transformation that is type 
safe and correct with respect to the operational semantics of the source and target lan-
guages.
5.1 Target Language

The source language for the unCurrying transformation described in the sections below is the simple functional language from Chapter 2, excluding the recursive μ operator. The focus of this section is the target language. Features of the target language not present in the source language include parameter lists, partial applications and dummy constants.

5.1.1 Parameter Lists

Since the unCurry transformation groups Curried parameters together, functions in the target language accept lists of parameters rather than a single parameter.\footnote{The parameter lists and their types that I describe here are the same as those used for the higher-order unCurrying transformation in [14]. This also includes the related static and dynamic semantics rules in Figures 5.1 and 5.2.} Thus a λ-abstraction introduces a list of local variable bindings:

$$\lambda [x_1, \ldots, x_n].m$$

Likewise, a function is applied to a list of parameters:

$$m \@ [m_1, \ldots, m_n]$$

In both cases, $n$ is an integer greater than or equal to one and $m$ represents a target language expression.
The type of a parameter list includes the types of the individual parameters in order and enclosed in square brackets (mirroring the syntax of parameter lists themselves). For example, the type of the parameter list \([x, y, z]\) is \([\phi_x, \phi_y, \phi_z]\), were \(\phi_x\), \(\phi_y\) and \(\phi_z\) are the types of \(x\), \(y\) and \(z\), respectively. Thus, the type of the abstraction \(\lambda[x, y, z].m\) is \([\phi_x, \phi_y, \phi_z] \rightarrow \phi_m\), where \(\phi_m\) is the type of the return expression. This expression represents a function of three parameters. In its Curried form \((\lambda x. \lambda y. \lambda z. m)\), the expression would represent three functions, each of one parameter, and its type would be \(\phi_x \rightarrow \phi_y \rightarrow \phi_z \rightarrow \phi_m\).

Following [14], I have included two separate type inference rules (shown below) for abstraction: one for functions of one parameter, the other for functions of multiple parameters. The rule for one-parameter functions is the same as the abstraction rule for the source language static semantics except that the parameter and its type are both enclosed in square brackets, denoting the parameter list (albeit a list of one element) and the type of the parameter list.

\[
\frac{\Gamma \vdash x : \phi_1 \vdash m : \phi_2}{\Gamma \vdash \lambda[x].m : [\phi_1] \rightarrow \phi_2} \quad \frac{\Gamma \vdash x_1 : \phi_1 \vdash \lambda[x_2, \ldots, x_n].m : [\phi_2, \ldots, \phi_n] \rightarrow \phi}{\Gamma \vdash \lambda[x_1, \ldots, x_n].m : [\phi_1, \ldots, \phi_n] \rightarrow \phi}
\]

The second abstraction rule reduces the problem of typing a function of \(n\) parameters to the problem of typing a function of \(n - 1\) parameters. The judgment in the antecedent of the rule infers the type of the function excluding the first (or leftmost) parameter using the type context \(\Gamma\) extended to include the binding of the first parameter to its type.
In the target language, a function may only be applied when all of its arguments are available. The application rule enforces this by requiring that the number of arguments in the operand equal the number of parameter types in the type of the operator. An expression that applies a function to an insufficient number of arguments is ill-formed and does not have a type.

\[
\Gamma \vdash m : [\phi_1, \ldots, \phi_n] \rightarrow \phi \quad \Gamma \vdash m_i : \phi_i \quad i = [1..n] \\
\Gamma \vdash m \@ [m_1, \ldots, m_n] : \phi
\]

The rule also requires that the arguments match the parameter types in order.

The operational semantics rule for an abstraction with a parameter list (shown below along with the rule for application) simply relates the abstraction to a closure that is created by pairing the abstraction together with the given environment. For an application, each arguments in the operand must be evaluated to a value using the given environment. These arguments must be evaluated in the order that they appear in the operand list from left to right, which is the order in which they would be evaluated in a Curried form. When evaluating the body of the abstraction, the given environment \(\rho'\) is extended to include bindings for all of the parameters to the values of the corresponding arguments.

\[
\rho \triangleright \lambda [x_1, \ldots, x_n] . m \leftrightarrow [\rho, \lambda [x_1, \ldots, x_n].m] \\
\rho \triangleright m \leftrightarrow [\rho', \lambda [x_1, \ldots, x_n].m'] \\
\rho \triangleright m_i \leftrightarrow v_i \\
\rho' \{ \overbrace{\{x_i \rightarrow v_i\}} \} \triangleright m' \leftrightarrow v \\
\rho \triangleright m \@ [m_1, \ldots, m_n] \leftrightarrow v
\]
5.1.2 Partial Application

In addition to these parameter lists, the target language also includes partial applications. It may be the case that, in the Curried source expression, a function is not applied to all of its arguments in consecutive applications. Consider the simple example below:

\[
\text{letrec } f = \lambda x. \lambda y. e \\
\text{ in } f @ e' 
\]

The function \( f \) requires two arguments, but is only applied to one in the letrec expression. How can we transform the letrec expression with an unCurried \( f \) when the second argument is unavailable? In its unCurried form, \( f \) cannot be applied to only one argument. The operational semantics rule for application discussed above forbids this.

One option might be to modify the program in which this letrec expression appears in such a way that the second argument is applied immediately after the first. It may be the case, for example, that the above letrec expression is the operator in an application:

\[
(\text{letrec } f = \lambda x. \lambda y. e \text{ in } f @ e') @ e'' 
\]

As long as \( f \) does not appear free in \( e'' \), there is no reason why a program transformation cannot move \( e'' \) inside the letrec:

\[
\text{letrec } f = \lambda x. \lambda y. e \\
\text{ in } f @ e' @ e'' 
\]
Now it is a simple matter to unCurry $f$'s definition and application:

\[
\text{letrec } f = \lambda[x, y].e \\
\text{in } f @ [e', e'']
\]

While this option looks like a winner for this particular example, it is not, in general, possible to modify a functional program so that all of the arguments to a Curried function are available at the same time. So this is not a suitable solution for the transformation described in this chapter.

A second option (which, unlike the first, always works) is to avoid unCurrying functions that are not applied to all of their arguments in consecutive applications. This method gives us the following transformation:

\[
\text{letrec } f = \lambda[x].\lambda[y].e \\
\text{in } f @ [e']
\]

However, the goal of the transformation in this chapter (and that which separates this work from the higher-order unCurrying of [14]) is to allow for the unCurrying of all functions. This solution is clearly not suitable either.

A third option, and the one I use in the remainder of this chapter, is to delay the application of a function until all of its arguments are available. In the example above, this necessitates the removal of the application ($f @ e'$). But the expression $e'$ certainly cannot be thrown away, and the rules of a call-by-value semantics prevent us from moving the expression. So the unCurry transformation replaces the application
with a closure-like structure pairing the function with its first argument:

\[
\text{letrec } f = \lambda[x, y]. e \\
\text{in } \langle f, e' \rangle
\]

The function \( f \) cannot be applied until its second argument is available. This allows the transformation to completely unCurry \( f \), combining both of its parameters into a single parameter list.

The closure-like structure \( \langle f, e' \rangle \) is a partial application, which is defined more explicitly below.

**Definition 13 (Partial Application).** A partial application is a left-associative tuple of \( n \) elements in which the leftmost element is a function of arity \( m \), and \( 2 \leq n \leq m \).

A left-associative tuple consisting of the expressions \( x, y \) and \( z \) is written \( \langle\langle\cdot, x), y), z \rangle \). The symbol \( \cdot \) denotes the leftmost point of the tuple. This allows us to distinguish between the expressions \( \langle\langle\cdot, m_1), m_2 \rangle \) and \( \langle m_1, m_2 \rangle \). The former is a tuple of two elements, whereas the latter is the expression \( m_2 \) appended to the tuple \( m_1 \). As a convenience, I use angle brackets to denote a left-associative tuple without the parentheses. Thus the tuple \( \langle\langle\cdot, x), y), z \rangle \) is also written \( \langle x, y, z \rangle \).

Since a partial application tuple must contain at least two elements, I define the grammar and semantics to enforce this restriction:

\[
p_a ::= \langle\langle\cdot, m_1), m_2 \rangle \mid (p_a, m)
\]
The static and dynamic semantics for these tuples are given below.

\[
\begin{align*}
\Gamma \vdash m_1 : \phi_1 \quad \Gamma \vdash m_2 : \phi_2 & \quad \Gamma \vdash (\langle \cdot, m_1 \rangle, m_2) : \phi_1 \times \phi_2 \\
\Gamma \vdash m : \phi_1 \times \cdots \times \phi_n \quad \Gamma \vdash m' : \phi' & \quad \Gamma \vdash (m, m') : \phi_1 \times \cdots \times \phi_n \times \phi'
\end{align*}
\]

\[
\begin{align*}
\rho \triangleright m_1 & \rightarrow v_1 \quad \rho \triangleright m_2 \rightarrow v_2 & \quad \rho \triangleright (\langle \cdot, m_1 \rangle, m_2) \rightarrow (\langle \cdot, v_1 \rangle, v_2) \\
\rho \triangleright (m, m') & \rightarrow (v_1, v_2) \quad \rho \triangleright (m_1, m_2) \rightarrow (v_1, v_2)
\end{align*}
\]

The type of a partial application tuple is a product type, denoted \( \phi_1 \times \cdots \times \phi_n \) for a partial application tuple of \( n \) elements. In the first rule, the type of a two-element partial application is the product of the types of the individual elements. In the second rule, \( m \) must be a partial application tuple of \( n \) elements (where \( n \) is at least two) whose type is \( \phi_1 \times \cdots \times \phi_n \). If the type of \( m' \) is \( \phi' \), then the type of the tuple \( (m, m') \) must be \( \phi_1 \times \cdots \times \phi_n \times \phi' \). The final pair of rules denote the dynamic semantics of partial applications. The function will not be applied to any of its arguments until all of its arguments are available, so partial application tuples are evaluated simply by separately evaluating the individual elements.

Rather than explicitly introducing a projection operator (like Standard ML's \#i()), which was used in an earlier example) to deconstruct these partial application tuples, I instead introduce a second form of application. Whereas the expression \( m_1 \odot [m_2] \) is a normal application in which a function is applied to an argument, the expression \( m_3 \odot_p m_4 \) is an application in which a partial application tuple is applied to an argument.

The earlier example

\[
\text{letrec } h = \lambda[f].\#1(f) \odot [\#2(f), e_2] \\
\quad k = \lambda[x, y].x \\
\quad \text{in } h \odot (k, e_1)
\]
can be re-written using $@_p$ as follows:

$$\text{letrec } h = \lambda[f]. f \@_p e_2$$

$$k = \lambda[x, y]. x$$

in $h \@ \langle\langle k, e_1 \rangle\rangle$.

In keeping with the restriction that a function cannot be applied until all of its arguments are available, $m_4$ in the above expression must be the final argument to the function in the partial application $m_3$. This restriction is enforced by the static and dynamic semantics for this new application:

$$\Gamma \vdash m_1 : ([\phi_1, \ldots, \phi_n, \phi'] \rightarrow \phi) \times \phi_1 \times \cdots \times \phi_n \quad \Gamma \vdash m_2 : \phi'$$

$$\Gamma \vdash m_1 \@_p m_2 : \phi$$

$$\rho \triangleright m_1 \leftarrow \langle\langle [\rho', \lambda[x_1, \ldots, x_n, y]. m_1], v_1, \ldots, v_n \rangle\rangle$$

$$\rho \triangleright m_2 \leftarrow v' \quad \rho'[x_i \mapsto v_i][y \mapsto v'] \triangleright m \leftarrow v \quad i = [1..n]$$

$$\rho \triangleright m_1 \@_p m_2 \leftarrow v$$

The operational semantics for $@_p$ is similar to the operational semantics for normal application, except that the operator evaluates to a partial application rather than a closure, and the first $n$ arguments to the function are found in the partial application rather than in the operand.

### 5.1.3 Dummy Constants

The target language includes dummy constants since the unCurry transformation may introduce additional parameters to some functions. When these functions are
\[ \Gamma(x) = \phi \quad \frac{}{\Gamma \vdash x : \phi} \quad \Gamma \vdash d^\phi : \phi \]

\[ \Gamma\{x : \phi_1\} \vdash m : \phi_2 \quad \frac{}{\Gamma \vdash \lambda[x]. m : [\phi_1] \to \phi_2} \quad \Gamma\{x_1 : \phi_1\} \vdash \lambda[x_2, \ldots, x_n].m : [\phi_2, \ldots, \phi_n] \to \phi \]

\[ \Gamma \vdash \lambda[x_1, \ldots, x_n].m : [\phi_1, \ldots, \phi_n] \to \phi \]

\[ \Gamma \vdash m : [\phi_1, \ldots, \phi_n] \to \phi \quad \Gamma \vdash m_i : \phi_i \quad i = [1..n] \quad \frac{}{\Gamma \vdash m \oplus [m_1, \ldots, m_n] : \phi} \]

\[ \Gamma \vdash m_1 : ([\phi_1, \ldots, \phi_n, \phi'] \to \phi) \times \phi_1 \times \cdots \times \phi_n \quad \Gamma \vdash m_2 : \phi' \quad \frac{}{\Gamma \vdash m_1 \oplus_p m_2 : \phi} \]

\[ \Gamma \vdash m_1 : \phi_1 \quad \Gamma \vdash m_2 : \phi_2 \quad \frac{}{\Gamma \vdash (\cdot, m_1), m_2 : \phi_1 \times \phi_2} \quad \Gamma \vdash m : \phi_1 \times \cdots \times \phi_n \quad \Gamma \vdash m' : \phi' \quad \frac{}{\Gamma \vdash (m, m') : \phi_1 \times \cdots \times \phi_n \times \phi'} \]

\[ \Gamma' = \Gamma\{g_i : \phi_i\} \quad \Gamma' \vdash m_i : \phi_i \quad \Gamma' \vdash m : \phi \quad \frac{}{\Gamma \vdash \text{letrec } g_i = m_i \text{ in } m : \phi} \]

Fig. 5.1. Target Language Static Semantics
\[
\begin{align*}
\rho(x) &= v \\
\therefore \, &\quad \rho \triangleright x \mapsto v \\
\rho \triangleright d^0 &\mapsto d \\
\rho \triangleright \lambda[x_1, \ldots, x_n].m &\mapsto [\rho, \lambda[x_1, \ldots, x_n].m] \\
\rho \triangleright m &\mapsto [\rho', \lambda[x_1, \ldots, x_n].m'] \\
\rho \triangleright m_i &\mapsto v_i \\
\rho' \{x_i \mapsto v_i\} \triangleright m' &\mapsto v \quad i = [1..n] \\
\rho \triangleright m \otimes [m_1, \ldots, m_n] &\mapsto v \\
\rho \triangleright m_1 &\mapsto \langle \langle [\rho', \lambda[x_1, \ldots, x_n].y].m, v_1, \ldots, v_n \rangle \rangle \\
\rho \triangleright m_2 &\mapsto v' \\
\rho' \{x_i \mapsto v_i\} \triangleright m &\mapsto v \quad i = [1..n] \\
\rho \triangleright m_1 \otimes_p m_2 &\mapsto v \\
\rho \triangleright m_1 &\mapsto v_1 \\
\rho \triangleright m_2 &\mapsto v_2 \\
\rho \triangleright (\langle \langle \cdot, m_1 \rangle, m_2 \rangle) &\mapsto (\langle \langle \cdot, v_1 \rangle, v_2 \rangle) \\
\rho \triangleright (m_1, m_2) &\mapsto (v_1, v_2) \\
\rho \triangleright m_i [\text{letrec } y_j = m_j \text{ in } g_k / g_k] &\mapsto v_i \\
\rho' \{y_i \mapsto v_i\} &\triangleright m \mapsto v \\
\rho \triangleright \text{letrec } y_i = m_i \text{ in } m &\mapsto v \quad i, j, k = [1..n]
\end{align*}
\]

Fig. 5.2. Target Language Dynamic Semantics
applied, the transformation must introduce additional operands to correspond with the
new parameters. Since it does not make sense to conjure up an arbitrary expression, the
transformation uses dummy constants for these additional operands. As in Chapter 3,
dummy constants are annotated with their type and, like ordinary constants, evaluate
to themselves.

The complete grammar for the target language is given below and the static and
dynamic semantics rules for all the expressions are collected in Figures 5.1 and 5.2,
respectively.

\[
\phi \quad ::= \quad [\phi_1, \ldots, \phi_n] \rightarrow \phi \mid \phi_1 \times \cdots \times \phi_n \\
pa \quad ::= \quad ((\cdot, m_1), m_2) \mid (pa, m) \\
m \quad ::= \quad x \mid d^{\phi} \mid pa \mid \lambda[x_1, \ldots, x_n].m \mid m \oplus [m_1, \ldots, m_n] \mid m_1 \oplus_p m_2 \\
\quad \text{letrec } g_i = \lambda x_i. m_i \text{ in } m
\]

5.2 Type Annotations

The transformations for useless-variable elimination and lambda-lifting discussed
in Chapters 3 and 4 used annotated types to relate source and target expressions. The
unCurry transformation described in this chapter does the same, however the annota-
tions used here are a bit more complicated. In particular, unCurrying uses two forms
of annotation, each form serving its distinct purpose. Here is an example of a type
annotated for unCurrying:\footnote{2}
\[ \tau \left[ \tau_1 \ldots \tau_n \right] \rightarrow \gamma \tau' \]

The \( \gamma \), an unCurry annotation, denotes whether or not a function of this type should be unCurried. The \( [\tau_1, \ldots, \tau_n] \), a parameter-type list, contains the types of the arguments that the function of this type has already been applied to.

### 5.2.1 UnCurry Annotations

An unCurry annotation (generically represented by \( \gamma \)) is either \( \varsigma \) or \( \varepsilon \), where \( \varsigma \) indicates that the function should be unCurried and \( \varepsilon \) indicates that the function should not be unCurried.\footnote{3} If the source expression

\[ \lambda x. \lambda y. \lambda z. e \]

was given the annotated type (ignoring parameter-type lists for the moment)

\[ \tau_x \rightarrow \varsigma \tau_y \rightarrow \varsigma \tau_z \rightarrow \varepsilon \tau, \]

then the corresponding target expression would be

\[ \lambda [x, y, z]. m \]

\footnote{2}{Note that the types that are annotated by the unCurry transformation are source language types, not target language types.}

\footnote{3}{These annotations are taken from the unCurry transformation in [14].}
where $e$ corresponds to $m$. If the annotated type was

$$
\tau_x \to^\varsigma \tau_y \to^\varepsilon \tau_z \to^\varepsilon \tau,
$$

then the target expression would be

$$
\lambda[x,y].\lambda[z].m.
$$

Since the annotation on the arrow between the types for parameters $x$ and $y$ is $\varsigma$, the transformation unCurries only those two parameters. On the other hand, if the $\varsigma$ were only on the arrow between $\tau_y$ and $\tau_z$, as in

$$
\tau_x \to^\varepsilon \tau_y \to^\varsigma \tau_z \to^\varepsilon \tau,
$$

then only the $y$ and $z$ would be unCurried, yielding the target expression

$$
\lambda[x].\lambda[y,z].m.
$$

Or course, if none of the arrows were annotated with $\varsigma$, then the transformation would unCurry nothing, giving us

$$
\lambda[x].\lambda[y].\lambda[z].m.
$$

Observe that the annotation on the arrow between $\tau_z$ and $\tau$ must be $\varepsilon$ unless $e$ is another abstraction.
5.2.2 Parameter-Type Lists

When we include the parameter-type lists, the annotated type of the above example (in the first case) would be

\[
\begin{align*}
\tau_x & \to \tau_y \to \tau_z \to \tau_y & \in \tau.
\end{align*}
\]

The lists over the function arrows contain the types of the arguments that the function has been applied to, in order. Recall that in the source language, the expression

\[
\lambda x. \lambda y. \lambda z. e
\]

is a function of one parameter that returns another function. In this example, the type of the parameter is \(\tau_x\) and the type of the return expression is \(\tau_y \to \tau_z \to \tau_y \in \tau\), which I'll call \(\tau'\). Now we can write the type of the expression as \(\tau_x \to \tau'\), and we can plainly see that the list over the arrow is empty, indicating that the function has not been applied to any arguments.

When the function is applied to an actual argument, the result is an expression of type

\[
\tau_y \to \tau_z \to \tau_y \in \tau.
\]

The list over the function arrow in this type is \(\tau_z\), indicating that the function being unCurried has been applied to an argument of type \(\tau_x\). Likewise, when the function is
applied to a second argument, the result is an expression of type

\[ \tau_z \xrightarrow{[\tau_x, \tau_y]} \varepsilon \tau, \]

which tells us that the function has been applied to two arguments, the first of type \( \tau_x \), and the second of type \( \tau_y \).

If a partial application tuple has the type

\[ \tau \xrightarrow{[\tau_1, \ldots, \tau_n]} \varepsilon \tau' \]

then we know that the tuple contains \( n + 1 \) elements, which include the function and the first \( n \) arguments that the function will be applied to. We also know that the types of these \( n \) arguments are \( \tau_1, \ldots, \tau_n \), in order. Since the unCurry annotation is \( \varepsilon \), the function in the partial application will not be unCurried any further. This tells us that the next argument that the function is given (which must be of type \( \tau \)) is the final argument and the function can then be applied.

The exact purpose of these parameter-type lists will become more clear in the next section in which I describe the unCurry transformation itself.

### 5.3 UnCurry Transformation

The formal specification of unCurrying defines the relation between Curried source expressions and unCurried target expressions via the judgment

\[ \Delta; \Gamma_c \triangleright e : \tau \Rightarrow m. \]
This judgment states that, given parameter-type list \( \Delta \) and type context \( \Gamma_c \), the expression \( m \) is an unCurried form of \( e \) and has annotated type \( \tau \). Like the specifications described in the previous chapters, the specification presented here does not define a one-to-one correspondence between source and target terms. A source term may relate to several, possibly infinite, target terms. While the specification does allow for the complete unCurrying of a given expression, any amount of unCurrying (including none at all) is allowed.

Since the unCurry specification is a bit longer than useless-variable elimination or lambda-lifting, I have divided it into three parts, presented in Figures 5.3, 5.5 and 5.6. These rules assume that all variable names in the source and target expressions are unique. I explain each rule in the specification in the following subsections, but first it is necessary to say a little more about \( \Delta \) and \( \Gamma_c \).

In most of the rules, \( \Delta \) is empty. However, if the source expression is an abstraction, then \( \Delta \) contains the types of any parameters that appear to the immediate left of the abstraction in the enclosing expression. If there is no enclosing expression, or if there are no parameters to the left of the source expression, then \( \Delta \) is empty. This parameter-type list is used to annotate the type of the abstraction. For example, if the source expression is \( \lambda z.e \) and is a subexpression of \( \lambda x.\lambda y.\lambda z.e \) in the program, then \( \Delta \) contains the types of \( x \) and \( y \).

The context \( \Gamma_c \) maps source variables to sets containing target variables and their annotated types. If the source variable is a parameter (introduced by an abstraction), then \( \Gamma_c \) relates that variable to a set containing a single target variable and annotated type. If the source variable is a function name (introduced by a letrec expression), then
\( \Gamma_c \) may map the variable to a set containing several target variables and annotated
types. The reason for this is that the unCurry transformation may introduce specialized
functions. Thus a function in the source expression may translate into several functions
in the target expression. I discuss function specialization in detail in Section 5.3.6.

The first set of rules, including those for variables, abstractions and letrec expres-
sions is presented in Figure 5.3.

5.3.1 Variable

There are two rules for variables in the specification: one for function names and
the other for function parameters.

\[
\begin{align*}
g^T &\in \Gamma_c(f) \quad \Rightarrow \quad \Gamma_c \vdash f : \tau \Rightarrow g \quad (\text{funvar}) \\
\Gamma_c(x) &= \{x^T\} \quad \Rightarrow \quad \Gamma_c \vdash x : \tau \Rightarrow x \quad (\text{var})
\end{align*}
\]

The \textit{funvar} rule translates \( f \) to \( g \), where \( g \) is one of the variables appearing in the set
\( \Gamma_c(f) \). Note that, in \( \Gamma_c \), \( g \) must be annotated with the type \( \tau \), which is the same type
in the consequent of the rule. Since the variables in \( \Gamma(f) \) may have different types, this
constraint identifies the variable that should be chosen in translating \( f \).

The \textit{var} rule is actually a special case of \textit{funvar} and its inclusion in the spec-
ification is not actually necessary. I have included the \textit{var} rule to make it clear that
parameters and function names are treated differently. In particular, a parameter is
always translated to itself.\footnote{There is no particular need to change the name of the parameter since I assume all variable
names are unique in the source and target expressions.} This is because the set \( \Gamma(x) \), where \( x \) is a parameter, must
\[
\Gamma_c(x) = \{x^\tau\} \quad (\text{var}) \quad \frac{g^\tau \in \Gamma_c(f)}{\Gamma_c \triangleright f : \tau \Rightarrow g} \quad (\text{funvar}) \\
\frac{\llbracket \Gamma_c \triangleright e : \tau_2 \Rightarrow m \quad \|\tau'_1\| = \tau_1 \quad \text{w.f.}(\tau'_1)}{\Delta; \Gamma_c \vdash \lambda x^{\tau_1}.e : \tau'_1 \xrightarrow{\Delta} \tau_2 \Rightarrow \lambda[x].m} \quad (\text{abs}_c) \\
\Delta + \tau'_1; \Gamma_c \{x : \{x^\tau\}\} \triangleright \lambda y^{\tau_2}.e : \tau_3 \Rightarrow \lambda[y_1, \ldots, y_n].m \quad \|\tau'_1\| = \tau_1 \quad \text{w.f.}(\tau'_1) \quad (\text{abs}_c) \\
\Delta; \Gamma_c \triangleright \lambda x^{\tau_1}.\lambda y^{\tau_2}.e : \tau'_1 \xrightarrow{\Delta} \tau_3 \Rightarrow \lambda[x, y_1, \ldots, y_n].m \\
\llbracket \Gamma'_c \triangleright e_i : \tau_{i_1} \Rightarrow m_{i_1} \ldots \llbracket \Gamma'_c \triangleright e_i : \tau_{i_k_i} \Rightarrow m_{i_k_i} \quad \text{fresh}(g_{i_j}) \\
\Gamma'_c = \Gamma_c \{f_i : \{g_{i_j}\}\} \\
\llbracket \Gamma_c \triangleright \text{letrec } f_i^{\tau_i} = e_i \quad \text{in } e : \tau \Rightarrow \text{letrec } g_{i_j} = m_{i_j} \text{ in } m \\
\Delta \vdash \Gamma_c \triangleright \text{letrec } f_i^{\tau_i} = e_i \quad \text{in } e : \tau \Rightarrow \text{letrec } g_{i_j} = m_{i_j} \text{ in } m \quad i \in \{1..n\} \quad \llbracket \Gamma'_c \triangleright e : \tau \Rightarrow m \quad \|\tau_{i_j}\| = \tau_i \quad \text{w.f.}(\tau_{i_j}) \quad j \in \{1..k_i\} \quad (\text{letrec}) \\
\]

Fig. 5.3. UnCurry Inference Rules (Part 1)
contain only one variable. Thus, unlike \textit{funvar}, \textit{var} offers no choice as to which variable to choose from the set.

### 5.3.2 Abstraction

The abstraction rules include $\text{abs}_\varepsilon$ and $\text{abs}_\zeta$. The first of these can handle all abstractions since it does not unCurry parameters.

$$\begin{array}{c}
\frac{\Gamma_e \vdash e : \tau_2 \Rightarrow m \quad \| \tau'_1 \| = \tau_1 \quad \text{w.f.}(\tau'_1)}{
\Delta; \Gamma_e \vdash \lambda x^{\tau_1}.e : \tau'_1 \vdash \tau_2 \Rightarrow \lambda x.m}
\end{array} \quad (abs_\varepsilon)
$$

In the target expression, the parameter $x$ is included in its own parameter list, and the body of the abstraction is the translated form of $e$. The type of the abstraction is annotated with $\varepsilon$ to indicate that this function will not be further unCurried. If $\Delta$ is nonempty, then it contains the types of the parameters that will be unCurried with $x$ (i.e. parameters bound by $\lambda$s to the left of $\lambda x.e$). Thus $\Delta$ is the parameter-type list in the type of the abstraction.

In the antecedent of the rule, $e$ must translate to some target term $m$, with the type context extended to include the mapping of $x$ to the set $\{x^{\tau_1}\}$. The parameter-type list in this judgment must be empty since either

1. $e$ is not an abstraction, or
2. $x$ (and the parameters whose types are in $\Delta$) will not be unCurried with $e$'s parameter.
The remaining constraints in this rule involve the types $\tau_1$ and $\tau'_1$. Variables in the source language are annotated with their types. These are un-annotated source language types. The constraint $\|\tau'_1\| = \tau_1$ relates the annotated type $\tau'_1$ and the un-annotated type $\tau_1$. Put simply, this constraint says that, if all annotations are removed from $\tau'_1$ (denoted by $\|\cdot\|$), the result is $\tau_1$. The remaining constraint requires that $\tau'_1$ be a well-formed type. I delay discussing well-formed types until later in the chapter since this constraint is included to guarantee type soundness. Section 5.4.1 includes formal definitions of both $\|\cdot\|$ and $w.f.(\cdot)$ (see Definitions 14 and 18).

The second abstraction rule, $\text{abs}_\varsigma$, unCurries the parameters of consecutive $\lambda$s.

$$
\frac{
\Delta + \tau'_1; \Gamma_c \{ x : \{ x' \} \} \triangleright \lambda y^{\tau_2} . e : \tau_3 \Rightarrow \lambda [y_1, \ldots, y_n]. m \\
\| \tau'_1\| = \tau_1 \\
w.f.(\tau'_1)
}{
\Delta; \Gamma_c \triangleright \lambda x^{\tau_1} . \lambda y^{\tau_2} . e : \tau'_1 \overset{\varsigma}{\Delta} \tau_3 \Rightarrow \lambda [x, y_1, \ldots, y_n]. m
} \quad \text{ (abs}_\varsigma \text{)}
$$

The $\varsigma$ annotation in the type of the abstraction indicates that $x$ will be unCurried with $y$. Once again, the set $\Delta$ contains the types of parameters bound by $\lambda$s appearing to the left of $\lambda x$ in the enclosing expression that will also be unCurried with $x$ and $y$. We can see in this rule how this parameter-type list is built. In the antecedent of $\text{abs}_\varsigma$, annotated type $\tau'_1$ is added to the end of $\Delta$ (denoted $\Delta + \tau'_1$). Since $x$ will be unCurried with $y$, the type of $x$ must be included in the parameter-type list in the judgment for $\lambda y.e$.

If $e$ is an abstraction, then more unCurrying may be done. Thus the judgment in the antecedent of $\text{abs}_\varsigma$ relates the source expression $\lambda y.e$ to the target expression $\lambda [y_1, \ldots, y_n]. m$, where $n \geq 1$, to allow for the possibility that $y$ may be unCurried with
additional parameters. In the consequent of the rule, the source expression is translated by adding \( x \) to the parameter list of the target expression.

Figure 5.4 shows part of the deduction for the translation of \( \lambda x. \lambda y. \lambda z. e \) into the unCurried expression \( \lambda [x, y, z]. m \) using two instances of the \( \text{abs}_\zeta \) rule and an instance of the \( \text{abs}_\epsilon \) rule. The types \( \tau'_x, \tau'_y \) and \( \tau'_z \) are annotated forms of \( \tau_x, \tau_y \) and \( \tau_z \), respectively. Note that the parameter-type list in the topmost judgment is empty since the parameters \( x, y \) and \( z \) are not unCurried with any parameters in \( e \). I have excluded the type constraints in the figure.

5.3.3 Letrec

The \textit{letrec} rule allows the translation to introduce specialized functions. A function defined by \textit{letrec} in the source expression can be translated several different ways, resulting in different versions of the function in the target expression. While I describe the \textit{letrec} rule here, I defer a more thorough discussion of function specialization and
how it applies to unCurrying until later in the chapter.

\[
\begin{align*}
\emptyset & ; \Gamma'_{c} \triangleright e_{i} : \tau_{i} \Rightarrow m_{i} \cdots \emptyset ; \Gamma'_{c} \triangleright e_{i} : \tau_{i} \Rightarrow m_{i} \quad \text{fresh}(g_{i}) \\
\Gamma'_{c} & = \Gamma'_{c}[f_{i} : \{ g_{i,j} \}] \quad \emptyset ; \Gamma'_{c} \triangleright e : \tau \Rightarrow m \quad \| \tau_{i,j} \| = \tau_{i} \quad \text{w.f.}(\tau_{i,j}) \\
\emptyset ; \Gamma'_{c} \triangleright \text{letrec } f_{i}^{\ast} = e_{i} \text{ in } e : \tau \Rightarrow \text{letrec } g_{i,j}^{\ast} = m_{i,j} \text{ in } m
\end{align*}
\]

(\text{letrec})

In the \text{letrec} rule, source functions are subscripted with \(i\) and target functions are subscripted with \(i\) and \(j\). The \(i\) ranges from 1 to \(n\), where \(n\) is the number of functions defined by the \text{letrec} in the source expression. For each function \(f_{i}\), the \(j\) subscript ranges from 1 to \(k_{i}\), where \(k_{i}\) is the number of specialized versions of \(f_{i}\) that are defined by the \text{letrec} in the target expression. For example, say that three specialized versions of function \(f_{2}\) are created for the target expression. In this case, \(k_{2} = 3\) and the specialized versions of \(f_{2}\) in the target expression are named \(g_{21}, g_{22}\) and \(g_{23}\).

In the antecedent of the \text{letrec} rule, each abstraction (i.e. the expressions \(e_{1}\) through \(e_{n}\)) may be translated several times. Specifically, for each function \(f_{i}\), there are \(k_{i}\) judgments of the form \(\emptyset ; \Gamma'_{c} \triangleright e_{i} : \tau_{i} \Rightarrow m_{i} \). In total, there are \(\sum_{i=1}^{n} k_{i}\) such judgments. The type context \(\Gamma'_{c}\) is the context \(\Gamma_{c}\) extended to include mappings of each function name \(f_{i}\) to the corresponding set of specialized function names \(g_{i,j}\) and their types \(\tau_{i,j}\). The types of specialized functions, for each \(f_{i}\), differ only in their annotations, as enforced by the constraint \(\| \tau_{i,j} \| = \tau_{i}\), where \(\tau_{i}\) is the given type of \(f_{i}\) in the source expression.
5.3.4 Application Rules

All of the application rules, except those dealing explicitly with dummy operands, are presented in Figure 5.5. These rules are divided into two groups: those for $\varepsilon$-annotated operators and those for $\zeta$-annotated operators. The first group translates applications into applications, since all of the function’s arguments are available. The second group must translate applications into partial application tuples.

$\varepsilon$-Annotation Rules

The first rule in this group translates applications in which the operator is a function of one parameter.

$$
\frac{\emptyset; \Gamma_c \triangleright e_1 : \tau' \triangleright_{\varepsilon} \tau \Rightarrow m_1 \quad \emptyset; \Gamma_c \triangleright e_2 : \tau' \Rightarrow m_2}{\emptyset; \Gamma_c \triangleright e_1 \ @ e_2 : \tau \Rightarrow m_1 \ @ [m_2]} \quad (app_{\varepsilon 1})
$$

The type of the operator must be of the form $\tau' \triangleright_{\varepsilon} \tau$. The $\varepsilon$ annotation indicates that the next argument that the function is applied to is the last argument the function needs. In this rule above, that argument is the operand $e_2$. The fact that the parameter-type list over the function arrow is empty indicates that the function has not previously been applied to other arguments. Thus the operator $e_1$ is not a partial application. The target expression is simply the unCurried operator applied to a parameter list consisting of the unCurried operand.

The second rule in this group handles the special case in which the operator has type $\tau' \triangleright_{\varepsilon} \tau$ and translates directly into a partial application tuple of $n + 1$
\[
\begin{align*}
&
\frac{
\begin{array}{l}
[]; \Gamma_C \triangleright e_1 : \tau' \quad \square \tau \Rightarrow m_1 \\
\end{array}
\begin{array}{l}
\square \tau \Rightarrow m_2 \\
\end{array}
}{
\begin{array}{l}
[]; \Gamma_C \triangleright e_1 \circ e_2 : \tau \Rightarrow m_1 \circ [m_2] \\
\end{array}
}
(app_\varepsilon_1)
\\
&
\frac{
\begin{array}{l}
[]; \Gamma_C \triangleright e_1 : \tau' \quad \{\tau_1, \ldots, \tau_n\} \quad \varepsilon \tau \Rightarrow (m_0, \ldots, m_n) \\
\end{array}
\begin{array}{l}
\square \tau \Rightarrow m_0 \circ [m_1, \ldots, m_n, m] \\
\end{array}
}{
\begin{array}{l}
[]; \Gamma_C \triangleright e_1 \circ e_2 : \tau \Rightarrow m_0 \circ [m_1, \ldots, m_n, m] \\
\end{array}
}
(app_\varepsilon_2)
\\
&
\frac{
\begin{array}{l}
[]; \Gamma_C \triangleright e_1 : \tau' \quad \{\tau_1, \ldots, \tau_n\} \quad \varepsilon \tau \Rightarrow m_1 \\
\end{array}
\begin{array}{l}
\square \tau \Rightarrow m_2 \\
\end{array}
}{
\begin{array}{l}
[]; \Gamma_C \triangleright e_1 \circ e_2 : \tau \Rightarrow m_1 \circ_p m_2 \\
\end{array}
}
(app_\varepsilon_3)
\\
&
\frac{
\begin{array}{l}
[]; \Gamma_C \triangleright e_1 : \tau' \quad \{\tau_1, \ldots, \tau_n\} \quad \varepsilon \tau \Rightarrow m_1 \\
\end{array}
\begin{array}{l}
\square \tau \Rightarrow m_2 \\
\end{array}
}{
\begin{array}{l}
[]; \Gamma_C \triangleright e_1 \circ e_2 : \tau \Rightarrow (\cdot, m_1), m_2 \\
\end{array}
}
(app_\varepsilon_1)
\\
&
\frac{
\begin{array}{l}
[]; \Gamma_C \triangleright e_1 : \tau' \quad \{\tau_1, \ldots, \tau_n\} \quad \varepsilon \tau \Rightarrow m_1 \\
\end{array}
\begin{array}{l}
\square \tau \Rightarrow m_2 \\
\end{array}
}{
\begin{array}{l}
[]; \Gamma_C \triangleright e_1 \circ e_2 : \tau \Rightarrow (m_1, m_2) \\
\end{array}
}
(app_\varepsilon_2)
\end{align*}
\]

Fig. 5.5. UnCurry Inference Rules (Part 2)
elements.

\[
\begin{align*}
\text{[]} ; \Gamma_c \triangleright e_1 : \tau' &\rightarrow^{\varepsilon} \tau \Rightarrow \langle (m_0, \ldots, m_n) \rangle \\
\text{[]} ; \Gamma_c \triangleright e_2 : \tau' &\Rightarrow m \quad (app_{\varepsilon 2})
\end{align*}
\]

Since the \(\varepsilon\) annotation indicates that the unCurried form of \(e_2\) is the final argument for the function and the function itself along with its first \(n\) arguments are readily available as the target expressions \(m_0, \ldots, m_n\), the rule can statically construct the unCurried application without having to introduce \(\oplus_p\) in the target expression to dynamically deconstruct the partial application tuple.

The final rule in the group translates all other applications in which the operator’s type is annotated with \(\varepsilon\).

\[
\begin{align*}
\text{[]} ; \Gamma_c \triangleright e_1 : \tau' &\rightarrow^{\varepsilon} \tau \Rightarrow m_1 \\
\text{[]} ; \Gamma_c \triangleright e_2 : \tau' &\Rightarrow m_2 \\
\text{[]} ; \Gamma_c \triangleright e_1 \oplus e_2 : \tau \Rightarrow m_1 \oplus_p m_2 \quad (app_{\varepsilon 3})
\end{align*}
\]

Since the parameter-type list in the type of the operator is non-empty, the operator must be a partial application. If \(e_1\) translated directly into a partial application tuple, then the rule \(app_{\varepsilon 2}\) would apply. So the unCurried form of the operator, \(m_1\), may be some expression whose value is a partial application tuple, but \(m_1\) itself is some other form of expression. Thus it is necessary to introduce \(\oplus_p\) in the target expression so that \(m_1\) can be dynamically evaluated and deconstructed to find the function and the first \(n\) arguments.
\(\zeta\)-Annotation Rules

The remaining two rules for application handle the cases in which the type of the operator is annotated with \(\zeta\). Such an annotation indicates that the next argument the function is applied to is not the last argument. Since, in the target expression, functions are not applied until the last argument is available, these rules translate a source expression application into a target expression partial application.

In \(app_{\zeta 1}\), the parameter-type list in the type of the operator is empty.

\[
\frac{\begin{align*}
\Gamma_c &\triangleright e_1 : \tau' \\
\Gamma_c &\triangleright e_2 : \tau \Rightarrow m_1 \\
\Gamma_c &\triangleright (e_1 @ e_2) : \tau \Rightarrow (\tau_m, m_2)
\end{align*}}{
\begin{align*}
\Gamma_c &\triangleright e_1 : \tau' \\
\Gamma_c &\triangleright e_2 : \tau \Rightarrow m_1 \\
\Gamma_c &\triangleright (e_1 @ e_2) : \tau \Rightarrow (\tau_m, m_2)
\end{align*}} \quad (app_{\zeta 1})
\]

This means that the next argument the operator is applied to is the first argument. The target expression, therefore, is a partial application consisting of the unCurried function \(m_1\) along with \(m_2\), the unCurried form of the first argument.

In the final application rule, the parameter-type list over the function arrow is non-empty. Thus \(m_1\) is a partial application.

\[
\frac{\begin{align*}
\Gamma_c &\triangleright e_1 : \tau' [\tau_1, \ldots, \tau_n] \\
\Gamma_c &\triangleright e_2 : \tau \Rightarrow m_1 \\
\Gamma_c &\triangleright (e_1 @ e_2) : \tau \Rightarrow (m_1, m_2)
\end{align*}}{
\begin{align*}
\Gamma_c &\triangleright e_1 : \tau' \\
\Gamma_c &\triangleright e_2 : \tau \Rightarrow m_1 \\
\Gamma_c &\triangleright (e_1 @ e_2) : \tau \Rightarrow (m_1, m_2)
\end{align*}} \quad (app_{\zeta 2})
\]

Since the annotation on the function arrow is \(\zeta\), the unCurried operand, \(m_2\), is not the function's final argument. So the target expression is the partial application tuple \((m_1, m_2)\), which is the expression \(m_2\) appended to the tuple \(m_1\).
Application Examples

The partial deduction given below illustrates the use of rules $app_{\zeta 1}$ and $app_{\zeta 2}$ in constructing a partial application tuple. Here, $f$ is a function of three parameters, but is only applied to the first two arguments. The result is the target expression $(((\cdot, g), m_1), m_2)$, which can also be written $\langle (g, m_1), m_2 \rangle$.

\[
\vdash \Gamma_c \mid f: \tau_x \to \tau_y \quad [\tau_x, \tau_y] \to \varepsilon \quad \tau \Rightarrow g \quad \vdash \Gamma_c \mid e_1: \tau_x \Rightarrow m_1 \quad app_{\zeta 1} \\
\vdash \Gamma_c \mid f \circ e_1: \tau_x \to \tau_y \quad [\tau_x, \tau_y] \to \varepsilon \quad \tau \Rightarrow ((\cdot, g), m_1) \\
\vdash \Gamma_c \mid (f \circ e_1) \circ e_2: \tau_x \to \varepsilon \quad \tau \Rightarrow (((\cdot, g), m_1), m_2) 
\]

The second example illustrates the use of $app_{\zeta 3}$ in introducing $\circ_p$ to apply a partial application tuple to its final argument. Suppose $m_1$ is some expression that evaluates to a partial application tuple, however the contents of this tuple are not statically available. The partial deduction below uses rule $app_{\zeta 2}$ to create the target expression $(m_1, m_2)$. Recall that this represents the expression $m_2$ appended to $m_1$. Since the contents of $m_1$ are not statically available, we cannot use rule $app_{\zeta 2}$ to translate $(e_1 \circ e_2) \circ e_3$. The only option is to use $app_{\zeta 3}$ to introduce $\circ_p$ to deconstruct the tuple dynamically.

\[
\vdash \Gamma_c \mid e_1: \tau_y \to \tau_x \quad [\tau_x, \tau_y] \to \varepsilon \quad \tau \Rightarrow m_1 \quad \vdash \Gamma_c \mid e_2: \tau_y \Rightarrow m_2 \quad app_{\zeta 2} \\
\vdash \Gamma_c \mid e_1 \circ e_2: \tau_x \to \varepsilon \quad \tau \Rightarrow (m_1, m_2) \\
\vdash \Gamma_c \mid (e_1 \circ e_2) \circ e_3: \tau \Rightarrow (m_1, m_2) \circ_p m_3 
\]
The final example illustrates the use of \texttt{app}_2 when all of a function's arguments are available in the same sequence of applications. Because of space constraints, I do not show the translation of the operands in the partial deduction below. Assume source expressions \(e_1\), \(e_2\) and \(e_3\) translate to target expressions \(m_1\), \(m_2\) and \(m_3\), respectively.

\[
\begin{align*}
\vdash &: \textsf{\texttt{letrec}} f = \lambda x. \lambda y. e_1 \\
& g = \lambda z. e_2 \\
& \text{in } \cdots e_3 \text{ @ } e_4 \cdots
\end{align*}
\]

5.3.5 Useless-Variable Rules

It may seem odd to include rules that actually introduce useless parameters, but it turns out that some expressions cannot be completely unCurried without them. Hence these parameters are useless in the sense that their corresponding operands do nothing,\(^5\) but the parameters actually do serve a purpose in the target expression. Suppose we have two functions defined in a \texttt{letrec} expression, where the type of \(f\) is \(\tau_1 \rightarrow_{\xi} \tau_2 \rightarrow_{\varepsilon} \tau_3\) and the type of \(g\) is \(\tau_2 \rightarrow_{\xi} \tau_3\):

\[
\begin{align*}
\vdash &: \textsf{\texttt{letrec}} f = \lambda x. \lambda y. e_1 \\
& g = \lambda z. e_2 \\
& \text{in } \cdots e_3 \text{ @ } e_4 \cdots
\end{align*}
\]

\(^5\)These operands are the dummy constants mentioned in Section 5.1.
Somewhere in the body of the letrec there is the application $e_3 @ e_4$, where the unCurried form of $e_3$ (which I'll call $m_3$) evaluates to either the unCurried function $g$ or a partial application including unCurried $f$ and its first argument. This means that the type of $m_3$ is either $\tau_2 \rightarrow_{\varepsilon} \tau_3$ or $\tau_2 \rightarrow_1 \tau_3$. If the value of $m_3$ cannot be determined statically, how do we translate this application? Do we use $app_{\varepsilon 1}$ or $app_{\varepsilon 3}$?

If we use $app_{\varepsilon 1}$, the unCurried application would be $m_3 @ [m_4]$. Unfortunately this presents a rather significant problem if $m_3$ turns out to be the partial application: the application only applies the function to one argument, but $f$ requires two arguments. On the other hand, if we use $app_{\varepsilon 3}$, the source application turns into the target expression $m_3 @_p m_4$. The problem here, of course, is that $g$ is not a partial application and it does not make sense to use $@_p$ to apply it to its argument.

There are two ways of resolving this issue. If we do not unCurry $f$, then its annotated type is $\tau_1 \rightarrow_\varepsilon \tau_2 \rightarrow_\varepsilon \tau_3$ and we can safely use $app_{\varepsilon 1}$ to translate $e_3 @ e_4$. While this works, it does not allow us to completely unCurry each function. The only thing preventing us from unCurrying $f$ is the fact that $g$ has only one parameter. But with the useless-variable rules introduced in this section, the translation can create a second, useless, parameter for $g$. Now, $m_3$ will either be the partial application of $f$ to its first argument, or the partial application of $g$ to a dummy constant. This allows us to safely use $app_{\varepsilon 3}$, resulting in the target expression

$$\textbf{letrec } f = \lambda [x,y]. m_1$$
$$g = \lambda [w,z]. m_2$$
$$\text{in } \cdots m_3 @_p m_4 \cdots$$
\[ \Delta + \tau_1; \Gamma_c \triangleright \lambda x^\tau. e : \tau_2 \Rightarrow \lambda[x_1, \ldots, x_n]. m \quad \text{fresh}(y) \quad \text{w.f.}(\tau_1) \quad (uvabs) \]

\[ \Delta; \Gamma_c \triangleright \lambda x^\tau. e : \tau_1^u \xrightarrow{\Delta} \tau_2 \Rightarrow \lambda[y, x_1, \ldots, x_n]. m \]

\[ \frac{[\ ]; \Gamma_c \triangleright e : \tau' \Rightarrow m}{[\ ]; \Gamma_c \triangleright e : \tau' \Rightarrow ([\ ], m, d^{d^\tau}(\tau))} \quad (uvapp_1) \]

\[ \frac{[\ ]; \Gamma_c \triangleright e : \tau [\tau_1; \ldots; \tau_n] \xrightarrow{\tau_1} \tau' \Rightarrow m}{[\ ]; \Gamma_c \triangleright e : \tau' \Rightarrow (m, d^{d^\tau}(\tau))} \quad (uvapp_2) \]

Fig. 5.6. UnCurry Inference Rules (Part 3)

If \( m_3 \) is a partial application tuple, then we can use \( app_{c2} \) and avoid introducing \( \oplus_p \).

In order for the translation to be able to keep track of useless parameters, I introduce yet another type annotation: \( u \). If a type is affixed with \( u \), then the parameter of that type is useless. In the example above, the annotated type of \( g \) is \( \tau_1^u \xrightarrow{\Delta} \tau_2 \xrightarrow{\tau_1 \in} \tau_3 \) since the first parameter of the unCurried form of \( g \) is useless. Note that the type of the first parameter inside the parameter-type list is not affixed with \( u \). The type of \( f \) in this example is \( \tau_1 \xrightarrow{\Delta} \tau_2 \xrightarrow{\tau_1 \in} \tau_3 \).

The final set of rules for the specification of the unCurry transformation, those dealing with useless parameters, are presented in Figure 5.6.

**Introducing Useless Parameters**

The abstraction rule below introduces a useless parameter to an abstraction. The specification allows any abstraction to be translated in such a fashion, however this rule
should only be used when necessary.

\[
\frac{\Delta + \tau_1; \Gamma_c \triangleright \lambda x^\tau . e : \tau_2 \Rightarrow \lambda [x_1, \ldots, x_n]. \text{m} \quad \text{fresh}(y) \quad \text{w.f.}(\tau_1)}{(uv.abs)}
\]

\[
\Delta; \Gamma_c \triangleright \lambda x^\tau . e : \tau_1^u \xrightarrow{\Delta} \tau_2 \Rightarrow \lambda [y, x_1, \ldots, x_n]. \text{m}
\]

If the annotated type of the source abstraction is \( \tau_2 \), then the type of the target abstraction with the useless parameter is \( \tau_1^u \xrightarrow{\Delta} \tau_2 \), where \( \tau_1 \) is the type of the useless parameter and \( \Delta \) is the given parameter-type list. The unCurry annotation is \( \varsigma \) because useless parameters are only added to abstractions, so there is always at least one parameter to the right of the useless parameter. In the target expression, the useless parameter, which must be a new variable, appears at the beginning of the parameter list. When translating the abstraction in the antecedent of the rule, the type of the useless parameter (excluding the \( u \) annotation) is included in \( \Delta \).

**Introducing Dummy Arguments**

For every useless parameter there must be a useless operand. Each of the rules below translates a source expression with a function type in which the type of the parameter is affixed with \( u \) into a partial application including the unCurried form of the source expression and a dummy constant. The rules \( uvapp_1 \) and \( uvapp_2 \) correspond to
the application rules $app_{<1}$ and $app_{<2}$.

\[
\frac{\emptyset; \Gamma_c \triangleright e : \tau^u \vdash_{\prec} \tau' \Rightarrow m}{\emptyset; \Gamma_c \triangleright e : \tau' \Rightarrow (\cdot, m, d[^{(\tau)}]_\emptyset) (uvapp_1)}
\]

\[
\frac{\emptyset; \Gamma_c \triangleright e : \tau^u [\tau_1 \rightarrow \cdots \rightarrow \tau_n] \vdash_{\prec} \tau' \Rightarrow m}{\emptyset; \Gamma_c \triangleright e : \tau' \Rightarrow (m, d[^{(\tau)}]_\emptyset) (uvapp_2)}
\]

In the first rule the useless parameter is the function's first parameter. We know this because the parameter-type list over the function arrow is empty. Before the function can be applied to any of its other arguments, it must be applied to a dummy constant. So the rule translates the source expression into a partial application tuple consisting of the unCurried form of $e$ and a dummy constant.

Recall that in the target language dummy constants are annotated with their types. However these types are target language types. As a result the dummy constant cannot be annotated with $\tau$, as this is a type annotated with unCurry annotations and parameter-type lists. Instead, the dummy constant is annotated with $\|\langle\tau\rangle\|_\emptyset$, where the $\|\langle\cdot\rangle\|_\Delta$ notation relates annotated types to target language types just as $\|\cdot\|$ relates annotated types to source language types. These relations are formally defined in Section 5.4.1.

The second rule is similar to the first, except that the useless parameter is not the function's first argument. Thus, in the target expression, the dummy constant is appended to the unCurried form of the source expression (which is a partial application).
5.3.6 Function Specialization

Let us return to the example from Section 5.3.5 in which a useless parameter is added to one function \( g \) to enable the unCurrying of another \( f \). Suppose now that there is a third function \( h \) and a second application site:

\[
\text{letrec } f = \lambda x. \lambda y. e_1 \\
g = \lambda z. e_2 \\
h = \lambda k. e_5 \\
in \cdots e_3 \circ e_4 \cdots e_6 \circ e_7 \cdots
\]

Recall that \( e_3 \) evaluates to either \( g \) or the partial application of \( f \) to its first argument. Suppose \( e_6 \) is an expression that evaluates to either \( g \) or \( h \). Since both \( g \) and \( h \) are functions of one parameter, this application does not seem to pose a problem. But the transformation adds a useless parameter to \( g \) so that \( f \) may be unCurried. Now \( g \) is a function of two parameters and we have a problem. How do we translate the second application? The obvious solution is to add a useless parameter to \( h \), which results in the following unCurried expression:

\[
\text{letrec } f = \lambda [x, y]. m_1 \\
g = \lambda [w_1, z]. m_2 \\
h = \lambda [w_2, k]. m_5 \\
in \cdots m_3 \circ_p m_4 \cdots m_6 \circ_p m_7 \cdots
\]
While this is correct, it seems absurd to introduce a useless parameter to $h$ simply because $g$ has a useless parameter. Suppose there are several other application sites in the program. The useless parameter that is added to $g$ can have a ripple-effect throughout the program, forcing the transformation to add useless parameters to several other functions.

There is a way to avoid such unnecessary consequences. The occurrence of $g$ in $e_3$ should have a useless parameter, but the occurrence of $g$ in $e_6$ should not. So, we create two specialized versions of $g$ in the translated term, one with a useless parameter and the other without. The result is the following unCurried expression:

$$
\text{letrec } f = \lambda[x, y].m_1 \\
g_1 = \lambda[w, z_1].m_2 \\
g_2 = \lambda[z_2].m_2 \\
h = \lambda[k].m_5 \\
\text{in } \cdots m'_3 \circ_p m_4 \cdots m'_6 \circ [m_7] \cdots
$$

In this expression, the occurrence of $g$ in $m'_3$ is replaced with $g_1$, and the occurrence of $g$ in $m'_6$ is replaced with $g_2$. By creating specialized versions of $g$, the transformation does not add a useless parameter to $h$. 
Consider a slightly more complicated example. For this example, assume we have an if-then-else expression in our source and target languages.

\[
\text{letrec } f = \lambda x_1.((\lambda z. \lambda y_1.e_1) \circ e_2) \\
g = \lambda x_2.\lambda y_2.e_3 \\
h = \lambda x_3.\lambda y_3.e_4 \\
in \cdots (\text{if } e_5 \text{ then } f \text{ else } g) \circ e_6 \circ e_7 \cdots
\]

The problem here is that \( f \) and \( g \) can be called from the same application site and \( f \) is a function of one parameter whereas \( g \) is a function of two parameters. If we wish to completely unCurry the functions, their types would be:

\[
f \quad : \quad \tau_x \xrightarrow{\varepsilon} \tau_y \xrightarrow{\varepsilon} \tau \\
g \quad : \quad \tau_x \xrightarrow{\varepsilon} \tau_y \xrightarrow{\varepsilon} \tau
\]

But since both functions appear at the same application site, they must have the same type. The only difference between their types is the unCurry annotations on the first arrow and the parameter-type lists on the second. We'll focus on the unCurry annotations first.
It may be tempting to add a parameter to \( f \), using \( \eta \)-expansion, to get the following expression:

\[
\text{letrec } f = \lambda x_1.\lambda y.((\lambda z.\lambda y_1.e_1) \circ e_2 \circ y) \\
g = \lambda x_2.\lambda y_2.e_3 \\
h = \lambda x_3.\lambda y_3.e_4 \\
in \ldots (\text{if } e_5 \text{ then } f \text{ else } g) \circ e_6 \circ e_7 \ldots
\]

Now \( f \) is a two-parameter function with the same type as \( g \), annotations and all. Unfortunately, this is not a correct transformation. By \( \eta \)-expanding, we have altered the order in which expressions are evaluated in a call-by-value semantics. In the expression above, \( e_7 \) is evaluated before \( e_2 \). In the original expression, \( e_2 \) is evaluated before \( e_7 \).

Instead of altering \( f \) to turn the first unCurry annotation in its type into \( \varsigma \), we can alter \( g \) to turn the first unCurry annotation in its type to \( \varepsilon \). This is done simply by not unCurrying \( g \) (i.e. when translating the function, we use the rule \( \text{abs}^\varepsilon \) instead of \( \text{abs}^\varsigma \)). The function then becomes \( \lambda [x_2].\lambda [y_2].m_4 \) in the target expression. Now the types of the functions are:

\[
\begin{align*}
  f & : \tau_x \mid \tau_y \xrightarrow{\varepsilon} \tau_y \xrightarrow{\varepsilon} \tau \\
  g & : \tau_x \mid \tau_y \xrightarrow{\varepsilon} \tau_y \xrightarrow{\varepsilon} \tau
\end{align*}
\]

Now the unCurry annotations match but the parameter-type lists still do not. This is easily remedied by introducing a useless parameter (of type \( \tau_2 \)) to \( g \)'s second parameter
list, giving us \( \lambda[x_2].\lambda[w, y_2].m_3 \). The type of this expression is

\[
\tau_x \nabla \varepsilon \overset{u}{\tau_z} \nabla \varepsilon \overset{y}{\tau_y} \overset{[\tau_z]}{\varepsilon} \tau.
\]

Now all that remains is to introduce the dummy constant to get rid of the \( \overset{u}{\tau_z} \) in \( g \)'s type: \( \lambda[x_2].\langle\langle \lambda[w, y_2].m_3, d^{[\tau_z]}[\varepsilon]\rangle\rangle \). Now the types of \( f \) and \( g \) match as they should.

Suppose, however, that \( g \) and \( h \) can appear at the same application site elsewhere in the program. If \( g \) is modified as above, \( h \) cannot be unCurried. But in the original expression, both are functions of two parameters and should be unCurried. Once again, function specialization comes to the rescue. We can introduce two versions of \( g \), one of which is completely unCurried. This allows us to safely unCurry \( h \). The entire unCurried expression becomes:

\[
\text{letrec } f = \lambda[x_1].\langle\langle \lambda[z, y_1].m_1, m_2\rangle\rangle \\
g_1 = \lambda[x_2].\langle\langle \lambda[w, y_2].m_3, d^{[\tau_z]}[\varepsilon]\rangle\rangle \\
g_2 = \lambda[x_2, y_2].m_3 \\
h = \lambda[x_3, y_3].m_4 \\
in \cdots ((\text{if } m_5 \text{ then } f \text{ else } g_1) \circ [e_6] \circ_p m_7 \cdots
\]

Wherever \( g \) and \( h \) are called at the same application site, \( g_2 \) should be used instead of \( g_1 \).

In both of the above examples, only two specialized versions of source functions are created, however the flexibility of the \textit{letrec} rule in Figure 5.3 allows any number of specialized functions to be created.
5.4 Correctness

In this section, I address the type soundness and operational correctness of the unCurry specification.

Type soundness ensures that the types of expressions are preserved by the transformation. If the source expression is well-typed, then the target expression is well-typed. This property allows unCurrying to safely be used in conjunction with other type-based compiler optimizations.

Operational correctness guarantees that the transformation does not alter the meaning of an expression. If the source expression evaluates to a value, then the target expression will evaluate to an unCurried form of that same value. This, of course, is a vital property for any compiler optimization.

With these two results, a compiler writer can use any unCurry algorithm that has been proved sound with respect to the unCurry specification presented in this chapter with the assurance that the algorithm produces a target expression that is well-typed and operationally equivalent to the source expression.

5.4.1 Type Soundness

In order to state a theorem of type soundness, I first define relations among source language types, target language types and annotated types. We have already seen some of these relations in the inference rules of the unCurry specification.

The $\| \cdot \|$ and $\cdot |_\Delta$ operators erase type annotations from a given type.
Definition 14 ($\parallel \cdot \parallel$).

\[
\parallel \tau \xrightarrow{\Delta \gamma} \tau' \parallel = \parallel \tau \parallel \rightarrow \parallel \tau' \parallel \\
\parallel \tau^u \xrightarrow{\Delta \gamma} \tau' \parallel = \parallel \tau' \parallel
\]

The first operator yields a source language type simply by removing unCurry annotations and parameter-type lists. If we wish to unCurry $e$ and the transformation infers the annotated type $\tau$, then the source language type of $e$ is $\parallel \tau \parallel$. Any type that is affixed with a $u$ represents a useless parameter that has been added by the transformation. Since such parameters do not appear in the source expression, they should not appear in the expression’s source language type.

The $|\cdot|_\Delta$ operator assumes that the given annotated type contains no occurrences of the useless parameter annotation $u$. The following definition introduces a means of removing such annotations.

Definition 15 ($\langle \cdot \rangle$).

\[
\langle \tau \xrightarrow{\Delta \gamma} \tau' \rangle = \langle \tau \rangle \xrightarrow{\Delta \gamma} \langle \tau' \rangle \\
\langle \tau^u \xrightarrow{\Delta \gamma} \tau' \rangle = \langle \tau \rangle \xrightarrow{\Delta \gamma} \langle \tau' \rangle
\]

The definition of $|\cdot|_\Delta$ (given below as Definition 16) is not as simple as Definition 14 because relating annotated types to target language types is not merely a matter of erasing annotations. The difficulty arises in interpreting the annotations correctly in determining the types of unCurried functions and partial application tuples.
Suppose we have an unCurried expression \( m \) whose annotated type is \( \tau \xrightarrow{[\tau_1,\tau_2]} \tau' \).

What does this type tell us about \( m \)? By examining the inference rules for unCurrying, we see that \( m \) is either a function or a partial application tuple. If \( m \) is a function, then its target language type is \([\tau_1, \tau_2, \tau] \rightarrow \tau'\). On the other hand, if \( m \) is a partial application tuple, then its target language type is \( ([\tau_1, \tau_2, \tau] \rightarrow \tau') \times \tau_1 \times \tau_2 \). The unCurry transformation annotates types to include information pertaining specifically to unCurrying. So these annotated types only make sense when considered within the context of the unCurry transformation.

If \( m \) is a function, then it is unCurried using one of the abstraction rules from the specification. Notice that in each abstraction rule the given parameter-type list (the list paired with the type context before the \( \triangleright \) symbol) is the same as the parameter-type list in the annotated type. So whichever abstraction rule unCurries \( m \), the given parameter-type list must be \([\tau_1, \tau_2]\). In every other rule the given parameter-type list is empty regardless of the annotated type. We can then relate an annotated type to a target language type by comparing the parameter-type list over the function arrow in the annotated type to the given parameter-type list. If \( m \) has type \( \tau \xrightarrow{[\tau_1,\tau_2]} \tau' \) and the given parameter-type list is \([\tau_1, \tau_2]\), then the target language type should be a function type. Otherwise, the target language type should be a product type.
**Definition 16** ($|\cdot|_{\Delta}$). For w.f. ($\Delta \Rightarrow \tau'$),

\[
|\tau \Delta \varepsilon \tau'|_{\Delta} = |\tau|_{[]} \rightarrow |\tau'|_{[]}
\]

\[
|\tau \Delta \varsigma \tau'|_{\Delta} = |\tau|_{[]}, \phi_1, \ldots, \phi_n \rightarrow \phi,
\]

where $|\tau'|_{\Delta+\tau} = [\phi_1, \ldots, \phi_n] \rightarrow \phi$

\[
|\tau [\tau_1 \Rightarrow \tau_n] \varepsilon \tau'|_{[]} = ([|\tau_1|_{[]}, \ldots, |\tau_n|_{[]}], |\tau|_{[]}) \rightarrow |\tau'|_{[]}) \times |\tau_1|_{[]} \times \cdots \times |\tau_n|_{[]},
\]

where $|\tau'|_{[]} = ([|\tau_1|_{[]}, \ldots, |\tau_n|_{[]}], |\tau|_{[]}, \phi_1, \ldots, \phi_k) \rightarrow \phi'_{[]}) \times |\tau_1|_{[]} \times \cdots \times |\tau_n|_{[]} \times |\tau|_{[]}$

In the first two cases of the definition, the given parameter-type list matches the parameter-type list in the annotated type. The type in each case represents a function (as opposed to a partial application) regardless of what $\Delta$ is. If $\Delta$ is empty, we know that the type represents a function because a type with an empty parameter-type list over the function arrow cannot represent a partial application. If $\Delta$ is non-empty, we know that the type represents a function because, in the unCurry specification, the only rules allowing the given parameter-type list to be non-empty are those dealing with an abstraction.

If the unCurry annotation is $\varepsilon$, then $\tau$ is the type of the function’s final argument. So the target language type of a function with annotated type $\tau \Delta \varepsilon \tau'$ is $|\tau|_{[]} \rightarrow |\tau'|_{[]}$.

If the unCurry annotation is $\varsigma$, then the function may have several additional parameters. Assuming the types are well-formed (see Definition 17), the parameter-type list over the arrow in $\tau'$ is $\Delta + \tau$. So $|\tau'|_{\Delta+\tau}$ is a function type. By adding $|\tau|_{[]}$ to the parameter
list of this function type, we have the target language type of a function with annotated
type $\tau \xrightarrow{\Delta} \tau'$. The second pair of cases in the definition, where the parameter-type lists do not
match, relate annotated types to product types. In each case the annotated type must
represent a partial application because the given parameter-type list is empty and the
parameter-type list over the arrow is not. The target language product type consists of
the type of the function that is partially applied and the types of the previously applied
arguments (the target language forms of $\tau_1, \ldots, \tau_n$).

The type translations $\| \cdot \|$ and $\langle \cdot \rangle |_{\Delta}$ are extended for the unCurry type context
$\Gamma_c$. If $\Gamma_c(x) = \{ x_1^{\tau_1}, \ldots, x_n^{\tau_n} \}$ and w.f.(\Gamma_c), then

- $\| \Gamma_c \| (x) = \| \tau_1 \|
- $\langle \Gamma_c \rangle [i](x_i) = \langle \tau_i \rangle [i]$, for $i = [1..n]$

The final two definitions concern the notion of well-formed types. Basically an
annotated type is well-formed if its parameter-type lists conform to the unCurry speci-
fication. For example, if we have an annotated type $\tau \xrightarrow{[\tau_1, \tau_2]} \tau' \xrightarrow{\Delta} \tau''$, then $\Delta$ must be
$[\tau_1, \tau_2, \tau]$.

**DEFINITION 17 (WELL-FORMED $\tau$).**

Annotated type $\tau$ is well-formed (written w.f.(\tau)) if:

- $\tau = \tau' \xrightarrow{\Delta} \tau''$ where w.f.(\tau') $\&$ w.f.(\tau'')$, or
- $\tau = \tau_1 \xrightarrow{\Delta} \tau_2$, where w.f.(\tau_1), w.f.(\tau_2) and $\tau_2$ has the form $\tau_3 \xrightarrow{\Delta} \gamma \tau_4$. 
In a well-formed type context, all annotated types must be well-formed. Furthermore, the types of specialized functions are constrained. In particular, if $\Gamma_c(f) = \{g^\tau_1, \ldots, g^\tau_n\}$, then the types $\tau_1, \ldots, \tau_n$ must all erase (via $\| \cdot \|$) to the same source language type.

**Definition 18 (Well-formed $\Gamma_c$).**

A type context $\Gamma_c$ is well-formed (written $w.f.(\Gamma_c)$) if $\forall x \in \text{dom}(\Gamma_c)$, if $\Gamma_c(x) = \{x^{\tau_1}, \ldots, x^{\tau_n}\}$ then $\exists \tau$ such that for all $i \in [1..n]$, $\|\tau_i\| = \tau$ and $w.f.(\tau_i)$.

The theorem of type soundness states that if $e$ unCurries to $m$ with annotated type $\tau$, then the source language type of $e$ must be $\|\tau\|$, and the target language type of $m$ must be $|\langle \tau \rangle|_{\Delta}$, where $\Delta$ is the given parameter-type list.

**Theorem 5.1 (Type Soundness).**

If $\Delta; \Gamma_c \triangleright e : \tau \Rightarrow m$ for $w.f.(\Gamma_c)$, then

1. $\|\Gamma_c\| \vdash e : \|\tau\|$  

2. $|\langle \Gamma_c \rangle| [] \vdash m : |\langle \tau \rangle|_{\Delta}$

The proof of both parts is given in Appendix C.

**5.4.2 Operational Correctness**

Type soundness relates the unCurry transformation to the static semantics of the source and target languages. In this section, I demonstrate the operational correctness of

---

6 This would be the case if the unCurry transformation created $n$ specialized versions of the source function $f$ in the target expression named $g_1, \ldots, g_n$. 
the transformation by relating it to the dynamic semantics of the source and target languages (Figures 2.2 and 5.2, respectively). The statement of the operational correctness theorem requires the following two definitions.

First, recall that the dynamic semantics relates expressions to values. In the source language, values are function closures. In the target language, values include both function closures and partial application tuples.

Because the dynamic semantics of the languages use explicit environments (which I'll call \( \rho \) and \( \rho' \)), and the unCurry transformation uses an explicit type context \( (\Gamma_c) \), we require a means of relating the three to guarantee their consistency. The definition below defines the relation \( \rho : \Gamma_c \Rightarrow \rho' \).

**Definition 19.** For environments \( \rho \) and \( \rho' \) and context \( \Gamma_c \) where \( \text{dom}(\Gamma_c) = \text{dom}(\rho) \),

\[
\rho : \Gamma_c \Rightarrow \rho'
\]

iff \( \forall x \in \text{dom}(\Gamma_c), \text{if } \Gamma_c(x) = \{(x^i_i)\} \text{ then} \)

- \( \rho(x) = v \)
- \( \rho'(x^i_i) = v_i \) and
- \( \emptyset \triangleright v : \tau_i \Rightarrow v_i \).

If, given the type context \( \Gamma_c \), \( m \) is the unCurried form of \( e \), then \( e \) may be evaluated using \( \rho \) and \( m \) may be evaluated using \( \rho' \). The definition requires that the domains of \( \rho \) and \( \Gamma_c \) be the same. However it may not be the case that \( \text{dom}(\rho) = \text{dom}(\rho') \) since \( m \) may contain several different specialized versions of functions defined in \( e \). However, if
\[ \Gamma_c(f) = \{ g^1, \ldots, g^n \}, \text{ then the variables } g_1, \ldots, g_n \text{ must be in the domain of } \rho'. \] The relation \( \models \vdash v : \tau_i \Rightarrow v_i \) (which is defined below) states that the target language values \( v_i \) are unCurried forms of the source language value \( v \).

The second definition defines the relation \( \Delta \vdash v : \tau \Rightarrow v' \), where \( v \) is a source language value and \( v' \) is a target language value. The definition requires both the values’ annotated type and the given parameter-type list in order to determine whether the target language value is a function closure or a partial application tuple.\(^7\)

**Definition 20.** For values \( v \) and \( v' \), ordered set \( \Delta \) and annotated type \( \tau \), the relation \( \Delta \vdash v : \tau \Rightarrow v' \) is defined as follows:

- \( \Delta \vdash [\rho, \lambda x.e] : \tau \overset{\Delta}{\rightarrow} \gamma \Rightarrow [\rho', m] \) iff for some well-formed \( \Gamma_c \) such that \( \rho : \Gamma_c \Rightarrow \rho' \), the judgment \( \Delta ; \Gamma_c \vdash \lambda x.e : \tau \overset{\Delta}{\rightarrow} \gamma \Rightarrow m \) is derivable.

- \( \models \vdash [\rho, \lambda x.e] : \tau \overset{\tau_1, \ldots, \tau_n}{\rightarrow} \gamma \Rightarrow \langle [\rho', \lambda[y_1, \ldots, y_n, x_1, \ldots, x_k].m], v_1, \ldots, v_n \rangle \) iff for some well-formed \( \Gamma_c \) such that \( \rho : \Gamma_c \Rightarrow \rho' \{ y_1 \mapsto v_1 \} \ldots \{ y_n \mapsto v_n \} \), the judgment \( [\tau_1, \ldots, \tau_n] ; \Gamma_c \vdash \lambda x.e : \tau \overset{\tau_1, \ldots, \tau_n}{\rightarrow} \gamma \Rightarrow \lambda[x_1, \ldots, x_k].m \) is derivable.

If the given parameter-type list matches the parameter-type list in the annotated type, then the target language value is a function closure. Otherwise, the given parameter-type list must be empty and the target language value is a partial application tuple. In the first case, the expression in the target closure \( (m) \) must be an unCurried form of the expression in the source closure \( (\lambda x.e) \). In other words, the judgment \( \Delta ; \Gamma_c \vdash \lambda x.e : \tau \overset{\Delta}{\rightarrow} \gamma \Rightarrow m \) must be derivable. It does not matter whether the unCurry annotation \( \gamma \) is \( \varsigma \) or \( \varepsilon \).

\(^7\)This reasoning is similar to that for Definition 16.
Furthermore, the environments in the closures must be related (together with the context from the judgment) via Definition 19.

In the second case of the definition, the $n$ types in the parameter-type list over the function arrow tell us that there must be $n + 1$ elements in the partial application tuple: a function closure followed by the values of the first $n$ arguments the function will be applied to. If $\lambda[x_1, \ldots, x_k]_c m$ is an unCurried form of $\lambda x.e$, then the function in the partial application tuple must have the form $\lambda[y_1, \ldots y_n, x_1, \ldots, x_k]_c m$. We know that there are $n$ parameters to the left of $x_1$ because there are $n$ types in the parameter-type list. Note that it is not the case that $\rho : \Gamma \Rightarrow \rho'$ since $\rho$ includes mappings for any parameters that appear to the left of $x$ in the original expression, whereas $\rho'$ does not (as the function in the partial application tuple has not been applied to its arguments yet). Thus, $\rho : \Gamma \Rightarrow \rho'\{\overrightarrows_i \overleftarrow{\overrightarrow{y_i_i}}\}$. Note that not all of the $y_i$s will appear in the domain of $\rho$ since some of them may have been introduced by the transformation as useless parameters.

The first part of the operational correctness theorem, given below, states that, if $m$ is an unCurried form of $e$ and $e$ evaluates to some value $v$, then $m$ evaluates to a value $v'$. Furthermore, $v'$ is an unCurried form of $v$ (related via Definition 20). It is not the case that $v = v'$ since values in both the source and target language contain abstractions, and abstractions in the target value are unCurried forms of the abstractions in the source value. The second part of the theorem states that the transformation is safe in that it preserves nontermination. The target expression only evaluates to a value if the source expression also evaluates to a value.
Theorem 5.2 (Operational Correctness).

If $\Delta; \Gamma \vdash e : \tau \Rightarrow m$, w.f.$(\Gamma_c)$ and $\rho : \Gamma_c \Rightarrow \rho'$,

1. if $\rho \vdash e \leftrightarrow v$ then there exists a $v'$ such that $\rho' \vdash m \leftrightarrow v'$ and $\Delta \vdash v : \tau \Rightarrow v'$.

2. if $\rho' \vdash m \leftrightarrow v'$ then there exists a $v$ such that $\rho \vdash e \leftrightarrow v$ and $\Delta \vdash v : \tau \Rightarrow v'$.

The proof of both parts can be found in Appendix D.
Chapter 6

Concluding Remarks

In this thesis, I have presented formal definitions of three type-based functional program transformations using sets of inference rules. The specifications for useless-variable elimination, lambda-lifting and unCurrying provide general, declarative descriptions of the transformations that provide us frameworks for studying them in a formal setting. We can prove properties of these transformations, as well as algorithms for the transformations, using standard induction techniques.

The specification of unCurrying that I have presented in this thesis is an extension of the higher-order unCurrying of Hannan and Hicks that transforms partial applications into tuples and allows for the specialization of functions. This extended unCurrying reduces the number of partial applications in a program and allows for the unCurrying of more functions. I have proved that the specification preserves the type of programs. If the source program is well-typed, then the unCurried form of that program is also well-typed. This property allows a compiler to safely apply other type-based transformations to the unCurried program. I have also shown that the specification preserves the meaning of programs. In other words, the value resulting from the evaluation of an unCurried program is an unCurried form of the value resulting from the evaluation of the original program.
I have also described joint work (with John Hannan) on the formal specifications of lambda-lifting and useless-variable elimination. In the former case, we have provided a more general description of lambda-lifting than was previously available. While earlier descriptions of lambda-lifting were algorithmic in nature, describing specific lambda-lifting transformations, we have presented a broader description that allows for varying degrees of lambda-lifting and takes advantage of the higher-order nature of functional programs to avoid the introduction of partial applications. In the latter case, we presented a type-based description of useless-variable elimination for higher-order programs that eliminates useless code without altering the behavior of the program.

In addition, I have described algorithms for both lambda-lifting and useless-variable elimination, and shown how the formal specifications of these transformations facilitate the proofs of correctness for the algorithms. For lambda-lifting, I described the parameter-lifting phase of Johnsson’s lambda-lifting algorithm using inference rules and then derived an actual parameter-lifting algorithm from those rules. I then showed that this parameter-lifting algorithm is sound with respect to the formal specification. Since the specification is both type safe and operationally correct, this soundness result guarantees that the algorithm is also type safe and operationally correct. For useless-variable elimination, I derived an original algorithm from the inference rules and, by showing the algorithm to be sound with respect to the formal specification, I proved that the useless-variable elimination algorithm is type safe and operationally correct.

I now conclude with a discussion of extensions to and possible applications of the work described in this thesis.
6.1 Extensions to Real Languages

This work is theoretical. The transformations described in the previous chapters work on a core functional language based on the lambda-calculus. But how well do they work on real languages, such as Standard ML? The transformations and their corresponding proofs of correctness can be extended to include additional language features such as lists and other data structures. In our published work, we discuss introducing polymorphic typing to both useless-variable elimination [9] and lambda-lifting [11]. The same can certainly be done for unCurrying.

Of particular interest (and difficulty) is the extension of our transformations to include mutable variables. This necessitates the introduction of a program state and can dramatically effect the analyses. For example, is lambda-lifting possible in the presence of mutable variables? If we lift a mutable variable (by introducing a local binding in a function’s parameter list), we must be certain that the variable will have the appropriate value when the function is applied to it. A way around this is to avoid lifting mutable variables. But, in that case, it may not be possible to flatten the program.

6.2 Algorithms for Lambda-Lifting and unCurrying

Although we have described a general lambda-lifting transformation that allows for many different lambda-lifting strategies, the only algorithm I describe is one that has been around since the mid-1980s. An obvious avenue of future work then is the development of a new algorithm that takes advantage of the “higher-order” lambda-lifting
described in Chapter 4. A more advanced lambda-lifting algorithm may provide benefits to partial evaluation since many partial evaluators require programs to be lambda-lifted.

Similarly, an algorithm should be developed for the extended unCurrying described in Chapter 5. Such an algorithm would enable us to study the higher-order to first-order transformation discussed in the following section.

6.3 Higher-Order to First-Order Transformation

A potential application of both lambda-lifting and unCurrying is the creation of a new method of transforming a program with higher-order functions into a program containing only first-order functions. Functional language compilers must eliminate higher-order functions from a program before generating machine code. For example, a functional language compiler might generate closures for each function in a program. These closures include the function’s code as well as its environment (i.e. values for the function’s free variables). The process of introducing closures into a program is called closure conversion ([26, 29, 12]). There are no higher-order functions in a closure-converted program because functions are passed function closures as arguments rather than functions.

Other methods, similar to closure conversion, have been proposed including defunctionalization [25], firstification [21], and higher-order removal [3]. All three of these methods first lambda-lift the program. How they proceed from there differs. Defunctionalization replaces each function that is passed as an argument with a data constructor (similar to a closure). These functions are called by passing these data constructors to
a specialized apply function. Firstification eliminates higher-order functions by introducing specialized functions. Functions are specialized by β-reducing any function that has a function as an argument. This method only works for a call-by-name language. Higher-order removal is basically a combination of firstification and closure-conversion.

However, rather than applying any of these methods after lambda-lifting, we can unCurry the program. Consider the following program with a higher-order function:

\[
\begin{align*}
&\text{let } x = e_1 \\
&f = \lambda y . y + x \\
g = \lambda h . \lambda z . h @ (h @ z) \\
in g @ f @ e_2
\end{align*}
\]

By lambda-lifting this program using Johnsson’s method, we have:

\[
\begin{align*}
&\text{let } x = e_1 \\
&f = \lambda x . \lambda y . y + x \\
g = \lambda h . \lambda z . h @ (h @ z) \\
in g @ (f @ x) @ e_2
\end{align*}
\]
Now we can unCurry the program to give us:

\[
\begin{align*}
\text{let } x &= e_1 \\
\lambda &= \lambda[x, y]. (y + x) \\
g &= \lambda[h, z]. \#2(h) \odot [\#1(h), \#2(h) \odot [\#1(h), z]] \\
in g @ [(\cdot, f), e_2]
\end{align*}
\]

This is a program without any higher-order functions. Rather than passing the function \( f \) as an argument to \( g \), we instead pass the partial application tuple \((\cdot, f), x\). We can further improve this program by arity raising [13], which splits the parameter \( h \) into two parameters, \( h_1 \) and \( h_2 \), allowing the program to pass the elements of the partial application tuple as separate arguments to \( g \):

\[
\begin{align*}
\text{let } x &= e_1 \\
\lambda &= \lambda[x, y]. (y + x) \\
g &= \lambda[h_1, h_2, z]. h_1 \odot [h_2, h_1 \odot [h_2, z]] \\
in g @ [f, x, e_2]
\end{align*}
\]

It turns out that we could have skipped the arity raising step by employing a different lambda-lifting method. Notice that the following program is also a lambda-lifted form
of the original program:

\[
\text{let } x = e_1 \\
\quad f = \lambda x.\lambda y.y + x \\
\quad g = \lambda x.\lambda h.\lambda z.h \, @ \, x \, @ \, (h \, @ \, x \, @ \, z) \\
\quad \text{in } g \, @ \, x \, @ \, f \, @ \, e_2
\]

UnCurrying this program is straightforward and does not require introducing any partial application tuples:

\[
\text{let } x = e_1 \\
\quad f = \lambda [x,y].y + x \\
\quad g = \lambda [x,h,z].h \, @ \, [x,h \, @ \, [x,z]] \\
\quad \text{in } g \, @ \, [x,f,e_2]
\]

Since we know that both lambda-lifting and unCurrying are type safe and operationally correct transformations for a call-by-value language, we know that this method of eliminating higher-order functions is both type safe and operationally correct. In contrast, closure conversion is typically not type safe.\(^1\)

Further study is needed in order to determine the exact benefits of this method. By employing lambda-lifting, unCurrying and arity raising instead of conventional closure conversion, we can reduce the number of closure-like data structures allocated on the heap in favor of additional function parameters. Of course, introducing too many parameters leads to register spilling and may negate any savings in terms of execution

\(^1\)Although Minamide, Morrisett and Harper [20] give a type-preserving closure conversion method using existential types.
time over conventional closure conversion. Perhaps a hybrid approach combining the
method described above with an efficient closure conversion strategy may serve as an
improvement over existing techniques.
Appendix A

Useless-Variable Elimination Proof of Correctness

The following lemmas are required by the proof of Theorem 3.1. The first three lemmas deal with the constraint solver defined in Figure 3.10. The first lemma states that the constraint solver terminates when given a finite set of constraints that does not contain any instances of the $u$ annotation.\(^1\) The second lemma states that the substitution computed by the constraint solver is actually a solution to the given constraints. Finally, the third lemma shows that any solution to a set of constraints is also a solution to any subset of the set of constraints. These last two lemmas are necessary to prove the correctness theorem stated at the end of this section.

**Lemma A.1.** Given a finite constraint set $\Phi$ not containing instances of $u$, $\text{solve}(\Phi)$ always terminates.

*Proof.*

Each rule 1-10 in Figure 3.10 either reduces or eliminates a constraint without introducing any additional constraints. Since we assume $\Phi$ is finite, all constraints are either eliminated or reduced to a point where none of the rules 1-10 apply (i.e. all sets $\Delta$ are either empty or contain only annotation variables), in which case rule 11 returns the substitution $\delta_u$ mapping all remaining annotation variables to $u$.

$\square$

\(^1\)Recall that algorithm $\mathcal{U}$ does not introduce any instances of $u$. 

**Lemma A.2.** If \( \text{solve}(\Phi) = \delta \) for any constraint set \( \Phi \) not containing instances of \( u \), then constraint \( c \) is satisfied for all \( c \in \delta \Phi \).

**Proof.**

Assume there exists a constraint \( c' \in \delta \Phi \) such that \( c' \) is not satisfied. I will show that there can be no such \( c' \).

1. \( c' \) cannot be \( \langle \delta \gamma = n \rangle^{\delta \Delta} \):

   If \( \delta \Delta \) contains \( u \), the constraint is irrelevant. Otherwise, Rule 4 guarantees that \( \text{solve} \) reduces the constraint so that Rule 5 or 6 is applicable. Rule 11 does not map variables to \( u \) unless Rules 1-10 are no longer applicable. So, \( \delta \gamma \) cannot be \( u \) since Rule 5 introduces the mapping \( \gamma \mapsto n \) to the solution. Since \( \delta \gamma \) must be \( n \), \( c' \) is satisfied.

2. \( c' \) cannot be \( \langle x \in \{x^{\delta \Delta_1} \cup \delta \Theta \supset \delta \gamma = n \rangle^{\delta \Delta_2} \):

   If either \( \delta \Delta_1 \) or \( \delta \Delta_2 \) contain \( u \), the constraint is irrelevant. Otherwise, Rules 1 and 2 guarantee that \( \text{solve} \) reduces the constraint so that Rule 3 is applicable. Rule 3 guarantees that \( \text{solve} \) reduces the constraint to \( \langle \gamma = n \rangle^\emptyset \). Thus, by 1, \( c' \) is satisfied.

3. \( c' \) cannot be \( \langle \delta \gamma_1 \leq \delta \gamma_2 \rangle^{\delta \Delta} \):

   As above, Rule 7 guarantees that \( \text{solve} \) reduces \( c' \), if the constraint is relevant, so that Rules 8-10 may be applicable. The only way in which \( c' \) is not satisfied is for \( \delta \gamma_1 \) to be \( n \) and \( \delta \gamma_2 \) to be \( u \). However, this cannot be the case since \( \text{solve} \) does not map variables to \( u \) unless Rules 1-10 are no longer applicable, and Rule 8 introduces the mapping \( \gamma_2 \mapsto n \) to the solution. Thus, \( c' \) is satisfied.

\( \square \)
**Lemma A.3.** For any constraint set $\Phi$, if $\delta$ is a solution to $\Phi$, then $\delta$ is a solution to any subset of $\Phi$.

**Proof.**

Let $\Phi'$ be some subset of $\Phi$. A constraint $c \in \Phi'$ only if $c \in \Phi$. By Lemma A.2, $c$ is satisfied for all $c \in \delta \Phi$, and so $c$ is satisfied for all $c \in \delta \Phi'$. Thus $\delta$ is a solution to $\Phi'$.

\[ \square \]

The following lemma shows that getord, used in the application case of algorithm $\mathcal{U}$, generates the proper constraints in order to enforce the relation on types defined in Section 3.1.2.

**Lemma A.4.** If $\text{getord}(\tau_1, \tau_2) = \Phi$, $\Phi \subseteq \Phi'$, and $\text{solve}(\Phi') = \delta$ for any types $\tau_1$ and $\tau_2$, and constraint set $\Phi'$, then $\delta \tau_1 \leq \delta \tau_2$.

**Proof.**

The proof is by induction on the structure of types. Since $\text{solve}(\Phi') = \delta$ and $\Phi \subseteq \Phi'$, $\delta$ is a solution to $\Phi$ and any subset of $\Phi$ by Lemmas A.2 and A.3.

1. $\tau_1 = \tau_2 = \text{int}$

   $\text{getord}(\text{int, int}) = \emptyset$ and we can construct:

   \[
   \delta\text{int} \leq \delta\text{int}
   \]

2. $\tau_1 = \tau' \rightarrow \tau''_1$
   $\tau_2 = \tau' \rightarrow \tau''_2$

   $\tau_1 = \tau'_1 \rightarrow \tau''_1$
   $\tau_2 = \tau'_2 \rightarrow \tau''_2$
\[ \text{getord}(\tau_1' \rightarrow \gamma_1, \tau_2' \rightarrow \gamma_2) = \{(\gamma_1 \leq \gamma_2)^0\} \cup \text{getord}(\tau_2' \rightarrow \gamma_2) \cup \text{getord}(\tau_1' \rightarrow \gamma_1) \]

By the induction hypothesis:

\[ \Xi_1 :: \delta^{\tau_1'}_2 \leq \delta^{\tau_1'}_1 \]

\[ \Xi_2 :: \delta^{\tau_2''}_1 \leq \delta^{\tau_2''}_2 \]

Since \( \delta \) is a solution to \( \Phi \), \( \delta \gamma_1 \leq_a \delta \gamma_2 \) and we can construct:

\[
\begin{align*}
\Xi_1 &:: \delta^{\tau_1'}_2 \leq \delta^{\tau_1'}_1 \\
\Xi_2 &:: \delta^{\tau_2''}_1 \leq \delta^{\tau_2''}_2 \\
\hline
\delta(\tau_1' \rightarrow \gamma_1, \tau_2'' \rightarrow \gamma_2) &:: \delta(\tau_1' \rightarrow \gamma_1) \leq \delta(\tau_2'' \rightarrow \gamma_2)
\end{align*}
\]

\[ \square \]

**Lemma A.5 (Weakening).** If \( \Gamma \vdash e : \tau \Rightarrow e' \) then \( \Gamma', \Gamma' \vdash e : \tau \Rightarrow e' \), where \( \text{dom}(\Gamma) \cap \text{dom}(\Gamma') = \emptyset \).

The proof is straightforward by induction on the derivation.

**Proof of Theorem 3.1**

Both parts are proved simultaneously by induction on the structure of the input term \( e^\tau \).

1. \( e^\tau = x^\tau \)

   - \( \mathcal{U}(x^\tau) = (\{x^0\}, \{\}, x) \)
\begin{itemize}
  \item $\text{solve}(\Phi') = \delta$, $\Phi = \{\} \text{ and } \Phi \subseteq \Phi'$. Thus, by Lemmas A.2 and A.3, $\delta$ is a solution to $\Phi$.
  \item $\text{translate}(\delta x) = x$
\end{itemize}

(a) $FV(x) = \text{simplify}(\delta\{x^0\}) = \{x\}$

(b) $FV(x^\tau) \subseteq \text{dom}(\Gamma)$, so $x \in \text{dom}(\Gamma)$ and $\Gamma(x) = \delta \tau$. We can construct:

\[
\begin{align*}
\Gamma(x) &= \delta \tau \\
\Gamma \triangleright x^\tau &:\delta \tau \Rightarrow x
\end{align*}
\]

2. $e^\tau = n^{\text{int}}$

\begin{itemize}
  \item $\mathcal{U}(n^{\text{int}}) = (\{\}, \{\}, n)$
  \item $\text{solve}(\Phi') = \delta$, $\Phi = \{\} \text{ and } \Phi \subseteq \Phi'$. Thus, by Lemmas A.2 and A.3, $\delta$ is a solution to $\Phi$.
  \item $\text{translate}(\delta n) = n$
\end{itemize}

(a) $FV(n) = \text{simplify}(\delta\{\}) = \{\}$

(b) We can construct:

\[
\begin{align*}
\Gamma \triangleright |n^{\text{int}}| &:\delta \text{int} \Rightarrow n
\end{align*}
\]
3. \( e^T = (\lambda x. e^T) \uparrow^T \gamma \)

\[
U((\lambda x^T. e^T) \uparrow^T) = \\
\text{let } (\Theta_1, \Phi_1, e_1) = U(e^T) \\
in (\Theta_1 \setminus x, \{x \in \Theta_1 \Rightarrow \gamma = n\} \cup \Phi_1, \lambda x. e_1)
\]

- solve(\( \Phi' \) = \( \delta \), \( \Phi = \{x \in \Theta_1 \Rightarrow \gamma = n\} \) \( \cup \Phi_1 \) and \( \Phi \subseteq \Phi' \). So, by Lemmas A.2 and A.3, \( \delta \) is a solution to \( \Phi \) and \( \Phi_1 \).

- translate(\( \delta (\lambda x.e_1) \) = \( \lambda x. \text{translate}(\delta e_1) \) and we assume \( e_1' = \text{translate}(\delta e_1) \).

By the induction hypothesis, we know:

- \( \text{FV}(e_1') = \text{simplify}(\delta \Theta_1) \)

We have two cases:

(a) If \( x^T \in \text{FV}(e^T) \) we have \( \Xi : \Gamma \{x : \delta \tau_1\} \vdash e^T : \delta \tau_2 \Rightarrow e_1' \) where

\[
\text{FV}(e^T) \subseteq \text{dom}(\Gamma\{x : \delta \tau_1\})
\]

(b) If \( x^T \notin \text{FV}(e^T) \) we have \( \Xi' : \Gamma \vdash e^T : \delta \tau_2 \Rightarrow e_1' \) where \( \text{FV}(e^T) \subseteq \text{dom}(\Gamma) \). By Lemma A.5, we have \( \Xi : \Gamma\{x : \delta \tau_1\} \vdash e^T : \delta \tau_2 \Rightarrow e_1' \). Since

\[
\text{FV}(e^T) \subseteq \text{dom}(\Gamma), \text{FV}(e^T) \subseteq \text{dom}(\Gamma\{x : \delta \tau_1\})
\]

(a) \( \text{FV}(\lambda x.e_1') = \text{FV}(e_1') \setminus x = \text{simplify}(\delta \Theta_1) \setminus x = \text{simplify}(\delta \Theta_1 \setminus x) \)
(b) We have two cases, based on the value of $\delta(\gamma)$:

i. $\delta(\gamma) = n$, so $\delta(\tau_1 \rightarrow \gamma \tau_2) = \delta\tau_1 \rightarrow_n \delta\tau_2$

We can construct:

$$
\Gamma \vdash (\lambda x.\delta\tau_1 \rightarrow \gamma \tau_2) : \delta(\tau_1 \rightarrow \gamma \tau_2) \Rightarrow \lambda x.\delta\tau_2
$$

ii. $\delta(\gamma) = u$, so $\delta(\tau_1 \rightarrow \gamma \tau_2) = \delta\tau_1 \rightarrow_u \delta\tau_2$

Since $\delta$ is a solution to $\Phi$ and $\langle x \in \Theta_1 \vdash \gamma = n \rangle \in \Phi$, if $\delta(\gamma) = u$ then $x \notin \text{simplify}(\delta\Theta_1)$. Thus $x \notin FV(\delta\Theta_1)$ and we can construct:

$$
\Gamma \vdash (\lambda x.\delta\tau_1 \rightarrow \gamma \tau_2) : \delta(\tau_1 \rightarrow \gamma \tau_2) \Rightarrow \lambda x.\delta\tau_2
$$

4. $e^\gamma = (e_1^{\tau_1 \rightarrow \gamma \tau_2} \oplus e_2^{\tau_1'})^{\tau_2}$

\[\mathcal{U}((e_1^{\tau_1 \rightarrow \gamma \tau_2} \oplus e_2^{\tau_1'})^{\tau_2}) = \]

let $(\Theta_1, \Phi_1, e') = \mathcal{U}(e_1^{\tau_1 \rightarrow \gamma \tau_2})$

$(\Theta_2, \Phi_2, e') = \mathcal{U}(e_2^{\tau_1'})$

$\Phi_3 = \Phi_2 \cup \text{getord}(\tau_1', \tau_1)$

$\Phi_4 = \text{if noeffect}(e_2) \text{ then } \{\} \text{ else } \{\langle \gamma = n \rangle^0\}$

in $(\Theta_1 \cup \gamma \Theta_2', \Phi_1 \cup \gamma \Phi_2 \cup \Phi_3', e_1^{\tau_1} \oplus e_2^{\tau_1'})$
• \( \text{solve}(\Phi') = \delta, \Phi = \Phi_1 \cup \gamma \Phi_3 \cup \Phi_4 \) and \( \Phi \subset \Phi' \), so by Lemmas A.2 and A.3, \( \delta \) is a solution to \( \Phi, \Phi_1, \Phi_2, \gamma \Phi_3 \) and \( \Phi_4 \).

• \( \text{translate}(\delta(e'_{1 \gamma 2} @ e')) = \text{translate}(\delta(e'_{1 \gamma 2}) \oplus \delta(\gamma)) \text{translate}(\delta e'_{1\gamma 2}) \)

and \( \text{translate}(\delta e'_{1\gamma 2} = e''_{1\gamma 2}), \text{translate}(\delta e'_{1\gamma 2}) = e''_{1\gamma 2} \).

By the induction hypothesis:

• \( FV(e''_{1\gamma 2}) = \text{simplify}(\delta\Theta_1) \)

\( FV(e''_{1\gamma 2}) = \text{simplify}(\delta\Theta_2) \)

\( \Xi_1 = \Gamma \triangleright [e_{1\gamma 2}^\tau_1 \rightarrow \gamma \tau_2] : \delta(\tau_1 \rightarrow \gamma \tau_2) \Rightarrow \delta''_{1\gamma 2} \)

\( \Xi_2 = \Gamma \triangleright [e_{1\gamma 2}^\tau_1 : \delta\tau_1 \Rightarrow \delta''_{1\gamma 2} \]

We have two cases based on the value of \( \delta(\gamma) \):

(a) \( \delta(\gamma) = n, \) so \( \delta(\tau_1 \rightarrow \gamma \tau_2) = \delta\tau_1 \rightarrow \delta\tau_2 \)

i. \( FV(e''_{1\gamma 2} @ n e''_{1\gamma 2}) = FV(e''_{1\gamma 2}) \cup FV(e''_{1\gamma 2}) = \text{simplify}(\delta\Theta_1) \cup \text{simplify}(\delta\Theta_2) = \text{simplify}(\delta(\Theta_1 \cup \Theta_2)) = \text{simplify}(\delta(\Theta_1 \cup \Theta_2)) \)

ii. By Lemma A.4, since \( \text{getord}(\tau_1', \tau_1) = \Phi'_3, \Phi'_3 \subset \Phi_3 \) and \( \delta \) is a solution to \( \gamma \Phi_3 \), then \( \delta\tau_1' \leq \delta\tau_1 \). We can construct:

\[
\begin{align*}
\Xi_1 & \quad \Gamma \triangleright [e_{1\gamma 2}^\tau_1 \rightarrow \gamma \tau_2] : \delta\tau_1 \rightarrow \delta\tau_2 \Rightarrow e''_{1\gamma 2} \quad \Gamma \triangleright [e_{1\gamma 2}^\tau_1 : \delta\tau_1 \Rightarrow e''_{1\gamma 2} \quad \delta\tau_1' \leq \delta\tau_1 \\
\Xi_2 & \quad \Gamma \triangleright [(e_{1\gamma 2}^\tau_1 \rightarrow \gamma \tau_2 @ e_{1\gamma 2}^\tau_1] : \delta\tau_2 \Rightarrow e''_{1\gamma 2} @ e''_{1\gamma 2} \quad \text{from} \quad \delta\tau_1' \leq \delta\tau_1
\end{align*}
\]
(b) \(\delta(\gamma) = u\), so \(\delta(\tau_1 \rightarrow \gamma \tau_2) = \delta\tau_1 \rightarrow_1 \delta\tau_2\) and \(e''_1 = d^{\delta\tau_1}

i. \(FV(e''_1 @ u d^{\delta\tau_1}) = FV(e''_1) = \text{simplify}(\delta\Theta_1) = \text{simplify}(\delta(\Theta_1 \cup \gamma\Theta_2))\)

ii. Since \(\delta\) is a solution to \(\Phi\) and \(\delta(\gamma) = u\), \((\gamma = n)^\emptyset \notin \Phi\). Thus, noeffect\(\left(e_2\right)\)

and we can construct:

\[
\begin{align*}
\Xi_1 &\quad \text{\(\Gamma \triangleright\{e_1\rightarrow_1 e_2\} : \delta\tau_1 \rightarrow_u \delta\tau_2 \Rightarrow e''_1\) noeffect\(\left(e_2\right)\)} \\
\Xi &\quad \text{\(\Gamma \triangleright\{(e_1 \rightarrow_1 e_2) @ d^{\delta\tau_1}\} : \delta\tau_2 \Rightarrow e''_1 @ u d^{\delta\tau_1}\)}
\end{align*}
\]

5. \(e^\tau = (\mu f^\tau . e^\tau)^\tau_1\)

\[
\begin{align*}
\mathcal{U}(\mu f^\tau . e^\tau)^\tau_1 &\quad = \\
\text{let } (\Theta_1, \Phi, e_1) = \mathcal{U}(e^\tau_1) &\quad = \\
\text{in } (\Theta_1 \backslash f, \Phi, \mu f. e_1') &\quad \text{in } (\Theta_1 \backslash f, \Phi, \mu f. e_1')
\end{align*}
\]

- \(\text{solve}(\Phi') = \delta\), \(\Phi \subseteq \Phi'\), so by Lemmas A.2 and A.3, \(\delta\) is a solution to \(\Phi\).

- \(\text{translate}(\delta(\mu f. e_1')) = \mu f. \text{translate}(\delta e_1')\) and \(\text{translate}(\delta e_1') = e''_1\).

By the induction hypothesis:

- \(FV(e''_1) = \text{simplify}(\delta \Theta_1)\)

- \(\Xi \quad \text{\(\Gamma\{f : \delta\tau\} \triangleright\{e^\tau_1 : \delta\tau \Rightarrow e''_1\}\)}\)
(a) \( \text{FV}(\mu f.e''_1) = \text{FV}(e''_1) \setminus f = \text{simplify}(\delta \Theta_1) \setminus f = \text{simplify}(\delta(\Theta_1 \setminus f)) \)

(b) We can construct:

\[
\frac{\Xi \Gamma \{ f : \delta \tau \} \triangleright |e^\tau| : \delta \tau \Rightarrow e''}{\Gamma \triangleright (\mu f^\tau.e^\tau)_1 | : \delta \tau \Rightarrow \mu f.e''_1}
\]

6. \( e^\tau = (e^\text{int}_1 + e^\text{int}_2)^\text{int} \)

\[
\mathcal{U}((e^\text{int}_1 + e^\text{int}_2)^\text{int}) =
\]

let \((\Theta_1, \Phi_1, e'_1) = \mathcal{U}(e^\text{int}_1)\)

\((\Theta_2, \Phi_2, e'_2) = \mathcal{U}(e^\text{int}_2)\)

in \((\Theta_1 \cup \Theta_2, \Phi_1 \cup \Phi_2, e'_1 + e'_2)\)

• \( \text{solve}(\Phi'_1) = \delta, \Phi = \Phi_1 \cup \Phi_2 \) and \( \Phi \subseteq \Phi'_1 \), so by Lemmas A.2 and A.3, \( \delta \) is a solution to \( \Phi, \Phi_1 \) and \( \Phi_2 \).

• \( \text{translate}(\delta(e'_1 + e'_2)) = \text{translate}(\delta e'_1) + \text{translate}(\delta e'_2) \) and \( \text{translate}(\delta e'_1) = e''_1 \)

\( \text{translate}(\delta e'_2) = e''_2 \). By the induction hypothesis:

- \( \text{FV}(e''_1) = \text{simplify}(\delta \Theta_1) \)

- \( \text{FV}(e''_2) = \text{simplify}(\delta \Theta_2) \)
\[ \Xi_1 : \Gamma \triangleright e^{\text{int}}_1 : \delta \text{int} \Rightarrow e''_1 \]
\[ \Xi_2 : \Gamma \triangleright e^{\text{int}}_2 : \delta \text{int} \Rightarrow e''_2 \]

(a) \[ FV(e''_1 + e''_2) = FV(e''_1) \cup FV(e''_2) = \text{simplify}(\delta \Theta_1) \cup \text{simplify}(\delta \Theta_2) = \text{simplify}(\delta (\Theta_1 \cup \Theta_2)). \]

(b) We can construct:

\[
\begin{array}{c}
\Xi_1 \quad \Xi_2 \\
\Gamma \triangleright e^{\text{int}}_1 : \delta \text{int} \Rightarrow e''_1 \quad \Gamma \triangleright e^{\text{int}}_2 : \delta \text{int} \Rightarrow e''_2 \\
\hline
\Gamma \triangleright [e^{\text{int}}_1 + e^{\text{int}}_2] : \delta \text{int} \Rightarrow e''_1 + e''_2
\end{array}
\]

\[ \square \]
Appendix B

Lambda Lifting Algorithm Proof of Correctness

Proof of Lemma 4.1

1. The first rule of solve

\[ \text{solve}(\{\theta_{g_i} = \theta \cup \theta \} \cup \{\theta_{g_j} = \theta \} \cup \Phi) = \text{solve}(\{\theta_{g_i} = \theta \setminus \theta_{g_i} \cup \theta \} \cup \{\theta_{g_j} = \theta \} \cup \Phi) \]

replaces one occurrence of \( \theta_{g_j} \) on the right hand side of a constraint with \( \theta_j \). Since \( \theta_{g_j} \) cannot occur in \( \theta_j \), one occurrence of \( \theta_{g_j} \) has been removed from \( \Phi \). Since there are a finite number, say \( N \), of \( \theta_{g_j} \), they can all be eliminated from the RHS of constraints after \( N \) applications of rule 1. Likewise for the rest of the \( \theta_{g_i} \). When all \( \theta_{g_i} \) have been removed from the RHS of all constraints, the resulting constraint list is returned as a substitution. Hence \( \text{solve}(\Phi) \) halts.

2. By the second rule of solve, \( \varepsilon \) must be a set of equalities \( \{\theta_{g_i} = \theta'_{i} \} \) where no \( \theta_{g_i} \) occur in any \( \theta'_{i} \). I show that \( \varepsilon \) is a solution to \( \Phi \).

I do this by demonstrating the correctness of rule 1, specifically, if \( \varepsilon \) is a solution to

\[ \{\theta_{g_i} = \theta \setminus \theta_{g_i} \cup \theta \} \cup \{\theta_{g_j} = \theta \} \cup \Phi \]
then it is a solution to

\[
\{\theta \cap g_i = \theta \cup \theta \} \cup \{\theta \cap g_j = \theta \cup \theta \} \cup \Phi.
\]

All of the constraints are the same in the two sets except for \(\theta \cap g_i\). Since we know

\[
\varepsilon_{\theta \cap g_j} = \varepsilon_{\theta \cap j},
\]

\[
\varepsilon_{\theta \cap g_i} = \varepsilon_{(\theta \cap g_j) \cup \theta}.
\]

We can add \(\varepsilon_{\theta \cap g_i}\) to the RHS allowing us to remove the restriction on \(\theta \cap g_j\), thus:

\[
\varepsilon_{\theta \cap g_i} = \varepsilon_{(\theta \cap g_j) \cup \theta}.
\]

Now, if \(\text{solve}(\Phi) \Rightarrow \text{solve}(\Phi') \Rightarrow \cdots \Rightarrow \varepsilon\), we know \(\varepsilon\) is a solution to \(\Phi\), hence

\[
\varepsilon_{\theta \cap g_i} = \varepsilon_{(\theta \cap g_j) \cup \theta} \text{ for all } \theta \cap g_i \text{ in } \Phi.
\]

\(\Box\)

Proof of Theorem 4.1

We only present the proof of Part 1 here. The proof is by induction over the definition of \(\mathcal{P}\mathcal{L}\) and the deduction \(\Pi\) of \(\Gamma \triangleright e : \tau\).

1. \(\Pi\) is

\[
\begin{align*}
\Gamma(x) & = \tau \\
\Gamma \triangleright x : \tau
\end{align*}
\]
\( \Delta(x) = \theta, \) then \( \mathcal{P}\mathcal{L}(\Delta, x) = (\{x\}, x^\theta) \). Since \( \text{Closed}(\Delta), \theta = \emptyset \) and we can construct \( \Xi \):

\[
\frac{(\Gamma \star \Delta)(x) = (\tau, \emptyset)}{(\Gamma \star \Delta) \triangleright x : (\tau, \{x\}) \Rightarrow x^\emptyset}
\]

2. \( \Pi \) is

\[
\frac{\Gamma(f) = \tau}{\Gamma \triangleright f : \tau}
\]

\( \Delta(f) = \theta, \) then \( \mathcal{P}\mathcal{L}(\Delta, f) = (\theta, f^\theta) \). We can construct \( \Xi \):

\[
\frac{(\Gamma \star \Delta)(f) = (\tau, \theta)}{(\Gamma \star \Delta) \triangleright f : (\tau, \theta) \Rightarrow f^\theta}
\]

3. \( \Pi \) is

\[
\frac{\Pi_1}{\frac{\Gamma\{y : \tau\} \triangleright e : \tau_1}{\Gamma \triangleright \lambda y.e : \tau \rightarrow \tau_1}}
\]

and \( \mathcal{P}\mathcal{L}(\Delta, \lambda y.e) = (\theta - \{y\}, \lambda y.m) \). By induction on \( \Pi_1 \) and the recursive call to \( \mathcal{P}\mathcal{L} \) we have

\[
\Xi_1 :: (\Gamma\{y : \tau\} \star \Delta\{y : \emptyset\}) \triangleright e : (\tau_1, \theta) \Rightarrow m.
\]

By Definition 11, \( \Gamma\{y : \tau\} \star \Delta\{y : \emptyset\} = (\Gamma \star \Delta)\{y : (\tau, \emptyset)\} \). We can construct \( \Xi \):

\[
\frac{\Xi_1}{\frac{(\Gamma \star \Delta)\{y : (\tau, \emptyset)\} \triangleright e : (\tau_1, \theta) \Rightarrow m}{(\Gamma \star \Delta) \triangleright \lambda y.e : (\tau \rightarrow \tau_1, \theta - \{y\}) \Rightarrow \lambda y.m}}
\]
4. \( \Pi \) is

\[
\begin{array}{c}
\Pi_1 & \Pi_2 \\
\Gamma \triangleright e_1 : \tau_1 \rightarrow \tau & \Gamma \triangleright e_2 : \tau_1 \\
\hline
\Gamma \triangleright e_1 \circ e_2 : \tau \\
\end{array}
\]

and \( \mathcal{P}\mathcal{L}(\Delta, e_1 \circ e_2) = (\theta_1 \cup \theta_2, m_1 \circ m_2) \). By induction on \( \Pi_1 \) and \( \Pi_2 \), and the recursive calls to \( \mathcal{P}\mathcal{L} \), we have

\[
\Xi_1 :: (\Gamma \times \Delta) \triangleright e_1 : (\tau_1 \rightarrow \tau, \theta_1) \Rightarrow m_1
\]

\[
\Xi_2 :: (\Gamma \times \Delta) \triangleright e_2 : (\tau_1, \theta_2) \Rightarrow m_2
\]

We can construct \( \Xi \):

\[
\begin{array}{c}
\Xi_1 \\
(\Gamma \times \Delta) \triangleright e_1 : (\tau_1 \rightarrow \tau_1) \Rightarrow m_1 & (\Gamma \times \Delta) \triangleright e_2 : (\tau_1, \theta_2) \Rightarrow m_2 \\
\hline
(\Gamma \times \Delta) \triangleright e_1 \circ e_2 : (\tau, \theta_1 \cup \theta_2) \Rightarrow m_1 \circ m_2 \\
\end{array}
\]

5. \( \Pi \) is

\[
\begin{array}{c}
\Pi_i \\
\Gamma' = \Gamma \{g_i : \tau_i\} & \Gamma' \triangleright \lambda y_i.e_i : \tau_i \rightarrow \tau_i' & \Pi' \\
\hline
\Gamma \triangleright \text{letrec } g_i = \lambda y_i.e_i \text{ in } e : \tau \\
\end{array}
\]

and \( \mathcal{P}\mathcal{L}(\Delta, \text{letrec } g_i = \lambda y_i.e_i \text{ in } e) = (\bigcup e \theta \cup \theta, \text{letrec } g_i = \epsilon \theta g_i \lambda y_i.e_m \text{ in } m) \). \( \Delta' = \Delta\{g_i : \theta_i \} \) and by induction on \( \Pi_i \) and the recursive calls to \( \mathcal{P}\mathcal{L} \), we have

\[
\Xi_i :: (\Gamma' \times \Delta') \triangleright \lambda y_i.e_i : (\tau_i \rightarrow \tau_i', \theta_i) \Rightarrow m_i
\]
ε = \text{solve}(\{\frac{θ_{g_{i}}}{g_{i}} = \frac{θ_{i}}{g_{i}}\}) \text{ and by Lemma 4.1, } εθ_{g_{i}} = εθ_{i} \setminus θ_{g_{i}} = ε\theta_{i}. \text{ By applying the substitution } ε \text{ to the deductions } Ξ_{i}, \text{ we have}

\[ Ξ'_{i} :: (Γ' * εΔ') \triangleright λ_{i}. e_{i} : (τ_{i} \rightarrow τ'_{i}, εθ_{i}) \Rightarrow εm_{i}. \]

By induction on Π' and the recursive call to $\mathcal{P}\mathcal{L}$, we have

\[ Ξ' :: (Γ' * εΔ') \triangleright e : (τ, θ) \Rightarrow m \]

There are no $θ_{g_{i}}$ in Δ, so by Definition 11

\[ (Γ' * εΔ') = (Γ * Δ)\{g_{i} : (τ_{i}, εθ_{g_{i}})\} \]

Now we can construct:

\[ Ξ'_{i} \quad (Γ' * εΔ') \triangleright λ_{i}. e_{i} : (τ_{i} \rightarrow τ'_{i}, εθ_{i}) \Rightarrow εm_{i}. \quad (Γ' * εΔ') \triangleright e : (τ, θ) \Rightarrow m \]

\[ (Γ * Δ) \triangleright \text{letrec } g_{i} = λ_{i}. e_{i} \text{ in } e : (τ \cup εθ_{g_{i}} \cup θ) \Rightarrow \text{letrec } g_{i} = εθ_{g_{i}} λ_{i}. m_{i} \text{ in } m \]

□
Appendix C

UnCurrying Proof of Type Soundness

The following three lemmas are used in the proof of the type soundness theorem. The first lemma states that it is safe to add a variable-type mapping to a type context without affecting the inferred type of an expression so long as the variable does not occur free in the expression.

Lemma C.1 (Weakening).

If $\Gamma \vdash m : \tau$ and $x \notin FV(m)$, then $\Gamma\{x : \tau\} \vdash m : \tau$.

The second lemma shows that annotated types inferred by the unCurry specification are well-formed when given a well-formed type context.

Lemma C.2.

If $\Delta;\Gamma_c \triangleright e : \tau \Rightarrow m$ and $w.f.(\Gamma_c)$, then $\tau$ is well-formed.

Proof.

By induction on the structure of the deduction $\Xi :: \Delta;\Gamma_c \triangleright e : \tau \Rightarrow m$.

1. (funvar) $\Xi ::$

   $\Gamma_c(f) = \{g^1_1, \ldots, g^n_n\}$

   $\frac{}{\text{w.f.}(\Gamma_c), \tau_i \text{ is well-formed.}}$

   Since $w.f.(\Gamma_c)$, $\tau_i$ is well-formed.
2. \((\text{abs } \epsilon) \quad \Xi :: \)

\[
\Delta; \Gamma_c \triangleright e : \tau_2 \Rightarrow m \\ \|\tau\|_1 = \tau_1 \\
\text{w.f.}(\tau_1)
\]

\[
\Delta; \Gamma_c \triangleright \lambda x^{\tau_1}.e : \tau_1 \xrightarrow{\epsilon} \tau_2 \Rightarrow \lambda [x].m
\]

Since \(\|\tau\|_1 = \tau_1\) and \(\text{w.f.}(\tau_1)\) and \(\text{w.f.}(\Gamma_c)\), the context \(\Gamma_c \{x : \{\tau_1\}\}\) is well-formed.

Thus, by induction on \(\Xi'\), \(\tau_2\) is well-formed. Since \(\text{w.f.}(\tau_1')\) and \(\text{w.f.}(\tau_2)\), by the definition of well-formed type, we have \(\text{w.f.}(\tau_1' \xrightarrow{\epsilon} \tau_2)\).

3. \((\text{abs } \zeta) \quad \Xi :: \)

\[
\Delta + \tau_1' ; \Gamma_c \{x : \{\tau_1'\}\} \triangleright \lambda y^{\tau_2}.e : \tau_3 \Rightarrow \lambda [y_1, \ldots, y_n].m \\
\|\tau\|_1 = \tau_1 \\
\text{w.f.}(\tau_1')
\]

\[
\Delta; \Gamma_c \triangleright \lambda x^{\tau_1}.\lambda y^{\tau_2}.e : \tau_1 \xrightarrow{\epsilon} \tau_3 \Rightarrow \lambda [x, y_1, \ldots, y_n].m
\]

Since \(\text{w.f.}(\Gamma_c)\), \(\|\tau\|_1 = \tau_1\) and \(\text{w.f.}(\tau_1')\), we have \(\text{w.f.}(\Gamma_c \{x : \{\tau_1'\}\})\). Thus, by induction on \(\Xi'\), we have \(\text{w.f.}(\tau_3)\).

(a) \(\Xi''\) has the form:

\[
\Xi''
\]

\[
\Delta + \tau_1' ; \Gamma_c \{x : \{\tau_1'\}\} \triangleright e : \tau_4 \Rightarrow m \\
\|\tau\|_2 = \tau_2 \\
\text{w.f.}(\tau_1')
\]

\[
\Delta + \tau_1' ; \Gamma_c \{x : \{\tau_1'\}\} \triangleright \lambda y^{\tau_2}.e : \tau_3 \Rightarrow \lambda [y].m
\]
Observe that \( \tau_3 \) must have the form \( \tau_2^{\Delta+\tau'_1} \in \tau_4 \). Since \( \text{w.f.}(\tau'_1) \) and \( \text{w.f.}(\tau_3) \), by the definition of well-formed type, we have

\[
\text{w.f.}(\tau_1^{\Delta} \in \tau_3).
\]

(b) The remaining two cases, where \( \Xi' \) is an instance of \( (\text{abs}_{\zeta}) \) and \( (\text{uvabs}) \), follow similarly.

4. \((\text{letrec})\)

\[
\begin{align*}
\Xi_{i_1}^{\Xi_{k_i}} && \Xi_{i_1}^{\Xi_{k_i}} \\
\text{if } \Gamma' \triangleright e_i : \tau_i \Rightarrow m_i \ldots \Gamma' \triangleright e_i : \tau_i \Rightarrow m_i \\
\text{then } \Gamma' = \Gamma_c^c(f_i : \{g_i^{i_j}\}) & \text{ fresh}(g_i^{i_j}) \quad \parallel \tau_{i_j} \parallel = \tau_i \text{ w.f.}(\tau_{i_j}) \quad i \in \{1..n\} \\
\text{letrec } \frac{\text{letrec } \Xi_i^{\Xi_i}}{\Xi_i^{\Xi_i}} \quad \text{in } e : \tau \Rightarrow \text{letrec } g_i^{i_j} = m_i^{i_j} \text{ in } m & j \in \{1..k_i\} \\
\end{align*}
\]

Since \( \text{w.f.}(\Gamma_c) \), \( \parallel \tau_{i_j} \parallel = \tau_{i_j} \) and \( \text{w.f.}(\tau_{i_j}) \), for \( i = [i..n] \) and \( j = [1..k_i] \), we have \( \text{w.f.}(\Gamma'_c) \). Thus, by induction on \( \Xi' \), we know \( \tau \) is well-formed.

5. \((\text{uvabs})\)

\[
\begin{align*}
\Delta + \tau_1 ; \Gamma_c \triangleright \lambda x^{\tau} \cdot e : \tau_2 \Rightarrow \lambda[x_1, \ldots, x_n].m & \text{ fresh}(y) \text{ w.f.}(\tau_1) \\
\Delta ; \Gamma_c \triangleright \lambda x^{\tau} \cdot e : \tau'_1 \Delta \tau_2 \Rightarrow \lambda[y, x_1, \ldots, x_n].m & \text{ w.f.}(\tau_2) \\
\end{align*}
\]

By induction on \( \Xi' \), we have \( \text{w.f.}(\tau_2) \).
(a) $\Xi'$ has the form:

$$\Xi''$$

$$[[]\, \Gamma_c \{ x : \{ x' \} \} \triangleright e : \tau_3 \Rightarrow m \| \tau' \| = \tau \quad \text{w.f.}(\tau')$$

$$\Delta + \tau_1; \Gamma_c \triangleright \lambda x^\tau.e : \tau_2 \Rightarrow \lambda[x].m$$

Observe $\tau_2$ must have the form: $\tau' \overset{\Delta + \tau_1}{\Rightarrow} _\varepsilon \tau_3$. Observe also that, since w.f.$(\tau_1)$, $\tau_2^n$ is well-formed. Thus, by the definition of well-formed type, $\tau_2^n_1 \overset{\Delta}{\Rightarrow} _\varsigma \tau_2$ is well-formed.

(b) The remaining tho cases, where $\Xi'$ is $(abs.\zeta)$ and $(uvabs)$, follow similarly.

6. \textbf{\textit{app.}}

\[\Xi::\]

$$\Xi_1$$

$$[[]\, \Gamma_c \triangleright e_1 : \tau' \overset{\epsilon}{\Rightarrow} \tau \Rightarrow m_1 \| \Gamma_c \triangleright e_2 : \tau' \Rightarrow m_2$$

$$[[]\, \Gamma_c \triangleright e_1 @ e_2 : \tau \Rightarrow m_1 @ [m_2]$$

By induction on $\Xi_1$, $\tau' \overset{\epsilon}{\Rightarrow} \tau$ is well-formed. By the definition of well-formed type, $\tau$ is well-formed.

The remaining application cases (including those for useless parameters) follow similarly.

\[\square\]

The third and final lemma requires the following definition of function arity.
DEFINITION 21 (ARITY).

\[
\text{arity}(\tau_1 \overset{\Delta}{\rightarrow} \epsilon \tau_2) = 1 \\
\text{arity}(\tau_1 \overset{\Delta}{\rightarrow} \varsigma \tau_2) = 1 + \text{arity}(\tau_2)
\]

The arity of a target language function is the number of parameters in the function’s parameter list. When unCurrying a function, parameters are included in the same parameter list so long as the unCurry annotation on the type arrow is \(\varsigma\). Given an unCurried function’s annotated type, that function’s arity can be determined by counting the \(\varsigma\) annotations in the function’s type until an \(\epsilon\) annotation is encountered. Thus, the arity of a function of type \(\tau_1 \overset{\varsigma}{\rightarrow} \tau_2 \overset{\varphi}{\rightarrow} [\tau_3] \overset{\epsilon}{\rightarrow} \tau_4\) has arity 3 whereas a function of type \(\tau_1 \overset{\varsigma}{\rightarrow} \tau_2 \overset{\varphi}{\rightarrow} [\tau_3] \overset{\epsilon}{\rightarrow} [\tau_4] \overset{\epsilon}{\rightarrow} \tau_4\) has arity 2.

The lemma below concerns the definition of \(|\cdot|_{\Delta}\). Recall that if target expression has annotated type \(\tau [\tau_1, \ldots, \tau_n] \gamma \rightarrow \tau'\), then its target language type is either a function type or a partial application (product) type. The following lemma shows the consistency between function and product types that are related to the same annotated type.

LEMMA C.3 \(|\cdot|_{\Delta}\).

For w.f.(\(\tau [\tau_1, \ldots, \tau_n] \gamma \rightarrow \tau'\)) and \(k, n \geq 1\), if

\[
|\tau [\tau_1, \ldots, \tau_n] \gamma \rightarrow \tau'|_{[\tau_1, \ldots, \tau_n]} = [\phi_1, \ldots, \phi_k] \rightarrow \phi, \text{ then}
\]

\[
|\tau [\tau_1, \ldots, \tau_n] \gamma \rightarrow \tau'|_{[\tau_1, \ldots, \tau_n]} = ([\tau_1]_{\|}, \ldots, [\tau_n]_{\|}, \phi_1, \ldots, \phi_k \rightarrow \phi) \times [\tau_1]_{\|} \times \cdots \times [\tau_n]_{\|}.
\]

Proof.

By induction on \(\text{arity}(\tau [\tau_1, \ldots, \tau_n] \gamma \rightarrow \tau').\)
• **Base:** \( \text{arity}(\tau \[
abla \rightarrow \gamma \tau \]) = 1 \)

By the definition of \( \text{arity()} \), \( \gamma = \varepsilon \). By the definition of \( |\cdot| \_\Delta \), if \( \tau [\tau_1, \ldots, \tau_n] \varepsilon \)
\( \tau' [\tau_1, \ldots, \tau_n] = \phi \), then \( \phi = [\tau|\|] \rightarrow \tau'|\| \). Thus we have,

\[
|\tau [\tau_1, \ldots, \tau_n] \varepsilon \tau'|\| = ([\tau_1|\|, \ldots, |\tau_n|\|] \rightarrow |\tau'|\|) \times |\tau_1|\| \times \cdots \times |\tau_n|\|
\]

• **Induction:** \( \text{arity}(\tau [\tau_1, \ldots, \tau_n] \varepsilon \tau') > 1 \)

By the definition of \( \text{arity()} \), \( \gamma = \varsigma \). If \( \tau [\tau_1, \ldots, \tau_n] \varsigma \tau' [\tau_1, \ldots, \tau_n] = \phi \), then, by the definition of \( |\cdot| \_\Delta \),

- \( \phi = [\tau|\|, \phi_1, \ldots, \phi_k] \rightarrow \phi' \), and
- \( |\tau'|[\tau_1, \ldots, \tau_n] = [\phi_1, \ldots, \phi_k] \rightarrow \phi' \)

Since w.f.(\( \tau [\tau_1, \ldots, \tau_n] \varsigma \tau' \), \( \tau'' [\tau_1, \ldots, \tau_n \tau] \gamma \tau''' \).

If \( \text{arity}(\tau [\tau_1, \ldots, \tau_n] \varsigma \tau') = n \), then \( \text{arity}(\tau') = n - 1 \). Thus, by induction:

\[
|\tau'|\| = ([\tau_1|\|, \ldots, |\tau_n|\|, \phi_1, \ldots, \phi_k] \rightarrow \phi') \times |\tau_1|\| \times \cdots \times |\tau_n|\| \times |\tau'|\|
\]

Thus we have,

\[
|\tau [\tau_1, \ldots, \tau_n] \varsigma \tau'|\| = ([\tau_1|\|, \ldots, |\tau_n|\|, \phi_1, \ldots, \phi_k] \rightarrow \phi') \times |\tau_1|\| \times \cdots \times |\tau_n|\|
\]

\( \square \)
Proof of Theorem 5.1.

I prove both parts simultaneously by induction over the deduction

$$\Xi :: \Delta \Gamma_c \vdash e : \tau \Rightarrow m.$$

1. (funvar)

$$\Gamma_c(f) = \{g_1^{\tau_1}, \ldots, g_n^{\tau_n}\}$$

$$\Xi :: \emptyset \Gamma_c \vdash f : \tau_i \Rightarrow g_i$$

(a) $$\| \Gamma_c \| (f) = \| \tau_i \|$$. Since w.f.(\Gamma_c), $$\| \Gamma_c \| (f) = \| \tau_i \|$$. Thus we can construct:

$$\frac{\| \Gamma_c \| (f) = \| \tau_i \|}{\| \Gamma_c \| \vdash f : \| \tau_i \|}$$

(b) $$\| (\Gamma_c)_i \| (g_i) = \| (\tau_i)_i \|$$. We can construct:

$$\frac{\| (\Gamma_c)_i \| (g_i) = \| (\tau_i)_i \|}{\| (\Gamma_c)_i \| \vdash g_i : \| (\tau_i)_i \|}$$

The case for (var) follows similarly.

2. (abs $e$)

$$\Xi'$$

$$\emptyset \Gamma_c \{x : \{x^{\tau'}\}\} \vdash e : \tau_2 \Rightarrow m \quad \| \tau'_1 \| = \tau_1 \quad \text{w.f.(\tau'_1)}$$

$$\Xi :: \Delta \Gamma_c \vdash \lambda x^{\tau_1} e : \tau'_1 \rightarrow \tau_2 \Rightarrow \lambda [x].m$$

Since $$\| \tau'_1 \| = \tau_1$$, w.f.(\tau'_1) and w.f.(\Gamma_c), we have w.f.(\Gamma_c \{x : \{x^{\tau'}\}\}). By induction on $$\Xi'$$, we have:

- $$\Theta' :: \| \Gamma_c \{x : \{x^{\tau'}\}\} \vdash e : \| \tau_2 \|$$
\[ I': \vdash (\Gamma_c \{ x : \{ x' \} \}) \vdash m : \langle \tau_2 \rangle \]

(a) Observe:

- \[ \| \Gamma_c \{ x : \{ x' \} \} \| = \| \Gamma_c \{ x : \| \tau' \| \} = \| \Gamma_c \{ x : \tau_1 \} \| \]
- \[ \| \tau'_1 \rightarrow^e \tau_2 \| = \| \tau'_1 \| \rightarrow \| \tau_2 \| = \| \tau_1 \| \rightarrow \| \tau_2 \| \]

We can construct:

\[
\frac{\| \Gamma_c \{ x : \tau'_1 \} \vdash e : \| \tau_2 \|}{\| \Gamma_c \| \vdash \lambda x \tau_1 . e : \tau_1 \rightarrow \| \tau_2 \|}
\]

(b) Observe:

- \[ \| (\Gamma_c \{ x : \{ x' \} \}) \| \vdash e : \| \tau_2 \| \]
- \[ \| (\tau'_1 \rightarrow^e \tau_2) \| \Delta = [\| \tau'_1 \|] \rightarrow [\| \tau_2 \|] \]

We can construct:

\[
\frac{|(\Gamma_c \{ x : \{ x' \} \}) \| \vdash m : \langle \tau_2 \rangle |}{\| (\Gamma_c \| | \vdash \lambda x . m : [\| \tau'_1 \|] \rightarrow [\| \tau_2 \|]} \]

3. (abs) \[ \Xi :: \]

\[
\Xi' \]

\[
\begin{align*}
\Delta + \tau'_1 ; \Gamma_c \{ x : \{ x' \} \} & \triangleright \lambda y_2 . e : \tau_3 \Rightarrow \lambda [y_1, \ldots, y_n] . m \quad \| \tau'_1 \| = \tau_1 \quad \text{w.f.}(\tau'_1) \\
\Delta ; \Gamma_c \triangleright \lambda x \tau_1 . \lambda y_2 . e : \tau'_1 \rightarrow^e \tau_3 \Rightarrow \lambda [x, y_1, \ldots, y_n] . m
\end{align*}
\]

Since \[ \| \tau'_1 \| = \tau_1 \], w.f.\((\tau'_1)\) and w.f.\((\Gamma_c)\), \( \Gamma_c \{ x : \{ x' \} \} \) is well-formed. By induction on \[ \Xi' \], we have:
\[ \Theta' :: \| \Gamma \{ x : \{ x' \} \} \| \vdash \lambda y^{T_2} . e : \| \tau_3 \| \]

\[ \Pi :: \langle \Gamma \{ x : \{ x' \} \} \rangle [\downarrow \lambda [y_1, \ldots, y_n] . m : \langle \tau_3 \rangle |_{\Delta + \tau_1} \]

(a) Observe:

- \[ \| \Gamma \{ x : \{ x' \} \} \| = \| \Gamma \{ x : \| \tau_1' \| \} \| = \| \Gamma \{ x : \tau_1 \} \| \]
- \[ \| \tau_1' \Delta \tau_3 \| = \| \tau_1' \| \rightarrow \| \tau_3 \| = \tau_1 \rightarrow \| \tau_3 \| \]

We can construct:

\[ \Theta' \]

\[ \| \Gamma \{ x : \tau_1 \} \| \vdash \lambda y^{T_2} . e : \| \tau_3 \| \]

\[ \| \Gamma \| \vdash \lambda x^{T_1} . \lambda y^{T_2} . e : \tau_1 \rightarrow \| \tau_3 \| \]

(b) Observe \[ \langle \tau_3 \rangle |_{\Delta + \tau_1} \] has the form \[ [\phi_1, \ldots, \phi_n] \rightarrow \phi \], thus we have:

\[ |\langle \tau_1' \Delta \tau_3 \rangle|_{\Delta} = |\langle \tau_1' \rangle|_{\downarrow} [\phi_1, \ldots, \phi_n] \rightarrow \phi \]

Also, observe \[ |\langle \Gamma \{ x : \{ x' \} \} \rangle|_{\downarrow} = |\langle \Gamma \{ x : \langle \tau_1' \rangle|_{\downarrow} \rangle|_{\downarrow} \]. We can construct:

\[ \Pi' \]

\[ |\langle \Gamma \rangle|_{\downarrow} [\lambda [y_1, \ldots, y_n] . m : \langle \phi_1, \ldots, \phi_n \rangle \rightarrow \langle \tau_1' \rangle|_{\downarrow} \]

\[ |\langle \Gamma \rangle|_{\downarrow} [\lambda x, y_1, \ldots, y_n] . m : |\langle \tau_1' \rangle|_{\downarrow} [\phi_1, \ldots, \phi_n] \rightarrow \phi \]
4. (letrec) \( \Xi :: \)

\[
\Xi_{ij}^{i_j} \quad \Gamma' \vdash e : \tau_{ij} \Rightarrow m_i \quad \Xi' \quad \Gamma' \vdash e : \tau \Rightarrow m \\
\Gamma_c' = \Gamma_c \{ f_i : \{ g_{ij}^{i_j} \} \} \quad \text{fresh}(g_{ij}) \quad \| \tau_{ij} \| = \tau_i \quad \text{w.f.}(\tau_{ij}) \quad i \in \{1..n\}
\]

\[
\Xi_c' \quad \Gamma_c \vdash \text{letrec } \hat{g}_i = e_i \text{ in } e : \tau \Rightarrow \text{letrec } \hat{y}_i = m_i \text{ in } m \\ j \in \{1..k^i\}
\]

Since w.f.\( (\Gamma_c) \) and, for \( i = [1..n] \), \( \Gamma'_c (f_i) = \{ g_{ij}^{i_1}, \ldots, g_{ij}^{i_k} \} \), and \( \| \tau_{ij} \| = \tau_i \) and w.f.\( (\tau_{ij}) \) for \( j = [1..k^i] \), \( \Gamma' \) is well-formed. By induction on the deductions \( \Xi \) (for \( i = [1..n], j = [1..k^i] \)), we have:

- \( \Theta_{ij} :: \| \Gamma'_c \| \vdash e_i : \| \tau_{ij} \| \)
- \( \Pi_{ij} :: |\{ \Gamma'_c \} \| \vdash m_i : |\{ \tau_{ij} \} \| \)

By induction on \( \Xi' \), we have:

- \( \Theta' :: \| \Gamma'_c \| \vdash e : \| \tau \| \)
- \( \Pi' :: |\{ \Gamma'_c \} \| \vdash m : |\{ \tau \} \| \)

(a) Observe \( \| \Gamma'_c \| = \| \Gamma_c \{ f_i : g_{ij}^{i_j} \} \| = \| \Gamma_c \| \{ f_i : \| \tau_{i_1} \| \} \| \). Since w.f.\( (\Gamma'_c) \), \( \| \tau_{i_1} \| = \| \tau_{i_2} \| = \cdots = \| \tau_{i_{k^i}} \| = \tau_i \). Thus we can use the deductions \( \Theta_{i_1} \), for \( i = [1..n] \), and \( \Theta' \) to build:

\[
\Theta_{i_1} \quad \| \Gamma_c \{ f_i : \| \tau_{i_1} \| \} \| \vdash e_i : \| \tau_{i_1} \| \quad \| \Gamma_c \{ f_i : \| \tau_{i_1} \| \} \| \vdash e : \| \tau \| \\
\| \Gamma'_c \| \vdash \text{letrec } \hat{f}_i = e_i \text{ in } e : \| \tau \| \\
\]
(b) Observe \(|\langle \Gamma'_c \rangle| \models \| \langle \Gamma_c \{ f_i : \{ g_{ij} \} \} \rangle | \| = \| \langle \Gamma_c \rangle | \| \{ g_{ij} : \langle \tau_{ij} \rangle \} | \| \). We use the deduc-

tions \( \Pi_{ij} \), for \( i = [1..n] \) and \( j = [1..k^i] \), and the deduction \( \Pi' \) to build:

\[
\Pi_{ij} \quad \Pi'
\frac{|\langle \Gamma_c \rangle| \| \{ g_{ij} : \langle \tau_{ij} \rangle \} | \| \vdash m_{ij} : \langle \tau_{ij} \rangle | \| \quad \frac{|\langle \Gamma_c \rangle| \| \{ g_{ij} : \langle \tau_{ij} \rangle \} | \| \vdash m : \langle \tau \rangle | \|}{|\langle \Gamma_c \rangle| \| \text{letrec } g_{ij} = m_{ij} \text{ in } m : \langle \tau \rangle | \|}
\]

5. \( \text{(app}_1 \text{)} \quad \Xi ::

\[
\begin{array}{c}
\Xi_1 \\
\Xi_2
\end{array}
\]

\[
\frac{|\langle \Gamma_c \rangle| \not\vdash e_1 : \tau' \in e_\epsilon \tau \Rightarrow m_1}{|\langle \Gamma_c \rangle| \not\vdash e_1 \circ e_2 : \tau \Rightarrow m_1 \circ [m_2]}
\]

By induction on \( \Xi_1 \) and \( \Xi_2 \), we have:

- \( \Theta_1 :: |\langle \Gamma_c \rangle| \not\vdash e_1 : \tau'
- \( \Theta_2 :: |\langle \Gamma_c \rangle| \not\vdash e_2 : \tau'
- \( \Pi_1 :: |\langle \Gamma_c \rangle| \not\vdash m_1 : \langle \tau' \in e_\epsilon \tau \rangle | \|
- \( \Pi_2 :: |\langle \Gamma_c \rangle| \not\vdash m_2 : \langle \tau' \rangle | \|

(a) Observe \( \| \tau' \in e_\epsilon \tau \| = \| \tau' \circ \| \rightarrow \| \tau \|. \) We can construct:

\[
\begin{array}{c}
\Theta_1 \\
\Theta_2
\end{array}
\]

\[
\frac{|\langle \Gamma_c \rangle| \not\vdash e_1 : \tau'}{|\langle \Gamma_c \rangle| \not\vdash e_1 \circ e_2 : \tau'}
\]

\[
\frac{|\langle \Gamma_c \rangle| \not\vdash e_2 \circ e_1 : \tau'}{|\langle \Gamma_c \rangle| \not\vdash e_2 : \tau'}
\]
(b) Observe $|\langle \tau' \stackrel{\varepsilon}{\rightarrow} \tau \rangle| \equiv |\langle \tau' \rangle|_\| \rightarrow |\langle \tau \rangle|_\|$. We can construct:

$$
\begin{align*}
\Pi_1 & \\
|\langle \Gamma_c \rangle|_\| \vdash m_1 : |\langle \tau' \rangle|_\| & \rightarrow |\langle \tau \rangle|_\| \\
|\langle \Gamma_c \rangle|_\| & \vdash m_2 : |\langle \tau' \rangle|_\|
\end{align*}
$$

$$
|\langle \Gamma_c \rangle|_\| \vdash m_1 \ominus [m_2] : |\langle \tau \rangle|_\|
$$

6. \(app_{\varepsilon_2}\) \(\Xi::\)

$$
\begin{align*}
\Xi_1 & \\
[]; \Gamma_c \triangleright e_1 : \tau' \stackrel{\varepsilon_1}{\rightarrow} \tau \Rightarrow |\langle m_0, \ldots, m_n \rangle|_\| & []; \Gamma_c \triangleright e_2 : \tau' \Rightarrow m \\
\Xi_2 & \\
[]; \Gamma_c \triangleright e_1 \ominus e_2 : \tau \Rightarrow m_0 \ominus [m_1, \ldots, m_n, m]
\end{align*}
$$

By induction on \(\Xi_1\) and \(\Xi_2\), we have:

- \(\Theta_1 :: |\Gamma_c| \vdash e_1 : \| \tau' \stackrel{\varepsilon_1}{\rightarrow} \varepsilon \tau \|
- \(\Theta_2 :: |\Gamma_c| \vdash e_2 : \| \tau' \|
- \(\Pi_1 :: |\langle \Gamma_c \rangle|_\| \vdash |\langle m_0, \ldots, m_n \rangle| : |\langle \tau' \stackrel{\varepsilon_1}{\rightarrow} \varepsilon \tau \rangle|_\|
- \(\Pi_2 :: |\langle \Gamma_c \rangle|_\| \vdash m : |\langle \tau' \rangle|_\|

(a) Observe $\| \tau' \stackrel{\varepsilon_1}{\rightarrow} \varepsilon \tau \| = \| \tau' \| \rightarrow \| \tau \|$. We can construct:

$$
\begin{align*}
\Theta_1 & \\
|\Gamma_c| \vdash e_1 : \| \tau' \| \rightarrow \| \tau \| & |\Gamma_c| \vdash e_2 : \| \tau' \|
\end{align*}
$$

$$
|\Gamma_c| \vdash e_1 \ominus e_2 : \| \tau \|
$$
(b) Observe \(|\tau^{[\tau_1 \ldots \tau_n]} \in \tau| \equiv ((|\tau_1| \in \tau, \ldots, |\tau_n| \in \tau) \rightarrow |\tau| \in \tau) \times |\tau_1| \in \tau \times \ldots \times |\tau_n| \in \tau|). The deduction \(\Pi_1\) has the form:

\[
\Pi_1'
\begin{array}{c}
|\langle \Gamma \rangle|_\emptyset \vdash m_i : |\langle \tau_i \rangle|_\emptyset \\
|\langle \Gamma \rangle|_\emptyset \vdash m_0 : [|\langle \tau_1 \rangle|_\emptyset, \ldots, |\langle \tau_n \rangle|_\emptyset, |\langle \tau' \rangle|_\emptyset] \rightarrow |\langle \tau \rangle|_\emptyset \quad i = [1..n]
\end{array}
\]

Thus we can construct the required deduction:

\[
\begin{array}{c}
\Pi_0' \\
\Pi_i' \quad \Pi_2
\end{array} \begin{array}{c}
|\langle \Gamma \rangle|_\emptyset \vdash m_0 \circ \circ m_1, \ldots, m_n, m : |\langle \tau \rangle|_\emptyset
\end{array}
\]

7. \((\text{app}_3)\) \(\Xi:\)

\[
\Xi_1 \\
; \Gamma \vdash e_1 : \tau^{[\tau_1 \ldots \tau_n]} \in \tau \Rightarrow m_1 \\
\Xi_2 \\
; \Gamma \vdash e_2 : \tau' \Rightarrow m_2
\]

By induction on \(\Xi_1\) and \(\Xi_2\), we have:

- \(\Theta_1 : |\Gamma_c|_\emptyset \vdash e_1 : |\tau^{[\tau_1 \ldots \tau_n]} \in \tau|\)
- \(\Theta_2 : |\Gamma_c|_\emptyset \vdash e_2 : |\tau'|\)
- \(\Pi_1 : |\langle \Gamma \rangle|_\emptyset \vdash m_1 : |\tau^{[\tau_1 \ldots \tau_n]} \in \tau|_\emptyset|\)
\( \Pi_2 :: |\langle \Gamma_c \rangle|_\| m_2 : |\langle \tau' \rangle|_\| \)

(a) Observe \( |\tau'[\tau_1, \ldots, \tau_n]|_\| \tau = |\tau| \rightarrow |\tau| \). We can construct:

\[
\begin{array}{c}
\Theta_1 \\
\| \Gamma_c \| \vdash e_1 : |\tau'| \rightarrow |\tau| \\
\Theta_2 \\
\| \Gamma_c \| \vdash e_2 : |\tau'| \\
\hline
\| \Gamma_c \| \vdash e_1 @ e_2 : |\tau|
\end{array}
\]

(b) \( |\langle \tau'[\tau_1, \ldots, \tau_n] \rangle|_\| = (|\langle \tau_1 |\| |, \ldots, |\langle \tau_n |\| | \rightarrow |\langle \tau |\| | \times |\langle \tau_1 |\| | \times \cdots \times |\langle \tau_n |\| | \}

We can construct the required deduction:

\[
\begin{array}{c}
\Pi_2 \\
|\langle \Gamma_c \rangle|_\| m_2 : |\langle \tau' \rangle|_\| \\
\Pi_1 \\
|\langle \Gamma_c \rangle|_\| m_1 : (|\langle \tau_1 |\| |, \ldots, |\langle \tau_n |\| | \rightarrow |\langle \tau |\| | \times |\langle \tau_1 |\| | \times \cdots \times |\langle \tau_n |\| | \\
\hline
|\langle \Gamma_c \rangle|_\| m_1 @ m_2 : |\langle \tau |\| |
\end{array}
\]

8. \( (app_{c_1}) \quad \Xi :: \)

\[
\Xi_1 \\
[\|; \Gamma_c \triangleright e_1 : \tau' \|_\cdot c \triangleright m_1 \\
\Xi_2 \\
[\|; \Gamma_c \triangleright e_2 : \tau' \Rightarrow m_2 \\
\hline
[\|; \Gamma_c \triangleright e_1 @ e_2 : \tau \Rightarrow \langle m_1, m_2 \rangle
\]

By induction on \( \Xi_1 \) and \( \Xi_2 \), we have:

* \( \Theta_1 :: \| \Gamma_c \| \vdash e_1 : |\tau'| \rightarrow |\tau| \)

* \( \Theta_2 :: \| \Gamma_c \| \vdash e_2 : |\tau'| \)
\begin{itemize}
  \item $\Pi_1 : \langle\Gamma_c\rangle[\cdot] \vdash m_1 : \langle\tau' \Downarrow \varsigma \quad \tau\rangle[\cdot]$
  \item $\Pi_2 : \langle\Gamma_c\rangle[\cdot] \vdash m_2 : \langle\tau'\rangle[\cdot]$
\end{itemize}

(a) We can construct the deduction:

\[
\begin{array}{c}
\Theta_1 \\
\hline
\|\Gamma_c\| \vdash e_1 : \|\tau'\| \rightarrow \|\tau\| \\
\|\Gamma_c\| \vdash e_2 : \|\tau'\| \\
\hline
\|\Gamma_c\| \vdash e_1 \& e_2 : \|\tau\|
\end{array}
\]

(b) Observe:

\[
\langle\tau' \Downarrow \varsigma \quad \tau\rangle[\cdot] = \langle\langle\tau'\rangle[\cdot], \phi_1, \ldots, \phi_n\rangle \to \phi
\]

where $\langle\langle\tau'\rangle[\cdot], \phi_1, \ldots, \phi_n\rangle \to \phi$. By Lemma C.2, $\tau' \Downarrow \varsigma \quad \tau$ is well-formed. By the definition of well-formed type, $\tau = \tau'' \rightarrow^{\gamma} \tau'''$ and w.f.($\tau$). Thus, by Lemma C.3, $\langle\langle\tau'\rangle[\cdot], \phi_1, \ldots, \phi_n\rangle \to \phi \times \langle\langle\tau'\rangle[\cdot]\rangle$. We can construct the required deduction:

\[
\begin{array}{c}
\Pi_1 \\
\hline
\langle\Gamma_c\rangle[\cdot] \vdash m_1 : \langle\langle\tau'\rangle[\cdot], \phi_1, \ldots, \phi_n\rangle \to \phi \\
\langle\Gamma_c\rangle[\cdot] \vdash m_2 : \langle\langle\tau'\rangle[\cdot]\rangle \\
\hline
\langle\langle\Gamma_c\rangle[\cdot], \langle\{m_1, m_2\}\rangle, \langle\langle\tau'\rangle[\cdot], \phi_1, \ldots, \phi_n\rangle \to \phi \times \langle\langle\tau'\rangle[\cdot]\rangle\rangle
\end{array}
\]

9. $(app_{\varsigma_2}) \ni$:

\[
\begin{array}{c}
\Xi_1 \\
\hline
[]; \Gamma_c \triangleright e_1 : \tau' \Downarrow \varsigma \quad \tau \Rightarrow m_1 \\
[]; \Gamma_c \triangleright e_2 : \tau' \Rightarrow m_2 \\
\hline
[]; \Gamma_c \triangleright e_1 \& e_2 : \tau \Rightarrow (m_1, m_2)
\end{array}
\]
By induction on $\Xi_1$ and $\Xi_2$, we have:

- $\Theta_1 :: \| \Gamma_c \| \vdash e_1 : \| \tau^\prime \|_{[\tau_1,\ldots,\tau_n]} \tau$
- $\Theta_2 :: \| \Gamma_c \| \vdash e_2 : \| \tau^\prime \|
- $\Pi :: |\langle \Gamma_c \rangle|_\Box \vdash m_1 : |\langle \tau^\prime \|_{[\tau_1,\ldots,\tau_n]} \tau \rangle|_\Box$
- $\Pi_2 :: |\langle \Gamma_c \rangle|_\Box \vdash m_2 : |\langle \tau^\prime \rangle|_\Box$

(a) We can construct the required deduction:

\[
\frac{\Theta_1 : \| \Gamma_c \| \vdash e_1 : \| \tau^\prime \| \rightarrow \| \tau^\prime \| \quad \Theta_2 : \| \Gamma_c \| \vdash e_2 : \| \tau^\prime \|}{\| \Gamma_c \| \vdash e_1 \land e_2 : \| \tau^\prime \|}
\]

(b)

\[
|\langle \tau^\prime \|_{[\tau_1,\ldots,\tau_n]} \tau \rangle|_\Box = (|\langle \tau_1 \rangle|_\Box \times \ldots \times |\langle \tau_n \rangle|_\Box) \times |\langle \tau^\prime \rangle|_\Box \\
= (|\langle \tau_1 \rangle|_\Box \times \ldots \times |\langle \tau_n \rangle|_\Box) \times |\langle \tau^\prime \rangle|
\]

where

\[
|\langle \tau \rangle|_\Box = (|\langle \tau_1 \rangle|_\Box \times \ldots \times |\langle \tau_n \rangle|_\Box) \times |\langle \tau^\prime \rangle|_\Box
\]

We can construct the required deduction:

\[
\frac{\Pi_1 : |\langle \Gamma_c \rangle|_\Box \vdash m_1 : |\langle \tau^\prime \|_{[\tau_1,\ldots,\tau_n]} \tau \rangle|_\Box \quad \Pi_2 : |\langle \Gamma_c \rangle|_\Box \vdash m_2 : |\langle \tau^\prime \rangle|_\Box}{|\langle \Gamma_c \rangle|_\Box \vdash (m_1, m_2) : |\langle \tau \rangle|_\Box}
\]
10. \( (uvabs) \quad \Xi ::

\[
\Delta + \tau_1; \Gamma_c \triangleright \lambda x^\tau.e : \tau_2 \Rightarrow \lambda[x_1, \ldots, x_n].m \quad \text{fresh}(y) \quad \text{w.f.}(\tau_1)
\]

\[
\Delta; \Gamma_c \triangleright \lambda x^\tau.e : \tau_1 \xrightarrow[\Delta]{\tau} \tau_2 \Rightarrow \lambda[y, x_1, \ldots, x_n].m
\]

By induction on \( \Xi' \), we have:

- \( \Theta' :: \| \Gamma_c \| \vdash \lambda x^\tau.e : \| \tau_2 \|
\)

- \( \Pi' :: \| \langle \Gamma_c \rangle \| \vdash \lambda[x_1, \ldots, x_n].m : \| \langle \tau_2 \rangle \|_{\Delta + \tau_1}
\)

(a) Observe \( \| \tau_1^u \xrightarrow{} \tau_2 \| = \| \tau_2 \| \). Thus, the required deduction is \( \Theta' \).

(b) Observe \( \| \langle \tau_2 \rangle \|_{\Delta + \tau_1} \) has the form \( [\phi_1, \ldots, \phi_n] \rightarrow \phi \).

Thus \( \| \langle \tau_1 \rangle \|_{\Delta} = \| \langle \tau_1 \rangle \|_{\Delta} \rightarrow \phi_1, \ldots, \phi_n \rightarrow \phi \).

Since \( y \notin \text{FV}(\lambda[x_1, \ldots, x_n].m) \) and \( \Pi' \), by the Weakening Lemma, we have:

\( \Pi'' :: \| \langle \Gamma_c \rangle \| \{ y : \| \langle \tau_1 \rangle \| \} \vdash \lambda[x_1, \ldots, x_n].m : [\phi_1, \ldots, \phi_n] \rightarrow \phi \)

We can construct the desired deduction:

\[
\| \langle \Gamma_c \rangle \| \{ y : \| \langle \tau_1 \rangle \| \} \vdash \lambda[x_1, \ldots, x_n].m : [\phi_1, \ldots, \phi_n] \rightarrow \phi
\]

\( \Pi'' \)

\[
\| \langle \Gamma_c \rangle \| \vdash \lambda[y, x_1, \ldots, x_n].m : \| \langle \tau_1 \rangle \|_{\Delta}, \phi_1, \ldots, \phi_n \rightarrow \phi
\]

11. \( (uvapp_1) \quad \Xi ::

\[
\Xi'
\]

\[
\begin{aligned}
\| ; \Gamma_c \triangleright e : \tau^u \xrightarrow{} \tau' & \Rightarrow m \\
\| ; \Gamma_c \triangleright e : \tau' & \Rightarrow ((\cdot, m), d(\tau')|\|)
\end{aligned}
\]
By induction on $\Xi'$, we have:

- $\Theta :: \parallel \Gamma_c \parallel \vdash e : \parallel \tau^u \Downarrow \varsigma \tau' \parallel$

- $\Pi :: \parallel \Gamma_c \parallel \vdash m : \parallel \langle \tau^u \Downarrow \varsigma \tau' \rangle \parallel$

(a) Since $\parallel \tau^u \Downarrow \varsigma \tau' \parallel = \parallel \tau' \parallel$, $\Theta$ is the required deduction.

(b) $\parallel \langle \tau^u \Downarrow \varsigma \tau' \rangle \parallel = \parallel \langle \tau \Downarrow \varsigma \tau' \rangle \parallel = \parallel \langle \tau \Downarrow \varsigma \tau' \rangle \parallel = \parallel \langle \tau \rangle \parallel, \phi_1, \ldots, \phi_n \rightarrow \phi$

where $\langle \tau' \rangle[\tau] = [\phi_1, \ldots, \phi_n] \rightarrow \phi$

By $\Xi'$ and Lemma C.2, $\tau^u \Downarrow \varsigma \tau'$ is well-formed. By Definition 17, $\tau' = \tau'' \Downarrow \gamma$

$\tau'''$ and w.f.$(\tau')$. By Lemma C.3, $\parallel \langle \tau' \rangle \parallel = (\parallel \langle \tau \rangle \parallel, \phi_1, \ldots, \phi_n \rightarrow \phi) \times \parallel \langle \tau \rangle \parallel$.

We can construct the required deduction:

$$
\begin{array}{c}
\parallel \Gamma_c \parallel \vdash m : \parallel \langle \tau \rangle \parallel, \phi_1, \ldots, \phi_n \rightarrow \phi \\
\parallel \Gamma_c \parallel \vdash d^{\langle \tau \rangle \parallel}
\end{array}
\frac{\parallel \Gamma_c \parallel \vdash \Pi}{\parallel \Gamma_c \parallel \vdash (\langle \cdot, m, d^{\langle \tau \rangle \parallel} \rangle : \parallel \langle \tau \rangle \parallel, \phi_1, \ldots, \phi_n \rightarrow \phi) \times \parallel \langle \tau \rangle \parallel}

12. (uvapp_2) \quad \Xi ::

$$
$$
\begin{array}{c}
\parallel \Gamma_c \parallel \vdash e : \tau^u \Downarrow \{\tau_1, \ldots, \tau_n\} \quad \tau' \Rightarrow m
\end{array}
\frac{\parallel \Gamma_c \parallel \vdash e : \tau' \Rightarrow (m, d^{\langle \tau' \rangle \parallel})}{\parallel \Gamma_c \parallel \vdash e : \tau' \Rightarrow (m, d^{\langle \tau' \rangle \parallel})}

By induction on $\Xi'$, we have:

- $\Theta :: \parallel \Gamma_c \parallel \vdash e : \parallel \tau^u \Downarrow \{\tau_1, \ldots, \tau_n\} \tau' \parallel$

- $\Pi :: \parallel \Gamma_c \parallel \vdash m : \parallel \langle \tau^u \Downarrow \{\tau_1, \ldots, \tau_n\} \tau' \rangle \parallel$

(a) Since $\parallel \tau^u \Downarrow \{\tau_1, \ldots, \tau_2\} \tau' \parallel = \parallel \tau' \parallel$, the required deduction is $\Theta$. 

(b) \(|\langle \tau^u [\tau_1 \cdots \tau_n] \langle \tau' \rangle | \rangle = |\langle \tau [\tau_1 \cdots \tau_n] \langle \tau' \rangle | \rangle = |\langle \tau [\tau_1 \cdots \tau_n] \langle \tau' \rangle | \rangle =
\)

\(|\langle \tau_1 \rangle | \rangle \cdots |\langle \tau_n \rangle | \rangle \langle \tau | \phi_1, \cdots, \phi_k \rangle \rightarrow \phi \rangle \times |\langle \tau_1 \rangle | \rangle \cdots |\langle \tau_n \rangle | \rangle \langle \tau | \phi \rangle \)

where \(|\langle \tau' \rangle | \rangle = (|\langle \tau_1 \rangle | \rangle \cdots |\langle \tau_n \rangle | \rangle \langle \tau | \phi_1, \cdots, \phi_k \rangle \rightarrow \phi \rangle \times |\langle \tau_1 \rangle | \rangle \cdots |\langle \tau_n \rangle | \rangle \langle \tau | \phi \rangle \)

\times |\langle \tau_1 \rangle | \rangle \cdots |\langle \tau_n \rangle | \rangle \times |\langle \tau | \rangle | \rangle \)

We can construct the required deduction:

\[
\Pi
\]

\[
|\langle \Gamma' \rangle | \rangle \vdash m : |\langle \tau [\tau_1 \cdots \tau_n] \langle \tau' \rangle | \rangle \quad |\langle \Gamma' \rangle | \rangle \vdash d |\langle \tau | \rangle | \rangle
\]

\[
|\langle \Gamma' \rangle | \rangle \vdash (m, \cdot |\langle \tau | \rangle | \rangle) : |\langle \tau' \rangle | \rangle
\]
Appendix D

UnCurrying Proof of Operational Correctness

**Lemma D.1 (Substitution).**

If \( \Delta; \Gamma_c \{ f_i : \{ g_i^j \} \} \triangleright e : \tau \Rightarrow m \) and \( \Gamma_c \triangleright e_i : \tau_{i_j} \Rightarrow m_{i_j} \) then

\[ \Delta; \Gamma_c \triangleright e[\overline{m_{i_j} / g_i^j}] : \tau \Rightarrow m[\overline{m_{i_j} / g_i^j}] \], for \( i = [1..n] \) and, for each \( i, j = [1..k_i] \).

**Proof.**

The proof is by induction on the deduction \( \Xi :: \Delta; \Gamma_c \{ f_i : \{ g_i^j \} \} \triangleright e : \tau \Rightarrow m \). I only include two cases here. The rest are straightforward.

1. \( \Xi :: \)

   \[
   g^T \in \Gamma_c \{ f_i : \{ g_i^j \} \}(f) \]

   \[ \Delta; \Gamma_c \{ f_i : \{ g_i^j \} \} \triangleright f : \tau \Rightarrow g \]

   \( \Xi :: []; \Gamma_c \triangleright e_i : \tau_{i_j} \Rightarrow m_{i_j} \)

   There are two cases:

   (a) The variable \( f \) is not one of the variables \( f_i \). Thus \( g^T \in \Gamma_c(f) \). Since all variable names are unique in any type context, \( g \) cannot be one of the variables \( g_i^j \). We then have \( f[\overline{e_i / f_i}] = f \) and \( g[\overline{m_{i_j} / g_i^j}] = g \). We can construct the required deduction:

   \[ g^T \in \Gamma_c(f) \]

   \[ []; \Gamma_c \triangleright f : \tau \Rightarrow g \]
(b) The variable \( f = f_l \) where \( 1 \leq l \leq n \). The variable \( g \) must then be \( g_{i_p} \) and \( \tau = \tau_{i_p} \) where \( 1 \leq p \leq k_l \). We then have \( f_l^{[e_i/f_i]} = e_l \) and \( g_{i_p}^{[m_{i_j}/g_{i_j}]} = m_{i_p} \). The required deduction is \( \Xi_{i_p} \).

2. \( \Xi ::= \)

\[
\Xi' \]

\[
\frac{}{[; \Gamma \{ f_i : \{ \overrightarrow{g_{i_j}} \} \}; \{ x : \overrightarrow{x} \} ] \vdash e : \tau \Rightarrow m} \Delta; \Gamma \{ f_i : \{ \overrightarrow{g_{i_j}} \} \} \vdash \lambda x. e : \tau \Rightarrow \lambda[x]. m
\]

\( \Xi_{i_j} ::= [ ]; \Gamma \vdash e_i : \tau_{i_j} \Rightarrow m_{i_j} \)

Since variable names are unique,

\[
\frac{}{\Gamma \{ f_i : \{ \overrightarrow{g_{i_j}} \} \}; \{ x : \overrightarrow{x} \} = \Gamma \{ \overrightarrow{x} \}; \{ f_i : \{ \overrightarrow{g_{i_j}} \} \}}
\]

By induction, we have:

\[
\Xi'' ::= [ ]; \Gamma \{ \overrightarrow{x} \} \vdash e_i[f_i] : \tau \Rightarrow m_{i_j}/g_{i_j}
\]

We can construct:

\[
\Xi'' \]

\[
\frac{}{[ ]; \Gamma \{ \overrightarrow{x} \} \vdash e_i[f_i] : \tau \Rightarrow m_{i_j}/g_{i_j}} \Delta; \Gamma \vdash \lambda x. e_i[f_i] : \tau \Rightarrow \lambda [x]. m_{i_j}/g_{i_j}
\]

Observe \( \lambda x. e_i[f_i] = (\lambda x. e)\overrightarrow{[f_i]} \) and \( \lambda[x]. m_{i_j}/g_{i_j} = (\lambda[x]. m)\overrightarrow{[m_{i_j}/g_{i_j}]} \).
Proof of Theorem 5.2.

The proof of Part 1 is by induction on the deductions \( \Xi :: \Delta; \Gamma_c \triangleright e : \tau \Rightarrow m \) and  
\( \Pi :: \rho \triangleright e \leftarrow v. \)

1. \( \Xi :: \)

\[
y^T \in \Gamma_c(x) \\
\therefore; \Gamma_c \triangleright x : \tau \Rightarrow y
\]

\( \Pi :: \)

\[
\rho(x) = v \\
\rho \triangleright x \Leftarrow v
\]

Since \( \rho : \Gamma_c \Rightarrow \rho', \rho(x) = v, \) and \( y^T \in \Gamma_c(x), \) by Definition 19 we have \( \rho'(y) = v' \) and \( \therefore \triangleright v : \tau \Rightarrow v'. \) We can construct the deduction:

\[
\rho'(y) = v' \\
\rho' \triangleright y \Leftarrow v'
\]

2. \( \Xi :: \)

\[
\Xi' \\
\therefore; \Gamma_c\{x : \{x^T\}\} \triangleright e : \tau_2 \Rightarrow m \\
\Delta; \Gamma_c \triangleright \lambda x^{T_1}.e : \tau_1 \overset{1}{\Rightarrow} \tau_2 \Rightarrow \lambda [x].m
\]

\( \Pi :: \)

\[
\rho \triangleright \lambda x^{T_1}.e \Leftarrow [\rho, \lambda x.e]
\]

We can construct the deduction:

\[
\rho' \triangleright \lambda [x].m \Leftarrow [\rho', \lambda [x].m]
\]
From $\Xi$ and $\rho : \Gamma \Rightarrow \rho'$, by Definition 20 we have:

$$\Delta \triangleright [\rho, \lambda x.e] : \tau_1 \xrightarrow{\xi} \tau_2 \Rightarrow [\rho', \lambda [x].m]$$

3. $\Xi ::$

$$\Delta + \tau'_1 \Gamma_c \{x : \{x'\}\} \triangleright \lambda y^{\tau_2}.e : \tau_3 \Rightarrow \lambda [y_1, \ldots, y_n].m$$

$$\Delta ; \Gamma_c \triangleright \lambda x^{\tau_1}, \lambda y^{\tau_2}.e : \tau'_1 \xrightarrow{\xi} \tau_3 \Rightarrow \lambda [x, y_1, \ldots, y_n].m$$

$\Pi ::$

$$\rho \triangleright \lambda x^{\tau_1}.\lambda y^{\tau_2}.e \rightarrow [\rho, \lambda x.\lambda y.e]$$

We can construct the deduction:

$$\rho' \triangleright \lambda [x, y_1, \ldots, y_n].m \rightarrow [\rho', \lambda [x, y_1, \ldots, y_n].m]$$

From $\Xi$ and $\rho : \Gamma \Rightarrow \rho'$, from Definition 20 we have:

$$\Delta \triangleright [\rho, \lambda x.\lambda y.e] : \tau'_1 \xrightarrow{\xi} \tau_3 \Rightarrow [\rho', \lambda [x, y_1, \ldots, y_n].m]$$

4. $\Xi ::$ (where w.f.$(\tau_i)$ and $\|\tau_{ij}\| = \tau_i$)

$$\Xi_{ij}$$

$$\Xi'$$

$$\Xi'$$

$$\Xi'$$

$$[]; \Gamma_c \{f_i : \{g'_{ij}\}\} \triangleright e_i \triangleright m_{ij}$$

$$[]; \Gamma_c \{f_i : \{g'_{ij}\}\} \triangleright e \triangleright m$$

$$[]; \Gamma_c \triangleright \text{letrec } f'_{ij} = e_i \text{ in } e : \tau \Rightarrow \text{letrec } g'_{ij} = m_{ij} \text{ in } m$$
\[\Pi : \]

\[
\rho \triangleright e, \text{let rec } f_i = e_i \text{ in } f / f_i \rightarrow v_i, \quad \rho \{ f_i \mapsto v_i \} \triangleright e \rightarrow v
\]

\[
\rho \triangleright \text{let rec } f_i = e_i \text{ in } e \rightarrow v
\]

We can construct the deductions \(\Xi'\) :

\[
\Xi'_{i,j}
\]

\[
\Gamma_c \{ f_i : \{ g_{i,j}^{\tau_{i,j}} \} \} \triangleright e_{i,j} : \tau_{i,j} \Rightarrow m_{i,j} \]

\[
\Gamma_c \triangleright \text{let rec } f_i = e_i \text{ in } f_i : \tau_{i,j} \Rightarrow \text{let rec } g_{i,j} = m_{i,j} \text{ in } g_{i,j}
\]

From the deductions \(\Xi'_{i,j}\) and \(\Xi''_{i,j}\), by Lemma D.1:

\[
\Xi''_{i,j} : \Gamma_c \triangleright e_i, \text{let rec } f_i = e_i \text{ in } f / f_i : \tau_{i,j} \Rightarrow m_{i,j}, \text{let rec } g_{i,j} = m_{i,j} \text{ in } g_{i,j} / g_{i,j}
\]

By induction on \(\Pi_{i,j}\) and \(\Xi''_{i,j}\), we have (for each \(i\) and \(j\)):

- \(\Theta_{i,j} : \rho' \triangleright m_{i,j}, \text{let rec } g_{i,j} = m_{i,j} \text{ in } g_{i,j} / g_{i,j} \rightarrow v_{i,j}\)

- (1) \(\emptyset \triangleright v_i : \tau_{i,j} \Rightarrow v_{i,j}\)

From \(\rho : \Gamma_c \Rightarrow \rho'\) and (1), by Definition 19,

\[
\rho \{ f_i \mapsto v_i \}; \Gamma_c \{ f_i : \{ g_{i,j}^{\tau_{i,j}} \} \} \Rightarrow \rho' \{ g_{i,j} \mapsto v_{i,j} \}
\]

Since w.f.(\(\Gamma_c\)), w.f.\(\tau_{i,j}\) and \(\| \tau_{i,j} \| = \tau_i\), by Definition 18, \(\Gamma_c \{ f_i : \{ g_{i,j}^{\tau_{i,j}} \} \}\) is well-formed. By induction on \(\Xi'\) and \(\Pi'\), we have:
5. \( \Xi :: \)

\[
\begin{align*}
\Xi_1 & \quad \Xi_2 \\
\emptyset ; \Gamma_c \triangleright e_1 : \tau' \quad \Gamma_c \triangleright e_2 : \tau & \Rightarrow \Gamma_c \triangleright [e_1, m_1] \\
\emptyset ; \Gamma_c \triangleright e_1 \otimes e_2 & \Rightarrow \Gamma_c \triangleright [m_1] @ [m_2]
\end{align*}
\]

\( \Pi :: \)

\[
\begin{align*}
\Pi_1 & \quad \Pi_2 & \quad \Pi_3 \\
\rho \triangleright e_1 & \Rightarrow [\rho_1, \lambda x. e] & \rho \triangleright e_2 & \Rightarrow v_2 & \rho_1 [x \mapsto v_2] \triangleright e & \Rightarrow v \\
\rho \triangleright e_1 \otimes e_2 & \Rightarrow v
\end{align*}
\]

By induction on \( \Xi_1 \) and \( \Pi_1 \), we have:

- \( \Theta_1 :: \rho' \triangleright m_1 \Rightarrow v_1' \)

- (1) \( \emptyset \triangleright [\rho_1, \lambda x. e] : \tau' \quad \Gamma_c \triangleright \tau \Rightarrow v_1' \)

By (1) and Definition 20, \( v_1' = [\rho_1', m] \) and there exists a context \( \Gamma_c' \) such that

- (2) \( \rho_1 : \Gamma_c' \Rightarrow \rho_1', \) and

- \( \Xi_3 :: \emptyset ; \Gamma_c' \triangleright \lambda x. e : \tau' \quad \Gamma_c \triangleright \tau \Rightarrow m \).
\( \Xi_3 \) has the form:

\[
\Xi'_3 \\
\vdash \Gamma' \{ x : \{ x' \} \} \Downarrow e : \tau \Rightarrow m'
\]

\[
\vdash \Gamma' \Downarrow \lambda x. e : \tau' \Downarrow \varepsilon \Rightarrow \lambda [x]. m'
\]

Observe \( m = \lambda [x]. m' \). By induction on \( \Xi_2 \) and \( \Pi_2 \) we have:

- \( \Theta_2 :: \rho' \Downarrow m_2 \leftrightarrow v'_2 \)
- \( (3) \quad \Downarrow v_2 : \tau \Rightarrow v'_2 \)

From (2) and (3), by Definition 19,

\[
\rho'_1 \{ x \mapsto v_2 \} : \Gamma' \{ x : \{ x' \} \} \Rightarrow \rho'_1 \{ x \mapsto v'_2 \}
\]

By induction on \( \Xi'_3 \) and \( \Pi_3 \) we have:

- \( \Theta_3 :: \rho'_1 \{ x \mapsto v'_2 \} \Downarrow m' \leftrightarrow v' \)
- \( \Downarrow v : \tau \Rightarrow v' \)

We can construct the required deduction:

\[
\Theta_1 \quad \Theta_2 \quad \Theta_3 \\
\rho' \Downarrow m_1 \leftrightarrow [\rho'_1, \lambda [x]. m'] \quad \rho' \Downarrow m_2 \leftrightarrow v'_2 \quad \rho'_1 \{ x \mapsto v'_2 \} \Downarrow m' \leftrightarrow v' \\
\frac{\rho' \Downarrow m_1 \uplus [m_2] \leftrightarrow v'}{\rho' \Downarrow m_1 \uplus [m_2] \leftrightarrow v'}
\]

6. \( \Xi :: \)

\[
\begin{align*}
\Xi_1 \\
\vdash \Gamma_c \Downarrow e_1 : \tau' \Downarrow [\tau_1, \ldots, \tau_n] \Rightarrow \langle \langle m_0, \ldots, m_n \rangle \rangle \\
\vdash \Gamma_c \Downarrow e_2 : \tau' \Rightarrow m \\
\vdash \Gamma_c \Downarrow e_1 \uplus_1 e_2 : \tau \Rightarrow m_0 \uplus [m_1, \ldots, m_n] \\
\end{align*}
\]
\[ \Pi : \]
\[
\begin{array}{c}
\Pi_1 \\
\rho \triangleright e_1 \mapsto [\rho_1, \lambda x . e] \\
\Pi_2 \\
\rho \triangleright e_2 \mapsto v_2 \\
\Pi_3 \\
\rho_1 \{ x \mapsto v_2 \} \triangleright e \mapsto v \\
\end{array}
\]

By Induction on \( \Xi_3 \) and \( \Pi_1 \) we have:

- \( \Theta : \rho \triangleright \langle \langle m_0, \ldots, m_n \rangle \rangle \mapsto v'' \)

- (1) \( \langle \rangle \triangleright [\rho_1, \lambda x . e] : \tau' [\tau_1 \ldots, \tau_n] \mapsto \tau k \mapsto v'' \)

By (1) and Definition 20,

- \( v'' = \langle \langle \rho_1', \lambda[y_1, \ldots, y_n, x_1, \ldots, x_k].m' \rangle \rangle \mapsto v' \ldots v' \)

- (2) \( \rho_1 : \Gamma' c \mapsto \rho_1 \{ y_1 \mapsto v' \} \cdots \{ y_n \mapsto v' \} \)

- \( \Xi_3 : [\tau_1, \ldots, \tau_n]; \Gamma' c \triangleright \lambda x . e : \tau' [\tau_1 \ldots, \tau_n] \mapsto \lambda[x_1, \ldots, x_k].m' \)

Observe, by (\( \text{abs}_{\tau} \)), \( k = 1 \). The deduction \( \Xi_3 \) has the form:

\[
\begin{array}{c}
\Xi' \\
\langle \rangle ; \Gamma' \{ x : \{x' \} \} \triangleright e : \tau \mapsto m' \\
\end{array}
\]

\[
[\tau_1, \ldots, \tau_n]; \Gamma' c \triangleright \lambda x . e : \tau' [\tau_1 \ldots, \tau_n] \mapsto \lambda[x].m'
\]

The deduction \( \Theta \) has the form: \(^1\)

\[
\begin{array}{c}
\Theta_0 \\
\rho \triangleright m_0 \mapsto [\rho_1', \lambda[y_1, \ldots, y_n, x].m'] \\
\end{array}
\]

\[
\begin{array}{c}
\Theta_i \\
\rho \triangleright m_i \mapsto v'_i \\
\end{array}
\]

\[
\rho \triangleright \langle \langle m_0, \ldots, m_n \rangle \langle \rangle \mapsto \langle \langle [\rho_1', \lambda[y_1, \ldots, y_n, x].m'], v'_1, \ldots, v'_n \rangle \rangle
\]

\(^1\)This can be derived from the two rules for left-associative tuples in Figure 5.2.
By Induction on \( \Xi_2 \) and \( \Pi_2 \), we have:

- \( \Theta' :: \rho' \downarrow m \Rightarrow v''' \)
- \( (3) \ [\downarrow v_2 : \tau' \Rightarrow v''' \)

From (2) and (3), by Definition 19,

\[
\rho_1 \{ x \mapsto v_2 \} : \Gamma' \{ x : \{ x \mapsto v' \} \} \Rightarrow \rho_1 \{ y_1 \mapsto v' \} \cdots \{ y_n \mapsto v' \} \{ x \mapsto v''' \}
\]

By Induction on \( \Xi' \) and \( \Pi_3 \), we have:

- \( \Theta'' :: \rho''_1 \{ y_1 \mapsto v' \} \cdots \{ y_n \mapsto v' \} \{ x \mapsto v''' \} \downarrow \rho' \downarrow v' \)
- \( [\downarrow v : \tau \Rightarrow v' \)

We can construct the required deduction (where \( \rho''_1 = \rho_1 \{ y_1 \mapsto v' \} \cdots \{ y_n \mapsto v' \} \{ x \mapsto v''' \} \downarrow \rho' \downarrow v' \)):

\[
\begin{array}{c}
\Theta_0 \\
\Theta_i \\
\Theta' \\
\Theta'' \\

\rho' \downarrow m_0 \mapsto [\rho', \lambda [y_1, \ldots, y_n, x], m'] \quad \rho' \downarrow m_i \mapsto v' \\
\rho' \downarrow m \mapsto v''' \\
\rho'' \downarrow m' \mapsto v'
\end{array}
\]

\[
\rho' \downarrow m_0 \mapsto [m_1, \ldots, m_n, m] \mapsto v'
\]

7. \( \Xi :: \)

\[
[\downarrow \Gamma \mapsto e_1 : \tau [\tau_1, \ldots, \tau_n] \mapsto m_1 \mapsto m_2]
\]

\[
\begin{array}{c}
\Xi_1 \\
\Xi_2
\end{array}
\]

\[
[\downarrow \Gamma \mapsto e_1 \circ e_2 : \tau \mapsto m_1 \circ m_2]
\]

\[
\Pi ::
\]

\[
\begin{array}{c}
\Pi_1 \\
\Pi_2 \\
\Pi_3
\end{array}
\]

\[
\begin{array}{c}
\rho \mapsto e_1 \mapsto [\rho_1, \lambda x.e] \\
\rho \mapsto e_2 \mapsto v_2 \\
\rho \mapsto x \mapsto v_2 \mapsto e \mapsto v
\end{array}
\]

\[
\rho \mapsto e_1 \circ e_2 \mapsto v
\]
By Induction on $\Xi_1$ and $\Pi_1$, and on $\Xi_2$ and $\Pi_2$, we have:

- $\Theta_1 :: \rho_1 \triangleright m_1 \rightarrow v_1'$
- (1) \[ \frac{}{[\rho_1, \lambda x. e] : \tau \xrightarrow{[\tau_1 \rightarrow \cdots \rightarrow \tau_n]} v} \epsilon \tau \Rightarrow v_1' \]
- $\Theta_2 :: \rho_2 \triangleright m_2 \rightarrow v_2'$
- (2) \[ \frac{}{v_2 : \tau \Rightarrow v_2'} \]

From (1) and Definition 20,

- $v_1' = \langle \langle \rho_1, \lambda [y_1, \ldots, y_n, x_1, \ldots, x_k].m, v'_1, \ldots, v''_n \rangle \rangle$
- (3) $\rho_1 : \Gamma' \Rightarrow \rho_1 \{ y_1 \mapsto v_1'' \} \cdots \{ y_n \mapsto v''_n \}$, where w.f.$(\Gamma')$
- $\Xi_3 :: [\tau_1, \ldots, \tau_n], \Gamma' \Rightarrow \lambda x. e : \tau' \xrightarrow{[\tau_1 \rightarrow \cdots \rightarrow \tau_n]} v \epsilon \tau \Rightarrow \lambda [x_1, \ldots, x_k].m$

$\Xi_3$ has the form (observe $k = 1$):

$$\Xi'_3 \quad \frac{[] ; \Gamma' \{ x : \{ x' \} \} \triangleright e : \tau \Rightarrow m \quad \text{w.f.(}\tau')}{\frac{[\tau_1, \ldots, \tau_n], \Gamma' \Rightarrow \lambda x. e : \tau' \xrightarrow{[\tau_1 \rightarrow \cdots \rightarrow \tau_n]} v \epsilon \tau \Rightarrow \lambda [x].m}{\frac{[\tau_1, \ldots, \tau_n], \Gamma' \triangleright \lambda x. e : \tau' \xrightarrow{[\tau_1 \rightarrow \cdots \rightarrow \tau_n]} v \epsilon \tau \Rightarrow \lambda [x].m}}$$

Since w.f.$(\Gamma')$ and w.f.$(\tau')$, by Definition 18, $\Gamma'_c \{ x : \{ x' \} \}$ is well-formed. Thus, from (2), (3) and Definition 19, we have:

$$\rho_1 \{ x \mapsto v_2 \} : \Gamma'_c \{ x : \{ x' \} \} \Rightarrow \rho_1 \{ y_1 \mapsto v''_1 \} \cdots \{ y_n \mapsto v''_n \} \{ x \mapsto v' \}$$

Thus by Induction on $\Xi'_3$ and $\Pi'_3$, we have:
\( \Theta_3 :: \rho' \{ y_1 \mapsto v''_1 \} \ldots \{ y_n \mapsto v''_n \} \{ x \mapsto v'_1 \} \triangleright m \mapsto v' \)

\( \| \triangleright v_1 : \tau \Rightarrow v' \)

We can construct the required deduction:

\[
\begin{array}{c}
\Theta_1 \\
\rho' \triangleright m_1 \mapsto \langle \langle \langle \rho', \lambda y_1, \ldots, y_n, x \mid m, v''_1, \ldots, v''_n \rangle \rangle \rangle \\
\Theta_2 \quad \Theta_3 \\
\rho' \triangleright m_2 \mapsto \{ y_1 \mapsto v''_1 \} \{ y_1 \mapsto v''_1 \} \{ x \mapsto v'_1 \} \triangleright m \mapsto v' \\
\hline
\rho' \triangleright \underbrace{m_1 \circ m_2} \mapsto v'
\end{array}
\]

8. \( \Xi :: \)

\[
\begin{array}{c}
\Xi_1 \\
\| ; \Gamma \triangleright e_1 : \tau' \mapsto \tau \Rightarrow m_1 \quad \Xi_2 \\
\| ; \Gamma \triangleright e_2 : \tau' \Rightarrow m_2 \\
\| ; \Gamma \triangleright e_1 \circ e_2 : \tau \Rightarrow ((\cdot, m_1), m_2)
\end{array}
\]

\( \Pi :: \)

\[
\begin{array}{c}
\Pi_1 \\
\rho \triangleright e_1 \mapsto \langle \rho, \lambda x. e \rangle \\
\Pi_2 \\
\rho \triangleright e_2 \mapsto v_2 \\
\Pi_3 \\
\rho \triangleright \underbrace{e_1 \circ e_2} \mapsto e \mapsto v \\
\hline
\rho \triangleright e_1 \circ e_2 \mapsto v
\end{array}
\]

By Induction on \( \Xi_1 \) and \( \Pi_1 \), and on \( \Xi_2 \) and \( \Pi_2 \), we have:

- \( \Theta_1 :: \rho' \triangleright m_1 \mapsto v'_1 \)

- (1) \( \| \triangleright [\rho_1, \lambda x. e] : \tau' \mapsto \tau \Rightarrow v'_1 \)

- \( \Theta_2 :: \rho' \triangleright m_2 \mapsto v'_2 \)

- (2) \( \| \triangleright v_2 : \tau' \Rightarrow v'_2 \)
We can construct the required deduction:

\[
\begin{align*}
\Theta_1 & \quad \Theta_2 \\
\rho' \triangleright m_1 & \quad \rho_2' \triangleright \nu' \\
\rho' \triangleright m_2 & \quad \rho_2' \triangleright \nu' \\
\therefore \rho' \triangleright ((\nu', m_1), m_2) & \quad \nu' \triangleright ((\nu', m_1), m_2)
\end{align*}
\]

By (1) and Definition 20, we have:

- \( \nu' = [\rho', m] \)
- (3) \( \rho_1 : \Gamma' \triangleright \rho_1' \) where w.f.(\( \Gamma' \))
- \( \Xi_3 :: [\ell; \Gamma' \triangleright \lambda x.e : \tau' \triangleright \tau \triangleright \ell] \)

By \( \Xi_1 \), w.f.(\( \Gamma_c \)) and Lemma C.2, \( \tau' \triangleright \tau \triangleright \ell \) is well-formed. Thus, by Definition 17, \( \tau \) has the form \( \tau'' \triangleright \gamma \tau''' \).

\( \Xi_3 \) has the form:

\[
\Xi_3' \\
\begin{align*}
[\tau']; \Gamma' \{ x : \{ x' \} \} & \triangleright \lambda y.e' : \tau'' \triangleright \tau''' \\
\therefore [\ell'; \Gamma' \triangleright \lambda x.\lambda y.e' : \tau' \triangleright \tau'' \triangleright \tau''' \triangleright \ell] & \triangleright \lambda [y_1, \ldots, y_n].m' \\
\text{w.f.}(\tau')
\end{align*}
\]

Notice that:

- \( e = \lambda y.e' \)
- \( m = \lambda [x, y_1, \ldots, y_n].m' \)
\( \Pi_3 \) has the form:

\[
\rho_1 \{ x \mapsto v_2 \} \vdash \lambda y.e' \iff [\rho_1 \{ x \mapsto v_2 \}, \lambda y.e']
\]

Thus \( v = [\rho_1 \{ x \mapsto v_2 \}, \lambda y.e'] \).

Since w.f.(\( \Gamma'_c \)) and w.f.(\( \tau' \)), by Definition 18, \( \Gamma'_c \{ x : \{ x^\tau \} \} \) is well-formed. Thus, by (2), (3) and Definition 19,

\[
\rho_1 \{ x \mapsto v_2 \} \vdash \Gamma'_c \{ x : \{ x^\tau \} \} \Rightarrow \rho'_1 \{ x \mapsto v'_2 \}
\]

Thus, from (4), \( \Xi'_3 \) and Definition 20, we have:

\[
[] \vdash [\rho_1 \{ x \mapsto v_2 \}, \lambda y.e'] \vdash \tau''' \gamma \tau''' \Rightarrow ((\cdot, [\rho'_1, \lambda [x], y_1, \ldots, y_n], m'), v')
\]

9. \( \Xi :: \)

\[
\Xi_1 [\cdot] \Gamma_c \vdash e_1 : \tau \vdash \tau_1 \cdots \tau_n \Rightarrow \gamma \Rightarrow \tau \Gamma_c \vdash m_1 \]

\[
\Xi_2 [\cdot] \Gamma_c \vdash e_2 : \tau' \Rightarrow m_2
\]

\[
[\cdot] \Gamma_c \vdash e_1 \odot e_2 : \tau \Rightarrow (m_1, m_2)
\]

\( \Pi :: \)

\[
\Pi_1 \rho \vdash e_1 \iff [\rho_1, \lambda x.e] \quad \Pi_2 \rho \vdash e_2 \iff v_2 \quad \Pi_3 \rho_1 \{ x \mapsto v_2 \} \vdash e \iff v
\]

\[
\rho \vdash e_1 \odot e_2 \iff v
\]

By Induction on \( \Xi_1 \) and \( \Pi_1 \), and on \( \Xi_2 \) and \( \Pi_2 \), we have:

- \( \Theta_1 :: \rho' \vdash m_1 \iff v'_1 \)
• (1) \( \triangleright [\rho_1, \lambda x.e : \tau^1_1 \rightarrow \tau^1_n] \lessdot \tau \Rightarrow \tau'_1 \)

• \( \Theta_2 :: \rho' \triangleright m_2 \Rightarrow \tau'_2 \)

• (2) \( \triangleright v_2 : \tau' \Rightarrow \tau'_2 \)

We can construct the required deduction:

\[
\begin{array}{c}
\Theta_1 \\
\rho' \triangleright m_1 \Rightarrow \tau'_1 \\
\Theta_2 \\
\rho' \triangleright m_2 \Rightarrow \tau'_2 \\
\hline
\rho' \triangleright (m_1, m_2) \Rightarrow (\tau'_1, \tau'_2) \\
\end{array}
\]

By (1) and Definition 20, we have:

• \( \tau'_1 = \langle \langle \rho'_1, \lambda[y_1, \ldots, y_n, x_1, \ldots, x_k].m \rangle, \tau''_1, \ldots, \tau''_n \rangle \)

• (3) \( \rho_1 : \Gamma' \vdash \rho'_1 \{ y_1 \mapsto \tau''_1 \} \cdots \{ y_n \mapsto \tau''_n \} \), where w.f.(\( \Gamma' \))

• \( \Xi_3 :: [\tau_1, \ldots, \tau_n] ; \Gamma' \vdash \lambda x.e : \tau' [\tau_1 \rightarrow \tau_n] \lessdot \tau \Rightarrow \lambda[x_1, \ldots, x_k].m \)

By \( \Xi_1 \), w.f.(\( \Gamma' \)) and Lemma C.2, \( \tau' [\tau_1 \rightarrow \tau_n] \lessdot \tau \) is well-formed. Thus, by Definition 17, \( \tau \) has the form \( \tau'' [\tau_1 \rightarrow \tau_n] \lessdot \tau''' \).

\( \Xi_3 \) has the form (where \( x_1 \) is \( x \)):

\[
\Xi'_3 \\
[\tau_1, \ldots, \tau_n, \tau'] ; \Gamma' \{ x : \{ x' \} \} \vdash \lambda y.e' : \tau'' [\tau_1 \rightarrow \tau_n] \lessdot \tau''' \Rightarrow \lambda[x_2, \ldots, x_k].m \ w.f.(\tau')
\]

\[
[\tau_1, \ldots, \tau_n] ; \Gamma_c \vdash \lambda x.\lambda y.e' : \tau' [\tau_1 \rightarrow \tau_n] \lessdot \tau'' [\tau_1 \rightarrow \tau_n] \lessdot \tau''' \Rightarrow \lambda[x, x_2, \ldots, x_k].m
\]
Notice that $e = \lambda y.e'$. $\Pi_3$ has the form:

$$\rho_1 \{x \mapsto v_2\} \triangleright \lambda y.e' \mapsto [\rho_1 \{x \mapsto v_2\}, \lambda y.e']$$

Thus $v = [\rho_1 \{x \mapsto v_2\}, \lambda y.e']$.

Since w.f.($\Gamma'_{\tau_c}$) and w.f.($\tau'$), by Definition 18, $\Gamma'_{\tau_c} \{x : \{x \mapsto \tau\}\}$ is well-formed. Thus by (2), (3) and Definition 19,

$$\rho_1 \{x \mapsto v_2\} : \Gamma'_{\tau_c} \{x : \{x \mapsto \tau\}\} \Rightarrow \rho_1 \{y_1 \mapsto v''_1\} \ldots \{y_n \mapsto v''_n\} \{x \mapsto v'\}$$

Thus from $\Xi'$ and Definition 20, we have:

$$[\cdot] \triangleright [\rho_1 \{x \mapsto v_2\}, \lambda y.e'] : \tau' \xrightarrow{\tau_1, \ldots, \tau, \tau'} \gamma \xrightarrow{\tau''} \Rightarrow \langle [\rho_1', \lambda[y_1, \ldots, y_n, x, x, \ldots, x, k, m], v''_1, \ldots, v''_n, v'_{2}] \rangle$$

10. $\Xi'::$

$$\Delta + \tau_1; \Gamma \triangleright \lambda x.e : \tau_2 \Rightarrow \lambda[x_1, \ldots, x_n].m \quad \text{fresh}(y)$$

$$\Delta; \Gamma \triangleright \lambda x.e : \tau \xrightarrow{\Delta} \tau_2 \Rightarrow \lambda[y, x_1, \ldots, x_n].m$$

$\Pi'::$

$$\rho \triangleright \lambda x.e \mapsto [\rho, \lambda x.e]$$

We can construct:

$$\rho' \triangleright \lambda[y, x_1, \ldots, x_n].m \mapsto [\rho', \lambda[y, x_1, \ldots, x_n].m]$$
From \( \rho : \Gamma_c \Rightarrow \rho' \) and \( \Xi \), by Definition 20 we have:

\[
\Delta \triangleright [\rho, \lambda x.e] : \tau \xrightarrow{u} \tau_1 \xrightarrow{\varsigma} \tau_2 \Rightarrow [\rho', \lambda y, x_1, \ldots, x_n].m
\]

11. \( \Xi' :: \)

\[
\frac{\Xi'}{[] : \Gamma \triangleright e : \tau \xrightarrow{u} \tau' \Rightarrow m}
\]

\[
\frac{\Xi'}{[] : \Gamma \triangleright e : \tau' \Rightarrow \langle m, d^{|(\langle \tau \rangle |)} \rangle}
\]

\( \Pi :: \rho \triangleright e \leftrightarrow v \)

By induction on \( \Xi' \) and \( \Pi \), we have:

- \( \Theta :: \rho' \triangleright m \leftrightarrow v' \)
- \( (1) \quad [] \triangleright v : \tau \xrightarrow{u} \tau' \Rightarrow v' \)

We can construct the deduction:

\[
\frac{\Theta}{\rho' \triangleright m \leftrightarrow v'} \frac{\rho' \triangleright d^{|(\langle \tau \rangle |)} \leftrightarrow d}{\rho' \triangleright \langle m, d^{|(\langle \tau \rangle |)} \rangle \leftrightarrow \langle v', d \rangle}
\]

From (1) and Definition 20, \( v = [\rho_1, \lambda x.e'] \) and \( v' = [\rho_1', m'] \) and we have (for some well-formed \( \Gamma' \)):

- \( \Xi_1 :: [] : \Gamma' \triangleright \lambda x.e' : \tau \xrightarrow{u} \tau' \Rightarrow m' \)
- \( (2) \quad \rho_1 : \Gamma' \Rightarrow \rho_1' \)
$\Xi_1$ cannot have the form:

$$\Xi''_1$$

$$\frac{\top; \Gamma' \triangleright \lambda \beta \cdot e : \tau \frac{\top}{\varsigma} \tau \frac{\top}{\varsigma} \tau' \Rightarrow m_1}{\top; \Gamma' \triangleright \lambda \beta \cdot e : \tau \frac{\top}{\varsigma} \tau \frac{\top}{\varsigma} \tau' \Rightarrow \langle m_1, d(\tau_1) \rangle}$$

The type $\tau \frac{\top}{\varsigma} \tau \frac{\top}{\varsigma} \tau'$ is not well-formed. By Lemma C.2, there is no deduction $\Xi''_1$.

Likewise, $\Xi_1$ cannot have the form:

$$\Xi'''_1$$

$$\frac{\top; \Gamma' \triangleright \lambda \beta \cdot e : \tau \frac{\top}{\varsigma} \tau \frac{\top}{\varsigma} \tau' \Rightarrow m_2}{\top; \Gamma' \triangleright \lambda \beta \cdot e : \tau \frac{\top}{\varsigma} \tau \frac{\top}{\varsigma} \tau' \Rightarrow \langle m_2, d(\tau_1) \rangle}$$

The type $\tau \frac{\top}{\varsigma} \tau \frac{\top}{\varsigma} \tau'$ is not well-formed. By Lemma C.2, there is no deduction $\Xi'''_1$.

Thus $\Xi_1$ must have the form:

$$\Xi'_1$$

$$\frac{[\tau]; \Gamma' \triangleright \lambda \beta \cdot e : \tau \frac{\top}{\varsigma} \tau \frac{\top}{\varsigma} \tau' \Rightarrow \lambda[x_1, \ldots, x_n].m'' \text{ fresh } (y)}{\top; \Gamma' \triangleright \lambda \beta \cdot e : \tau \frac{\top}{\varsigma} \tau \frac{\top}{\varsigma} \tau' \Rightarrow \lambda[y, x_1, \ldots, x_n].m''}$$

$\Xi'_1$ can be an instance of $\text{abs } e$, $\text{abs } \varsigma$, or $\text{uvabs}$. In all three cases, $\tau'$ has the form $\tau' \frac{[\tau]}{\top} \gamma_1 \gamma_2$. Since $y$ is a fresh variable, $y \notin \text{dom}(\tau')$. From (2) and Definition 19,
\( y \notin \text{dom}(\rho_1) \) and we have:

\[
(3) \quad \rho_1 : \Gamma \vdash \rho_1 \{ y \mapsto d \}
\]

From (3) and \( \Xi' \), by Definition 20, we have:

\[
\emptyset \triangleright [\rho_1, \lambda x.e'] : \tau' \Rightarrow \langle\langle [\rho_1', \lambda [y, x_1, \ldots, x_n].m'], d \rangle\rangle
\]

12. \( \Xi' ::
\[
\begin{array}{c}
\Xi' \\
\hline
\emptyset; \Gamma \triangleright e : \tau^u [\tau_1 \rightarrow \cdots \tau_n] \quad \langle \tau' \Rightarrow m \rangle \\
\hline
\emptyset; \Gamma \triangleright e : \tau' \Rightarrow (m, d')
\end{array}
\]

\( \Pi :: \rho \triangleright e \mapsto v \)

By Induction on \( \Xi' \) and \( \Pi \), we have:

- \( \Theta_1 :: \rho_1' \triangleright m \mapsto v' \)
- \( (1) \quad \emptyset \triangleright v : \tau^u [\tau_1 \rightarrow \cdots \tau_n] \quad \langle \tau' \Rightarrow v' \rangle \)

From (1) and Definition 20,

- \( v = [\rho_1, \lambda x.e'] \)
- \( v' = \langle\langle [\rho_1', \lambda [y_1, \ldots, y_n, x_1, \ldots, x_k].m'], v_1', \ldots, v_n' \rangle\rangle \)
- \( (2) \quad \rho_1 : \Gamma \vdash [\rho_1, \lambda [y_1, \ldots, y_n, x_1, \ldots, x_k].m'], \{ y_1 \mapsto v' \}, \ldots, \{ y_n \mapsto v' \}, \text{where w.f.(}\Gamma'\text{)} \)
- \( \Xi_1 :: [\tau_1, \ldots, \tau_n]; \Gamma' \triangleright \lambda x.e' : \tau^u [\tau_1 \rightarrow \cdots \tau_n] \quad \langle \tau' \Rightarrow \lambda [x_1, \ldots, x_k].m' \rangle \)
We can construct the required deduction:

\[
\begin{align*}
\Theta_1 \\
\rho' \triangleright m &\mapsto (\langle [\rho'_1, \lambda[y_1, \ldots, y_n, x_1, \ldots, x_k], m'_1, v'_1, \ldots, v'_n] \rangle, d) \\
\rho' &\triangleright (m, d') \mapsto (\langle [\rho'_1, \lambda[y_1, \ldots, y_n, x_1, \ldots, x_k], m'_1, v'_1, \ldots, v'_n] \rangle, d) \\
\rho' &\triangleright d' \mapsto d
\end{align*}
\]

From \( \Xi' \) and Lemma C.2, \( \tau'_{\overrightarrow{[t_1 \ldots t_n]}, c} \) \( \iota' \) is well-formed. Thus by Definition 17,

\( \tau' = \tau''_{\overrightarrow{[t_1 \ldots t_n]}, c} \gamma \tau''' \).

\( \Xi_1 \) has the form:

\[
\begin{align*}
\Xi'_{\overrightarrow{[t_1 \ldots t_n]}, c} &\triangleright \lambda x.e' : \tau''_{\overrightarrow{[t_1 \ldots t_n]}, c} \gamma \tau''' \Rightarrow \lambda [x_2, \ldots, x_k, m'_1 \text{ fresh}(x_1)] \\
\tau'_{\overrightarrow{[t_1 \ldots t_n]}, c} &\triangleright \lambda x.e' : \tau_{\overrightarrow{[t_1 \ldots t_n]}, c} \gamma \tau_{\overrightarrow{[t_1 \ldots t_n]}, c} \Rightarrow \lambda [x_1, \ldots, x_k, m]
\end{align*}
\]

Since \( x_1 \) is a fresh variable, \( x_1 \notin \text{dom}(\Gamma'_c) \). Thus from (2) and Definition 19, \( x_1 \notin \text{dom}(\rho_1) \) and we have:

\[
\rho'_1 : \Gamma'_c \triangleright \rho'_1 \{ y_1 \mapsto v'_1 \} \cdots \{ y_n \mapsto v'_n \} \{ x_1 \mapsto d \}
\]

Thus from \( \Xi'_1 \) and Definition 20,

\[
\llbracket \rho_1, \lambda x.e' \rrbracket : \tau''_{\overrightarrow{[t_1 \ldots t_n]}, c} \gamma \tau''' \Rightarrow \langle [\rho'_1, \lambda [y_1, \ldots, y_n, x_1, \ldots, x_k], m'_1, v'_1, \ldots, v'_n, d] \rangle
\]

\[\square\]
The proof of Part 2 is by induction on the deductions $\Xi :: \Delta; \Gamma_c \Rightarrow e : \tau \Rightarrow m$ and $\Theta :: \rho' \Rightarrow m \leftarrow v'$. The proof is similar to the proof of Part 1, so I only include two cases here:

1. $\Xi ::$

\[
\begin{align*}
\rho'(y) &\in \Gamma_c(x) \\
\therefore \Gamma_c \Rightarrow x : \tau \Rightarrow y
\end{align*}
\]

$\Theta ::$

\[
\begin{align*}
\rho'(y) &= v' \\
\therefore \rho' \Rightarrow y \leftarrow v'
\end{align*}
\]

Since $\rho : \Gamma_c \Rightarrow \rho', y \in \Gamma_c(x)$ and $\rho'(y) = v$, by Definition 19 we have:

- $\rho(x) = v$
- $\models v : \tau \Rightarrow v'$

We can construct the required deduction:

\[
\begin{align*}
\rho(x) &= v \\
\therefore \rho \Rightarrow x \leftarrow v
\end{align*}
\]

2. $\Xi ::$

\[
\begin{align*}
\Xi_1 &\models \Gamma_c \Rightarrow e_1 : \tau' \Rightarrow \tau \Rightarrow m_1 \\
\Xi_2 &\models \Gamma_c \Rightarrow e_2 : \tau' \Rightarrow m_2 \\
\therefore \Xi &\models \Gamma_c \Rightarrow e_1 \circ e_2 : \tau \Rightarrow m_1 \circ [m_2]
\end{align*}
\]

$\Theta ::$

\[
\begin{align*}
\Theta_1 &\models \rho' \Rightarrow m_1 \leftarrow \rho(x).m \\
\Theta_2 &\models \rho' \Rightarrow m_2 \leftarrow v' \\
\Theta_3 &\models \rho' \{x \mapsto v'\} \Rightarrow m \leftarrow v' \\
\therefore \rho' \Rightarrow m_1 \circ [m_2] \leftarrow v'
\end{align*}
\]
By Induction on $\Xi_1$ and $\Theta_1$, and on $\Xi_2$ and $\Theta_2$, we have:

- $\Pi_1 :: \rho \triangleright e_1 \rightarrow v_1$
- (1) $\boxed{\varepsilon} \triangleright v_1 : \tau \triangleright \tau \Rightarrow [\rho_1', \lambda x].m$
- $\Pi_2 :: \rho \triangleright e_2 \rightarrow v_2$
- (2) $\boxed{\varepsilon} \triangleright v_2 : \tau \Rightarrow v_2'$

From (1) and Definition 20, we have:

- $v_1 = [\rho_1', \lambda x.e]$
- $\Xi_3 :: \boxed{\varepsilon}; \Gamma' \triangleright \lambda x.e : \tau \triangleright \tau \Rightarrow \lambda [x].m$
- (3) $\rho_1 : \Gamma' \Rightarrow \rho_1'$, where w.f.(\Gamma')

$\Xi_3$ has the form:

$$
\Xi' \frac{\boxed{\varepsilon}; \Gamma' \{ x : \{ x' \} \} \triangleright e : \tau \Rightarrow m}{\boxed{\varepsilon}; \Gamma' \triangleright \lambda x.e : \tau \triangleright \lambda [x].m}
$$

From (2) and (3), by Definition 19,

$$
\rho_1 \{ x \mapsto v_2 \} : \Gamma' \{ x : \{ x' \} \} \Rightarrow \rho_1' \{ x \mapsto v'_2 \}
$$

Thus by Induction on $\Xi'_3$ and $\Theta_3$, we have:

- $\Pi_3 :: \rho_1 \{ x \mapsto v_2 \} \triangleright e \leftarrow v$
- $\boxed{\varepsilon} \triangleright v : \tau \Rightarrow v'$
We can construct:

\[
\begin{array}{c}
\Pi_1 \\
\rho \triangleright e_1 \mapsto [\rho_1, \lambda x. e] \\
\Pi_2 \\
\rho \triangleright e_2 \mapsto v_2 \\
\Pi_3 \\
\rho_1 \{x \mapsto v_2\} \triangleright e \mapsto v \\
\hline
\rho \triangleright e_1 \mathbin{@} e_2 \mapsto v
\end{array}
\]
References


Vita

Joseph Adam Fischbach was born in Annapolis, Maryland in 1974. He earned his Bachelor of Arts degree with specialized honors in computer science from Drew University in 1997. He enrolled in the Ph.D. program at the Pennsylvania State University in the Fall of that year. During much of his graduate career, he was employed by Penn State as a teaching assistant and instructor, teaching introductory programming in Pascal and C++ to undergraduates. He is a member of the honor societies Phi Beta Kappa and Pi Mu Epsilon. He is also a member of the Association for Computing Machinery.