A Parametric Extension of Haskell's Type Classes

Kung Chen

Research Report YALEU/DCS/RR-1057
January 1995

This work was supported by the Advanced Research Project Agency and the Office of Naval Research under DARPA Contract N00014-91-J-4043.

Abstract

A Parametric Extension of Haskell's Type Classes

Kung Chen
Yale University
1994

Haskell's type classes permit the definition of overloaded operators in a rigorous and general manner that integrate well with the underlying Hindley-Milner type system. As a result, operators that are monomorphic in other typed languages can be given a more general type. Most notably missing in Haskell, however, are overloaded functions over container structures. Such overloaded functions are quite useful, but the current Haskell type system is not expressive enough to support them.

This thesis introduces the notion of parametric type classes and a new type system as a significant generalization of Haskell's type classes. A parametric type class is a class that has type parameters in addition to the placeholder variable which is always present in a class declaration. Haskell's type classes are special instances of parametric type classes with just a placeholder but no parameters. We show that this generalization is essential to representing container structures with overloaded data constructor and selector operations.

The underlying type system supporting our proposed generalization is is a version

of the Hindley-Milner type system, extended to include a form of constrained quantification. In the extended system, type classes act as constraints on the quantification and instantiation of type variables. This is achieved by putting class constraints on quantified type variables and adding a separate constraint inference sub-system to the standard type inference engine. We prove that the resulting type system is decidable, and provide an effective type inference algorithm to compute the principal types for well-typed terms.

The meaning of a program in our system is described by translating the program, based on its typing derivation, to a program defined in a language that includes constructs for manipulating overloading explicitly. We present a method for extending the type inference algorithm to perform the translation and prove that the resulting algorithm computes the principal translations for well-typed terms. Furthermore, since interpretations of well-typed expressions follow the typing derivations and an expression can be type-checked in more than one way, it is necessary to ensure the coherence of this translation scheme. This is accomplished by defining the corresponding coherence conditions.

A Parametric Extension of Haskell's Type Classes

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Kung Chen
November 1994

Acknowledgments

I would like to thank my advisors, Paul Hudak and Martin Odersky. Paul suggested the subject of this thesis and provided years of support, encouragement, and advice. Martin taught me many things about doing research. His thoughtful insights, enormous enthusiasm and perseverence, and careful guidance have been invaluable to this thesis. I consider myself very lucky to have been supervised by them.

The other members of my committee, Young-il Choo, Mark Jones, and Tobias Nipkow, were also very helpful. Young-il answered innumerable questions about logic and formal methods, and suggested several improvments over the notations used here. Mark read the thesis very carefully. His comments significantly improved its accuracy and presentation. Moreover, many of the results presented here are inspired by his own work on qualified types. Tobias pushed me to get the right induction hypothesis for Lemma 4.7 in our electronic communication conducted in Febuary 1992.

I owe a great deal to my officemates. Tom Blenko not only helped improve my defense presentation but also provided many constructive criticisms for my work. Dan Rabin was generous enough to let me use his abundant collection of books and conference proceedings. From them, I also learned a great deal about the U.S. and about Computer Science. Rajiv Mirani and Sheng Liang kept me informed of developments outside type theory. They all gave me many useful suggestions for imporving this document.

My gratitude also goes to the other members of Yale Haskell Group. Special thanks goes to Siau Cheng Khoo, who provided valuable advice on various issues, even after he had left Yale; to Amir Kishon, who shared his knowledge about type classes with me when I first started this project; to John Peterson, who showed me first-rate Lisp programming; to Chih-Ping Chen, who allocated much of his office space to store my collections of papers; and to Linda Joyce, who provided efficient adminstrative support.

Finally, I would like to thank my family and friends for their support. I am especially grateful to my parents and my sister for their love and encouragement. Most of all, I thank my wonderful wife, Lian-Chien. She patiently looked after me when I was busy. Her unfailing love and support are the main driving force for me to finish. I may never be able to repay her, but fortunately, I have a lifetime to try.

Contents

List of Figures

A Parametric Extension of Haskell's Type Classes

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Kung Chen
November 1994

Chapter 1

Introduction

1.1 Overview

The Hindley-Milner type system [Hindley, 1969, Milner, 1978, Damas and Milner, 1982] was a major advance in the development of static type systems for programming languages. It is one of the most significant aspect of the language ML [Milner *et al.*, 1990] and has been adopted by every other statically typed functional language, including Miranda[1] [Turner, 1985] and Haskell [Hudak *et al.*, 1990]. This is due, in a large part, to the reasonable balance it achieves among the following somewhat conflicting goals:

1. **Security**: A class of common programming errors such as passing arguments of incorrect type to a function are detected at compile time by the type system.

2. **Flexibility**: Functions may take arguments of arbitrary type if in fact the function does not depend on that type.

3. **Convenience**: An expression's type is determined by a type inference algorithm, instead of relying on explicit type declarations.

[1] "Miranda" is a trademark of Research Software, Ltd.

While convenient for many programs, there are also examples that cannot be described comfortably using the Hindley-Milner type system. For instance, what type should be assigned to the following function `double` which makes sense only for numeric arguments:

```
double x  =  x + x
```

In ML, the result is a type error, even though expressions such as `3+3` and `1.2+1.2` are permitted. In other words, while the arithmetic operator `+` is *overloaded* with values of type either `Int` or `Float`, function `double` is rejected due to insufficient information to determine which of the overloaded definitions for `+` is intended.

Haskell is a purely functional language designed to be a common standard for the non-strict functional programming community [Hudak *et al.*, 1990]. The most innovative feature of Haskell, its type system, extends the Hindley-Milner system with *type classes* [Wadler and Blott, 1989] to address the typing issues raised by overloaded functions. Type classes permit the definition of overloaded operators in a rigorous manner that integrates well with the underlying Hindley-Milner type system. As a result, functions like `double` can be given a more general type that indicates that they are defined over types with a `+` operation.

Despite this additional flexibility, type classes cannot overload functions over "container structures," such as lists and trees. For example, a program may manipulate collections of various kinds—lists, trees, arrays, etc. In this case, it is highly desirable to have an overloaded "member" function that tests whether an object belongs to any of these structures, as demonstrated by the following function:

```
scaleUp x c1 c2 =  if   (member x c1)
                   then    x + x
                   else if (member x c2)
                        then  x * x
                        else  x
```

Function `scaleUp` scales up its argument in different ways, depending on the membership tests. Ideally, different `member` functions will be invoked for different kinds of collections passed to `scaleUp`; however the current Haskell type system is not expressive enough to support this. As a result, the meaning of `member` in `scaleUp` has to be fixed, thereby restricting the kind of collection that `scaleUp` can accept as arguments.

This thesis explores ways to extend the Hindley-Milner type system to provide greater expressiveness for Haskell. We introduce the notion of *parametric type classes* as a significant generalization of Haskell's type classes. The underlying type system supporting our proposed generalization is a version of the Hindley-Milner type system, extended to include a form of *constrained quantification.* Basically, in the extended system, type classes act as constraints on the quantification and instantiation of type variables. This is achieved by putting class constraints on quantified type variables and adding a separate constraint inference sub-system to the standard type inference engine. Furthermore, we prove that any overloaded-operator ambiguity that may arise in the process can be easily detected.

The rest of this chapter briefly reviews the Hindley-Milner type system, describes the major features of type classes, motivates and explains our proposed extension, discusses related work, summarizes our contributions, and outlines the body of the thesis.

1.2 The Hindley-Milner Type System

Following its success in the language ML, the Hindley-Milner type system has become the basis of the type systems of many languages. The system was originally discovered by Roger Hindley [Hindley, 1969], developed independently by Robin Milner for ML [Milner, 1978] and subsequently elaborated in detail by Luis Damas [Damas and Milner, 1982, Damas, 1984]. Our discussion here is mainly based on [Damas and

Milner, 1982].

The type systems of strongly typed languages like Pascal are simple: they collect type declarations of variables and function parameters from programmers to perform some consistency check which will ensure that every program unit has a unique type. Such type systems are said to be *monomorphic* in the sense that every expression can have at most one type. In contrast, the Hindley-Milner type system allows *polymorphism*—some expressions can be treated as having many different types. In addition, the Hindley-Milner system allows *type inference* to eliminate the burden of type declarations borne by the programmer.

Polymorphism occurs naturally in many programming situations. For example, consider the Haskell function that computes the length of a list:

```
length []     =  0
length (x:xs) =  1 + length xs
```

It is clear that the calculation does not depend on the elements of the list (represented by variable `x` in the second line of the definition). Hence conceptually the same **length** function works regardless of the type of these elements. For example, **length** acts in the same way on lists of integers and lists of characters. In the Hindley-Milner type system, this is expressed by giving **length** a polymorphic type: for all types `a`, **length** is a function that takes a list of elements of type `a` and returns an integer. This is captured by the *type scheme* $\forall$`a.List a` $\rightarrow$ `Int`. The explicit universal quantification indicates that the choice of `a` is arbitrary and the types for **length** are obtained by instantiating `a` to different types. A large number of programs whose behavior is independent of the types of values in some components can be treated in a similar fashion.

The essential part of the type system is an inference engine that uses a set of natural-deduction [Prawitz, 1965] style inference rules. These rules define the typing discipline that forms the core of a language's static semantics. Moreover, being

presented in a highly structural manner, such type inference rules are also useful for programmers when reasoning about their programs. For example, the following rule specifies how to type function application:

$$(\rightarrow\text{-}elim)\quad \frac{A \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad A \vdash e_2 : \tau_1}{A \vdash e_1\ e_2 : \tau_2}$$

It can be read "Under some type assumption set A, if we can infer that e_1 has a functional type from τ_1 to τ_2 and that e_2 has type τ_1, then we may infer that the application of e_1 to e_2 has type τ_2." This rule captures the requirement that the type of an actual argument to a function must match that of its formal parameter. It is labelled as $(\rightarrow\text{-}elim)$ because the $\rightarrow$ in the hypothesis is eliminated. Dually, the next rule specifies how to type functions, resulting in the introduction of functional types:

$$(\rightarrow\text{-}intro)\quad \frac{A.x : \tau_1 \vdash e : \tau_2}{A \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

The reasoning of this rule is similar: Under some type assumption set A, if, by assuming that function parameter x has type τ_1, we can infer that the function body e has type τ_2, then we may infer that the function $\lambda x.e$ has type $\tau_1 \rightarrow \tau_2$. This duality of eliminating and introducing the functional type constructor, $\rightarrow$, shows the system's resemblance to natural deduction systems.

This system cannot be practical without an effective type inference algorithm to implement the inference rules. Milner [Milner, 1978] developed such an algorithm for the type system that allows programs without explicit type annotations. Furthermore, it was established in subsequent work [Damas and Milner, 1982, Damas, 1984] that the type computed by Milner's algorithm is the most general one that can be inferred for any typable expression. Such types are called *principal types.* An expression's or function's principal type is the least general type that, intuitively, "subsumes all types

the expression may assume." Principal types often take the form of type schemes. For example, the principal type of `length` is $\forall$`a.List a` $\rightarrow$ `Int`; the types $\forall$`a.a` $\rightarrow$ `Int` and $\forall$`a.a` are too general, whereas something like `List Char` $\rightarrow$ `Int` is too specific.

The existence of unique principal types that can be automatically inferred is the hallmark feature of the Hindley-Milner type system and the chief advantage that any proposed extension is expected to retain.

1.3 Type Classes

The kind of polymorphism supported by the Hindley-Milner system is commonly called *parametric* polymorphism following [Strachey, 1967]. Another kind of polymorphism, termed *ad hoc* polymorphism, is better known as *overloading.* An example is numerical operators, such as `+`, that are used on many different kinds of numbers. In general, ad-hoc polymorphism is necessary when the same variable name is used to denote many functions whose types are distinct, whereas in parametric polymorphism the same function is used uniformly in different type contexts. Type classes provide a structured way to blend these two forms of polymorphism. The idea was originally described in [Wadler and Blott, 1989], and subsequently adopted by the Haskell committee.

Overloading in monomorphic languages is easily resolved, since every expression has only one type. However, for languages supporting parametric polymorphism, where types may contain type variables, the situation is more complicated. It is conceivable to follow monomorphically typed languages by treating overloading as a purely syntactic device and insisting on compile-time resolution, yet such solutions are bound to be overly restrictive. To illustrate the kind of restrictions that can be imposed, consider the *equality* operator (`==` in Haskell). There are many types for which equality needs to be defined, but also some for which it should probably not be defined (e.g., functions). Furthermore, computing the equality of integers, for

example, is quite different from computing the equality of, say, lists (thus parametric polymorphism is ruled out). Now consider the definition of the function **member** which tests for membership in a list:

```
member x []       =  False
member x (y:ys)   =  if x==y then True else member x ys
```

There is not enough information in the second clause of the definition to determine which version of the == operator is intended. Any commitment to a particular type of equality would make member monomorphic, which is clearly too restrictive. Intuitively speaking, the type of member "ought" to be $\forall a.a \rightarrow$ List a $\rightarrow$ Bool. But this would imply that == has type $\forall a.a \rightarrow a \rightarrow$ Bool, even though we don't expect == to be defined for all types. Precisely, member is to be overloaded only on types that admit the equality test, and then different equality operations can be invoked when **member** is called with different types of lists.

Type classes solve both problems in a convenient and systematic way. For example, to overload equality, one uses the following *class* definition:

```
class  a::Eq  where
   (==)  :  a -> a -> Bool
```

Here Eq is the name of the class being defined, and == is the single operation in the class. This declaration may be read "a type a is an instance of the class **Eq** if there is an (overloaded) operation == of the appropriate type defined on it."

The constraint that a type a must be an instance of the class Eq is written **a::Eq.**[2] We also use this notation to extend the Hindley-Milner style type scheme to provide *constrained quantification.* For example, the class declaration above signifies assignment of the following principal type to ==:

[2]In Haskell, this is written as Eq a.

$$\forall a::Eq.\ a \rightarrow a \rightarrow Bool$$

This should be read, "For every type a that is an instance of the class **Eq**, **==** has type **a → a → Bool**." This is the type that would be used for **==** in the **member** example, and indeed the constraint imposed on **a** propagates to the principal type for **member**:

$$\forall a::Eq.a \rightarrow List\ a \rightarrow Bool$$

This is just what is desired—it expresses the fact that **member** is not defined on *all* types, just those for which it is possible to compare elements for equality.

We use a collection of *instance declarations* to specify which types are instances of the class **Eq**, along with the actual behavior of **==** on each of those types. For example:

```
instance  Int::Eq  where
  x == y   =  intEq x y
```

The definition of **==** is called a *method.* **intEq** happens to be the primitive function that compares integers for equality, but in general, any valid Boolean-valued expression is allowed on the right-hand side. The declaration specifies that **Int** is an instance of the class **Eq** as witnessed by the method for **==** on integers. Given this declaration, we can now compare integers for equality. Similarly, the instance declaration:

```
instance  Float::Eq  where
  x == y   =  floatEq x y
```

allow us to compare floating point numbers using **==**.

Polymorphic types such as lists can also be handled. Moreover, when needed, additional constraints on their constituent types can be added by supplying an appropriate *context* as part of the instance definition. Context constraints are put before the symbol `=>`:

```
instance  a::Eq => (List a)::Eq  where
   []     == []        =   True
   []     == (y:ys)    =   False
   (x:xs) == []        =   False
   (x:xs) == (y:ys)    =   (x == y) && (xs == ys)
```

The constraint `a::Eq` in the first line is necessary because the elements in the lists are compared for equality in the last line (`x==y`). The additional constraint is essentially saying that we can compare lists with members of type `a` for equality as long as we know how to compare values of type `a` for equality. If this constraint were omitted, a static type error would result.

1.4 The Problem

While convenient for many common overloaded functions, there are also many useful forms of overloading that cannot be expressed using type classes. More specifically, type classes cannot overload functions involving both container structures and their components. A typical example is the membership test mentioned in the beginning of this chapter that works on a container structure and the objects that may belong to the structure. Such overloaded operations are quite useful, but type classes are not expressive enough to support them.

To illustrate the problems that can occur, consider the concept of a *collection*: a container structure with the operations `insert`, `delete` and `member`. There are many possible implementations of collections: sorted or unsorted lists, arrays, binary search

trees and so on. Since each representation has its own merits, different representations may be appropriate for different settings. Moreover, multiple representations may co-exist under some situations. Thus, a programmer may not wish to commit to a particular representation when writing functions on collections. For example, consider the function `classify` defined as follows:

```
classify x p accept reject =
            if    (p x)
            then  (insert x accept, reject)
            else  (accept, insert x reject)
```

This function takes an object, a predicate and two collections as arguments, and inserts the object in one of the collections depending on whether or not it satisfies the predicate. In general, the two argument collections, `accept` and `reject`, can be collections of different kinds. If, in the definition of `classify`, a particular kind of collection, say lists, must be fixed for each collection, then several versions of `classify` will be required, one for each possible combination of representations used for the two collections. The obvious cure for this name-space pollution and duplicated code is overloading. In our context, that means specifying the notion of a collection as a type class:

```
class  k::Collection  where
   insert   :  a -> k -> k
   delete   :  a -> k -> k
   member   :  a -> k -> Bool
```

This defines three overloaded functions `insert`, `delete` and `member` on collections. With this declaration, we wish to get an overloaded `classify` that can be reused for different kinds of collections. But it simply does not capture the desired behavior.

To see the problem, consider the principal type for **insert** specified by the class declaration:

$$\forall \mathtt{k}::\mathtt{Collection}.\forall \mathtt{a}.\ \mathtt{a} \rightarrow \mathtt{k} \rightarrow \mathtt{k}$$

This type is too general—the intended relationship that **a** is the type of objects contained in structures typed by **k** is not expressed in the type at all. For instance, we may instantiate **a** to **Char** while substituting (`List Int`) for **k** and get the following invalid type instance for `insert` :

$$\mathtt{Char} \rightarrow \mathtt{List\ Int} \rightarrow \mathtt{List\ Int}$$

In general, when overloading functions over container structures using type classes, we need to be able to express the relationship between the type of the structure and the type(s) of the stored objects. A single type constrained by simple classes such as `class k::Collection where ...` is not expressive enough to achieve our goal, since the type **k** is treated as atomic—it cannot assume that **k** is composite, containing objects of some other type.

1.5 Parametric Type Classes

The solution that we propose is *parametric type classes.* Such classes can have type parameters in addition to the constrained type variable, and thus are able to express classes such as `Collection` discussed earlier. For instance, we can express the notion of a collection using a parametric type class `Collection` with one parameter:

```
class  k::Collection a  where
   insert  :  a -> k -> k
   delete  :  a -> k -> k
   member  :  a -> k -> Bool
```

The intended meaning is that the parameter a to the class Collection should be functionally dependent on the constrained type k. In other words, the type to which k is instantiated will uniquely determine that of a.

This dependence relationship is established in instance declarations through some simple static constraints. For example, the following instance declaration makes lists a kind of collection:

```
instance  a::Eq => (List a)::(Collection a)  where
        insert x xs          =  Cons x  xs
        delete x []          =  []
        delete x (Cons y ys) =  if x==y then ys else delete x ys
        member x []          =  False
        member x (y:ys)      =  if x==y then True else member x ys
```

This declaration may be read "If type a is an instance of class Eq, then List a is an instance of the (parametric) class Collection a as witnessed by the three collection methods on lists." Here the type argument to the class Collection is the constituent type a of the collection type List a, thereby establishing the dependence relationship intended in the class declaration. The constraint a::Eq indicates that the methods need to do the equality test on values of type a.

In general, all type variables occurring in the type argument to Collection should also occur in the constrained instance type. As a consequence, the problematic situation in instantiating insert's type described earlier is ruled out in our new system. For example, from the parametric type class declaration, we get the following principal type for insert:

$$\forall a.\forall k::\texttt{Collection}\ a.\ a \rightarrow k \rightarrow k$$

Now type a and type k are related through the parametric class Collection, and hence any instantiation of a has to be coordinated with that of k according to the

instance declarations

1.6 Related Work

Type classes were introduced by Wadler and Blott as an extension of the Hindley-Milner type system [Wadler and Blott, 1989]. They proposed a new type form, called a *predicated type*, to specify the types of overloaded functions. A quite similar notion was used under the name of *category* in the Scratchpad II system for symbolic computation [Jenks and Trager, 1981]. Kaes' work on parametric overloading [Kaes, 1988] provided an early treatment of extending ML-style polymorphism with function overloading. Also related are type-dependent parameter inference [Cormack and Wright, 1990] and safe run-time overloading [Rouaix, 1990].

The type class idea was quickly taken up in the design of Haskell. Its theoretical foundation, however, took some time to develop. The initial approach of [Wadler and Blott, 1989] encoded Haskell's source-level syntax in a type system that is more powerful than Haskell itself, since it could also encode predicates over arbitrary types. This increased expressiveness can, however, make typability undecidable, as shown in [Volpano and Smith, 1991]. More discussion about the decidability problem appears in [Lillibridge, 1992,Thatte, 1992].

The source-level syntax of Haskell, on the other hand, has a sufficient number of static constraints to guarantee decidability. Research on type systems closely following Haskell's syntax began in [Nipkow and Snelting, 1991], where the authors model type classes in a three-level system of values, types, and partially ordered sorts. In their system, classes correspond to sorts and types are sorted according the class hierarchy. Order-sorted unification [Meseguer *et al.*, 1989] is used to resolve overloading in type reconstruction. The use of an order-sorted approach is mathematically elegant, yet we have shown in [Chen *et al.*, 1992a] that the ordering relation between classes is a syntactic mechanism and thus not necessary for developing a type system for

type classes. Indeed, [Nipkow and Prehofer, 1993] have recently refined this system and presented, independently of our work, a new type system, which is essentially equivalent to ours when all classes are parameterless. Additionally, [Peyton Jones and Wadler, 1991] gives a static semantics for Haskell and [Hall *et al.*, 1994] presents a simplified version.

Work has also been done to extend the type class concept to *predicates* over multiple types. [Smith, 1991] has looked into extensions of the original system in [Wadler and Blott, 1989] to include subtyping with implicit coercions. [Jones, 1992b] gives a general framework for *qualified types* of which type classes, subtyping and extensible records are special instances. Unlike Wadler and Blott's system, his system separates overloading and instance information from the type assumption set and models them under a general predicate entailment system. As a result, the decidability problem of typability is reduced to that of the predicate entailment system. Our type system shares the idea of using a separate sub-system to handle overloading constraints; however, our sub-system is designed to address the problem of overloading functions over container structures by conservatively extending Haskell. [Kaes, 1992] also explored extensions of the Hindley-Milner type system to include overloading and subtypes. The resulting system is, however, very complicated even when restricted to overloading.

Jones in a recent paper [Jones, 1993] proposed another interesting generalization of type classes: *constructor classes*. By combining overloading with higher-order polymorphism, classes in this system can constrain type constructors as well as types. In particular, a constructor class of monads can be defined to support monad comprehensions [Wadler, 1990a], an extension of list comprehensions to other parameterized structures. While we cannot directly express this in our system, many instance declarations for parameterized types require constraints on their constituent types that are not permitted by constructor classes. For example, as discussed in Section 1.5, an additional constraint `a::Eq` is necessary for type `List a` to be an instance of

parametric class `Collect a`. Now if instead a constructor class `Collect` is used, the instance declaration can only refer to the type constructor `List`, and thus is unable to include the constraint on `a`.

The ambiguity problem in resolving overloaded operation is described in the Haskell Report [Hudak *et al.*, 1990]. [Blott, 1991] investigated this problem for a version of their original system [Wadler and Blott, 1989]. Our result for parametric type classes is based on [Jones, 1992a, Jones, 1994], where Jones developed a new technique for establishing conditions sufficient to ensure ambiguity-free resolution.

1.7 Summary of Contributions

We summarize the contributions of this thesis as follows:

- We introduce a significant generalization of Haskell's type classes. Parametric type classes can have type parameters in addition to the constrained type variable, and thus are able to express classes such as `Collection` defined earlier.
- To support our proposal, we develop a type system by extending, in a highly modular fashion, the Hindley-Milner type system to include constrained quantification. In particular:
  - we add to the standard type inference engine a separate constraint inference sub-system required by constrained quantification,
  - we propose a new unification algorithm that augments first-order unification with constraint propagation,
  - we provide an effective algorithm to reconstruct types for typable expressions,
  - we prove that the principal type property holds for the extended system.

- We provide a translation semantics to associate meanings with typed expressions, and prove that ambiguous uses of overloading can be detected at compile-time.
- Parametric type classes are a conservative extension of Haskell's type system: if all classes are parameterless, the two systems are equivalent.

1.8 Thesis Organization

Chapter 2 gives a more detailed description of parametric type classes in terms of a small example language. Chapter 3 shows how to do type inference in the presence of instance constraints. After explaining formally when a type is an instance of a class, two type inference systems are presented and shown to be equivalent. Chapter 4 presents unification and type reconstruction algorithms for computing the principal type for any typable expression. Chapter 5 presents a translation semantics to associate meanings with typed expressions. Conditions to detect overloaded-operator ambiguity are given and shown to be sufficient for the semantics in Chapter 6. Finally, Chapter 7 reviews our results and suggests possible directions for future research.

Chapter 2

Parametric Type Classes

As discussed in Chapter 1, a type class constrains only a single atomic type, and thus cannot express overloaded functions over parameterized types. To provide greater expressiveness, we propose to parameterize type classes. This chapter gives a more detailed description of parametric type classes in the context of a small example language, followed by an example and some discussion. We focus here on the definition of classes and instances. In subsequent chapters we will develop two type inference systems, a type reconstruction algorithm and a translation semantics for this language. Certain knowledge of type classes is assumed; readers are referred to the Haskell report [Hudak *et al.*, 1990] for this information.

2.1 Mini-Haskell$^+$

This section describes a small example language to illustrate our idea of parametric type classes. The language is a variant of Mini-Haskell [Nipkow and Snelting, 1991], augmented with parameterized type classes.

Type variables	α		
Type constructors	κ		
Types	τ	::=	$\alpha \mid () \mid (\tau_1, \tau_2) \mid \tau_1 \rightarrow \tau_2 \mid \kappa\,\tau$
Type schemes	σ	::=	$\tau \mid \forall\alpha{::}\Gamma\,.\,\sigma$
Class constructors	c		
Type classes	γ	::=	$c\,\tau$
Class sets	Γ	::=	$\{c_1\,\tau_1, \ldots, c_n\,\tau_n\}$ $(n \geq 0, c_i$ pairwise disjoint$)$

Figure 2.1: Abstract Syntax of Mini-Haskell$^+$ Types and Classes

2.1.1 Syntax of Types and Classes

Figure 2.1 gives the syntax of types and classes. A parametric type class γ in this syntax has the form $c\,\tau$, where c is a class constructor, corresponding to a class in Haskell, and τ is a type. To simplify the presentation, classes with several parameters are encoded using tuple types, e.g., $c\,(\alpha, \alpha')$. and parameterless classes are encoded using the unit type, e.g., `Eq()`. Nevertheless, we require that in a program, every class constructor has a fixed arity. We apply similar conventions to types: except for the functional type constructor $\rightarrow$ and the tuple type constructor $(,)$, all additional type constructors, denoted by κ, are unary, and application of a type constructor to a type is expressed by juxtaposition, e.g., `List (Int ())`. We will often omit the unit type $()$ in writing types and classes.

The definition of type schemes embodies the key device of our type system: *class-based constrained quantification*, in which quantified type variables may be instantiated with types that satisfy a set of class constraints. This notion of type schemes is obtained by extending the Hindley-Milner style type scheme with class sets Γ and with the instance relationship (between a type and a set of classes) denoted by $(::)$. For type scheme $\forall\alpha{::}\Gamma.\sigma$, the informal meaning is that α can only be instantiated to

types that are instances of *every* class in Γ. Due to the *consistency* requirement to be discussed in the following section, we require that any two classes in the same class set must have different class constructors.

As type classes may contain type variables now, the *order* of quantified type variables in such extended type schemes, unlike their counterparts in the Hindley-Milner system, is relevant. For example, the type scheme:

$$\forall \mathtt{a}.\forall \mathtt{k}::\{\mathtt{Collection\ a}\}.\ \mathtt{a} \rightarrow \mathtt{k} \rightarrow \mathtt{k}$$

is certainly different from:

$$\forall \mathtt{k}::\{\mathtt{Collection\ a}\}.\forall \mathtt{a}.\ \mathtt{a} \rightarrow \mathtt{k} \rightarrow \mathtt{k}$$

Indeed, by the standard bound-variable renaming convention, the latter type scheme is equivalent to:

$$\forall \mathtt{k}::\{\mathtt{Collection\ a}\}.\forall \mathtt{b}.\ \mathtt{b} \rightarrow \mathtt{k} \rightarrow \mathtt{k}$$

The set of type variables appearing free (bound) in an expression X is denoted by $\mathrm{tv}(X)$ ($\mathrm{btv}\ X$) and is defined in the obvious way. In particular, $\mathrm{tv}(\forall\alpha::\Gamma.\sigma) = (\mathrm{tv}\ \Gamma \cup \mathrm{tv}\ \sigma) \setminus \{\alpha\}$. A class γ often stands for a singleton class set $\{\gamma\}$. We use the notation $\forall\langle\alpha_i::\Gamma_i\rangle_1^n.\,\tau$, or simply $\forall\langle\alpha_i::\Gamma_i\rangle.\,\tau$, to abbreviate the type scheme $\forall\alpha_1::\Gamma_1.\ \ldots\ .\forall\alpha_n::\Gamma_n.\tau$, suggesting that quantified type variables and their constraints form a sequence, not a set; and we will refer to the type τ as the *type proper* of the type scheme.

2.1.2 Program Syntax

The abstract syntax of Mini-Haskell$^+$ programs is shown in Figure 2.2. A program consists of a sequence of class and instance declarations, followed by an expression. Expressions are λ–terms extended with `let`-expressions to permit the definition and use of overloaded and polymorphic values.

Programs	p	::=	**class** α::$c\ \tau$ **where** $x_1:\sigma_1, \ldots, x_n:\sigma_n$ **in** p
		|	**inst** $C \Rightarrow \tau$::γ **where** $x_1 = e_1, \ldots, x_n = e_n$ **in** p
		|	e
Expressions	e	::=	x
		|	$e_1\ e_2$
		|	$\lambda x.e$
		|	**let** $x = e_1$ **in** e_2
Contexts	C	::=	$\{\alpha_1::\Gamma_1, \ldots, \alpha_n::\Gamma_n\}$ $\quad (n \geq 0)$

Figure 2.2: Abstract Syntax of Mini-Haskell$^+$ Programs

2.1.3 Class Declarations

Each class declaration introduces a new class γ and some new overloaded functions x_i of appropriate types. For a parametric type class, there are type parameter(s) in addition to the placeholder variable which is always present in a class declaration. These type parameters are meant to be dependents of the placeholder, which is assumed to be a parameterized type. To distinguish between placeholder and type parameters, we write the placeholder in front of the class, separated by an infix (::). For example:

```
class  t::Eq  where
   (==)  :  t -> t -> Bool    in
class  k::Collection a  where
   insert  :  a -> k -> k
   delete  :  a -> k -> k
   member  :  a -> k -> Bool  in
...
```

The first declaration introduces a class without parameters; in Haskell this would be written `class Eq t where ...`. The second declaration defines a type class

`Collection` with one parameter; this cannot be expressed in standard Haskell.

We follow the static conventions of Haskell's type classes. In the declarations given above, type variables `k` and `a` are, like `t`, scoped only over the method-type templates in the class body. Placeholder `k` is the parameterized type on which those overloaded operations operate and thus must appear in the type of every operator. Hence an additional overloaded constructor `emptyCollect` with type specification

```
emptyCollect  :  k
```

is permitted, whereas the following method specification is not allowed (in class `Collection`):

```
foo  :  a -> a -> Int
```

Furthermore, two classes in scope at the same time may not share any of the same methods.

Classes with several parameters are similarly defined. For example, we may use the following class `Map` with two parameters to describe associative arrays mapping keys of type `a` to values of type `b`:

```
class  t::Map (a, b)  where
   emptyTab  :  t
   lookup    :  a -> t -> b
   contains  :  a -> t -> Bool
   atPut     :  a -> b -> t -> t
   remove    :  a -> t -> t
```

2.1.4 Instance Declarations

The instances of a class are defined by a collection of instance declarations. An instance declaration provides methods e_i to implement the class functions x_i at an

instance type τ. The infix (::) notation is also used in instance declarations and contexts. For example, the following instance declarations assert that both **Int** and **Float** are instances of **Eq** by providing the appropriate definitions of the equality method for each type.

```
inst  Int::Eq  where
   (==)  =  primIntEq    in
inst  Float::Eq  where
   (==)  =  primFloatEq  in
...
```

Note that in an instance declaration, the instance type τ must not be a variable, since declarations such as `inst` α`::`γ `where` $x = e$ define x to be a polymorphic, not overloaded function.

The method definitions in an instance declaration may themselves contain overloaded operations, if they are provided with a suitable *context* C, which is a finite set of instance constraints on type variables occurring in the instance type. For example:

```
inst  {a::Eq} => List a :: Eq  where
   l1 == l2  =  (null l1  and  null l2)
                or (    not (null l1)
                   and  not (null l2)
                   and  (head l1 == head l2)
                   and  (tail l1 == tail l2) )
```

The context {a::Eq} indicates that equality over type a is needed in the method definition and thus restricts the instantiation of a to only instances of **Eq**. In other words, for type `List` τ to be an instance of Eq, type τ must be one, too. Conversely, if type τ admits equality, so does `List` τ. As we will see, contexts play a crucial role in developing our system.

In addition to the components discussed so far, an instance declaration for a parametric type class supplies suitable type arguments to instantiate the class. To establish the dependence relationship implicitly assumed between the placeholder and the type parameters in a class declaration, we require that the type arguments be functionally dependent on the instance type in placeholder position. For example, consider the instance declaration presented in the previous chapter for list collections:

```
inst  a::Eq => List a :: Collection a  where
      insert  =  listInsert
      delete  =  listDelete
      member  =  listMember
```

Since the type argument `a` is contained in the instance type `List a`, the method-type templates specified in the class declaration will be properly instantiated, e.g., the type for `listInsert` is:

$$\forall \texttt{a} :: \texttt{Eq} .\ \texttt{a} \rightarrow \texttt{List a} \rightarrow \texttt{List a}$$

Ambiguity

We follow Haskell's static restrictions on instance declarations to avoid *ambiguity*, a problem inherent in overloading. Ambiguity arises when the compiler does not have sufficient type information to determine the appropriate implementation for a particular occurrence of an overloaded operator—there might be several, possibly conflicting, implementations. Any approach to overloading must anticipate such possibilities and find ways to eliminate them; and if ambiguities cannot be completely avoided, the compiler should at least be able to detect them. In our system, it is clear that instance declarations may contribute to ambiguities, since they provide implementations for overloaded operators. For example, it is a manifest ambiguity to declare two ways to implement list collections:

```
inst  a::Eq => List a :: Collection a  where ...
inst  b::Eq => List b :: Collection b  where ...
```

Furthermore, the following two declarations may also lead to ambiguity when the compiler needs an implementation for collections of a more general type `List a`:

```
inst  List Int  :: Collection Int   where ...
inst  List Char :: Collection Char  where ...
```

To eliminate these forms of ambiguity, we require that there be *at most one* instance declaration for every pair of type and class constructor (κ, c). This restriction is necessary but not sufficient: there are other forms of ambiguities that cannot be eliminated but can be detected. Chapter 6 presents a more detailed treatment of this problem.

Consistency Criterion

The at-most-one restriction on instance declarations has another important consequence on our system. The set of instance declarations in a program forms the basis for asserting whether type τ is an instance of class $c\ \tau'$, written as $\tau :: c\ \tau'$, a judgement to be formally defined in the next chapter. Now with this restriction, the above requirement about dependence between instance type and type arguments guarantees that, in determining whether $\kappa\ \tau :: c\ \tau'$, we may assume that type $\kappa\ \tau$ and class c uniquely determine type τ'. One consequence of this is that instance constraints are now subject to a *consistency* criterion: if we have both $\tau :: c\ \tau_1$ and $\tau :: c\ \tau_2$, then we must have $\tau_1 = \tau_2$. The type reconstruction algorithm enforces consistency in this situation by unifying τ_1 and τ_2. The consistency requirement also explains why, in a class set $\Gamma = \{c_1\ \tau_1, ..., c_n\ \tau_n\}$, we demand that the c_i be pairwise disjoint: an instance constraint α::Γ in a context or a type scheme means that $\alpha :: c_i\ \tau_i$ for all i.

2.2 Example

In this section we develop a system that performs arithmetic operations on complex numbers as a simple example of a program that uses parametric type classes. For conciseness the program is written using Haskell-like syntax. In particular, we augment Mini-Haskell$^+$ with Haskell-like type declarations (introduced by the keyword **data**) and pattern matching.

There are at least two familiar representations for complex numbers as ordered pairs: rectangular form (real part and imaginary part) and polar form (magnitude and angle). Addition of complex numbers represented in rectangular form reduces to straightforward addition of coordinates:

$$(r_1, i_1) + (r_2, i_2) = (r_1 + r_2, i_1 + i_2)$$

On the other hand, when multiplying complex numbers, it is more natural to think in terms of representing a complex number in polar form. The product of two complex numbers is the pair obtained by multiplying their magnitudes and adding their angles:

$$(m_1, a_1) \cdot (m_2, a_2) = (m_1 \cdot m_2, a_1 + a_2)$$

Thus there is a preferred representation for each operator.

Our goal here is to develop a program that performs arithmetic operations on complex numbers in a single system where both representations coexist. In particular, we want our program to use only a single function for each arithmetic operation, regardless of the representations of their arguments. To design such a program, we define a one-parameter type class **Complex** with the following four overloaded selectors: **real-part**, **imag-part**, **magnitude**, and **angle**. In addition, we declare two data types to represent complex numbers and be instances of **Complex**. Using these selectors and constructors associated with the data types, we can implement complex-number arithmetic in a way that is independent of the underlying representations.

```
class  c::Complex a  where
   real-part,
   imag-part,
   magnitude,
   angle        :   c -> a

--Two representations of complex numbers

data  Rect a    =  MkRect a a

data  Polar a   =  MkPolar a a

instance  a::Num => Rect a  :: Complex a  where
   real-part (MkRect x y)   =  x
   imag-part (MkRect x y)   =  y
   magnitude (MkRect x y)   =  sqrt (square x  +  square y)
   angle (MkRect x y)       =  atan y x

instance  a::Num => Polar a :: Complex a  where
   real-part (MkPolar r t)  =  r * cos t
   imag-part (MkPolar r t)  =  r * sin t
   magnitude (MkPolar r t)  =  r
   angle     (MkPolar r t)  =  t

-- Arithmetic operations on complex numbers

cAdd z1 z2  =  MkRect (real-part z1  +  real-part z2)
                      (imag-part z1  +  imag-part z2)

cSub z1 z2  =  MkRect (real-part z1  -  real-part z2)
                      (imag-part z1  -  imag-part z2)

cMul z1 z2  =  MkPolar (magnitude z1  *  magnitude z2)
                       (angle z1  +  angle z2)

cDiv z1 z2  =  MkPolar (magnitude z1  /  magnitude z2)
                       (angle z1  -  angle z2)
```

Figure 2.3: Complex-number Arithmetic

The program appears in Figure 2.3. Two polymorphic types, **Rect** and **Polar**, are declared to represent complex numbers using the **data** declarations. **MkRect** and **MkPolar** are (data) constructors, and, when applied, yield values of types **Rect** and **Polar** respectively. These two parameterized types become instances of the parametric class **Complex** if their constituent type supports arithmetic operations and trigonometric functions such as **cos** and **atan**, which are put together under the class **Num**. The definition of **Num** is not pertinent and has been omitted. With these declarations, we can add and subtract complex numbers in terms of real and imaginary parts while multiplying and dividing complex numbers in terms of magnitudes and angles.

The use of overloaded selectors ensures that the definition of the complex-number-arithmetic operators **cAdd**, **cSub**, **cMul**, **cDiv** is independent of which representation we choose. Indeed, this is reflected in our type system by the principal type of **cAdd** and **cSub**:

$$\forall \mathtt{a} :: \mathtt{Num}.\forall \mathtt{c1} :: \mathtt{Complex\ a}.\forall \mathtt{c2} :: \mathtt{Complex\ a}.\ \mathtt{c1} \rightarrow \mathtt{c2} \rightarrow \mathtt{Rect\ a}$$

and similarly of **cMul** and **cDiv**:

$$\forall \mathtt{a} :: \mathtt{Num}.\forall \mathtt{c1} :: \mathtt{Complex\ a}.\forall \mathtt{c2} :: \mathtt{Complex\ a}.\ \mathtt{c1} \rightarrow \mathtt{c2} \rightarrow \mathtt{Polar\ a}$$

Quantified type variables **c1** and **c2** can be independently instantiated to any types that are instances of **Complex**.

In contrast, we cannot write similar programs using Haskell's type classes without being overspecific: either a particular representation of complex number or the constituent type **a** has to be fixed.

2.3 Discussion

2.3.1 Class Inclusion

We have described the proposed extension of type classes and illustrated its features with an example. One aspect of the system of type classes used in Haskell that we have not discussed so far is the notion of *class inclusion*. For example, we may wish to define a class `Ord` which *inherits* all the operations in `Eq`, but has in addition a set of comparison operations:[1]

```
class  a::Ord  where
   a::Eq
   (<), (<=)  :  a -> a -> Bool
```

The constraint `a::Eq` in the class body stipulates that `Eq` is a *superclass* of `Ord`, and any instance of `Ord` must also be an instance of `Eq`. This subclass/superclass facility is used extensively in the definition of the predefined classes of Haskell to group overloaded operators into class hierarchies.

Despite their wide use in Haskell, superclass declarations are largely a syntactic mechanism and thus do not play an essential role in the development of our type system. Indeed, we can model the subclass/superclass relationship using class sets. Consider, for example, the class `Eq()` of equality types and its subclass `Ord()` of ordered types. We can always represent `Ord()` as a set of two classes, $\{\texttt{Eq}(), \texttt{Ord}'()\}$, where `Ord'` contains only operations $(<, \leq)$, which are defined in `Ord` but not in `Eq`. Translating all classes in a program in this way, we end up with sets over a flat domain of classes. This shows that we can without loss of generality disregard class hierarchies in our system.

[1] In Haskell, this is written as `class Eq a => Ord a where (<), (<=) :: a -> a -> Bool`.

2.3.2 Multi-parameter Type Classes

Before going into the formal treatment of our type system, it is worth considering parametric type classes from a different perspective. In Haskell, type classes are viewed as *unary* predicates over types, as suggested by the notation used, e.g., **Eq t**. As such, it is natural to consider classes over multiple types. In some ways, parametric type classes can be seen as a special form of multi-parameter type classes. To see this, we rewrite our definition of `Collection` using Haskell-like syntax as follows:

```
class  Collection a k  where
   insert  :  a -> k -> k
   delete  :  a -> k -> k
   member  :  a -> k -> Bool
```

However, the similarity is superficial. The differences between the two systems are very fundamental and lead to significant consequences that merit some explanation.

Essentially, we view type classes as constraints on individual types, and introduce judgements like τ::**Eq** to express that type τ satisfies the constraint **Eq**. Parametric classes are our approach to generalizing standard classes so that we may constrain both a parameterized type and any constituent types in a way that the dominance of the parameterized type over its constituents is captured. In other words, it is still the case that only one type is constrained, but the type constrained determines the others. A multi-parameter type class, on the other hand, is viewed as a general predicate over multiple types; no pre-existing relationship among the types is assumed.

We use the `Collection` example given above to illustrate two consequences resulting from the different views. The first concerns grouping operators into classes. It is reasonable and desirable to include constructor **emptyCollect** in class **Collection** to represent empty collections. However, its typing in a system of multi-parameter classes is:

```
emptyCollect  :  Collection a k => k
```

This is unacceptable in the system since it is *ambiguous*: type variable **a** occurs in the constraint (`Collection a k`), but not in the type-part proper (`k`). Ambiguous types must be rejected, since they signal the possibility of ambiguity. (Chapter 6 deals with this issue.) By contrast, our system gives the following typing:

```
emptyCollect  : ∀a.∀k::Collection a . k
```

Since **a** depends on `k`, which appears in the type-part proper (`k`), the typing is valid in our system. The distinction between constrained and dependent type variables allow us to avoid this problem.[2]

The second consequence has to do with the consistency criterion discussed in Section 2.1.4. This is an integral part of our system, whereas in a system of multi-parameter classes no such notion as consistency is maintained, since the multiple types constrained by a class need not be related by any pre-existing dependence relationship. As a result, the two systems have different notions of well-typedness. For example, consider the following function `foo` that inserts two objects of different types into a collection:

```
foo c  =  insert 5 (insert 'f' c)
```

Our system will reject this function as an ill-typed one due to the inconsistent constraints derived in typing it: `k::Collection Char` and `k::Collection Int`. In the system of multi-parameter classes, on the other hand, the function `foo` can be typed as follows:

```
foo : (Collection Int k, Collection Char k) => k -> k
```

[2]One may argue that this problem can be circumvented in this case by having the **emptyCollect** constructor in a different class from the other operations. This is true, but it would severely restrict the way in which operators can be grouped.

Only when the function is applied to a particular collection will the two constraints `Collection Int k` and `Collection Char k` be checked for satisfiability.

Chapter 3

Type Inference Systems

Type inference systems are used to determine types for program expressions through a set of inference rules without any need for explicit type declaration. The Hindley-Milner type system is one such example that works well for languages like ML, in which polymorphic functions behave uniformly over a range of types. In dealing with a combination of polymorphism and overloading *à la* Haskell, we face additional complexity: programmers can supply class/instance declarations to stipulate the behaviors of certain operations on various data types. Thus type inference in this system has to be conducted in accordance with the constraints imposed by these declarations.

This chapter presents two type inference systems for Mini-Haskell$^+$. Since types in Mini-Haskell$^+$ may be constrained by classes, we begin with a constraint inference system that allows us to deduce, according to the instance declarations in a program, what class constraints a type satisfies. We then describe the first type inference system for Mini-Haskell$^+$. It is obtained by extending the Hindley-Milner type system in a highly modular fashion to include the constraint inference system.

The second type inference system adapts the first one towards developing a type reconstruction algorithm. In the original system, there are many ways in which

the typing rules can be applied to a single expression, so that the order in which the rules are applied in constructing a proof of type inference is not obvious. In contrast, the second system has a more restricted set of typing rules and the choice of rules is completely determined by the syntactic structure of the expression involved. This syntax-directed property permits more systematic typing-proof constructions and thus makes these rules more suitable for implementation. Furthermore, we show that the syntax-directed system, despite its simplicity, is equivalent to the original system; therefore, we can use the same type reconstruction algorithm to compute typings that can be inferred from both systems.

Proofs for results of this and following chapters can be found in [Chen *et al.*, 1992b]. They can also be derived from those included in Appendix A for the results of Chapter 6, where the type system is extended to provide a translation semantics for Mini-Haskell$^+$.

For ease of reference, the abstract syntax of Mini-Haskell$^+$ is shown again in Figure 3.1. We have made one simplification with respect to the syntax given in the previous chapter by allowing only one operator symbol in each class declaration: classes with multiple operators can be coded using tuples.

3.1 Instance Constraint Inference

This section presents an inference system for deducing whether a type τ is an instance of a class set Γ, and investigates its properties in the setting of the type inference systems mentioned above.

3.1.1 Instance Entailment

We refer to the problem of checking whether a type τ is an instance of a class set Γ as the satisfiability of an *instance predicate* $\tau::\Gamma$, the result of which is obviously deter-

Type variables	α		
Type constructors	κ		
Types	τ	::=	$\alpha \mid () \mid (\tau_1, \tau_2) \mid \tau_1 \rightarrow \tau_2 \mid \kappa\,\tau$
Type schemes	σ	::=	$\tau \mid \forall\alpha{::}\Gamma\,.\,\sigma$
Class constructors	c		
Type classes	γ	::=	$c\,\tau$
Class sets	Γ	::=	$\{c_1\,\tau_1, \ldots, c_n\,\tau_n\}$ $(n \geq 0,\ c_i$ pairwise disjoint$)$
Programs	p	::=	`class` $\alpha{::}\gamma$ `where` $x{:}\sigma$ `in` p
		$\mid$	`inst` $C \Rightarrow \tau{::}\gamma$ `where` $x = e$ `in` p
		$\mid$	e
Expressions	e	::=	$x \mid e_1\ e_2 \mid \lambda x.e \mid$ `let` $x = e_1$ `in` e_2
Contexts	C	::=	$\{\alpha_1{::}\Gamma_1, \ldots, \alpha_n{::}\Gamma_n\}$ $(n \geq 0)$

Figure 3.1: Abstract Syntax of Mini-Haskell$^+$

mined by the set of instance declarations Σ in a program. We use these declarations to generate an inference system for a theory whose sentences are instance judgements of the form $C \Vdash \tau{::}\Gamma$, which asserts that from C, it follows that $\tau{::}\Gamma$ is true. Informally, the context C records the class constraints on the type variables occurring in τ. Instance declarations such as `inst` $\tau'{::}\gamma'$ `where` ... are interpreted as axioms of the system, while those with contexts such as `inst` $C' \Rightarrow \tau'{::}\gamma'$ `where` ... are interpreted as inference rules by treating C' and $\tau'{::}\gamma'$ as antecedent and conclusion respectively. Finally, a type τ is an instance of class set Γ if it is an instance of all the classes in Γ.

The inference rules given in Figure 3.2 formalize our idea of instance entailment. Note that singleton class sets of the form $\{\gamma\}$ are written simply as γ. An instance judgement is true in the theory if it can be deduced using these rules. The context

$$\Gamma) \quad \frac{\forall i \in 1..n\,(C \Vdash \tau::\gamma_i)}{C \Vdash \tau::\{\gamma_1, ..., \gamma_n\}} \quad (n \geq 0)$$

$$\alpha) \quad \frac{\gamma \in C\alpha}{C \Vdash \alpha::\gamma}$$

$$\tau) \quad \frac{\forall i \in 1..n\,(C \Vdash \tau_i::\Gamma_i)}{C \Vdash \tau::\gamma} \quad (\ulcorner \texttt{inst}\ \{\tau_i::\Gamma_i\} \Rightarrow \tau::\gamma \urcorner \in \Sigma,\ n \geq 0)$$

Figure 3.2: Inference Rules for Class Constraints

C in these rules is a finite set of *instance assumptions* of the form $\alpha::\Gamma$, with no α occurring twice. We also regard a context as a finite mapping from type variables to class sets, i.e., $C\alpha = \Gamma$ iff $\alpha::\Gamma \in C$. The set of instance declarations Σ in a program is taken as an implicit parameter to this entailment system. In other words, the judgement $C \Vdash \tau::\Gamma$ is really an abbreviation of $C \Vdash_\Sigma \tau::\Gamma$.

As noted earlier, in an instance declaration `inst` $C \Rightarrow \tau::\gamma$ `where` ... , type τ must not be a variable; therefore, inferences using these rules proceed according to the structure of τ. In particular, for an instance predicate $(\kappa\,\tau' :: c\,\tau'')$, the inference consists of an application of rule (τ) to a possibly empty set of instance predicates $\{\tau_i::\Gamma_i\}$, each of which is derived from the instance declaration that matches $(\kappa\,\tau' :: c\,\tau'')$.

The following lemma states that extra type variables in the context of an instance judgement $C \Vdash \tau::\Gamma$ that are not free in the predicate $\tau::\Gamma$ may be ignored.

Lemma 3.1 (weaken) *If* $C(\alpha) = C'(\alpha)$ *for all* $\alpha \in tv(\tau::\Gamma)$, *then* $C \Vdash \tau::\Gamma$ *iff* $C' \Vdash \tau::\Gamma$.

In the sequel, we will write $C \Vdash \{\tau_i::\Gamma_i\}$ to express the satisfiability of a set of instance predicates $\{\tau_1::\Gamma_1, \ldots, \tau_n::\Gamma_n\}$ under some context C, assuming that the range of i is clear from the given setting.

3.1.2 Contexts

We now introduce some notions about contexts that will be used through the subsequent development of our system.

A context records the type variables and the class constraints associated with them in a given scenario. It can be seen as a special form of an instance predicate set, a "normalized" one, where type variables but not general types are constrained. The domain of a context C, written $\mathrm{dom}(C)$, is defined as the set of type variables α such that $(\alpha::\Gamma) \in C$. Viewing contexts as finite maps from type variables to class sets, we want to think of the *region* of a context C:

$$\mathrm{reg}(C) = \bigcup_{\alpha \in \mathrm{dom}(C)} \mathrm{tv}(C\alpha).$$

Using this notion, we define the *closure* of a context C over a set of type variables Δ, written $C^*(\Delta)$, as the least fixed point of the following equation:

$$C^*(\Delta) = \Delta \cup C^*(\mathrm{reg}\ C|_\Delta)$$

where $C|_\Delta$ is the restriction of C to Δ. Intuitively, $C^*(\Delta)$ is the set of type variables that are related, directly or indirectly, to those in Δ through the class constraints in C.

We say C_1 is *contained* in C_2, written $C_1 \preceq C_2$, if $\mathrm{dom}(C_1) \subseteq \mathrm{dom}(C_2)$ and $C_1\alpha \subseteq C_2\alpha$ for each $\alpha \in \mathrm{dom}(C_1)$. We write $C_1 \uplus C_2$ for the disjoint union of two contexts and $C\backslash_\alpha$ for restriction of a context C to all type variables in its domain other than α. As a consequence of the consistency criterion given in 2.1.4, the set union of two contexts is well-defined only for *compatible* contexts. Two contexts C_1 and C_2 are compatible, written $C_1 \bowtie C_2$, if, for any class constructor c and type variable $\alpha \in \mathrm{dom}(C_1) \cap \mathrm{dom}(C_2)$, we have $\tau = \tau'$ whenever $c\ \tau \in C_1\alpha$ and $c\ \tau' \in C_2\alpha$.

A context C is *closed* if $C^*(\mathrm{dom}\ C) = \mathrm{dom}\ C$, or, equivalently, if $\mathrm{reg}(C) \subseteq \mathrm{dom}(C)$. A context C is *acyclic* if the type variables in $\mathrm{dom}(C)$ can be topologically

sorted according to the ordering given by: $\alpha < \beta$ if $\alpha \in \mathrm{tv}(C\beta)$. We shall restrict our discussion to only closed acyclic contexts in the remainder of the thesis. For example, the contexts {a::Klc a} and {a::Kls b, b::Kls a} are both "recursive" (cyclic) and thus prohibited.[1] The exclusion of recursive contexts has to do with the structure of type schemes in our system: the ordered quantification in a type scheme rules out such a possibility.

Finally, we say that context C *covers* an expression Π if $\mathrm{tv}(\Pi) \subseteq \mathrm{dom}(C)$. In the following discussion, unless stated otherwise, we will always assume that this is the case for expressions such as $C \Vdash \{\tau_i::\Gamma_i\}$, since unconstrained type variables simply have empty class sets in the context.

3.1.3 Substitution, Context and Instance Entailment

Substitution plays an important role in Hindley-Milner style systems of parametric polymorphism. To integrate our instance entailment system into the Hindley-Milner type system, we need to investigate its interaction with substitution.

A (type) substitution is a map from type variables to types. The domain of a substitution S, $\mathrm{dom}(S)$, is the set of type variables α such that $S\alpha \neq \alpha$. The region of a substitution can be defined in the same way as the region of a context. As usual, the *composition* of substitutions S and R is denoted by $S \circ R$, or simply by juxtaposition SR. The *extension* to a substitution S by mapping type variable α to type τ is denoted by $[\tau/\alpha]S$.

In our system, we will apply substitutions not only to types, but also to (sets of) classes and (sets of) instance predicates. On all of these, substitution is defined

[1] The type reconstruction algorithm enforces this restriction by performing clique detection on the underlying context after each call to unification.

pointwise, i.e., it is a homomorphism on sets, class constructor application and (::):

$$S(c\ \tau) = c\ S\tau \qquad S\{\gamma_i\} = \{S\gamma_i\}$$
$$S(\tau::\Gamma) = S\tau::S\Gamma \qquad S\{\tau_i::\Gamma_i\} = \{S(\tau_i::\Gamma_i)\}$$

Since a context is a special form of an instance predicate set, substitutions can be applied to contexts. However, the result of such a substitution is in general not a context, as the left hand side α of an instance predicate $\alpha::\Gamma$ can be mapped to a non-variable type. Our entailment rules, on the other hand, require contexts instead of general instance predicate sets. Thus, applying an arbitrary substitution to an instance judgement $C \Vdash \tau::\Gamma$ may not give us another one. In other words, instance judgements are not closed under substitution.

The following two properties do hold in our system, the second of which shows the interaction between substitution and instance entailment:

- **monotonicity:** For any contexts C and C', if $C' \preceq C$ then $C \Vdash C'$.
- **transitivity under substitution:** For any substitution S, contexts C and C', if $C \Vdash \tau::\Gamma$ and $C' \Vdash SC$, then $C' \Vdash S(\tau::\Gamma)$.

The first property is fairly obvious. The second one can be established by a straightforward structural induction on type τ. It indicates that, to apply a substitution to an instance judgement, we need to find a new context that entails the original one under the substitution.

The following lemma shows how the context closure operation interacts with substitutions under proper context change.

Lemma 3.2 *Let C and D be contexts and S a substitution such that $D \Vdash SC$. Then for any $\Delta \subseteq dom(C)$, we have $tv(S\ C^*(\Delta)) \subseteq D^*(tv\ S\Delta)$.*

3.2 Constrained Type Inference

This section presents our first type inference system for Mini-Haskell$^+$. Like the Hindley-Milner type system, type inference in our system depends on type assumptions. A type assumption set is a set of type predicates $x:\sigma$ in which no x appears twice. The difference is that our system includes an additional context containing instance predicates $\alpha::\Gamma$ in the typing rules to restrict the set of types that these type variables may range over. This leads to a modular combination of the instance constraint inference system presented above with the Hindley-Milner type inference system.

Figures 3.3 and 3.4 give the typing rules that allow us to derive typing judgements of the form

$$A,\ C \vdash p : \sigma,$$

which asserts that program expression p has type σ when the types of the free variables in p are specified in A while the class constraints on the free type variables in A are specified in C.

The rules in Figure 3.3 specify how to type expressions along the lines of the standard Hindley-Milner system. These are the same six rules as in [Damas and Milner, 1982], but with the constraint system added. The constraint extension contributes to the restricted instantiation and quantification of type variables, which becomes manifest in two particular rules: the quantified type variable in a type scheme $\forall\alpha::\Gamma.\sigma$ can be instantiated to a type τ only if we know from the context that $\tau::\Gamma$ ($\forall$-elim), and type variables are generalized together with their class constraints recorded in the context ($\forall$-intro). We write $C.\alpha::\Gamma$ for $C \cup \{\alpha::\Gamma\}$, assuming $\alpha \notin \mathrm{dom}(C)$. A similar convention applies to the type assumption set A. Note that, to ensure type variables are "discharged" from the context in a proper order, we have included the additional constraint about $\mathrm{reg}(C)$ in the side condition of rule ($\forall$-intro). For example, if $C = \{\texttt{a::Eq}, \texttt{k::Collection a}\}$ and $A = \emptyset$, then we cannot discharge the

$$(var) \quad \frac{(x : \sigma) \in A}{A, C \vdash x : \sigma}$$

$$(\forall\text{-elim}) \quad \frac{A, C \vdash e : \forall\alpha{::}\Gamma.\sigma \qquad C \Vdash \tau{::}\Gamma}{A, C \vdash e : [\tau/\alpha]\,\sigma}$$

$$(\forall\text{-intro}) \quad \frac{A, C.\alpha{::}\Gamma \vdash e : \sigma}{A, C \vdash e : \forall\alpha{::}\Gamma.\sigma} \quad \alpha \notin \mathrm{tv}(A) \cup \mathrm{reg}(C)$$

$$(\lambda\text{-elim}) \quad \frac{A, C \vdash e_1 : \tau' \to \tau \qquad A, C \vdash e_2 : \tau'}{A, C \vdash e_1\, e_2 : \tau}$$

$$(\lambda\text{-intro}) \quad \frac{A.x{:}\tau', C \vdash e : \tau}{A, C \vdash \lambda x.e : \tau' \to \tau}$$

$$(let) \quad \frac{A, C \vdash e_1 : \sigma \qquad A.x{:}\sigma, C \vdash e_2 : \tau}{A, C \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \tau}$$

Figure 3.3: Typing Rules for Expressions

constraint a::Eq until we first discharge k::Collection a, since a depends on k.

One may observe that, in rule (*let*), a single context C is used to derive the types for both the `let`-definition e_1 and the `let`-body e_2. Another reasonable choice would be to use two contexts as follows:

$$\frac{A, C_1 \vdash e_1 : \sigma \qquad A.x{:}\sigma, C_2 \vdash e_2 : \tau}{A, C_2 \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \tau} \quad C_1 \preceq C_2$$

The following lemma shows that the two alternatives lead to equivalent systems.

Lemma 3.3 *Suppose that* $A, C' \vdash e : \sigma$. *If* $C' \preceq C$ *and* $dom(C) \cap \mathsf{b}(\sigma) = \emptyset$, *then* $A, C \vdash e : \sigma$.

$$(class)\quad \frac{A.x:\forall_{tv(\gamma)}\forall\alpha::\{\gamma\}.\sigma, C \vdash p:\sigma'}{A, C \vdash (\texttt{class}\ \alpha::\gamma\ \texttt{where}\ x:\sigma\ \texttt{in}\ p):\sigma'}$$

$$(inst)\quad \frac{A, C \vdash x:\forall\alpha::\{\gamma\}.\sigma \qquad A, C \cup C' \vdash e:[\tau'/\alpha]\sigma \qquad A, C \vdash p:\sigma'}{A, C \vdash (\texttt{inst}\ C' \Rightarrow \tau'::\gamma\ \texttt{where}\ x = e\ \texttt{in}\ p):\sigma'}$$

Figure 3.4: Typing Rules for Declarations

The rules in Figure 3.4 extend this system from expressions to programs. In rule (*class*), the overloaded identifier x is added to the type assumption set with its most general type. Rule (*inst*) expresses a type compatibility requirement between an overloaded identifier and its instance definition.

To sum up, a program consists of a series of declarations followed by an expression. Class declarations in a program provide the type templates for overloaded functions, while instance declarations instantiate those type templates properly and generate an entailment relation $\Vdash$. Together they supply the constraint information needed to infer the type of the main expression in a program. Since the method definition in an instance declaration is also an expression, in the remainder of the thesis we shall focus only on the problem of how to derive types for expressions, omitting the other bookkeeping operations for handling declarations.

3.3 A Syntax-directed Approach

In this section we present a syntax-directed type inference system. Compared to the typing rules in the previous section, the rules here are formulated so that the typing derivation for a given term e is uniquely determined by the syntactic structure of e. As a result, these rules are better suited for use in a type reconstruction algorithm. We show that the system is equivalent to the previous one in terms of expressiveness

and, in addition, has all the nice properties required for the construction of a type reconstruction algorithm.

The use of a syntax-directed, Hindley-Milner style inference system originates in [Clément *et al.*, 1986]. They call it "deterministic" in the sense that there is only one rule stating how to type each syntactic construct, whereas, in the original system, rules (∀-intro) and (∀-elim) may be invoked at any time in building a proof for $A \vdash e : \sigma$, thus rendering the system non-deterministic. The technique they developed is to apply rule (∀-elim) only to variables and rule (∀-intro) only to the definition sub-expression of a `let`-expression. We will follow their method to derive a syntax-directed system for Mini-Haskell$^+$, but we need to introduce the notion of generic instance first.

3.3.1 Ordering Type Schemes and Type Assumptions

A useful fact about the Hindley-Milner type system is that when an expression e has a type, there is a *principal type* which captures the set of all other types derivable for e through the notion of *generic instance*, an ordering relationship between type schemes.

Definition 3.1 (Generic instances) *A type scheme* $\sigma' = \forall\langle\alpha'_j{::}\Gamma'_j\rangle.\,\tau'$ *is a* generic instance *of another type scheme* $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\,\tau$ *under a context* C, *if none of the variables* α'_j *appears free in* σ *or* C, *and there exists a substitution* S *on* $\{\alpha_i\}$ *such that*

$$S\tau = \tau' \quad \textit{and} \quad C \uplus \{\alpha'_j{::}\Gamma'_j\} \Vdash S\{\alpha_i{::}\Gamma_i\}.$$

We write in this case, $\sigma' \preceq_C \sigma$, *and we drop the subscript in* $\preceq_C$ *if* $C = \{\}$.

The definition of $\preceq_C$ is an extension of the ordering relation defined in [Damas and Milner, 1982]. The new requirement on instance entailment is needed for the

extension of parametric type classes. It is easy to see that $\preceq_C$ defines a preorder on the set of type schemes.

The following properties of $(\preceq_C)$ follow directly from its definition.

Lemma 3.4 *If* $\sigma' \preceq_C \sigma$ *and* $C \preceq C'$ *then* $\sigma' \preceq_{C'} \sigma$.

Lemma 3.5 *If* $\sigma'' \preceq_C \sigma'$ *and* $\sigma' \preceq_C \sigma$ *then* $\sigma'' \preceq_C \sigma$.

The next lemma shows the interaction between substitutions and the ordering on type schemes. It is a consequence of the transitivity under substitution property of the instance entailment system $\Vdash$.

Lemma 3.6 *If* $\sigma' \preceq_C \sigma$ *and* $C' \Vdash SC$ *then* $S\sigma' \preceq_{C'} S\sigma$.

The ordering on type schemes $(\preceq_C)$ extends naturally to an ordering between type assumption sets: for type assumption sets A' and A, and context C, $A' \preceq_C A$ iff dom A' = dom A and $A'(x) \preceq_C A(x)$ for all x, $x \in$ dom A.

Clearly, the ordering between two type assumption sets still holds when the context is enlarged.

Lemma 3.7 *If* $A' \preceq_C A$ *and* $C \preceq C'$, *then* $A' \preceq_{C'} A$.

3.3.2 Syntax-directed Typing Rules

The typings rules for the syntax-directed system are given in Figure 3.5. The rules ($\forall$-intro) and ($\forall$-elim) have been removed and typing judgements are now of the form $A, C \vdash' e : \tau$ where τ ranges over the set of type expressions rather than the set of type schemes as in the typing judgements of the previous section. Other major differences are that rule (*var'*) instantiates a type scheme to a type according to the

$$(var') \qquad \frac{(x:\sigma) \in A \qquad \tau \preceq_C \sigma}{A, C \vdash' x : \tau}$$

$$(\lambda\text{-elim}') \qquad \frac{A, C \vdash' e_1 : \tau' \to \tau \qquad A, C \vdash' e_2 : \tau'}{A, C \vdash' e_1\ e_2 : \tau}$$

$$(\lambda\text{-intro}') \qquad \frac{A.x:\tau', C \vdash' e : \tau}{A, C \vdash' \lambda x.e : \tau' \to \tau}$$

$$(let') \qquad \frac{A, C' \vdash' e_1 : \tau_1 \qquad A.x:\sigma, C \vdash' e_2 : \tau}{\begin{array}{c} A, C \vdash' (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \tau \\ (\sigma, C'') = gen(\tau_1, A, C'),\ C'' \preceq C \end{array}}$$

Figure 3.5: Deterministic Typing Rules for Expressions

definition of generic instance and rule (*let'*) uses the generalization function, *gen*, to introduce type schemes. In other words, only type expressions are involved in the matching process; type schemes can only appear in the type assumption set.

The function *gen* acts as the rule (∀-intro) of the original type system. It takes as arguments a type scheme, an assumption set, and a context, and returns a generalized type scheme and a discharged context. It is defined as follows:

$$\begin{array}{rcl} gen\ (\sigma,\ A,\ C) & = & \textbf{if}\ \exists\alpha \in \mathrm{dom}(C) \backslash (\mathrm{tv}\ A \cup \mathrm{reg}\ C)\ \textbf{then} \\ & & \quad gen\ (\forall\alpha{::}C\alpha.\sigma,\ A,\ C\backslash_\alpha) \\ & & \textbf{else}\ (\sigma, C). \end{array}$$

In other words, instance assumptions in the given context, except those constraining, directly or indirectly, type variables in the type assumption set, are discharged and moved to form a more general type scheme in an order such that type variables are properly quantified.[2] This is formalized in the following lemma:

[2]To make it well-defined, any implementation of this function must fix a particular order to

Lemma 3.8 *If* $gen(\tau, A, C) = (\sigma, C')$, *then* $\sigma = \forall \langle \alpha_i :: C\alpha_i \rangle_1^n . \tau$ *for some* $n \geq 0$ *such that* $\{\alpha_i\} \uplus dom(C') = dom(C)$ *and* $dom(C') = C^*(tv\ A)$.

We also need the following lemmas about *gen* to investigate the properties of the syntax-directed system and its relationship with the original system. The first two lemmas follow directly from *gen*'s definition.

Lemma 3.9 *If* $A, C \vdash e : \tau$ *and* $(\sigma, C') = gen(\tau, A, C)$, *then* $A, C' \vdash e : \sigma$.

Lemma 3.10 *Let* $(\sigma, C_1) = gen(\tau, A, C)$ *and* $(\sigma', C_1') = gen(\tau, A, C \uplus C')$. *Then* $\sigma' \preceq \sigma$ *and* $\sigma \preceq_{C'} \sigma'$.

The next two lemmas describe the interaction between the *gen* function and substitutions under the common scenario $A, C \vdash' e : \tau$.

Lemma 3.11 *Suppose that* $(\sigma, C') = gen(\tau, A, C)$. *Let* D *be a context and* S *a substitution. If* $gen(S\tau, SA, D) = (\sigma', D')$, *then* $\sigma' \preceq_{D'} S\sigma$ *whenever* $D \Vdash SC$.

This result indicates that, in general, the composition of generalization and substitution is not commutative. However, under certain conditions, we can find a suitable substitution that satisfies the commutativity, as shown in the following lemma.

Lemma 3.12 *Suppose that* $(\sigma, C') = gen(\tau, A, C)$. *Let* C'' *be a context and* S *a substitution such that* $C'' \Vdash SC'$. *Then there exists a substitution* R *and a context* D *such that*

$$RA = SA \quad and \quad D \Vdash RC.$$

Furthermore, if $gen(R\tau, RA, D) = (\sigma', D')$ *then*

$$S\sigma = \sigma' \quad and \quad D' \preceq C''.$$

discharge instance assumptions from the given context.

3.3.3 Properties of the Syntax-directed System

This section presents some useful properties of the syntax-directed type system. The first property is about substitution and typing judgements. Like instance judgements, typing judgements are not closed under arbitrary substitution: a matching context is necessary.

Lemma 3.13 *If* $A, C \vdash' e:\tau$ *and* $C' \Vdash SC$, *then* $SA, C' \vdash' e:S\tau$.

The next two lemmas express a form of monotonicity of typing derivations with respect to the context and the type assumption set.

Lemma 3.14 *If* $A, C \vdash' e:\tau$ *and* $C \preceq C'$, *then* $A, C' \vdash' e:\tau$.

Lemma 3.15 *If* $A.x:\sigma, C \vdash' e:\tau$ *and* $\sigma \preceq_C \sigma'$, *then* $A.x:\sigma', C \vdash' e:\tau$.

3.3.4 Relationship with the Original Type System

Given the results of the preceding sections, we can show that the syntax-directed system $\vdash'$ is equivalent to the original system $\vdash$ in the following sense. First, the syntax-directed system is sound with respect to the original one:

Theorem 3.16 *If* $A, C \vdash' e:\tau$ *then* $A, C \vdash e:\tau$.

Second, for each typing derivation in the original system we can always find a derivation in the syntax-directed system such that the inferred type, obtained using the *gen* function, is more general than the type scheme determined by the original derivation:

Theorem 3.17 *If* $A, C \vdash e:\sigma$ *then there is a context* C', *and a type* τ *such that* $C \preceq C'$, $A, C' \vdash' e:\tau$ *and* $\sigma \preceq_C \sigma'$ *where* $(\sigma', C'') = gen(\tau, A, C')$.

Chapter 4

Unification and Type Reconstruction

We have seen in the previous chapter how we can do type inference in the presence of parametric type classes by adding an instance constraint inference system into the Hindley-Milner type system. In particular, using the syntax-directed system, we can infer the types for an expression in a way that the application of typing rules is determined by the syntax of the expression. Nevertheless, since rule (var') can choose *any* generic instance of the type of the identifier x, the system is non-deterministic. In this chapter, we refine the syntax-directed system to develop a deterministic algorithm that reconstructs, for an expression, the *most general* type that can be inferred from the type system.

As usual, type reconstruction relies on unification, so we will first work out what kind of unification is needed for parametric type classes. This leads to our development of a *context-preserving* unification algorithm, which augments first-order unification by propagating the class constraints associated with type variables. We show that the unifiers produced by the algorithm are, in a certain sense, the most general unifiers possible.

We then go on to present a type reconstruction algorithm based on the unification algorithm. We show that for any expression that can be properly typed by the type systems presented in Chapter 3, the type reconstruction algorithm will assign a type from which all other types can be derived. In other words, our type system has the so-called *principal type* property.

4.1 Context-Preserving Unification

Type reconstruction usually relies on unification to compute most general types. One consequence of rule ($\forall$-elim) of the type system given in Section 3.2 is that the well-known first-order unification algorithm of Robinson [Robinson, 1965] cannot be used since not every substitution of type variables to types satisfies the given class constraints. [Nipkow and Snelting, 1991] has shown that order-sorted unification can be used for reconstruction of types in Haskell, but it is not clear how to extend their result to parametric type classes. In this section we develop an algorithm to compute the most general unifier of two types that also preserves the class constraints of any given context.

4.1.1 Constrained Substitution

Usually, a substitution S is called a *unifier* for the type expressions τ and τ' if $S\tau = S\tau'$, and the main use of unification in type reconstruction is to compute a unifier for the type of a function's parameter and that of its actual argument. Unification in our system has, however, an additional task—namely, preserving the underlying context. As discussed in Section 3.1.3, applying a substitution to a context usually yields some general instance predicate set that needs to be further normalized until a preserving context is obtained. To express such interaction between substitutions and contexts in our unification algorithm, we introduce the notion of *constrained substitution*:

Definition 4.1 (Constrained substitutions) *A* constrained substitution *is a pair* (S, C) *where* S *is a substitution and* C *is a context such that* $C = SC$.

As we shall see, a unifier in our system is such a pair of substitution and context.

The following definition introduces an ordering on constrained substitutions by extending the standard ordering on substitutions to include the notion of *context preservation.*

Definition 4.2 *A constrained substitution* (S', C') preserves *another constrained substitution* (S, C) *if there is a substitution* R *such that* $S' = R \circ S$ *and* $C' \Vdash RC$, *in which case we write* $(S', C') \preceq (S, C)$.

Given the ordering on constrained substitutions, we can define the most general constrained substitution that satisfies certain requirements.

Definition 4.3 *A constrained substitution* (S, C) *is* most general *among those constrained substitutions that satisfy some requirement* $\mathcal{R}$ *if* (S, C) *satisfies* $\mathcal{R}$, *and, for any* (S', C') *that satisfies* $\mathcal{R}$, $(S', C') \preceq (S, C)$.

In particular, we are interested the following two requirements on constrained substitutions.

Definition 4.4 *A constrained substitution* (S, C) *is a*

(a) (S_0, C_0)*-preserving* unifier *for the type expressions* τ *and* τ' *if* $S\tau = S\tau'$ *and* $(S, C) \preceq (S_0, C_0)$.

(b) (S_0, C_0)*-preserving* normalizer *of an instance predicate set* P *if* $C \Vdash SP$ *and* $(S, C) \preceq (S_0, C_0)$.

Our goal in the remainder of this section is to compute the most general unifiers and normalizers that also preserve a given context.

4.1.2 Restrictions on Instance Declarations

An important property of Robinson's unification algorithm is that it is able to find the most general unifiers, if they exist. To establish a similar property for our unification algorithm we need to impose some restrictions on the instance declarations in a program, since unifiers in our system have to preserve the underlying class constraints. The restrictions are as follows:

(i) There is no instance declaration of the form `inst` $C \Rightarrow \alpha::c\,\tau$ `where` ... .

(ii) For every pair of type and class constructor (κ, c), there is at most one instance declaration of the form `inst` $C \Rightarrow \kappa\,\tau'::c\,\tau$ `where` ... . Furthermore,

  (a) τ' must be the unit type, or a possible empty tuple of *distinct* type variables.

  (b) Both $\mathrm{dom}(C)$ and $\mathrm{tv}(\tau)$ must be contained in $\mathrm{tv}(\tau')$.

Basically, these are the same restrictions that were informally described in Section 2.1.4. Restriction (i) is fairly obvious, and restriction (ii) is a generalization of the restrictions in Haskell to incorporate parametric type classes. In particular, because of (ii)[a], instance declarations are simple templates that can be instantiated using a standard *matching* operation. Restriction (ii)[b] ensures that all the type variables involved appear in τ'—i.e., there are no "unbound" type variables.

Furthermore, the set of instance declarations in a program must not satisfy any recursive contexts since we exclude recursive contexts from our system. Consider, for example, the following two instance declarations:

```
inst  List a :: C1 a  where ...
inst     Int :: C2 (List Int) where ...
```

Both of them follow the restrictions (i) and (ii) stated above. Nevertheless they must be rejected since they satisfy recursive contexts such as { a::C1 b, b::C2 a } (using the

substitution [`List Int/a`][`Int/b`]). This is achieved by doing a clique-detection on the set of types related by the instance declarations: In the example, the first instance declaration establishes an "edge" between `List a` and `a`, in particular `List Int` and `Int`. But `Int` is in turn "connected" to `List Int` through the second instance declaration, thereby forming a clique.

More formally, we consider the graph formed by viewing types as vertices and letting every instance declaration establish certain edges between certain types. The edges, denoted by $\sim$, are derived as follows. From the instance declaration, removing the auxiliary tuple type constructors,

$$\texttt{inst}\ C \Rightarrow (\kappa^n\ \alpha_1, \ldots, \alpha_n) :: (c^m\ \tau_1, \ldots, \tau_m)\ \texttt{where}\ \ldots$$

we obtain the following edges:

$$S(\kappa^n\ \alpha_1, \ldots, \alpha_n) \sim S\tau_i, \quad 1 \leq i \leq m$$

where S is an arbitrary substitution on α_i, since τ_i contains no type variables other than those of α_i. Then, to support the exclusion of cyclic contexts, we require that the resulting graph be acyclic.[1]

Given these restrictions on the instance declarations in a program, we show in the following section that our unification algorithm computes the most general context-preserving unifiers.

4.1.3 Algorithm

The context-preserving unification algorithm is shown in Figure 4.1; it consists of four mutually recursive functions: *mgu*, *mgu'*, *mgn* and *mgn'*.

Function *mgu* takes two types and returns a transformer on constrained substitutions. The application *mgu* $\tau_1\ \tau_2\ (S, C)$ returns a most general constrained

[1] Note that this is a conservative simplification since we have ignored the additional context C that may exist in an instance declaration.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{mgu : \tau \to \tau \to S \times C \to S \times C} \\
\multicolumn{3}{l}{mgn : \tau \to \Gamma \to S \times C \to S \times C} \\
mgu\ \tau_1\ \tau_2\ (S, C) & = & mgu'\ (S\tau_1)\ (S\tau_2)\ (S, C) \\
mgu'\ \alpha\ \alpha & = & id_{S \times C} \\
mgu'\ \alpha\ \tau\ (S, C)\ \mid\ \alpha \notin \mathrm{tv}(\tau) & = & mgn\ \tau\ (C\alpha)\ ([\tau/\alpha] \circ S, [\tau/\alpha] C \backslash_\alpha) \\
mgu'\ \tau\ \alpha\ (S, C) & = & mgu\ \alpha\ \tau\ (S, C) \\
mgu'\ ()\ () & = & id_{S \times C} \\
mgu'\ \kappa\ \tau\ \kappa\ \tau'\ (S, C) & = & mgu\ \tau\ \tau'\ (S, C) \\
mgu'\ (\tau_1, \tau_2)\ (\tau_1', \tau_2') & = & (mgu\ \tau_1\ \tau_1') \circ (mgu\ \tau_2\ \tau_2') \\
mgu'\ (\tau_1 \to \tau_2)\ (\tau_1' \to \tau_2') & = & (mgu\ \tau_1\ \tau_1') \circ (mgu\ \tau_2\ \tau_2') \\
mgn\ \tau\ \{\} & = & id_{S \times C} \\
mgn\ \tau\ \{\gamma\}\ (S, C) & = & mgn'\ (S\tau)\ (S\gamma)\ (S, C) \\
mgn\ \tau\ (\Gamma_1 \cup \Gamma_2) & = & (mgn\ \tau\ \Gamma_1) \circ (mgn\ \tau\ \Gamma_2) \\
mgn'\ \alpha\ c\,\tau\ (S, C) & = & \textbf{if } \exists \tau'.(c\,\tau' \in C\alpha) \textbf{ then } mgu\ \tau\ \tau'\ (S, C) \\
 & & \textbf{else } (S, C[C\alpha \cup \{c\,\tau\}/\alpha]) \\
\multicolumn{3}{l}{mgn'\ \kappa\,\tau'\ c\,\tau\ (S, C)\ \mid\ \exists\, \ulcorner \texttt{inst}\ C' \Rightarrow \kappa\,\tilde{\tau}'::c\,\tilde{\tau}\ \urcorner \in \Sigma} \\
 & = & \textbf{let } S' = match\ \tilde{\tau}'\ \tau' \\
 & & \quad (S'', C'') = mgu\ \tau\ (S'\tilde{\tau})\ (S, C) \\
 & & \quad \{\tau_1::\Gamma_1, \ldots, \tau_n::\Gamma_n\} = S'C' \\
 & & \textbf{in } (mgn\ \tau_1\ \Gamma_1\ (\ \ldots\ (mgn\ \tau_n\ \Gamma_n\ (S'', C'')))) \\
\multicolumn{3}{l}{\text{(and similarly for } \to, (,), ())}
\end{array}
$$

Figure 4.1: Unification and Normalization Algorithms

substitution that unifies the types τ_1 and τ_2 and preserves (S, C), if such a substitution exists. A subsidiary function *mgu'* is used to thread the constrained substitution through recursive calls that traverse the structures of the given types. The algorithm is similar to the one of Robinson, except for the case $mgu'\ \alpha\ \tau\ (S, C)$, where α may be substituted by τ only if τ can be shown to be an instance of $C\alpha$. This constraint translates to an application of the subsidiary function *mgn* to τ and $C\alpha$.

The call $mgn\ \tau\ \Gamma\ (S, C)$ computes a most general (S, C)-preserving normalizer of $\{\tau::\Gamma\}$, provided one exists. This is accomplished by normalizing $\tau::\gamma$ for each $\gamma \in \Gamma$ using *mgn'*. In *mgn'*, it may in turn call $mgu\ \tau\ \tau'$ to solve a subproblem of the form $C.\alpha::(\{c\,\tau'\} \cup \Gamma') \Vdash \alpha::c\,\tau$. The unification is required since all class constructors in a class set are pairwise disjoint. Thus, the above entailment has $\tau = \tau'$ as a logical consequence. The other call to *mgu* is made when solving the entailment $C \Vdash \kappa\,\tau'::c\,\tau$. Since there is at most one instance declaration for each (κ, c) pair, it either fails or finds the proper instance declaration. The standard pattern matching operation *match* is to instantiate the instance declaration $\ulcorner C' \Rightarrow \kappa\,\tilde{\tau}'::c\,\tilde{\tau}\urcorner$. It takes the pair of types $(\tilde{\tau}', \tau')$ and returns a most general substitution S' such that $S'\tilde{\tau}' = \tau'$. Then, due to the consistency criterion, it calls *mgu* to unify τ and $S'\tilde{\tau}$. Finally, it instantiates the declared context C' and recursively normalizes it.

We now investigate the properties of the algorithm. The first two properties concern the type variables that are involved in the computed unifiers. Note that in an instance declaration, $\ulcorner C' \Rightarrow \kappa\,\tilde{\tau}'::c\,\tilde{\tau}\urcorner$, the type $\tilde{\tau}'$ is either a unit type or a tuple of distinct type variables, and both $\mathrm{tv}(\tilde{\tau})$ and $\mathrm{dom}(C')$ are contained in $\mathrm{tv}(\tilde{\tau}')$. Thus it suffices to use the standard matching operation to instantiate an instance declaration, and no new type variables will be introduced in the normalization process. Furthermore, the substitution is only extended in *mgu*, which does not introduce any new type variables. Consequently, we have the following lemma:

Lemma 4.1 *Let (S, C) be a constrained substitution and τ_1, τ_2 types such that C covers both $S\tau_1$ and $S\tau_2$. If $mgu\ \tau_1\ \tau_2\ (S, C) = (S', C')$, then there exists a sub-*

stitution R such that $S' = R \circ S$ and both $dom(R)$ and $reg(R)$ are contained in $C^*(tv\, S\tau_1 \cup tv\, S\tau_2)$.

Moreover, in the course of computing the unifier, whenever the substitution is extended with $[\tau/\alpha]$ in *mgu'*, the class constraints on α are discharged from C and all occurrences of α in C are replaced by τ. Hence we have the following lemma:

Lemma 4.2 *Let (S, C) be a constrained substitution and τ_1, τ_2 types such that C covers both $S\tau_1$ and $S\tau_2$. If mgu τ_1 τ_2 $(S, C) = (S', C')$, then (S', C') is a constrained substitution and $dom(S') \setminus dom(S) = dom(C) \setminus dom(C')$.*

This lemma points out an important property of the algorithm—in its execution, the context C is never enlarged and whenever S is extended, C will be correspondingly diminished. This property is crucial to our proof of the termination of the algorithm.

Lemma 4.3 (Termination of *mgu*) *For any constrained substitution (S, C) and types τ_1, τ_2, the invocation mgu τ_1 τ_2 (S, C) either fails or terminates.*

The next lemma states the soundness of our algorithm. It consists of two mutually dependent parts—namely, *mgu* computes a unifier and *mgn* computes a normalizer, both of which preserve the given context.

Lemma 4.4 (Soundness of *mgu* and *mgn*)

1. *If mgu τ_1 τ_2 $(S, C) = (S', C')$, then $S'\tau_1 = S'\tau_2$ and $(S', C') \preceq (S, C)$.*
2. *If mgn τ Γ $(S, C) = (S', C')$, then $C' \Vdash S'(\tau::\Gamma)$ and $(S', C') \preceq (S, C)$.*

Furthermore, our algorithm is also complete. More specifically, if there are context-preserving unifiers and normalizers in a specific setting, our algorithm will computes the ones that are most general.

Lemma 4.5 (Completeness of *mgu* and *mgn*)

1. *If there is an (S, C)-preserving unifier $(\bar{S}, \bar{C})$ of τ_1 and τ_2, then the invocation mgu τ_1 τ_2 (S, C) succeeds with (S', C') such that $S'\tau_1 = S'\tau_2$ and $(\bar{S}, \bar{C}) \preceq (S', C') \preceq (S, C)$.*

2. *If there is an (S, C)-preserving normalizer $(\bar{S}, \bar{C})$ of τ and Γ, then the invocation mgn τ Γ (S, C) succeeds with (S', C') such that $C' \Vdash S'(\tau_3::\Gamma)$ and $(\bar{S}, \bar{C}) \preceq (S', C') \preceq (S, C)$.*

As a corollary of these lemmas, we have the following theorem for our unification algorithm:

Theorem 4.6 (Unification theorem) *Let (S, C) be a constrained substitution and τ_1, τ_2 types such that C covers both $S\tau_1$ and $S\tau_2$. If there is a (S, C)-preserving unifier of τ_1 and τ_2 then mgu τ_1 τ_2 (S, C) returns a most general such unifier. If there is no such unifier then mgu τ_1 τ_2 (S, C) fails in a finite number of steps.*

4.2 Type Reconstruction

Given the context-preserving unification algorithm, we now present an algorithm for deciding, under a fixed setting, if there is any type that can be inferred for a given expression. We show the soundness and completeness of this type inferencer with respect to the type system of Section 3.2 through the syntax-directed typing rules given in Section 3.3 and the equivalence result established therein. As a corollary of these results, we obtain a principal type property for our system analogous to the one in [Damas and Milner, 1982].

4.2.1 Algorithm

The type reconstruction algorithm is shown in Figure 4.2.[2] Function *tp* takes as arguments an expression e, a type assumption set A, and an initial constrained substitution (S, C). It returns, when it succeeds, a type τ and a final constrained substitution (S', C') such that $S'A, C' \vdash e : \tau$. The intended initial scenario is as follows: The "refined" type assumption set SA holds the types of the variables occurring free in e, including those overloaded class operators and certain primitive operations. The context C, on the other hand, contains class constraints on the type variables that are free in SA.

Function *tp* proceeds by cases dispatching on the form of the expression e. Except for the case of `let`-expressions, the algorithm is a straightforward extension of Milner's algorithm W [Milner, 1978] to support constraint processing for parametric type classes. Indeed, *tp* uses the same two major components—instantiation of type schemes with new type variables and a unification algorithm—to synthesize a type for e. The extension concerns the maintaining of instance assumptions in the context C: when new type variables are introduced, they are added to the context with suitable class constraints; and, as discussed in Section 4.1, *mgu* performs context normalization as well as unification.

Note that `let`-expressions need some extra context processing. To construct the type for expression `let` $x = e_1$ `in` e_2 given type assumption set A and constrained substitution (S, C), we first construct the type of e_1. However, as the `let`-expression may be a sub-expression of another expression, the context C may contain instance assumptions besides those needed in constructing the type for e_1. If we were to make a direct recursive call $tp\,(e_1, A, S, C)$, the resulting context would also contain those extra instance assumptions, which in turn will be discharged by function *gen* in

[2]This is actually a simplification of the real algorithm because we can get a cyclic context after the call to unification function and thus violate our restriction on contexts. So what is missing here is a clique-detection algorithm, which is simply a variant of occur checking. We omit it here for simplicity. This is safe since recursive instance declarations are not allowed.

$tp\ (e, A, S, C) =$ **case** e **of**

x : $inst\ (S(Ax), S, C)$

$e_1\, e_2$: **let** $(\tau_1, S_1, C_1) = tp\ (e_1, A, S, C)$
$(\tau_2, S_2, C_2) = tp\ (e_2, A, S_1, C_1)$
α be a new type variable
$(S_3, C_3) = mgu\ \tau_1\ (\tau_2 \to \alpha)\ (S_2, C_2.\alpha::\{\})$
in $(S_3\alpha, S_3, C_3)$

$\lambda x.e$: **let** α be a new type variable
$(\tau_1, S_1, C_1) = tp\ (e_1, A.x:\alpha, S, C.\alpha::\{\})$
in $(S_1\alpha \to \tau_1, S_1, C_1)$

`let` $x = e_1$ `in` e_2 : **let** $C' = \{\, (\alpha::C\alpha) \mid \alpha \in C^*(\mathrm{tv}\ SA) \,\}$
$D = C \backslash C'$
$(\tau_1, S_1, C_1) = tp\ (e_1, A, S, C')$
$(\sigma, C_2) = gen\ (\tau_1, S_1 A, C_1)$
$(\tau, S_2, C_3) = tp\ (e_2, A.x:\sigma, S_1, C_2)$
in $(\tau, S_2, C_3 \uplus D)$

where

$inst\ (\forall\alpha::\Gamma.\sigma, S, C)$ = **let** β be a new type variable **in** $inst\ ([\beta/\alpha]\sigma, S, C.\beta::\Gamma)$

$inst\ (\tau, S, C)$ = (τ, S, C)

Figure 4.2: Type Reconstruction Algorithm

forming a type scheme for e_1. As a result, the type scheme would be overgeneral and the context would become incomplete. Therefore, we cannot pass the whole context C to construct the type of e_1, but only the pertinent part.

Our strategy is to partition the context C into two parts: one that contains instance assumptions on type variables that are free in the type assumption set SA (denoted by C' in tp) and the other the extra ones that are generated in previous stages (denoted by D in tp). Since all variables occurring free in e_1 have their types recorded in SA, sub-context C' is all we need for constructing the type of e_1. As to D, we simply restore it to the context after the type of the `let`-expression is synthesized. Indeed, this scheme is based on the syntax-directed type inference rules of Section 3.3, where two contexts are used in the antecedent part of rule (*let's*) while only one is used in all other rules.

One may argue that the extra context processing in the case of `let`-expressions is due, in a large part, to the way the function *gen* is defined (Section 3.3). A conceivable alternative is to discharge from the given context only instance assumptions that constrain type variables appearing free in the given type, but not the type assumption set:

$$\begin{aligned} gen'\ (\sigma,\ A,\ C) \ = \ & \textbf{if } \exists \alpha \in \mathrm{tv}(\sigma) \backslash (\mathrm{tv}\ A \cup \mathrm{reg}\ C) \textbf{ then} \\ & \quad gen'\ (\forall \alpha :: C\alpha.\sigma,\ A,\ C \backslash_\alpha) \\ & \textbf{else } (\sigma, C) \end{aligned}$$

This function *gen'* will leave those instance assumptions generated earlier intact in C while generalizing the type of e_1, and hence no loss of context information will occur. However, it will not be able to discharge *ambiguous* instance assumptions from C and therefore will fail to return a more general type of e_1. We will have more to say about this interaction between ambiguity and function *gen* in Chapter 6.

Now let us investigate the properties of our algorithm. We begin by showing that tp, when given an appropriate constrained substitution, produces a constrained substitution that preserves the given one.

Lemma 4.7 *Let e be an expression, and let (S, C) be a constrained substitution and A a type assumption set such that C covers SA. If $tp(e, A, S, C) = (\tau, S', C')$, then (S', C') is a constrained substitution and $(S', C') \preceq (S, C)$.*

Henceforth, to simplify the presentation, we shall assume as an implicit side-condition that C covers SA when writing $tp\,(e, A, S, C)$.

The following theorem establish the soundness of tp with respect to the syntax-directed type system.

Theorem 4.8 *If $tp(e, A, S, C) = (\tau, S', C')$, then $S'A, C' \vdash' e : \tau$.*

Combining this with the result of Theorem 3.16 gives us the soundness of tp with respect to the type system of Section 3.2.

Corollary 4.9 (Soundness of tp) *If $tp(e, A, S, C) = (\tau, S', C')$, then $S'A, C' \vdash e : \tau$.*

The next theorem shows that the typing judgements obtained by tp are, in a precise sense, the most general ones derivable for a given expression from the syntax-directed system.

Theorem 4.10 *Suppose that $S'A, C' \vdash' e : \tau'$ and $(S', C') \preceq (S_0, C_0)$. Then $tp(e, A, S_0, C_0)$ succeeds with (τ, S, C), and there is a substitution R such that*

1. *$S' = RS$, except possibly on new type variables of $tp(e, A, S_0, C_0)$,*
2. *$C' \Vdash RC$, and*
3. *$\tau' = R\tau$.*

Combining this with the result of Theorem 3.17 we obtain the following completeness result for tp.

Corollary 4.11 (Completeness of *tp*) *Suppose that* $S'A,\ C' \vdash e : \sigma'$ *and* $(S', C') \preceq (S_0, C_0)$. *Then* $tp(e, A, S_0, C_0)$ *succeeds with* (τ, S, C), *and there is a substitution* R *such that*

1. $S' = RS$ *except possibly on the new type variables of* $tp(e, A, S_0, C_0)$, *and*
2. $\sigma' \preceq_{C'} R\sigma$

where $(\sigma, \tilde{C}) = gen(\tau, SA, C)$.

4.2.2 Principal Type Property

We are now in a position to show, for our type system, the existence of reconstructible *principal types*, the key property of the Hindley-Milner type system. First, let's make precise the notion of principal types in our system.

Definition 4.5 (Principal types) *Given type assumption set* A, *context* C, *and expression* e, *we call type scheme* σ *a* principal type *for* e *under* A *and* C *iff* $A, C \vdash e : \sigma$, *and for every type scheme* σ', *if* $A, C \vdash e : \sigma'$ *then* $\sigma' \preceq_C \sigma$.

As a consequence of the completeness of *tp*, we have the following two corollaries about principal types. The first one shows that the type computed by *tp* is principal in a specific setting.

Corollary 4.12 *Suppose that* $dom(C_0) = (C_0)^*(tv\ S_0A)$. *If* $tp(e, A, S_0, C_0) = (\tau, S, C)$ *and* $gen(\tau, SA, C) = (\sigma, C')$, *then* σ *is a principal type for* e *under* SA *and* C'.

An interesting case occurs when $\mathrm{tv}(A) = \emptyset$ (and hence no initial context is necessary): if $tp(e, A, id, \emptyset) = (\tau, S, C)$, then $gen(\tau, A, C) = (\sigma, \emptyset)$ and σ is a principal type for e under A.

Note that we need the condition $dom(C_0) = (C_0)^*(fv\ S_0 A)$ to apply Theorem 4.11 to prove the corollary. The condition ensures that no redundant instance assumptions occur in C_0, for, if there are, they will remain intact during type reconstruction and will in turn be discharged from C by *gen* in calculating σ and C'. This would then falsify the premise $(S, C') \preceq (S_0, C_0)$ required to apply Theorem 4.11.

We say that expression e is *well-typed* under type assumption set A and context C iff there exist a type scheme σ such that $A, C \vdash e : \sigma$. The next corollary states that, under a given scenario, an expression has a principal type if it has a type.

Corollary 4.13 (Principal type theorem) *If e is well-typed under A and C, then e has a principal type under A and C.*

Chapter 5

Translation Semantics

So far we have concentrated on the type system of Mini-Haskell$^+$—namely, how to do type inference and reconstruction for parametric type classes. In this chapter we turn to the problem of associating *meaning* with well-typed Mini-Haskell$^+$ expressions. One feature of Haskell-style overloading is that at compile time, it is possible to translate, based on typing derivations, a program using overloaded operators to an equivalent program that does not. Since our main concern is overloading resolution, such a translation scheme can be seen as a semantic specification for the source language. Our proposed parametric extension maintains this feature, and it requires the same mechanisms to realize the translation.

This chapter begins with a short introduction to the translation scheme and then leads to the development of a formal translation semantics for Mini-Haskell$^+$. The target language is a version of the polymorphic λ-calculus called CP that includes explicit constructs to handle overloading. Except those constructs, CP is very similar to Mini-Haskell$^+$. One of the main uses of CP is to assign meaning to Mini-Haskell$^+$ expressions by translating them into CP expressions where overloading is made explicit. The translation is based on a mapping between typing derivations in Mini-Haskell$^+$ and CP. More specifically, we show that each typing derivation for a Mini-Haskell$^+$

expression corresponds to a typing derivation for a CP expression with explicit overloading, which thereby serves as a translation for the given expression. In other words, every well-typed Mini-Haskell$^+$ expression has a translation and all the translations obtained in this way are well-typed in CP.

5.1 An Informal Presentation

It is instructive to describe the translation scheme informally before we proceed to the formal semantics. The complex number example of Section 2.2 will be used to illustrate the main ideas.

Class/Instance Declarations

Following [Wadler and Blott, 1989], type classes and their instances are replaced by so-called *(method) dictionaries* which contain all the functions associated with a class. In particular, each instance declaration generates an appropriate definition of a dictionary that contains methods for all the (overloaded) operators associated with a class at a given type. For example, corresponding to the following instance declarations:

```
instance  Int::Eq  where
  (==)  =  primEqInt

instance  Int::Ord  where
  (<)   =  primLeInt
  (<=)  =  primLeqInt
```

we have the dictionaries:

```
DEqInt    =  <primEqInt>
DOrdInt   =  <primLeInt, primLeqInt>
```

Here, $\langle e_1, \ldots, e_n \rangle$ builds a dictionary of n methods. The dictionary for **Eq** contains only the equality method while the one for `Ord` has two methods for `(<)` and `(<=)`.

If an instance declaration has a context, then it translates into a definition of a *dictionary constructor* whose parameters correspond to the dictionaries required by the context. For example, the declaration:

```
inst  a::Num => Rect a :: Complex a  where ...
```

translates into the definition of the unary dictionary constructor `DComRect`:

```
DComRect dNum   =   <...>
```

Given a dictionary for `a::Num`, this dictionary constructor yields a dictionary for instance (`Rect a :: Complex a`). Hence (`DComRect DNumFloat`) generates a dictionary for `Complex` in rectangular form.

Finally, for each operation in a class, there is an *selector* to extract the appropriate method from the corresponding dictionary. Hence for the `Comp` class, we generate the following selectors:

```
real-part  <r, i, m, a>  =  r
imag-part  <r, i, m, a>  =  i
magnitude  <r, i, m, a>  =  m
angle      <r, i, m, a>  =  a
```

Overloaded Expressions

Having introduced how class/instance declarations are handled, we now turn to overloaded expressions. Each reference to an overloaded operator is translated into an

extraction from some dictionary variable, which will either be resolved to a concrete dictionary or remain unknown and become a parameter to the whole expression. For example, the `cAdd` function of Section 2.2, defined by:

```
cAdd z1 z2  =  MkRect (real-part z1  +  real-part z2)
                      (imag-part z1  +  imag-part z2)
```

is translated as follows:

```
cAdd dNum dCom1 dCom2 z1 z2  =
               MkRect ( (+ dNum) ((real-part dCom1) z1)
                                 ((real-part dCom2) z2) )
                      ( (+ dNum) ((imag-part dCom1) z1)
                                 ((imag-part dCom2) z2) )
```

Three additional parameters, `dNum`, `dCom1`, and `dCom2`, are generated, corresponding to the required dictionaries. In general, these dictionary parameters witness the class constraints in the type of an overloaded function, as demonstrated by the type of `cAdd`:

$$\forall \texttt{a} :: \texttt{Num}.\forall \texttt{c1} :: \texttt{Complex a}.\ \forall \texttt{c2} :: \texttt{Complex a}.\ \texttt{c1} \rightarrow \texttt{c2} \rightarrow \texttt{Rect a}$$

Finally, each call of an overloaded function supplies the appropriate dictionary arguments. Thus the application `cAdd (MkRect 1.5 2.5) (MkPolar 4.0 3.5)` translates to

```
cAdd DNumFloat (DComRect DNumFloat) (DComPolar DNumFloat)
     (MkRect 1.5 2.5) (MkPolar 4.0 3.5)
```

Type variables	α			
Type constructors	κ			
Types	σ	$::=$	$\alpha \mid () \mid (\sigma_1, \sigma_2) \mid \sigma_1 \rightarrow \sigma_2 \mid \kappa\,\sigma \mid \forall\alpha{::}\Gamma\,.\,\sigma$	
Class lists	Γ	$::=$	$\gamma_1, \ldots, \gamma_n$	
Expressions	e	$::=$	x	term variables
		$\mid$	$e_1\ e_2$	applications
		$\mid$	$\lambda x.e$	abstractions
		$\mid$	$e\ \mathtt{d}$	dictionary applications
		$\mid$	$\lambda \mathtt{v}.e$	dictionary abstractions
		$\mid$	$\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$	local definitions
Dictionary constructors	χ			
Dictionaries	$\mathtt{d}$	$::=$	$\mathtt{v}$	dictionary variables
		$\mid$	$\backslash\ \mathtt{d}_1 \ldots \mathtt{d}_n$	dictionary construction

Figure 5.1: Abstract Syntax of CP

5.2 CP: The Target Language

This section describes the target language of our translation semantics, a version of polymorphic λ–calculus that includes explicit constructs for handling dictionary expressions. We called the system CP, intended as a mnemonic for "Constrained Polymorphic λ-calculus." The surface syntax of CP is designed to be very similar to that of Mini-Haskell$^+$, although semantically Mini-Haskell$^+$ is a proper sublanguage of CP.

5.2.1 Syntax of Types and Expressions

Figure 5.1 presents CP's syntax of types and expressions. Compared to Mini-Haskell$^+$, there are two major differences. First, CP has a more expressive set of types. Follow-

ing Mini-Haskell$^+$, quantified type variables may be associated with class constraints Γ, but there is no distinction of simple types and type schemes, since quantification of type variables is no longer restricted to be outermost. Thus types such as $(\forall \mathtt{a}::\mathtt{Eq}.\mathtt{a} \rightarrow \mathtt{a}) \rightarrow \mathtt{Int}$ are valid in CP. In other words, functions may take arguments of polymorphic and/or overloaded types. Section 6.3 explores this feature to relate a group of CP expressions whose types are ordered by the instantiation ordering ($\preceq_C$) defined in Section 3.3.1.

Second, CP has additional abstraction and application constructs for dictionary expressions. Overloaded Mini-Haskell$^+$ functions will be translated to dictionary abstractions in CP while occurrences of overloaded Mini-Haskell$^+$ functions will become dictionary applications in CP. With such explicit use of dictionary expressions, the order of class constraints on a type variable in the type of an object (and hence the order of dictionary parameters taken by an overloaded value) can no longer be ignored. Indeed, expressions such as $(e\ \mathtt{d}_1)\mathtt{d}_2$ may become meaningless if replaced by $(e\ \mathtt{d}_2)\mathtt{d}_1$. Thus all notions defined in previous chapters using sets must be replaced by those using *lists*, e.g., lists of class, lists of instance assumptions (contexts).

The set of free term and dictionary variables in an expression e are defined in an obvious manner and denoted by $\mathrm{fv}(e)$ and $\mathrm{dv}(e)$ respectively. As usual, we use the notation $[e'/x]e$ to represent substitution of an expression e' for the free occurrences of a variable x in an expression e, and assume the standard use of renaming to avoid variable capture.

5.2.2 Dictionary Expressions

A dictionary expression is either a dictionary variable or a construction obtained from an instance declaration similar to that of Mini-Haskell$^+$. After introducing some notational conventions for writing objects that involve dictionary expressions, this section describes CP's program declarations and how dictionaries are synthesized through a

straightforward extension of the instance entailment system given in Section 3.1.

Notations

Since we will mostly deal with multiple dictionary expressions grouped into separate lists, it is useful to introduce some notations for writing lists of dictionary expressions. In particular, fonts are used to distinguish dictionary expressions of different units, as detailed by the following table:

Object	expression	abbreviation
List of dictionary variables	$\mathtt{v}_1, \ldots, \mathtt{v}_n$	v
List of list of dictionary variables	$v_1, \ldots, v_m$	$\mathbf{v}$
List of dictionary expressions	$\mathtt{d}_1, \ldots, \mathtt{d}_n$	d
List of list of dictionary expressions	$d_1, \ldots, d_m$	$\mathbf{d}$
Dictionary abstraction	$\lambda\mathtt{v}_1 \ldots . \lambda\mathtt{v}_m . e$	$\lambda v . e$
Dictionary application	$(\ldots(e\,\mathtt{d}_1)\ldots)\mathtt{d}_n$	$e \cdot d$

We also use $\mathtt{u}$ and $\mathtt{w}$ to denote dictionary variables. Note that we only deal with lists without repetitive elements since each dictionary corresponds to a class constraint. Concatenation of lists is simply expressed by juxtaposition, e.g., $\mathbf{vw}$. In some situations, to emphasize that we are concatenating two disjoint lists, we use $\oplus$ as the concatenating operator, e.g., $\mathbf{v} \oplus \mathbf{w}$.

The empty list is denoted by ϵ. The *length* of a list is denoted by $|l|$. We use $\downarrow k$ for indexing the kth element in a list object, e.g., $v{\downarrow}k = \mathtt{v}_k$. A list l is a ***sublist*** of another list l', written $l \sqsubseteq l'$, iff $\forall i \in 1..|l|,\ l{\downarrow}i = l'{\downarrow}k$ for some $1 \leq k \leq |l'|$. Two lists l and l' are isomorphic, written $l \cong l'$, iff $l \sqsubseteq l'$ and $l' \sqsubseteq l$.

In addition, to describe the dictionary construction scheme, we introduce some notations for combining instance judgements with dictionary expressions. Given the

setting:

Γ_i	=	$\gamma_{i,1}, \ldots, \gamma_{i,n_i}$	Lists of type classes
v_i	=	$\mathbf{v}_{i,1}, \ldots, \mathbf{v}_{i,n_i}$	Lists of dictionary variables
$\mathbf{v}$	=	$v_1, \ldots, v_m$	List of list of dictionary variables
C	=	$\alpha_1::\Gamma_1, \ldots, \alpha_m::\Gamma_m$	Context—List of instance assumptions

we define *dictionary-augmented contexts* as follows:

$$\mathbf{v}{:}C \stackrel{def}{=} v_1{:}(\alpha_1::\Gamma_1), \ldots, v_m{:}(\alpha_m::\Gamma_m)$$

In other words, we pair off lists of dictionary variables v_i and instance assumptions $\alpha_i::\Gamma_i$. The notation

$$v{:}(\alpha::\Gamma)$$

expresses that, for all i, $1 \leq i \leq |\Gamma|$, dictionary variable $v{\downarrow}i$ is associated with the instance assumption $\alpha :: \Gamma{\downarrow}i$—namely, one dictionary for one class constraint. And we often overload the operations involving simple contexts to operate on augmented ones, e.g., membership test $v{:}(\alpha::\Gamma) \in \mathbf{v}{:}C$ and context restriction $(\mathbf{v}{:}C)\backslash_{v:(\alpha::\Gamma)}$. When there is no particular need to explicitly mention the dictionary variables associated with an augmented context we will often write just C in place of $\mathbf{v}{:}C$.

Similarly, we define dictionary-augmented instance predicates as follows:

$$\mathbf{d}{:}P \stackrel{def}{=} d_1{:}(\tau_1::\Gamma_1). \ \ldots \ , d_m{:}(\tau_m::\Gamma_m)$$

where

d_i	=	$\mathtt{d}_{i,1}, \ldots, \mathtt{d}_{i,n_i}$	Lists of dictionaries
$\mathbf{d}$	=	$d_1, \ldots, d_m$	List of list of dictionaries
P	=	$\tau_1::\Gamma_1, \ \ldots \ , \tau_m::\Gamma_m$	List of instance predicates

Program Declarations

Like Mini-Haskell$^+$, a CP program consists of a sequence of declarations followed by an expression. As illustrated in the informal presentation, these declarations introduce overloaded operators (as selectors) and their dictionary implementations to be used in the main expression. To simplify the presentation of the translation, however, we make the syntax of CP declarations very similar to that of Mini-Haskell$^+$. In particular, the declarations are "sugared" as class and instance declarations that declare overloaded operators and define their implementations respectively, as detailed below:

$$
\begin{array}{lrcl}
\text{Programs} & p & ::= & \texttt{class}\ \alpha{::}\gamma\ \texttt{where}\ x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\ \texttt{in}\ p \\
& & \mid & \chi{:}\ \texttt{inst}\ \mathbf{v}{:}C \Rightarrow \tau{::}\gamma\ \texttt{where}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ p \\
& & \mid & e
\end{array}
$$

Indeed, the syntax is closely modeled on that of Mini-Haskell$^+$. What is new here is the additional dictionary "annotations": Each instance declaration is assigned a dictionary constructor χ, and the context C is paired with matching dictionary variables $\mathbf{v}$, which may occur in the method expressions e_i.

The informal meaning of these declarations is as follows: a class declaration introduces a selector definition and an instance declaration defines a dictionary using the specified dictionary constructor. In the simplified case where a class introduces only one operator, every overloaded operator is simply the identity function and a dictionary is just a function. Essentially, these declarations are merely syntactic sugar for global definitions. For example, an instance declaration stands for a binding of a dictionary constructor:

$$\texttt{let}\ \chi\ \mathbf{v} = \langle e_1, \ldots, e_n \rangle\ \texttt{in}\ p$$

$$\Gamma) \quad \frac{\forall i \in 1..|\Gamma| \ (\ \mathbf{v}{:}C \Vdash d{\downarrow}i{:}(\tau :: \Gamma{\downarrow}i)\ )}{\mathbf{v}{:}C \Vdash d{:}(\tau{::}\Gamma)}$$

$$\alpha) \quad \frac{v{:}(\alpha{::}\Gamma) \in \mathbf{v}{:}C}{\mathbf{v}{:}C \Vdash v{\downarrow}i : (\alpha :: \Gamma{\downarrow}i)} \quad (i = 1, \ldots, |\Gamma|)$$

$$\tau) \quad \frac{\mathbf{v}{:}C \Vdash \mathbf{d} : \langle \tau_i{::}\Gamma_i \rangle}{\mathbf{v}{:}C \Vdash \chi\ \mathbf{d} : (\tau{::}\gamma)} \quad (\ \chi : \ulcorner\, \texttt{inst}\ \mathbf{v}'{:}\langle \tau_i{::}\Gamma_i \rangle \Rightarrow \tau{::}\gamma\ \urcorner \in \Sigma\ )$$

Figure 5.2: Augmented Instance Entailment System

However, by making their surface syntax similar to that of Mini-Haskell$^+$, we get a simpler description of the translation.

Instance Entailment and Dictionary Construction

An important step of our translation scheme is dictionary construction whereby overloading is made explicit. In CP, as an advantage of the syntax similarity discussed above, the set of instance declarations Σ in a program forms the basis for both instance constraint inference and dictionary construction. This is easily achieved by extending the inference rules for Mini-Haskell$^+$ instance entailment given in Section 3.1 such that the satisfication of an instance predicate is witnessed by a list of matching dictionaries. More specifically, given Σ, the extended inference rules presented in Figure 5.2 allow us to deduce augmented instance judgements of the form

$$\mathbf{v}{:}C \Vdash d : (\tau{::}\Gamma),$$

which assert that, from the augmented context $\mathbf{v}{:}C$, it follows that type τ is an instance of class list Γ as witnessed by dictionary list d. If $\mathbf{d}$ is a list of dictionaries of

proper length for a list of instance predicates $\langle \tau_i :: \Gamma_i \rangle_{i=1,n}$, then we use the notation

$$\mathbf{v}{:}C \Vdash \mathbf{d} : \langle \tau_i :: \Gamma_i \rangle$$

to represent

$$\forall i \in 1..n \ (\ \mathbf{v}{:}C \Vdash \mathbf{d}{\downarrow}i{:}(\tau_i :: \Gamma_i) \ )$$

This notation is used in the rule (τ) above to synthesize parameterized dictionaries.

The augmented instance entailment system forms the complete engine of overloading resolution: Any references to an overloaded operator in an expression is checked by the system and, in addition, if validated, a proper implementation is supplied by the system in the form of dictionaries.

The following properties of the augmented system are easily established.

- **(dvars)** If $\mathbf{v}{:}C \Vdash d{:}(\tau :: \Gamma)$, then $\mathrm{dv}(d) \subseteq \mathrm{dv}(\mathbf{v})$.
- **(weaken)** If $\mathbf{v}{:}C \Vdash d{:}(\tau :: \Gamma)$, then $\mathbf{v} : C \oplus \mathbf{v}' : C' \Vdash d{:}(\tau :: \Gamma)$.
- **(strengthen)** If $\mathbf{v}_1 : C_1 \oplus v : (\alpha :: \Gamma') \oplus \mathbf{v}_2 : C_2 \Vdash d{:}(\tau :: \Gamma)$ and $\alpha \notin \mathrm{tv}(\tau) \cup \mathrm{tv}(\Gamma)$, then $\mathbf{v}_1 : C_1 \oplus \mathbf{v}_2 : C_2 \Vdash d{:}(\tau :: \Gamma)$.
- **(swap)** If $\mathbf{v}_1 : C_1 \oplus \mathbf{v}_2 : C_2 \Vdash d{:}(\tau :: \Gamma)$, then $\mathbf{v}_2 : C_2 \oplus \mathbf{v}_1 : C_1 \Vdash d{:}(\tau :: \Gamma)$.

The lemma of transitivity under substitution presented in Section 3.1 extends naturally to the augmented system.

Lemma 5.1 *For any substitution S, augmented contexts $\mathbf{v}{:}C$ and $\mathbf{v}'{:}C'$, if $\mathbf{v}{:}C \Vdash d : (\tau :: \Gamma)$ and $\mathbf{v}'{:}C' \Vdash \mathbf{d}' : SC$, then $\mathbf{v}'{:}C' \Vdash [\mathbf{d}'/\mathbf{v}]d : S(\tau :: \Gamma)$.*

Uniqueness of Dictionary Construction

Since a dictionary constructed from the instance declarations implements certain overloaded operators, we need to impose, as before, certain restrictions on instance

declarations to avoid ambiguity. In fact, the CP programs we will consider are the translations of valid Mini-Haskell$^+$ programs, and hence maintain all the restrictions stated in Section 4.1.2 for Mini-Haskell$^+$. In particular, for every pair of type and class constructor (κ, c), there is at most one instance declaration of the form χ : **inst** $C \Rightarrow \kappa\,\tau'::c\,\tau$ **where** ... . Furthermore, we require that the dictionary constructor associated with an instance declaration be unique within a program. It is then easy to show that there is at most one dictionary that makes a type an instance of a particular class within a program:

Lemma 5.2 *When the set of instance declarations Σ in a program satisfies the two restrictions discussed above, then the augmented instance entailment system admits unique construction of dictionaries. In other words, if* $\mathbf{v}{:}C \Vdash d{:}(\tau{::}\Gamma)$ *and* $\mathbf{v}{:}C \Vdash d'{:}(\tau{::}\Gamma)$, *then* $d \equiv d'$.

5.2.3 Typing Rules for CP

The typing rules for CP expressions and declarations are given in Figure 5.3 and Figure 5.4 respectively. Similar to the case of instance entailment, these rules are derived from those of Mini-Haskell$^+$ (Section 3.2) by extending them with support for dictionary expressions. Indeed, if we remove all the dictionary augmentations, we obtain the same sets of typing rules. Note that, as mentioned earlier, we omit the list of dictionary variables from the augmented context when they are not explicitly referenced.

5.3 Translating Mini-Haskell$^+$ to CP

This section presents the formal definition of the translation scheme. We show that every well-typed Mini-Haskell$^+$ program has a CP translation and all translations obtained in this way are well-typed in CP. Similar schemes

$$(var) \quad \frac{A(x) = \sigma}{A,\ C \vdash x : \sigma}$$

$$(\forall - E) \quad \frac{A,\ C \vdash e : \forall \alpha :: \Gamma . \sigma \qquad C \Vdash d : (\tau :: \Gamma)}{A,\ C \vdash e \cdot d : [\tau/\alpha]\ \sigma}$$

$$(\forall - I) \quad \frac{A,\ C_1 \oplus v : (\alpha :: \Gamma) \oplus C_2 \vdash e : \sigma}{A,\ C_1 C_2 \vdash \lambda v.e : \forall \alpha :: \Gamma . \sigma} \qquad \alpha \notin \mathrm{fv}(A) \cup \mathrm{reg}(C_1 C_2)$$

$$(\lambda - E) \quad \frac{A,\ C \vdash e_1 : \sigma' \to \sigma \qquad A,\ C \vdash e_2 : \sigma'}{A.\ C \vdash e_1\ e_2 : \sigma}$$

$$(\lambda - I) \quad \frac{A.x : \sigma',\ C \vdash e : \sigma}{A,\ C \vdash \lambda x.e : \sigma' \to \sigma}$$

$$(let) \quad \frac{A,\ C \vdash e_1 : \sigma' \qquad A.x : \sigma',\ C \vdash e_2 : \sigma}{A,\ C \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \sigma}$$

Figure 5.3: Typing Rules for CP Expressions

The translation scheme is based on the similarities between the typing derivations in Mini-Haskell+ and CP. As mentioned in the previous section, every Mini-Haskell+ type can be treated as a CP type. Moreover, the typing rules of Mini-Haskell+ are just a restricted version of the rules for CP, except that typing derivations in the latter involve augmented contexts rather than simple contexts and require explicit dictionary application and abstraction in the rules $(\forall - elim)$ and $(\forall - intro)$ respectively. Based on these observations, we can easily establish a correspondence between Mini-Haskell+ and CP typing derivations using two auxiliary functions: The first

$$(class)\quad \frac{A.x{:}\forall_{\mathrm{tv}(\gamma)}\forall\alpha{::}\{\gamma\}.\sigma,\ C \vdash p : \sigma'}{A,\ C \vdash (\texttt{class}\ \alpha{::}\gamma\ \texttt{where}\ x{:}\sigma\ \texttt{in}\ p) : \sigma'}$$

$$(inst)\quad \frac{A,\ C \vdash x{:}\forall\alpha{::}\{\gamma\}.\sigma \quad A,\ \mathbf{v}{:}C \oplus \mathbf{v}'{:}C' \vdash e : [\tau/\alpha]\sigma \quad A,\ C \vdash p : \sigma'}{A,\ C \vdash (\chi :\ \texttt{inst}\ \mathbf{v}'{:}C' \Rightarrow \tau{::}\gamma\ \texttt{where}\ x = e\ \texttt{in}\ p) : \sigma'}$$

Figure 5.4: Typing Rules for CP Declarations

function *Cxt* maps an augmented context to the corresponding simple context:

$$\begin{aligned} Cxt(\mathbf{v}_1{:}C \oplus \mathbf{v}'{:}C') &= Cxt(\mathbf{v}{:}C) \uplus Cxt(\mathbf{v}'{:}C') \\ Cxt(v{:}(\alpha{::}\Gamma)) &= \{\alpha{::}\Gamma\} \\ Cxt\ \emptyset &= \emptyset \end{aligned}$$

The second function *Erase* maps CP expressions with explicit overloading to Mini-Haskell$^+$ expressions by eliminating all occurrences of dictionary expressions:

$$\begin{aligned} Erase(x) &= x \\ Erase(e_1\ e_2) &= Erase(e_1)\ Erase(e_2) \\ Erase(\lambda x.e) &= \lambda x.Erase(e) \\ Erase(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) &= \texttt{let}\ x = Erase(e_1)\ \texttt{in}\ Erase(e_2) \\ Erase(e\,\mathtt{d}) &= Erase(e) \\ Erase(\lambda \mathbf{v}.e) &= Erase(e) \end{aligned}$$

Analogous functions are defined in [Jones, 1992a] to investigate similar problems. In an earlier work, [Mitchell and Harper, 1988] introduced a *type erasure* function to explain the implicit polymorphism of ML-like languages in terms of the explicit polymorphism of the polymorphic lambda calculus [Reynolds, 1974].

The correspondence between Mini-Haskell$^+$ and CP can now be formally described by the following theorem:

Theorem 5.3 *If* $A,\ C \vdash e : \sigma$ *in Mini-Haskell*$^+$*, then there is a CP expression* e' *and an augmented context* $\mathbf{v}'{:}C'$ *such that* $C = Cxt(\mathbf{v}'{:}C'),\ e = Erase(e')$ *and* $A,\ \mathbf{v}'{:}C' \vdash e' : \sigma$ *using a derivation of the same structure.*

In other words, every well-typed Mini-Haskell$^+$ expression has a well-typed CP translation that can be obtained from its typing derivation.

The proof is straightforward, using induction on the length of $A,\ C \vdash e : \sigma$. We thereby define the expression e' in the statement of the theorem to be a *translation* of e and use the notation $A,\ C \vdash e \rightsquigarrow e' : \sigma$ to refer to a translation of an expression in a specific setting. Note that, in general, a Mini-Haskell$^+$ expression will have many distinct translations in any given setting, each corresponding to a different derivation of $A,\ C \vdash e : \sigma$ in Mini-Haskell$^+$. The issue of well-definedness will be addressed in the next chapter.

This theorem also suggests a more succinct way to describe the translations of Mini-Haskell$^+$ expressions—i.e., by combining the typing rules of Mini-Haskell$^+$ and CP expressions, as illustrated in Figure 5.5. It is easy to show that $A,\ C \vdash e \rightsquigarrow e' : \sigma$ according to the original definition of translations above if, and only if, the same judgement can be derived from these rules.

Since the method defined in an instance declaration is also an expression, we can easily generalize the correspondence result to program declarations. Specifically, we extend the definition of *Erase* to declarations as follows:

$$Erase(\texttt{class}\ \alpha{::}\gamma\ \texttt{where}\ x{:}\sigma\ \texttt{in}\ p) = \texttt{class}\ \alpha{::}\gamma\ \texttt{where}\ x{:}\sigma\ \texttt{in}\ Erase(p)$$

$$Erase(\chi : \texttt{inst}\ \mathbf{v}'{:}C' \Rightarrow \tau'{::}\gamma\ \texttt{where}\ x = e\ \texttt{in}\ p) =$$

$$\texttt{inst}\ Cxt(\mathbf{v}'{:}C') \Rightarrow \tau'{::}\gamma\ \texttt{where}\ x = Erase(e)\ \texttt{in}\ Erase(p)$$

$$(var) \quad \frac{A(x) = \sigma}{A,\ C \vdash x \rightsquigarrow x\ :\sigma}$$

$$(\forall\text{-}E) \quad \frac{A,\ C \vdash e \rightsquigarrow e'\ :\forall\alpha::\Gamma.\sigma \qquad C \Vdash d{:}(\tau::\Gamma)}{A,\ C \vdash e \rightsquigarrow e'{\cdot}d\ :[\tau/\alpha]\,\sigma}$$

$$(\forall\text{-}I) \quad \frac{A,\ C_1 \oplus v{:}(\alpha::\Gamma) \oplus C_2 \vdash e \rightsquigarrow e'\ :\sigma}{A,\ C_1C_2 \vdash e \rightsquigarrow \lambda v.e'\ :\forall\alpha::\Gamma.\sigma} \quad \alpha \notin \mathrm{fv}(A) \cup \mathrm{reg}(C_1C_2)$$

$$(\lambda\text{-}E) \quad \frac{A,\ C \vdash e_1 \rightsquigarrow e'_1\ :\tau' \to \tau \qquad A,\ C \vdash e_2 \rightsquigarrow e'_2\ :\tau'}{A,\ C \vdash e_1\ e_2 \rightsquigarrow e'_1\ e'_2\ :\tau}$$

$$(\lambda\text{-}I) \quad \frac{A.x{:}\tau',\ C \vdash e \rightsquigarrow e'\ :\tau}{A,\ C \vdash \lambda x.e \rightsquigarrow \lambda x.e'\ :\tau' \to \tau}$$

$$(let) \quad \frac{A,\ C \vdash e_1 \rightsquigarrow e'_1\ :\sigma \qquad A.x{:}\sigma,\ C \vdash e_2 \rightsquigarrow e'_2\ :\tau}{A,\ C \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \rightsquigarrow (\texttt{let}\ x = e'_1\ \texttt{in}\ e'_2)\ :\tau}$$

Figure 5.5: Typing & Translation Rules for Mini-Haskell$^+$ Expressions

Then the previous theorem naturally extends to the following result.

Theorem 5.4 *If* $A,\ C \vdash p : \sigma$ *in Mini-Haskell*$^+$*, then there is a CP program* p' *and an augmented context* $\mathbf{v}'{:}C'$ *such that* $C = Cxt(\mathbf{v}'{:}C')$, $p = Erase(p')$ *and* $A,\ \mathbf{v}'{:}C' \vdash p' : \sigma$ *using a derivation of the same structure.*

Such a correspondence effectively specifies a translation semantics for Mini-Haskell$^+$. In the next chapter, we will extend the syntax-directed system and the type reconstruction algorithm given in earlier chapters to include the calculation of translations, and deal with the situation where a single expression has multiple translations.

Chapter 6

Ambiguity and Coherence

In this chapter we address the problem of overloaded-operator ambiguity. As mentioned in Chapter 2, despite the at-most-one restriction on instance declarations, there are situations under which ambiguous uses of overloading can occur, and the compiler needs to detect them. Indeed, the problem manifests itself in our translation semantics when there are many different derivations for a single typing judgement, which may in turn yield semantically distinct translations; choosing one would give different results than the other. Our aim here is to develop conditions that are sufficient to detect such ambiguous situations.

More specifically, as a result of such ambiguous expressions in Haskell as well as our parametric extension, we cannot hope to establish a general *coherence* theorem [Breazu *et al.*, 1989] (a property referring to translation semantics in which all derivations of a given typing judgement yield the same meaning). Instead, we derive some simple syntactic conditions that are sufficient to exclude those undesirable expressions, and thus identify a collection of expressions for which the coherence property can be established. Like Haskell, the conditions are based on the principal type computed by our type reconstruction algorithm for any given expression. Essentially, if an expression's principal type is *unambiguous* in a specific sense, then all its translations

will have the same meaning.

To begin with, it is necessary to specify when two translations are equivalent. After illustrating the incoherence problem, we develop a typed equational theory for CP expressions whereby we can formally establish the equivalence between translations. Then, following [Jones, 1992a], we define *conversions* to relate different translations of an overloaded expression based on their types. In particular, such conversions are CP expressions derived from a type scheme and its generic instances in a way that, when applied to a translation of an expression, they repackage the dictionaries involved to yield another translation whose type is less general.

Based on the notion of conversions, we define *principal translations* along the lines of principal types. We extend our type reconstruction algorithm to include the calculation of translations and show that, analogous to the principal type property, the extended type inferencer computes the most general translation for every well-typed expression. In other words, any translation of an expression can be obtained by applying a conversion to its principal translation. Consequently, the equivalence of two translations at a given type is determined by the equivalence of the conversions from which they are derived. We show that when an expression's principal type is unambiguous, the conversions that can be derived from the type and any of its generic instances are all equivalent, thereby establishing the conditional coherence result.

Detailed proofs for the results of this chapter are included in Appendix A. Most of them are extensions of earlier results from Chapter 3 and Chapter 4.

6.1 The Coherence Problem

This section motivates the problem of overloaded-operator ambiguity and identifies it with the incoherence of our translation semantics.

Usually compilers rely on type information to resolve overloaded operators. For

polymorphic languages such as Haskell, overloading resolution is complicated by the presence of type variables. As described in the preceding chapters, constraints on type variables and explicit dictionary abstractions are used to handle unresolved overloaded operators. When demanded, these operators are properly resolved according to the types at which they are used. As long as the type that instantiates a type variable satisfies the associated constraints, unique resolution is guaranteed by the restrictions imposed on instance declarations (Lemma 5.2).

There are, however, situations under which the type inferencer cannot determine the suitable type to instantiate a constrained type variable and is therefore unable to supply the proper dictionary to resolve a particular occurrence of an overloaded operator. Arbitrary instantiations of such type variables may lead to inconsistent resolutions and thus the ambiguity problem.

As an example, consider the following class `Parsable` that declares two overloaded operations **parse** and **unparse**, which convert strings to/from values of a certain type:

```
class  a::Parsable  where
   parse   : String -> a
   unparse : a -> String
```

Now suppose that only `Int` and `Float` are instances of `Parsable`. Then the expression **unparse** (**parse** "123") is ambiguous: The composition of **parse** and **unparse** creates an intermediate value, **parse** "123", whose type, a type variable, is constrained by class `Parsable`, but does not appear in the type of the whole expression, **String**. As a result, even though the final type (`String`) is clear, the type inferencer is not able to determine the intermediate type via unification; instantiating it to **Int** or **Float** will give different results: "123" and "123.0" respectively.

Our translation semantics exploits an expression's typing derivations to resolve overloaded operators; it maps Mini-Haskell$^+$ typing derivations into CP typing derivations whereby overloading is made explicit. As such, ambiguity can occur when there

are many different derivations for a single typing judgement, which in turn yield many semantically distinct translations. Indeed, the ambiguity in the preceding example can be described in this manner: There are two ways to derive the typing judgement $\vdash$ `unparse (parse "123") : String`, one using integer `parse/unparse` functions and one floating-point `parse/unparse` functions, and consequently two translations which are clearly not equivalent:

`intUnparse (intParse "123")` and `floatUnparse (floatParse "123")`

In general, ambiguity arises in our translation semantics when it is possible to get, from derivations $A,\ C \vdash\ e \rightsquigarrow e_1 : \sigma$ and $A,\ C \vdash\ e \rightsquigarrow e_2 : \sigma$, translations e_1 and e_2 of a Mini-Haskell$^+$ expression e that are not equivalent.

The existence of such ambiguous expressions indicates that our translation semantics is not coherent, i.e., the meaning associated with a typed expression depends on the way that its type is derived. As a result, the mapping from expressions to translations is not well-defined. Indeed, ambiguous expressions do not have well-defined semantics and must be eliminated. A proposal to make the translation semantics coherent by restricting the type class mechanism have been made [Wadler, 1990b], but further study is needed to assess its feasibility. Instead, following Haskell, we choose to develop conditions that are sufficient to exclude those expressions and thus ensure that the semantics of an expression is well-defined.

6.2 Equality of Translations

As a first step, we need to specify formally what it means for two translations to be equivalent. This section defines a *typed equational theory* for CP expressions; two translations of an overloaded expression are said to be equivalent if they are provably equal within the theory.

$$(\alpha) \qquad \frac{y \notin \mathrm{fv}(\lambda x.e)}{A,\ C \vdash (\lambda x.e) = (\lambda y.[y/x]e) : \sigma}$$

$$(\alpha_d) \qquad \frac{\mathtt{w} \notin \mathrm{fv}(\lambda \mathtt{v}.e)}{A,\ C \vdash (\lambda \mathtt{v}.e) = (\lambda \mathtt{w}.[\mathtt{w}/\mathtt{v}]e) : \sigma}$$

$$(\alpha\text{-let}) \qquad \frac{y \notin \mathrm{fv}(e) \cup \mathrm{fv}(e')}{A,\ C \vdash (\mathtt{let}\ x = e'\ \mathtt{in}\ e) = (\mathtt{let}\ y = e'\ \mathtt{in}\ [y/x]e) : \sigma}$$

$$(\beta) \qquad A,\ C \vdash (\lambda x.e)e' = [e'/x]e : \sigma$$

$$(\beta_d) \qquad A,\ C \vdash (\lambda v.e)d = [d/v]e : \sigma$$

$$(\beta\text{-let}) \qquad A,\ C \vdash (\mathtt{let}\ x = e'\ \mathtt{in}\ e) = [e'/x]e : \sigma$$

$$(\eta_d) \qquad \frac{v \notin \mathrm{dv}(e)}{A,\ C \vdash (\lambda v.e \cdot v) = e : \sigma}$$

Figure 6.1: Equation rules for CP expressions, I

The theory comprises equational judgements of the form:

$$A,\ C \vdash e = e' : \sigma$$

where we assume that e and e' have type σ in the setting determined by type assumptions A and instance assumptions C. Intuitively, the judgement $A,\ C \vdash e = e' : \sigma$ asserts that expressions e and e' denote the same element of type σ in environments that satisfy A and C. The implicit side-condition that both $A,\ C \vdash e : \sigma$ and $A,\ C \vdash e' : \sigma$ hold is necessary since only typed expressions and their translations are considered meaningful.

$$(sym) \quad \frac{A,\ C \vdash e = e'}{A,\ C \vdash e' = e\ :\sigma}$$

$$(trans) \quad \frac{A,\ C \vdash e = e'\ :\sigma \qquad A,\ C \vdash e' = e''\ :\sigma}{A,\ C \vdash e = e''\ :\sigma}$$

$$(var) \quad \frac{A(x) = \sigma}{A,\ C \vdash x = x\ :\sigma}$$

$$(app\text{-d}) \quad \frac{A,\ C \vdash e = e'\ :\forall\alpha{::}\Gamma.\sigma \qquad C \Vdash d{:}(\tau{::}\Gamma)}{A,\ C \vdash e{\cdot}d = \epsilon'{\cdot}d\ :[\tau/\alpha]\,\sigma}$$

$$(abs\text{-d}) \quad \frac{A,\ C_1 \oplus v{:}(\alpha{::}\Gamma) \oplus C_2 \vdash e = \epsilon'\ :\sigma}{A,\ C_1 C_2 \vdash \lambda v.e = \lambda v.\epsilon'\ :\forall\alpha{::}\Gamma.\sigma} \quad \alpha \notin \mathrm{fv}(A) \cup \mathrm{reg}(C)$$

$$(\mu) \quad \frac{A,\ C \vdash \epsilon_1 = \epsilon_1'\ :\sigma' \to \sigma \qquad A,\ C \vdash \epsilon_2 = \epsilon_2'\ :\sigma'}{A,\ C \vdash \epsilon_1\,\epsilon_2 = \epsilon_1'\,\epsilon_2'\ :\sigma}$$

$$(\xi) \quad \frac{A.x{:}\sigma',\ C \vdash e = e'\ :\sigma}{A,\ C \vdash \lambda x.e = \lambda x.\epsilon'\ :\sigma' \to \sigma}$$

$$(cong\text{-let}) \quad \frac{A,\ C \vdash e_1 = e'_1\ :\sigma \qquad A.x{:}\sigma,\ C \vdash e_2 = e'_2\ :\tau}{A,\ C \vdash (\texttt{let}\ x = \epsilon_1\ \texttt{in}\ \epsilon_2) = (\texttt{let}\ x = \epsilon'_1\ \texttt{in}\ \epsilon'_2)\ :\tau}$$

Figure 6.2: Equation rules for CP expressions, II

The axioms and inference rules of the theory are given in Figures 6.1 and 6.2. The axioms in Figure 6.1 include the familiar definitions of α-conversion and β-conversion and their extensions to dictionary-augmented expressions. Also included is a rule of η-conversion for removing unnecessary dictionary abstractions.

These axioms, except α-conversion, are often formulated as *reduction rules* by orienting them from left to right. As such, they have simpler side-condition of well-typedness since it can be shown that if the expression on the left has type σ then the reduced expression on the right also has type σ. This is a consequence of the *subject reduction property*—reduction preserves typing—which can be proved using standard techniques as in [Hindley and Seldin, 1986].

The second group of axioms and inference rules in Figure 6.2 makes the typed equality an equivalence relation and a congruence with respect to the expression formation operation. To make the equality an equivalence relation, we have included the symmetry rule (*sym*) and the transitivity rule (*trans*). On the other hand, there is no need to include the reflexivity rule since it is a direct consequence of the other rules, which are closely modeled on the original typing rules of Figure 5.3 to make the equality a congruence by allowing the equivalence of sub-expressions within a given expression.

One may observe from rule (*app*-d) that the congruence property does not include dictionaries. This is a consequence of Lemma 5.2. As discussed in Section 5.2.2, dictionary construction is unique within a program when the set of instance declarations in a program satisfies the restrictions mentioned therein. Hence there is no induced equivalence relation over dictionaries to be included in rule (*app*-d).

The following lemma states some useful properties of `let`-expressions that follow directly from rule (β-let).

Lemma 6.1 *For any CP expressions e_1, e_2 and e' and distinct term variable x and*

y such that $y \notin fv(e_1)$*:*

1. $A,\ C \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ [e'/x]e_2) = (\texttt{let}\ x = [e_1/x]e'\ \texttt{in}\ e_2) : \sigma$
2. $A,\ C \vdash \lambda y.(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) = (\texttt{let}\ x = e_1\ \texttt{in}\ \lambda y.\, e_2) : \sigma$
3. $A,\ C \vdash e'(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) = (\texttt{let}\ x = e_1\ \texttt{in}\ e'e_2) : \sigma$
4. $A,\ C \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)\, e' = (\texttt{let}\ x = e_1\ \texttt{in}\ e_2e') : \sigma$

To give a flavor of typed equational reasoning in our system, we include here the proof of the first property of this lemma. In particular, we lay out the equational deduction as follows:

$$\begin{aligned} A,\ C \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ [e'/x]e_2) &= [e_1/x]\,([e'/x]e_2) && (\beta\text{-let}) \\ &= [[e_1/x]e'/x]e_2 && (\textit{substitution}) \\ &= (\texttt{let}\ x = [e_1/x]e'\ \texttt{in}\ e_2) : \sigma && (\beta\text{-let}) \end{aligned}$$

Note that we have also used rules (*sym*) and (*trans*) in the deduction. Furthermore, the required side-condition on types is preserved by the intermediate steps: From the given hypothesis $A,\ C \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ [e'/x]e_2) : \sigma$ and the subject reduction property it follows that $A,\ C \vdash [e_1/x]([e'/x]e_2) : \sigma$. Hence the first application of (β-let) is justified. Similarly, another hypothesis $A,\ C \vdash (\texttt{let}\ x = [e_1/x]e'\ \texttt{in}\ e_2) : \sigma$ justifies the second application of (β-let).

Given the typed equality over CP expressions, our task in the subsequent sections is to derive conditions sufficient to guarantee that:

If $A,\ \mathbf{v}{:}C \vdash e \rightsquigarrow e_1 : \sigma$ and $A,\ \mathbf{v}{:}C \vdash e \rightsquigarrow e_2 : \sigma$, then $A,\ \mathbf{v}{:}C \vdash e_1 = e_2 : \sigma$.

6.3 Ordering and Conversion Functions

This section explores the generic instance ordering between type schemes ($\preceq_C$) to relate the translations of an overloaded expression. Following [Jones, 1992a], we

show that any two translations of an expression can be related by certain functions if their types are related by the ordering. Such functions are called *conversions* since they convert one translation to another, whose type is less general. Furthermore, the notion of conversions between translations naturally leads us to define *principal translations* along the lines of principal types. Later in this chapter we will show that the principal translation property holds for our translation system. As part of the technical development, we extend the definition of conversions to type assumption sets.

6.3.1 Conversions and Principal Translations

Given that each typing derivation for a Mini-Haskell$^+$ expression yields a type as well as a translation, and all the types that we can associate with an expression are generic instances of its principal type, it is conceivable to consider the relation between the translation obtained from the principal-type derivation and those from other derivations based on the relation between their types. Indeed, as we will show immediately, a proper relation between these translations can be established through a semantic interpretation of the ordering between their types.

Conversions are functions we use to relate translations; they convert a translation of a more general type to another translation of the same expression whose type is less general. Furthermore, they can be expressed in our system as CP terms. More formally, given a Mini-Haskell$^+$ expression e and two of its translations e_1 and e_2 obtained from the typing derivations $A, C \vdash e \rightsquigarrow e_1 : \sigma$ and $A, C \vdash e \rightsquigarrow e_2 : \sigma'$ with $\sigma' \preceq_C \sigma$, we are interested in functions K such that $A, C \vdash Ke_1 = e_2 : \sigma'$.

We have, therefore, the following characterizations of the conversions. First, it is clear that the type for such expressions is $\sigma \rightarrow \sigma'$, under A and C. Note that this type, in general, cannot be expressed as a Mini-Haskell$^+$ type scheme since it uses the richer structure of CP types. Second, from the translation semantics we know that

$Erase(e_1) = e = Erase(e_2)$. Since $Erase(Ke_1) = Erase(K)Erase(e_1)$ and $Ke_1 = e_2$, we need to ensure that $Erase(K)Erase(e_1) = Erase(e_1)$. An obvious choice is to require that $Erase(K)$ be equivalent to the identity function $id = \lambda x.x$.

The insight of [Jones, 1992a] is that we can derive from the definition of generic instance ordering a "canonical" conversion that suits our purpose. Such conversions, when applied to a translation of an overloaded expression, repackage the dictionaries involved to yield another translation whose type is less general. In our system, the idea is embodied in the following definition of conversions:

Definition 6.1 (Conversions) *Given type assumption set A and context $\mathbf{v}{:}C$. Suppose that $\sigma' = \forall\langle\alpha'_j{::}\Gamma'_j\rangle.\tau'$ and $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\tau$ and that none of the α'_j occurs free in σ or C. A CP expression K of type $\sigma \to \sigma'$ under A and $\mathbf{v}{:}C$ such that $Erase(K) = id$ is called a* conversion *from σ to σ' under A and $\mathbf{v} : C$, written $K{:}\sigma' \preceq_{A,\mathbf{v}:C} \sigma$, if there are types τ_i, dictionary variables $\mathbf{w}$, and dictionary expressions $\mathbf{d}$ such that*

- $\tau' = [\tau_i/\alpha_i]\,\tau$
- $\mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha'_j{::}\Gamma'_j\rangle \Vdash \mathbf{d}{:}[\tau_i/\alpha_i]\,\langle\alpha_i{::}\Gamma_i\rangle$ *and*
- $A,\ \mathbf{v}{:}C \vdash K = \lambda x.\lambda\mathbf{w}.\,x\cdot\mathbf{d}$

Since conversions contain no free term variables and hence only context $\mathbf{v}{:}C$ is significant in the setting of A and $\mathbf{v}{:}C$, we will drop the type assumption set A from the subscript of $\preceq_{A,\mathbf{v}:C}$. We will also omit the dictionary application symbol and write $x\,\mathbf{d}$ for $x\cdot\mathbf{d}$ in what follows.

It is straightforward to verify that $\lambda x.\lambda\mathbf{w}.\,x\,\mathbf{d}$ is itself a conversion from σ to σ' under A and $\mathbf{v}{:}C$; clearly $Erase(\lambda x.\lambda\mathbf{w}.\,x\,\mathbf{d}) = \lambda x.\,x$ and the following derivation establishes the required typing:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A.x{:}\sigma,\ \mathbf{v}{:}C \vdash x : \sigma}{A.x{:}\sigma,\ \mathbf{v}{:}C \vdash x : \forall\langle\alpha_i{::}\Gamma_i\rangle.\tau}\ \sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\tau}{A.x{:}\sigma,\ \mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha'_j{::}\Gamma'_j\rangle \vdash x\,\mathbf{d} : [\tau_i/\alpha_i]\,\tau}\ (\forall\text{-elim})}{A.x{:}\sigma,\ \mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha'_j{::}\Gamma'_j\rangle \vdash x\,\mathbf{d} : \tau'}\ \tau' = [\tau_i/\alpha_i]\,\tau}{A.x{:}\sigma,\ \mathbf{v}{:}C \vdash \lambda\mathbf{w}.\,x\,\mathbf{d} : \sigma'}\ (\forall\text{-intro}),\ \sigma' = \forall\langle\alpha'_j{::}\Gamma'_j\rangle.\tau'}{A,\ \mathbf{v}{:}C \vdash \lambda x.\lambda\mathbf{w}.\,x\,\mathbf{d} : \sigma \to \sigma'}\ (\lambda\text{-intro})$$

This canonical conversion works by repackaging the dictionaries involved. Also, as noted earlier, the types of conversion functions utilize the more expressive polymorphism provided by CP.

The following two lemmas state some properties of conversions that will be useful in subsequent work. The first one is pretty straightforward.

Lemma 6.2 *If* $K : \sigma' \preceq_{\mathbf{v}:C} \sigma$ *and* $\mathbf{v}{:}C \sqsubseteq \mathbf{u}{:}C'$, *then* $K : \sigma' \preceq_{\mathbf{u}:C'} \sigma$.

The second lemma shows when two conversions can be meaningfully composed.

Lemma 6.3 *If* $K' : \sigma'' \preceq_{\mathbf{v}:C} \sigma'$ *and* $K : \sigma' \preceq_{\mathbf{v}:C} \sigma$, *then* $(K' \circ K) : \sigma'' \preceq_{\mathbf{v}:C} \sigma$.

The introduction of conversions is a key step towards our goal. With the notion of conversion we can generalize the definition of principal types to that of *principal translations*, which are translations obtained from the principal-type derivations and from which all other translations can be derived. If the principal translation property holds for our system, our task of determining the equivalence of two translations is reduced to that of the two conversions derived between the principal translation and the respective translations, which is simpler since conversions are also CP expressions but with regular structures. The following definition formalizes the notion of principal translations:

Definition 6.2 (Principal translations) *Given* A, $\mathbf{v}{:}C$, *and* e, *we call* $e'{:}\sigma$ *a* principal translation *for* e *under* A *and* $\mathbf{v}{:}C$ *iff* $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e' : \sigma$, *and for every* σ', *if* $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e'' : \sigma'$ *and* $K : \sigma' \preceq_{\mathbf{v}:C} \sigma$, *then* $A, \mathbf{v}{:}C \vdash K\,e' = e'' : \sigma'$.

6.3.2 Conversions Between Type Assumption Sets

We have seen, in Section 3.3.1, the extension of the generic instance to an ordering between type assumption sets. Similarly, we can extend the definition of conversions to type assumption sets. The additional complexity here is that we need to express multiple conversions, one for each pair of types associated with a term variable in the respective type assumption sets. Since each of these conversions maps a term variable to an expression, we use *expression substitutions* to define the conversions between type assumption sets, as suggested by [Jones. 1992a].

As a simple example, consider the following two type assumption sets:

$$A' = \{ (==) : \forall \mathtt{a}::\mathtt{Eq}.\, \mathtt{List\ a} \rightarrow \mathtt{List\ a} \rightarrow \mathtt{Bool} \}$$

$$\text{and} \quad A = \{ (==) : \forall \mathtt{a}::\mathtt{Eq}.\, \mathtt{a} \rightarrow \mathtt{a} \rightarrow \mathtt{Bool} \}$$

Assuming that $\mathtt{w}:(\mathtt{a}::\mathtt{Eq}) \vdash (\mathtt{DEqList\ w}) : (\mathtt{List\ a} :: \mathtt{Eq})$, one possible conversion between A and A' would be a substitution that maps $(==)$ to $\lambda \mathtt{w}.\,(==)\,(\mathtt{DEqList\ w})$, but leaves other variables unchanged.

The following definition formalizes the idea:

Definition 6.3 (Conversions between type assumption sets) *A substitution K is a* conversion *from a type assumption set A to another type assumption set A' under context $\mathbf{v}{:}C$, written $K{:}A' \preceq_{\mathbf{v}:C} A$, if*

- *$dom\ A = dom\ A'$ and*
- *For each $x \in dom\ A$ there is a conversion $\lambda x.K(x) : A'(x) \preceq_{\mathbf{v}:C} A(x)$. On the other hand, if $x \notin dom\ A$, then $K(x) = x$.*

Note that since every conversion is a CP expression without any free term variables, it follows that the only free term variable that appears free in the expression of the form $K(x)$ is the variable x itself. Based on this observation, we can easily establish the following results:

Lemma 6.4 *If* $K{:}A' \preceq_{\mathbf{v}:C} A$, *then*

1. $K(\lambda x.\, e) = \lambda x.\, K_x\, e$,
2. $K(\texttt{let } x = e_1 \texttt{ in } e_2) = (\texttt{let } x = Ke_1 \texttt{ in } K_x e_2)$,
3. $K_x : (A.x{:}\sigma) \preceq_{\mathbf{v}:C} (A'.x{:}\sigma)$ *for any* σ, *and*
4. $[e_1/x](K_x e_2) = (K[e_1/x])e_2$ *for any* e_1 *and* e_2

where K_x *stands for the substitution such that* $K_x(x) \equiv x$ *and* $K_x e \equiv Ke$ *for any expression* e *that* $x \notin fv(e)$.

The following lemma is another useful consequence of the definition of conversions between type assumption sets. It is easily proved using Lemma 6.2.

Lemma 6.5 *If* $K : A' \preceq_{\mathbf{v}:C} A$ *and* $\mathbf{v}{:}C \sqsubseteq \mathbf{u}{:}C'$, *then* $K : A' \preceq_{\mathbf{u}:C'} A$

6.4 Syntax-directed Translation

The next three sections follow the developments of Chapter 3 and Chapter 4 to extend our type inferencer to include the calculation of translations for any given expression. To begin with, we extend the syntax-directed typing rules of Section 3.3 to include the construction of translations. Figure 6.3 gives the extended inference rules.

Note that although the structure of a derivation $A,\ C \vdash' \ e \rightsquigarrow e' : \tau$ is uniquely determined by the syntactic structure of the expression e, it need not be the case for the translation e'. The reason is that in rule (var'), the dictionary expressions $\mathbf{d}$ are determined by the types τ_i we choose to instantiate the quantified type variables, and there may be distinct choices for such types and consequently distinct dictionary expressions. This is exactly where incoherence may occur in the translation semantics of Mini-Haskell$^+$.

$$(var') \quad \frac{A(x) = \forall\langle\alpha_i{::}\Gamma_i\rangle.\tau \qquad C \Vdash \mathbf{d} : ([\tau_i/\alpha_i]\,\langle\alpha_i{::}\Gamma_i\rangle)}{A,\ C \vdash' x \rightsquigarrow x\mathbf{d} : [\tau_i/\alpha_i]\tau}$$

$$(\lambda\text{-E'}) \quad \frac{A,\ C \vdash' e_1 \rightsquigarrow e'_1 : \tau' \to \tau \qquad A,\ C \vdash' e_2 \rightsquigarrow e'_2 : \tau'}{A,\ C \vdash' e_1\ e_2 \rightsquigarrow e'_1\ e'_2 : \tau}$$

$$(\lambda\text{-I'}) \quad \frac{A.x{:}\tau',\ C \vdash' e \rightsquigarrow e' : \tau}{A,\ C \vdash' \lambda x.e \rightsquigarrow \lambda x.e' : \tau' \to \tau}$$

$$(let') \quad \frac{A,\ \mathbf{v}'{:}C' \vdash' e_1 \rightsquigarrow e'_1 : \tau_1 \qquad A.x{:}\sigma,\ \mathbf{v}{:}C \vdash' e_2 \rightsquigarrow e'_2 : \tau_2}{A,\ C \vdash' (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\mathtt{let}\ x = \lambda\mathbf{w}.e'_1\ \mathtt{in}\ e'_2) : \tau_2}$$

where $(\sigma,\ \mathbf{v}''{:}C'',\ \mathbf{w}) = gen(\tau_1,\ A,\ \mathbf{v}'{:}C',\ \epsilon)$ and $\mathbf{v}''{:}C'' \sqsubseteq \mathbf{v}{:}C$

Figure 6.3: Syntax-directed Translation Rules

To accommodate dictionary abstractions in translating `let`-expressions, we extend the definition of function *gen* as follows:

$$\begin{aligned} gen\ (\sigma,\ A,\ \mathbf{v}{:}C,\ \mathbf{w}) \ = \ & \textbf{if } \exists\,(v{:}(\alpha{::}\Gamma)) \in \mathbf{v}{:}C \textbf{ and } \alpha \notin (\ \mathrm{tv}(A) \cup \mathrm{reg}(C)\ ) \\ & \textbf{then } gen\ (\forall\alpha{::}\Gamma.\sigma,\ A,\ (\mathbf{v}{:}C)\backslash_{v:(\alpha::\Gamma)},\ v\mathbf{w}) \\ & \textbf{else } (\sigma,\ \mathbf{v}{:}C,\ \mathbf{w}) \end{aligned}$$

In other words, now *gen* not only extracts generic type variables from the context but also accumulates the dictionary variables associated with those type variables.

The following three lemmas about the extended *gen* function are easily established.

Lemma 6.6 *If* $gen(\tau, A, \mathbf{v}{:}C, \epsilon) = (\sigma, \mathbf{v}'{:}C', \mathbf{w})$ *then* $\sigma = \langle\alpha_i{::}C\alpha_i\rangle_1^n.\tau$ *for some* $n \geq 0$ *such that* $\mathbf{w}{:}\langle\alpha_i{::}C\alpha_i\rangle \oplus \mathbf{v}'{:}C'$ *and* $dom(C') = C^*(tv\ A)$.

Lemma 6.7 *If* $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e' : \tau$ *and* $(\sigma, \mathbf{v}'{:}C', \mathbf{w}) = gen(\tau, A, \mathbf{v}{:}C, \epsilon)$, *then* $A, \mathbf{v}'{:}C' \vdash e \rightsquigarrow \lambda\mathbf{w}.\, e' : \sigma$.

Lemma 6.8 *Suppose that* $(\sigma, C_1, \mathbf{w}) = gen(\tau, A, C, \epsilon)$ *and* $\mathbf{u}{:}D$ *is disjoint from* $\mathbf{v}{:}C$. *Then* $(\sigma', C_1, \mathbf{v}') = gen(\tau, A, \mathbf{v}{:}C \oplus \mathbf{u}{:}D, \epsilon)$ *for some* σ', $\mathbf{v}'{:}C'$, *and* $\mathbf{w}'$ *such that*

$$C' = C_1, \quad \mathbf{w}' \cong \mathbf{uw}, \quad (\lambda x.\lambda\mathbf{w}'.\, x\,\mathbf{w}) : \sigma' \preceq \sigma \text{ and } (\lambda x.\lambda\mathbf{w}.\, x\mathbf{w}') : \sigma \preceq_{\mathbf{u}:D} \sigma'.$$

As in Chapter 3, our goal in the remainder of this section is to show that the set of syntax-directed translation rules is equivalent to the original set of translation rules given in Figure 5.5. By a straightforward induction, we can show that the syntax-directed system is sound with respect to the original one:

Theorem 6.9 *If* $A, \mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$ *then* $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e' : \tau$.

To show that the syntax-directed system is also as general as the original system, we need to develop a series of lemmas about the former. We begin with the following two lemmas that describe the interaction between the *gen* function and type substitutions under the common scenario $A, \mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$.

Lemma 6.10 *Let* $(\sigma, C', \mathbf{w}) = gen(\tau, A, \mathbf{v}{:}C, \epsilon)$ *and* $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\, \tau$. *Suppose that there exist dictionary expressions* $\mathbf{d}$ *such that* $\mathbf{u}{:}D \Vdash \mathbf{d} : SC$. *If* $gen(S\tau, SA, \mathbf{u}{:}D, \epsilon) = (\sigma', \mathbf{u}'{:}D', \mathbf{w}')$, *then*

$$(\lambda x.\lambda\mathbf{w}'.\, x\mathbf{d}') : \sigma' \preceq_{\mathbf{u}':D'} S\sigma.$$

where $\mathbf{d}'$ *is a sublist of* $\mathbf{d}$ *such that* $\mathbf{u}{:}D \Vdash \mathbf{d}' : S\langle\alpha_i{::}\Gamma_i\rangle$.

Lemma 6.11 *Let* $(\sigma, C_0, \mathbf{w}) = gen(\tau, A, \mathbf{v}{:}C, \epsilon)$. *If* $\mathbf{v}'{:}C' \Vdash \mathbf{d} : SC_0$, *then there exist a substitution* R, *an augmented context* $\mathbf{u}{:}D$ *and dictionary expressions* $\mathbf{d}'$ *such that*

$$RA = SA, \quad \mathbf{u}{:}D \Vdash \mathbf{d}' : RC \quad \text{and} \quad \mathbf{d}' \cong \mathbf{wd}.$$

Furthermore, if $gen(R\tau, RA, \mathbf{u}{:}D, \epsilon) = (\sigma', D', \mathbf{w}')$ *then*

$$S\sigma = \sigma', \quad D' \sqsubseteq C' \text{ and } \mathbf{w}' \cong \mathbf{w}.$$

The next group of lemmas states the properties of the syntax-directed translation; with the exception of Lemma 6.12, these results are extensions of the properties of the syntax-directed type system in Section 3.3.3 to address the calculation of translations.

Lemma 6.12 *If* $A, \mathbf{v} : C \vdash' e \rightsquigarrow e' : \tau$ *then* $dv(e') \subseteq \mathbf{v}$

Lemma 6.13 *If* $A, \mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$ *and* $\mathbf{v}{:}C \sqsubseteq \mathbf{v}'{:}C'$, *then* $A, \mathbf{v}'{:}C' \vdash' e \rightsquigarrow e' : \tau$.

Lemma 6.14 *If* $A, \mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$ *and* $\mathbf{v}'{:}C' \Vdash \mathbf{d} : SC$, *then* $SA, \mathbf{v}{:}C' \vdash' e \rightsquigarrow [\mathbf{d}/\mathbf{v}]e' : \tau$.

Lemma 6.15 *If* $A', \mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$ *and* $K : A' \preceq_{\mathbf{v}:C} A$, *then* $A, \mathbf{v}{:}C \vdash' e \rightsquigarrow e'' : \tau$ *with* $A, \mathbf{v}{:}C \vdash Ke' = e'' : \tau$.

Finally, using these lemmas, we can show that the syntax-directed translation system is complete with respect to the original one in the following sense:

Theorem 6.16 *If* $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e' : \sigma$, *then there is a context* $\mathbf{v}'{:}C'$, *a type* τ' *and an expression* e'' *such that* $\mathbf{v}{:}C \sqsubseteq \mathbf{v}'{:}C'$ *and* $A, \mathbf{v}'{:}C' \vdash' e \rightsquigarrow e'' : \tau'$. *Furthermore, if* $gen(\tau', A, \mathbf{v}'{:}C', \epsilon) = (\sigma', C'', \mathbf{w})$, *then* $A, \mathbf{v}{:}C \vdash K(\lambda \mathbf{w}.e'') = e' : \sigma$, *where* $K : \sigma \preceq_{\mathbf{v}:C} \sigma'$.

Therefore, any translation derived from the original set of translation rules can also be obtained by using the set of syntax-directed translation rules.

6.5 Unification and Dictionary Construction

Before we can extend our type reconstruction algorithm to include the calculation of translations, we need to develop some mechanisms to synthesize dictionaries. This section extends the unification algorithm given in Section 4.1.3 to incorporate dictionary construction. As discussed there, unification of types is associated with a context normalization process to ensure that the underlying context is properly preserved. This normalization sub-algorithm can be viewed as an implementation of the instance entailment system of Section 3.1. We have also shown in Section 5.2.2 that we can easily extend the instance entailment system to include dictionary construction using augmented judgements of the form $\mathbf{v}{:}C \Vdash \mathbf{d} : (\tau{::}\Gamma)$. Therefore, our main task here is to extend the normalization algorithm to implement the augmented instance entailment system.

Augmented Constrained Substitutions

We extend constrained substitutions defined in Section 4.1.1 with dictionary substitutions to handle dictionary construction. In addition to substitutions of type variables by types, we will also use dictionary substitutions given by maps from dictionary variables to dictionaries. More specifically, the extended unification algorithm operates on a type substitution as well as a dictionary substitution. During type reconstruction, as types get unified, along with the associated context normalization process, dictionaries will be constructed to replace dictionary variables involved in the translation being synthesized, thereby yielding more refined dictionary substitutions. At the end, we obtain the complete translation by applying the resulting dictionary substitution.

Using Θ to denote a dictionary substitution, we extend the definition of constrained substitutions as follows:

Definition 6.4 *An* augmented constrained substitution *is a triple* $(S, \mathbf{v}{:}\,C, \Theta)$ *where*

S is a substitution, $\mathbf{v}:C$ an augmented context, and Θ a dictionary substitution such that $C = SC$ and $\mathbf{v} = \Theta\mathbf{v}$.

Consequently, definitions derived from constrained substitutions have to be extended to include dictionary substitutions. The following definition extends the notion of context preserving to augmented constrained substitutions.

Definition 6.5 *An augmented constrained substitution $(S, \mathbf{v} : C, \Theta)$* preserves *another augmented constrained substitution $(S_0, \mathbf{v}_0 : C_0, \Theta_0)$ if there is a substitution S' and a dictionary substitution Θ' such that $S = S' \circ S_0$, $\Theta = \Theta' \circ \Theta_0$, and $\mathbf{v}{:}C \Vdash \Theta'\mathbf{v}_0 : S'C_0$. We write in this case $(S.\mathbf{v}{:}C, \Theta) \preceq (S_0, \mathbf{v}_0{:}C_0, \Theta_0)$.*

The augmented context-preserving unifiers and normalizers are similarly defined.

Definition 6.6 *An augmented constrained substitution $(S, \mathbf{v}{:}C, \Theta)$ is a*

(a) $(S_0, \mathbf{v}_0{:}C_0, \Theta_0)$-preserving unifier *of the type expressions τ and τ' if $S\tau = S\tau'$ and $(S, \mathbf{v}{:}C, \Theta) \preceq (S_0, \mathbf{v}_0{:}C_0, \Theta_0)$.*

(b) $(S_0, \mathbf{v}_0{:}C_0, \Theta_0)$-preserving normalizer *of an instance predicate set P if there exist dictionary expressions $\mathbf{d}$ such that $\mathbf{v} : C \Vdash \mathbf{d} : SP$ and $(S, \mathbf{v}{:}C, \Theta) \preceq (S_0, \mathbf{v}_0{:}C_0, \Theta_0)$.*

Augmented Unification Algorithm

Given the notion of augmented constrained substitutions, we can extend our unification algorithm to include dictionary construction. Figure 6.4 presents the augmented algorithm. A few words on our notations may be helpful here. As before, we have used simple contexts where augmented contexts are meant. Thus expression $C\alpha$ yields the class list Γ associated with α as well as the matching list of dictionary variables v. Such augmented class lists are denoted by $v : \Gamma$, but we may simply write Γ if there is no need to mention v.

$$
\begin{array}{l}
mgu : \tau \to \tau \to S \times C \times \Theta \to S \times C \times \Theta \\
mgn : \tau \to \Gamma \to S \times C \times \Theta \times d \to S \times C \times \Theta \times d \\
mgu\ \tau_1\ \tau_2\ (S, C, \Theta) \quad = \quad mgu'\ (S\tau_1)\ (S\tau_2)\ (S, C, \Theta) \\
mgu'\ \alpha\ \alpha \quad = \quad id_{S \times C \times \Theta} \\
mgu'\ \alpha\ \tau\ (S, C, \Theta) \mid \alpha \notin \mathrm{fv}(\tau),\ v{:}(\alpha{::}\Gamma) \in (\mathbf{v}{:}C) = \\
\qquad \textbf{let}\ \ (S', C', \Theta', d) = mgn\ \tau\ C\alpha\ ([\tau/\alpha] \circ S,\ [\tau/\alpha]C\backslash_\alpha,\ \Theta,\ \epsilon) \\
\qquad \textbf{in}\ \ (S', C',\ [d/v] \circ \Theta') \\
mgu'\ \tau\ \alpha\ (S, C, \Theta) \quad = \quad mgu\ \alpha\ \tau\ (S, C, \Theta) \\
mgu'\ ()\ () \quad = \quad id_{S \times C \times \Theta} \\
mgu'\ \kappa\ \tau\ \kappa\ \tau' \quad = \quad mgu\ \tau\ \tau'\ (S, C, \Theta) \\
mgu'\ (\tau_1, \tau_2)\ (\tau_1', \tau_2') \quad = \quad (mgu\ \tau_1\ \tau_1') \circ (mgu\ \tau_2\ \tau_2') \\
mgu'\ (\tau_1 \to \tau_2)\ (\tau_1' \to \tau_2') \quad = \quad (mgu\ \tau_1\ \tau_1') \circ (mgu\ \tau_2\ \tau_2') \\
\\
mgn\ \tau\ \{\} \quad = \quad id_{S \times C \times \Theta \times d} \\
mgn\ \tau\ \mathbf{v}{:}\langle\gamma\rangle\ (S, C, \Theta, d) \quad = \quad mgn'\ (S\tau)\ \mathbf{v}{:}(S\gamma)\ (S, C, \Theta, d) \\
mgn\ \tau\ (v_1{:}\Gamma_1 \oplus v_2{:}\Gamma_2) \quad = \quad (mgn\ \tau\ v_1{:}\Gamma_1) \circ (mgn\ \tau\ v_2{:}\Gamma_2) \\
mgn'\ \alpha\ \mathbf{v}{:}(c\ \tau)\ (S, C, \Theta, d) \quad = \quad \textbf{if}\ \exists \mathtt{w}, \tau'.\ \mathtt{w}{:}(c\ \tau') \in C\alpha \\
\qquad \textbf{then let}\ \ (S', C', \Theta') = mgu\ \tau\ \tau'\ (S, C, \Theta) \\
\qquad\qquad \textbf{in}\ \ (S', C', \Theta', \mathtt{w}d) \\
\qquad \textbf{else}\ \ (S, C[C\alpha \oplus \mathbf{v}{:}(c\ \tau)/\alpha], \Theta, \mathbf{v}d) \\
\\
mgn'\ \kappa\tau'\ c\tau\ (S, C, \Theta, d) \mid \exists\, \ulcorner \chi : \mathtt{inst}\ C' \Rightarrow \kappa\ \tilde{\tau}'{::}c\ \tilde{\tau}\ \urcorner\ \mathtt{in}\ \Sigma \\
\qquad \textbf{let}\ \ S' = match\ \tilde{\tau}'\ (\kappa\tau') \\
\qquad\quad (S'', C'', \Theta'') = mgu\ \tau\ (S'\tilde{\tau})\ (S, C, \Theta) \\
\qquad\quad \langle \tau_1{::}\Gamma_1, \ldots, \tau_n{::}\Gamma_n \rangle = S'C' \\
\qquad\quad (S_1, C_1, \Theta_1, \mathbf{d}_1 d) = \\
\qquad\qquad (mgn\ \tau_1\ \Gamma_1\ (\ \ldots\ (mgn\ \tau_n\ \Gamma_n\ (S'', C'', \Theta'', d)))) \\
\qquad \textbf{in}\ (S_1, C_1, \Theta_1, (\chi\ \mathbf{d}_1)d)
\end{array}
$$

(and similarly for $\to$, $(,)$, $()$)

Figure 6.4: Augmented Unification and Normalization Algorithms

The augmented algorithm retain the basic structure of the original algorithm. There are still four mutually recursive functions: *mgu*, *mgu'*, *mgn*, and *mgn'*; all follow the same calling patterns of their predecessors. On the other hand, two major changes are made to incorporate dictionary construction. First, the common state thread becomes an augmented constrained substitution. Second, the normalization functions *mgn* and *mgn'* are threaded with an additional argument d to accumulate the dictionaries constructed in checking the satisfiability of the given instance predicate.

More specifically, the context in the augmented constrained substitution keeps track of both the class constraints on the underlying type variables and their associated dictionary variables. In the meantime, any change to those dictionary variables will be recorded in the dictionary substitution of the augmented constrained substitution. As in the original algorithm, the call $mgu'\ \alpha\ \tau\ (S, C, \Theta)$ will, in turn, invoke *mgn* to check the satisfiability of $\tau::C\alpha$; but, in addition, *mgn* will return a list of dictionaries as a witness to the satisfication of this instance predicate and as a source for updating the dictionary substitution. These dictionaries are individually constructed by function *mgn'* using the augmented context and the given set of instance declarations.

By some straightforward manipulation, we can extend the properties of the original algorithms to include the construction of dictionaries. More precisely, the following two lemmas are established by similar induction proofs.

Lemma 6.17 (Soundness of *mgu* and *mgn*)

1. *If* $mgu\ \tau_1\ \tau_2\ (S, \mathbf{v}{:}C, \Theta) = (S', \mathbf{v}'{:}C', \Theta')$, *then* $S'\tau_1 = S'\tau_2$ *and* $(S', \mathbf{v}'{:}C', \Theta') \preceq (S, \mathbf{v}{:}C, \Theta)$.

2. *If* $mgn\ \tau\ \Gamma\ (S, \mathbf{v}{:}C, \Theta, d) = (S', \mathbf{v}'{:}C', \Theta', d_1 d)$, *then* $\mathbf{v}'{:}C' \Vdash d_1 : S'(\tau::\Gamma)$ *and* $(S', \mathbf{v}'{:}C', \Theta') \preceq (S, \mathbf{v}{:}C, \Theta)$.

The soundness lemma states that *mgu* computes a unifier and *mgn* computes a normalizer, both of which preserve the given context. In addition, *mgn* yields the dictionaries that satisfy the instance predicate to be normalized.

Lemma 6.18 (Completeness of *mgu* and *mgn*)

1. *Suppose that* $(S', C', \Theta') \preceq (S_0, C_0, \Theta_0)$ *and* $S'\tau_1 = S'\tau_2$. *Then the invocation mgu* τ_1 τ_2 (S_0, C_0, Θ_0) *succeeds with* (S, C, Θ) *such that* $S\tau_1 = S\tau_2$ *and* $(S, C, \Theta) \preceq (S', C', \Theta') \preceq (S_0, C_0, \Theta_0)$.

2. *Suppose that* $(S', C', \Theta') \preceq (S_0, C_0, \Theta_0)$ *and there exist dictionary expressions* d' *such that* $C' \Vdash d' : S'(\tau::\Gamma)$. *Then the invocation mgn* τ Γ (S_0, C_0, Θ_0, d) *succeed with* $(S, C, \Theta, d_1 d)$ *such that* $C \Vdash d_1 : S(\tau::\Gamma)$, $(S, C, \Theta) \preceq (S', C', \Theta') \preceq (S_0, C_0, \Theta_0)$ *and* $d' = \Phi d_1$ *for some dictionary substitution* Φ *on variables of* d_1.

The completeness lemma states that our algorithm computes the most general context-preserving unifiers and normalizers. In addition, *mgn* yields, for the given instance predicate, the most general dictionaries that are obtainable in terms of the ordering derived from dictionary substitutions.

Moreover, since the recursive calling patterns are unchanged, the termination property of the original algorithm carries over to the augmented one.

Lemma 6.19 (Termination of *mgu*) *For any augmented constrained substitution* (S, C, Θ) *and types* τ_1, τ_2, *the invocation mgu* τ_1 τ_2 (S, C, Θ) *either fails or terminates.*

Based on these results, we can easily establish the following theorem for the augmented unification algorithm.

Theorem 6.20 *Given an augmented constrained substitution* (S_0, C_0, Θ_0) *and two types* τ_1, τ_2, *if there is a* (S_0, C_0, Θ_0)*-preserving unifier of* τ_1 *and* τ_2, *then the invocation mgu* τ_1 τ_2 (S_0, C_0, Θ_0) *returns a most general such unifier. If there is no such unifier then mgu* τ_1 τ_2 (S_0, C_0, Θ_0) *fails in a finite number of steps.*

6.6 Type Reconstruction and Translation

As the last step towards the coherence result, this section extends our type reconstruction algorithm to include the calculation of translations, and shows that the augmented algorithm computes the most general translation for any given expression.

Figure 6.5 gives the augmented type reconstruction algorithm. As before, function *tp* proceeds by cases dispatching on the form of the input expression, but it yields a translation in addition to a type. More precisely, if $tp(e,\ A,\ S,\ \mathbf{v}:C,\ \Theta) = (\tau,\ e',\ S',\ \mathbf{v}' : C',\ \Theta')$, $\Theta' e'$ would be the translation of e at type τ. The dictionary bindings maintained in the dictionary substitution Θ' are acquired by extending Θ through calls to the augmented unification algorithm presented in the preceding section.

In the remainder of this section, we will establish the principal translation property for our translation semantics. We begin with the key property of *tp* that the augmented constrained substitution produced by *tp* preserves the input one, as formalized by the following lemma.

Lemma 6.21 *Let e be a Mini-Haskell$^+$ expression, and let $(S, \mathbf{v}{:}C, \Theta)$ be an augmented constrained substitution and A a type assumption set such that C covers SA. If $tp(e, A, S, \mathbf{v}{:}C, \Theta) = (\tau, S', \mathbf{v}'{:}C', \Theta')$, then $(S'.\ C', \Theta')$ is an augmented constrained substitution and $(S', \mathbf{v}'{:}C', \Theta') \preceq (S, \mathbf{v}{:}C, \Theta)$.*

The following theorem states that any typing and translation obtained by *tp* can also be derived using the rules for the syntax-directed system described in Section 6.4.

Theorem 6.22 *If $tp(e, A, S, \mathbf{v} : C, \Theta) = (\tau,\ e',\ S',\ \mathbf{v}' : C',\ \Theta')$, then $S'A,\ \mathbf{v}' : C' \vdash' e \rightsquigarrow \Theta' e' : \tau$.*

Combining this result with Theorem 6.9, we obtain the soundness property of *tp*:

$$
\begin{array}{lll}
tp(e, A, S, \mathbf{v}{:}C, \Theta) & = & \textbf{case } e \textbf{ of} \\
\\
x : & & inst\ (S(Ax), x, S, \mathbf{v}{:}C, \Theta) \\
\\
e_1\ e_2 : & & \textbf{let } (\tau_1, e'_1, S_1, \mathbf{v}_1{:}C_1, \Theta_1) = tp(e_1, A, S, \mathbf{v}{:}C, \Theta) \\
& & \quad (\tau_2, e'_2, S_2, \mathbf{v}_2{:}C_2, \Theta_2) = tp(e_2, A, S_1, \mathbf{v}_1{:}C_1, \Theta_1) \\
& & \quad \alpha \text{ be a new type variable} \\
& & \quad (S_3, \mathbf{v}_3{:}C_3, \Theta_3) = mgu\ \tau_1\ (\tau_2 \to \alpha)\ (S_2, C_2 \oplus (\alpha{::}\langle\rangle), \Theta_2) \\
& & \textbf{in } (S_3\alpha, (e'_1 e'_2), S_3, C_3, \Theta_3) \\
\\
\lambda x.e : & & \textbf{let } \alpha \text{ be a fresh type variable} \\
& & \quad (\tau_1, e'_1, S_1, C_1, \Theta_1) = tp\ (e_1, A.x{:}\alpha, S, C \oplus (\alpha{::}\langle\rangle), \Theta) \\
& & \textbf{in } (S_1\alpha \to \tau_1, (\lambda x.e'_1), S_1, C_1, \Theta_1) \\
\\
\texttt{let } x = e_1 \texttt{ in } e_2 : & & \textbf{let } \mathbf{v}'{:}C' = \langle\ C\alpha \mid \alpha \in C^*(\text{fv } SA)\ \rangle \\
& & \quad \mathbf{u}{:}D = (\mathbf{v}{:}C)\backslash(\mathbf{v}'{:}C') \\
& & \quad (\tau_1, e'_1, S_1, \mathbf{v}_1{:}C_1, \Theta_1) = tp\ (e_1, A, S, \mathbf{v}'{:}C', \Theta) \\
& & \quad (\sigma, \mathbf{v}_2{:}C_2, \mathbf{w}) = gen\ (\tau_1, S_1 A, \mathbf{v}_1{:}C_1, \epsilon) \\
& & \quad (\tau, e'_2, S_2, \mathbf{v}_3{:}C_3, \Theta_2) = tp\ (e_2, A.x{:}\sigma, S_1, \mathbf{v}_2{:}C_2, \Theta_1) \\
& & \textbf{in } (\tau, (\texttt{let } x = \lambda\mathbf{w}.\Theta_1 e'_1 \texttt{ in } e'_2), S_2, \mathbf{v}_3{:}C_3 \oplus \mathbf{u}{:}D, \Theta_2)
\end{array}
$$

where

$$
\begin{array}{lll}
inst\ (\forall\alpha{::}\Gamma.\sigma, e', S, \mathbf{v}{:}C, \Theta) & = & \textbf{let } \beta, u \text{ be new variables} \\
& & \textbf{in } inst\ ([\beta/\alpha]\sigma, (e'\ u), S, \mathbf{v}{:}C \oplus u{:}(\beta{::}\Gamma), \Theta) \\
inst\ (\tau, e', S, C, \Theta) & = & (\tau, e', S, C, \Theta)
\end{array}
$$

Figure 6.5: Type Reconstruction & Translation Algorithm

Theorem 6.23 *If* $tp(e, A, S, \mathbf{v}:C, \Theta) = (\tau,\ e',\ S',\ \mathbf{v}':C',\ \Theta')$, *then* $S'A,\ \mathbf{v}':C' \vdash e \rightsquigarrow \Theta' e' : \tau$.

Furthermore, any translation derived from the syntax-directed system can be expressed in terms of the translation synthesized by *tp*.

Theorem 6.24 *Suppose that* $S'A,\ \mathbf{v}':C' \vdash' e \rightsquigarrow e' : \tau'$ *and* $(S', \mathbf{v}':C', \Theta') \preceq (S_0, \mathbf{v}_0:C_0, \Theta_0)$. *Then* $tp(e, A, S_0, \mathbf{v}_0:C_0, \Theta_0)$ *succeeds with* $(\tau,\ e'',\ S,\ \mathbf{v}:C,\ \Theta)$, *and there exist a substitution* R *and dictionary expressions* $\mathbf{d}$ *such that*

1. $S' = RS$, *except possibly on new type variables of* $tp(e, A, S_0, \mathbf{v}_0:C_0, \Theta_0)$,
2. $\tau' = R\tau$,
3. $\mathbf{v}':C' \Vdash \mathbf{d} : RC$,
4. $S'A,\ \mathbf{v}':C' \vdash e' = [\mathbf{d}/\mathbf{v}]\Theta e'' : \tau'$.

Combining this result with Theorem 6.16, we obtain the completeness property of *tp*:

Corollary 6.25 *Suppose that* $S'A,\ \mathbf{v}':C' \vdash e \rightsquigarrow e' : \sigma'$ *and* $(S', \mathbf{v}':C', \Theta') \preceq (S_0, \mathbf{v}_0:C_0, \Theta_0)$. *Then* $tp(e, A, S_0, \mathbf{v}_0:C_0, \Theta_0)$ *succeeds with* $(\tau,\ e'',\ S,\ \mathbf{v}:C,\ \Theta)$, *and there exist a substitution* R, *a conversion* K *and dictionary expressions* $\mathbf{d}$ *such that*

1. $S' = RS$, *except possibly on new type variables of* $tp(e, A, S_0, \mathbf{v}_0:C_0, \Theta_0)$,
2. $K : \sigma' \preceq_{\mathbf{v}':C'} R\sigma$,
3. $\mathbf{v}':C' \Vdash \mathbf{d} : RC''$, *and*
4. $S'A,\ \mathbf{v}':C' \vdash K\,(\lambda \mathbf{w}.\,[\mathbf{d}/\mathbf{u}]\Theta e'') = e' : \sigma'$

where $(\sigma, \mathbf{u}:C'', \mathbf{w}) = gen(\tau, SA, \mathbf{v}:C, \epsilon)$.

As a corollary, we obtain the principal translation result:

Corollary 6.26 (Principal translations) *Suppose that* $dom(C_0) = (C_0)^*(tv\ S_0 A)$ *and* $tp(e, A, S_0, \mathbf{v}_0{:}C_0, \Theta_0) = (\tau,\ e',\ S,\ \mathbf{v} : C,\ \Theta)$. *Then* $\lambda \mathbf{w}.\, \Theta e'{:}\sigma$ *is a principal translation for* e *under* SA *and* $\mathbf{v}'{:}C'$, *where* $(\sigma, \mathbf{v}'{:}C', \mathbf{w}) = gen(\tau,\ SA,\ \mathbf{v}{:}C,\ \epsilon)$.

6.7 The Coherence Result

Having established the principal translation property, we can now proceed to develop the conditions that are sufficient to ensure coherent translation. This section defines the notion of ambiguous types and shows that unambiguous principal types entail coherent translations.

The notion of ambiguous types, first described in the Haskell report [Hudak *et al.*, 1990], has a rather intuitive interpretation. For example, the following type is ambiguous:

$$\forall \mathtt{a}{::}\mathtt{Parsable}.\mathtt{String}$$

In this type scheme, the quantified type variable `a` is constrained by the class **Parsable**, but does not appear in the type proper `String`. Given such a type scheme during type reconstruction, unification is not able to determine which instance type of **Parsable** is intended for `a` since only the type proper of a type scheme is used in unification. Indeed, as indicated in Section 6.1, overloaded expressions of this type may have several distinct translations and hence should be rejected.

Our definition of ambiguous types generalizes that of Haskell to include parameterized classes. Before presenting the formal definitions, it is instructive to consider some examples: the following type scheme is ambiguous for reasons similar to those of the preceding example on the quantified type variable `k`.

$$\forall \mathtt{a}{::}\{\mathtt{Eq}\}.\, \forall \mathtt{k}{::}\{\mathtt{Collection\ a}\}.\, \mathtt{a} \rightarrow \mathtt{Bool}$$

However, the following one is not considered ambiguous:

$$\forall \mathtt{a}::\{\mathtt{Eq}\}.\, \forall \mathtt{k}::\{\mathtt{Collection\ a}\}.\, \mathtt{k} \rightarrow \mathtt{k}$$

In this type scheme, although the quantified type variable **a** is constrained by class **Eq** and does not appear in the type proper, it is, through the constraint **Collection a**, a dependent of another type variable **k**, which does appear in the type proper. Once **k** is instantiated to some type τ through unification, we can obtain **a**'s value as a consequence of solving the instance predicate τ::**Collection a**.

As illustrated in the examples, quantified type variables manifest the potential ambiguity of a type scheme. In general, a type scheme is not ambiguous if its quantified type variables that are constrained by classes are also depended upon, directly or indirectly, by its type proper. This dependency relation can be expressed through the context closure operation C^* defined in Section 3.1.2, which computes, for a given set of type variables Δ, the set of type variables that are related, directly or indirectly, to those in Δ through the class constraints in C. The following definition of ambiguous type variables formalizes this idea.

Definition 6.7 (Ambiguous type variables) *A quantified type variable α in a type scheme $\sigma = \forall\langle\alpha_i::\Gamma_i\rangle.\,\tau$ is* ambiguous *if $C_\sigma(\alpha) \neq \emptyset$ and $\alpha \notin C^*_\sigma(tv\ \tau)$ where C_σ stands for the generic context, $\langle\alpha_i::\Gamma_i\rangle$, of σ.*

A type scheme is ambiguous if it contains ambiguous type variables. Note that in standard Haskell, $C^*_\sigma(\mathrm{tv}\ \tau) = \mathrm{tv}\ \tau$, so this definition generalizes the notion of ambiguous types described in the Haskell report.

For the coherence result, we are more interested in unambiguous types.

Definition 6.8 (Unambiguous type schemes) *A type scheme $\sigma = \forall\langle\alpha_i::\Gamma_i\rangle.\,\tau$ is* unambiguous *if none of the α_i is ambiguous.*

We are now ready to illustrate why unambiguous types entail coherent translations. From Corollary 6.26, we know that any translation of a Mini-Haskell$^+$ expression e in a particular setting can be written in the form $K\,e'$ where e' is e's principal translation and K is some suitable conversion. Now suppose that we have two arbitrary derivations $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e'_1 : \sigma'$ and $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e'_2 : \sigma'$. It follows that:

$$A, \mathbf{v}{:}C \vdash e'_1 = K_1\, e' : \sigma' \text{ and } A, \mathbf{v}{:}C \vdash e'_2 = K_2\, e' : \sigma'$$

where K_1 and K_2 are conversions from e's principal type σ to σ' under $\mathbf{v}{:}C$. Clearly, the two translations would be equivalent if $\mathbf{v}{:}C \vdash K_1 = K_2$.

Assume that $\sigma' = \forall\langle\alpha'_j{::}\Gamma'_j\rangle.\nu'$ and $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\nu$ and that none of the α'_j occurs free in σ or C. It follows from the definition of conversions that

$$[\tau_i/\alpha_i]\nu = \nu' \quad \text{and} \quad \mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha'_j{::}\Gamma'_j\rangle \Vdash \mathbf{d}_1 : [\tau_i/\alpha_i]\langle\alpha_i{::}\Gamma_i\rangle$$

for some types τ_i and that $\mathbf{v}{:}C \vdash K_1 = \lambda x.\lambda\mathbf{w}.\, x\, \mathbf{d}_1$. Similarly, for K_2 there are types τ'_i such that

$$[\tau'_i/\alpha_i]\nu = \nu' \quad \text{and} \quad \mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha'_j{::}\Gamma'_j\rangle \Vdash \mathbf{d}_2 : [\tau'_i/\alpha_i]\langle\alpha_i{::}\Gamma_i\rangle$$

and $\mathbf{v}{:}C \vdash K_2 = \lambda x.\lambda\mathbf{w}.\, x\, \mathbf{d}_2$. Obviously, it is sufficient to show that $\mathbf{d}_1 = \mathbf{d}_2$ to prove that these two conversions are equivalent. This in turn depends on whether the two instance predicate lists $[\tau_i/\alpha_i]\langle\alpha_i{::}\Gamma_i\rangle$ and $[\tau'_i/\alpha_i]\langle\alpha_i{::}\Gamma_i\rangle$ are identical since the dictionaries are uniquely determined by the instance predicates. But in general this is not true due to the differences between the types τ_i and τ'_i.

On the other hand, since $[\tau_i/\alpha_i]\nu = \nu' = [\tau'_i/\alpha_i]\nu$, it follows that $\tau_i = \tau'_i$ for all $\alpha_i \in \mathrm{tv}(\nu)$. This result can be further extended to those type variables' dependents. Recall the consistency requirement for parameterized classes:

$$\tau :: c\ \tau_1 \text{ and } \tau :: c\ \tau_2 \text{ implies } \tau_1 = \tau_2.$$

The requirement enables us to equate more types between τ_i and τ'. Indeed, a straightforward induction gives $\tau_i = \tau'_i$ for all $\alpha_i \in C^*_\sigma(\mathrm{tv}\ \nu)$.

Now, if σ is unambiguous, then for all α_i, either $C_\sigma \alpha_i = \emptyset$ or $\alpha_i \in C^*_\sigma(\alpha_i)$. The former case needs no dictionary; the latter one yields the same dictionary since $\tau_i = \tau'_i$. Consequently, all conversions from σ to any of its instances are equivalent:

Lemma 6.27 *If* $K_1, K_2 : \sigma' \preceq_{\mathbf{v}:C} \sigma$ *are conversions and* σ *is an unambiguous type scheme then* $\mathbf{v}{:}C \vdash K_1 = K_2$.

As a corollary, it follows that an unambiguous principal type entails coherent translations.

Theorem 6.28 (Coherence) *If* $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e'_1 : \sigma$ *and* $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e'_2 : \sigma$ *and the principal type of* e *under* A *and* C *is unambiguous, then* $A, \mathbf{v}{:}C \vdash e'_1 = e'_2 : \sigma$.

Analogous results were established in [Blott, 1991] for a version of their original system [Wadler and Blott, 1989] and in [Jones, 1992a] for his system of qualified types [Jones, 1992b].

The practical significance of this technical result is clear. If the principal type, computed by the type inferencer, of the given expression is ambiguous, we cannot guarantee a well-defined semantics for the expression and hence must reject it. Otherwise, we are sure that the translation gives a well-defined semantics.

Function *gen* Revisited

In Section 4.2.1, we briefly argued that our definition of *gen* is adequate out of the consideration of ambiguity detection as well as principal types. Given the results we have developed, it is easy to elaborate our argument. For ease of reference, we include

the original definition of *gen* below:

$$\begin{aligned} \mathit{gen}\ (\sigma,\ A,\ C) \quad = \quad & \textbf{if}\ \exists \alpha \in \mathrm{dom}(C) \backslash (\mathrm{tv}\ A \cup \mathrm{reg}\ C)\ \textbf{then} \\ & \mathit{gen}\ (\forall \alpha :: C\alpha.\sigma,\ A,\ C \backslash_{\alpha}) \\ & \textbf{else}\ (\sigma, C) \end{aligned}$$

Basically, our definition of *gen* is complicated by the presence of ambiguous expressions. As an example, consider the type reconstruction for the expression given in Section 6.1:

```
unparse (parse "123")
```

In the final step of reconstructing its type, *gen* will be invoked with the type `String`, the context {`a::Parsable`}, and the empty type assumption set. The issue here is the instance assumption `a::Parsable` in the context. Suppose that we had used the alternative definition (*gen'*) given in Section 4.2.1 and partially repeated below:

$$\begin{aligned} \mathit{gen'}\ (\sigma,\ A,\ C) \quad = \quad & \textbf{if}\ \exists \alpha \in \mathrm{tv}(\sigma) \backslash (\mathrm{tv}\ A \cup \mathrm{reg}\ C)\ \textbf{then} \\ & \vdots \end{aligned}$$

Since $\mathtt{a} \notin \mathrm{tv}(\mathtt{String})$, we would not discharge it and therefore fail to obtain the most general type for this ambiguous expression. In contrast, using *gen*, we would discharge the instance assumption to form the principal type `∀a::Parsable. String` for the expression, thereby detecting the underlying ambiguity.

In general, since the ambiguity test is based on an expression's principal type, we must ensure that the type inferencer computes the most general types for any given expression, including ambiguous ones. This in turn relies on *gen* to discharge as many instance assumptions from the underlying context as possible to obtain the most general type scheme. Therefore, we have adopted the more general version of generalization function.

Chapter 7

Conclusions

In this chapter, we review the results of this thesis and discuss future work.

7.1 Results

This thesis generalizes Haskell's type system to support overloaded operators over parameterized types. The solution that we proposed is a parametric extension of Haskell's type classes, called *parametric type classes*. Unlike standard type classes which constrain a single type only, parametric type classes can constrain a parameterized type and its constituent type(s) as well. Therefore, for example, we can use them to overload data constructors and selectors on container classes such as `Collection`.

To support our proposal, we have developed a type system by extending, in a highly modular fashion, the Hindley-Milner type system to include constrained quantification. In the extended system, type classes act as constraints on quantification and instantiation of type variables. This is achieved by putting class constraints on quantified type variables and adding a separate constraint inference sub-system to the standard type inference engine. To glue the two systems together, we have devel-

oped a new unification algorithm that augments first-order unification with constraint propagation. Then, based on the augmented unification algorithm, we have given an effective algorithm to reconstruct types for typable expressions. Most important of all, we have proved that the principal type property holds for the extended system.

We have also provided a translation semantics to associate meanings with typed expressions. This is done by translating a given program, based on its typing derivation, to a program defined in a language that includes constructs for manipulating overloading explicitly. We have presented a method for extending the type reconstruction algorithm to perform the translation, and proved that it yields the principal translations for well-typed expressions. Furthermore, since interpretations of expressions follow their typing derivations and an expressions can be type-checked in more than one way, it is necessary to ensure the coherence of our translation scheme. As in Haskell, this is accomplished by defining sufficient conditions that ensure coherent translations.

Overall, parametric type classes are a conservative extension of Haskell's type system, since if all classes are parameterless, the two systems are equivalent.

7.2 Future Work

Although parametric type classes can overload many operations over various kinds of parameterized types, there are some operations that cannot be similarly overloaded without extra machinery. In particular, operations that involve more than one parameterized type which share the same structure but contain distinct constituents are not directly supported by parametric type classes. A typical example is the overloaded `map` function, which takes a function and a parameterized structure as arguments, and builds another structure by applying the function to the elements of the given structure.

How do we type the overloaded `map` using parametric type classes? A naive solution is to introduce a class called `Map` and type `map` as follows:

$$\texttt{map} : \forall \texttt{a}.\, \forall \texttt{b}.\, \forall \texttt{ma::Map a}.\, \forall \texttt{mb::Map b}.\, (\texttt{a} \rightarrow \texttt{b}) \rightarrow \texttt{ma} \rightarrow \texttt{mb}$$

But this may be more general than we would like, since it would admit also implementations that take one structure (e.g. a list) and return a structure of a different kind (e.g. a vector). What is needed is some way to specify that `map` returns the same kind of structure as its argument, but with a possibly different element type. Without additional mechanisms, the best we can do is to adopt a less general type for `map`:

$$\texttt{map} : \forall \texttt{a}.\, \forall \texttt{ma::Map a}.\, (\texttt{a} \rightarrow \texttt{a}) \rightarrow \texttt{ma} \rightarrow \texttt{ma}$$

which is clearly rather restrictive. The solution that we proposed in [Chen *et al.*, 1992a] is an encoding scheme that can capture the requirement of structural similarity among type variables. The basic idea is to introduce a special root class `TC` with one parameter but no operations in it. Every type $(\kappa\ \tau)$ is an instance of `TC` by virtue of an instance declaration

$$\texttt{inst}\ \kappa\ a :: \texttt{TC}\ (\kappa\ ())$$

which is implicitly generated for every algebraic data type. Effectively, `TC` is used to "isolate" the top-level type constructor of a type. That is, if two types are related by a `TC` constraint, we know that they have the same top-level type constructor. The `TC` technique enables us to type `map` precisely:

$$\texttt{map} : \forall \texttt{a}.\, \forall \texttt{b}.\, \forall \texttt{t}.\, \forall \texttt{ma::\{Map a, TC t\}}.\, \forall \texttt{mb::\{Map b, TC t\}}.\, (\texttt{a} \rightarrow \texttt{b}) \rightarrow \texttt{ma} \rightarrow \texttt{mb}$$

This states that `ma` and `mb` are instance types of `Map` with element types `a` and `b`, and that `ma` and `mb` share the same type constructor. More details can found in [Chen *et al.*, 1992b].

Another solution is to use constructor classes, as recently proposed in [Jones, 1993]. By combining overloading with higher-order polymorphism, classes in this

system can constrain type constructors as well as types. For example, using our notations, a constructor class **Functor** that overloads the **map** function at both **List** and **Tree** type constructors can be described by the following declarations:

```
class  f :: Functor  where
  map  :  (a -> b) -> f a -> f b
inst  List :: Functor  where
  map f l  =  ...
inst  Tree :: Functor  where
  map f l  =  ...
```

Furthermore, using a constructor class of monads:

```
class  m :: Monad  where
   unit :  a -> m a
   bind :  m a -> (a -> m b) -> m b
   map  :  (a -> b) -> f a -> f b
   join :  m (m a)
```

we can define monad comprehensions [Wadler, 1990a], an extension of list comprehensions to other parameterized structures.

On the other hand, many instance declarations for parameterized types require constraints on their constituent types that are not expressible in this system since constructor classes constrain only type constructors, excluding their constituent types. For example, we would like to constrain set types to have equality defined for its members. But consider the following instance declaration that attempts to define an algebraic data type **Set a** to be an instance of **Monad**:

```
inst  Set :: Monad  where ...
```

Since only **Set** is being constrained, the equality constraint on **a** is not expressible using the declaration above. Therefore, an interesting avenue for future work is to

see whether we can apply our technique to parameterize constructor classes. For example, the following declaration may introduce a "parametric" constructor class **Monad a b**:

```
class  m :: Monad a b  where
   unit :  a -> m a
   bind :  m a -> (a -> m b) -> m b
   map  :  (a -> b) -> f a -> f b
   join :  m (m a)
```

Then we can specify the equality constraints in its instance declarations as follows:

```
inst  (a::Eq, b::Eq) => Set :: Monad a b  where ...
```

Whether the theory can be thus extended is yet to be investigated.

Bibliography

[Blott, 1991] Stephen Blott. *An Approach to Overloading with Polymorphism.* PhD thesis, University of Glasgow, Glasgow, UK, September 1991.

[Breazu *et al.*, 1989] V. Breazu, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 112–129, June 1989.

[Chen *et al.*, 1992a] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 170–181. ACM, June 1992.

[Chen *et al.*, 1992b] Kung Chen, Martin Odersky, and Paul Hudak. Type inference for parametric type classes. Technical Report YALEU/DCS/RR-900, Dept. of Computer Science, Yale University, New Haven, Conn., June 1992.

[Clément *et al.*, 1986] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. on LISP and Functional Programming*, pages 13–27, 1986.

[Cormack and Wright, 1990] G. Cormack and A. Wright. Type-dependent parameter inference. In *Proc. SIGPLAN'90 Conf. on Programming Language Design and Implementation*, pages 127–136, White Plains, NY, June 1990.

[Damas and Milner, 1982] L. Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[Damas, 1984] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1984.

[Hall *et al.*, 1994] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in haskell. In *Proc. European Symposium on Programming*, 1994. LNCS 788.

[Hindley and Seldin, 1986] R. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[Hindley, 1969] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, Dec. 1969.

[Hudak *et al.*, 1990] Paul Hudak, Simon Peyton Jones, and Philip L. Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.0. Technical Report YALEU/DCS/RR-777, Dept. of Computer Science, Yale University, New Haven, Conn., April 1990. Current version 1.2, March 1992.

[Jenks and Trager, 1981] R.D. Jenks and B.M. Trager. A language for computational algebra. In *Proc. ACM Symposium on Symbolic and Algebraic Manipulation*, pages 22–29, 1981.

[Jones, 1992a] Mark P. Jones. *Qualified types: theory and practice.* PhD thesis, Oxford University, Oxford, UK, July 1992.

[Jones, 1992b] Mark P. Jones. A theory of qualified types. In B. Krieg-Brückner, editor, *Proc. European Sym-*

posium on Programming, pages 287–306. Springer Verlag, Feburary 1992. LNCS 582.

[Jones, 1993] Mark P. Jones. A sysytem of constructor classes: overloading and implicit higher-order polymorphism. In *Proc. Conf. on Functional Programming and Computer Architecture*, pages 43–52, June 1993.

[Jones, 1994] Mark P. Jones. ML typing, explicit polymorphism, and qualified types. In *Proc. Theoretical Aspects of Computer Software '94*, pages 56–75, April 1994. LNCS 789.

[Kaes, 1988] S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proc. 2nd European Symosium on Programming, Lecture Notes in Computer Science, Vol. 300*, pages 131–144, Nancy, France, March 1988. Springer-Verlag.

[Kaes, 1992] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. ACM Conf. on Lisp and Functional Programming*. ACM, June 1992.

[Lillibridge, 1992] Mark D. Lillibridge. Making ad hoc polymorphism work. Private communication, June 1992.

[Meseguer *et al.*, 1989] J. Meseguer, J. Goguen, and G. Smolka. Order-sorted unification. *J. Symbolic Computation*, 8:383–413, 1989.

[Milner *et al.*, 1990] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Milner, 1978] Robin Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Mitchell and Harper, 1988] J.C. Mitchell and R. Harper. The essence of ML. In *Proc. 15th ACM Symp. on Principles of Programming Languages*. ACM, Jan. 1988.

[Nipkow and Prehofer, 1993] Tobias Nipkow and Christian Prehofer. Type-Checking type classes. In *Proc. ACM Symp. on Principles of Programming Langusges*, pages 409–18, 1993.

[Nipkow and Snelting, 1991] Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In J. Hughes, editor, *Proceedings of Functional Programming and Computer Architecture*, pages 1–14. Springer-Verlag, 1991. LNCS 523.

[Peyton Jones and Wadler, 1991] Simon Peyton Jones and Phil Wadler. A static semantics for haskell. Dept. of Computing Science, Glasgow University, Obtained electronically from `ftp.dcs.glasgow.ac.uk`, May 1991.

[Prawitz, 1965] Dag Prawitz. *Natural Deduction.* Almqvist and Wiksell, Stockholm, 1965.

[Reynolds, 1974] John C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, pages 408–25. Springer Verlag, 1974. Lecture Notes in Computer Science 19.

[Robinson, 1965] J. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12(1):23–41. 1965.

[Rouaix, 1990] Francois Rouaix. Safe run-time overloading. In *Seventeenth Annual ACM Symp. on Principles of Programming Languages*, pages 355–366, San Franscico, CA, January 1990.

[Smith, 1991] Geoffery S. Smith. *Polymorphic type inference for languages with overloading and subtyping.* PhD thesis. Cornell University, Ithaca, NY, August 1991.

[Strachey, 1967] Christopher Strachey. Fundamental concepts in programming languages. International summer school in computer programming, 1967.

[Thatte, 1992] Satish R. Thatte. Typechecking with ad hoc polymorphism. Obtained electronically from `sun.mcs.mclarkson.edu`, May 1992.

[Turner, 1985] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. Conf. on Functional Programming and Computer Architecture*, pages 1–16. Springer Verlag, 1985. LNCS 201.

[Volpano and Smith, 1991] Dennis M. Volpano and Geoffery S. Smith. On the complexity of ML typability and overloading. In J. Hughes, editor, *Proceedings of Functional Programming and Computer Architecture*, pages 15–28. Springer-Verlag, 1991. LNCS 523.

[Wadler and Blott, 1989] Phil Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 60–76. ACM, 1989.

[Wadler, 1990a] P. Wadler. Comprehending monads. In *Proc. ACM Conf. on LISP and Functional Programming*, pages 61–78, June 1990.

[Wadler, 1990b] Philip Wadler. Simplified overloading for haskell. Note sent to the Haskell mailing list, October 1990.

Appendix A

Proofs

This appendix contains detailed proofs for many of the results stated in Chapter 6. Most of these are direct extensions of the results described in Chapter 3 and Chapter 4, and the proofs for the latter may be obtained from the proofs given here by ignoring the parts on dictionary-based translation.

In addition, we include here the proof of Lemma 3.2, which is referenced by many of the proofs that follow.

Lemma 3.2 *Let C and D be contexts and S a substitution such that $D \Vdash SC$. Then for any $\Delta \subseteq dom(C)$, we have $tv(S\ C^*(\Delta)) \subseteq D^*(tv\ S\Delta)$.*

Proof: We prove that if $\beta \in C^*(\Delta)$ then $\mathrm{tv}(S\beta) \subseteq D^*(\mathrm{tv}\ S\Delta)$.

Let C be a context. Define C^k as follows:

$$\begin{aligned} C^0(\Delta) &= \Delta \\ C^{k+1}(\Delta) &= C^k(\Delta) \cup \{\, \mathrm{tv}(C\alpha) \mid \alpha \in C^k(\Delta) \,\} \end{aligned}$$

By the definition of C^*, if $\beta \in C^*(\Delta)$ then $\beta \in C^k(\Delta)$ for some $k \geq 0$. Thus an induction on k suffices.

k = 0 : In this case, $\beta \in \Delta$. Hence $\text{tv}(S\beta) \subseteq \text{tv}(S\Delta)$.

k = n + 1 : Suppose that $\beta \in \text{tv}(C\alpha)$ for some $\alpha \in C^n(\Delta)$. By induction, $\text{tv}(S\alpha) \subseteq D^*(\text{tv } S\Delta)$. There are four possible cases, depending on the effects of S on α and β:

- $\alpha \notin \text{dom}(S)$ and $\beta \notin \text{dom}(S)$: Note that by $D \Vdash SC$, we know that D covers SC. So, in this case, clearly $\beta \in \text{tv}(D\alpha)$, since $D \Vdash SC$. Then, by induction, $\alpha \in D^*(\text{tv } S\Delta)$ and hence $\text{tv}(S\beta) \subseteq D^*(\text{tv } S\Delta)$.

- $\alpha \notin \text{dom}(S)$ and $\beta \in \text{dom}(S)$: In this case, we know that $\alpha \in \text{dom}(D)$, since $D \Vdash SC$. In particular, $D\alpha = S(C\alpha)$ and hence $\text{tv}(S\beta) \subseteq \text{tv}(D\alpha)$. By induction, $\alpha \in D^*(\text{tv } S\Delta)$. So $\text{tv}(S\beta) \subseteq D^*(\text{tv } S\Delta)$.

- $\alpha \in \text{dom}(S)$ and $\beta \notin \text{dom}(S)$: there are two possibilities depending on the structure of $S\alpha$:

  - $S\alpha = \rho$: Since $D \Vdash SC$, we have $\rho \in \text{dom}(D)$ and $\beta \in \text{tv}(D\rho)$ By induction, $\rho \in D^*(\text{tv } S\Delta)$. So $\text{tv}(S\beta) \subseteq D^*(\text{tv } S\Delta)$.

  - $S\alpha = \kappa\,\tau$: Since $D \Vdash SC$, it follows from our requirements on instance declarations that $\beta \in \text{tv}(\tau)$. Moreover, since $\text{tv}(S\alpha) = \text{tv}(\tau)$, by induction we have $\text{tv}(\tau) \subseteq D^*(\text{tv } S\Delta)$. So $\text{tv}(S\beta) \in D^*(\text{tv } S\Delta)$.

- $\alpha \in \text{dom}(S)$ and $\beta \in \text{dom}(S)$, there are four possibilities depending on the structures of $S\alpha$ and $S\beta$. Except for the following case, the others are similar to the ones given above.

  - $S\alpha = \kappa\,\tau$ and $S\beta = \kappa'\,\tau'$: In this case, $\text{tv}(S\beta) = \text{tv}(\tau')$. Again, it follows from $D \Vdash SC$ and our requirements on instance declarations that $\text{tv}(\tau') \subseteq \text{tv}(\tau)$. Moreover, since $\text{tv}(S\alpha) = \text{tv}(\tau)$, by induction we have $\text{tv}(\tau) \subseteq D^*(\text{tv } S\Delta)$. So $\text{tv}(S\beta) \in D^*(\text{tv } S\Delta)$.

Hence for all $\beta \in C^*(\Delta)$, $\text{tv}(S\beta) \subseteq D^*(\text{tv } S\Delta)$. ∎

Lemma 6.3 *If* $K' : \sigma'' \preceq_{\mathbf{v}:C} \sigma'$ *and* $K : \sigma' \preceq_{\mathbf{v}:C} \sigma$, *then* $(K' \circ K) : \sigma'' \preceq_{\mathbf{v}:C} \sigma$.

Proof: Suppose that

$$\sigma = \forall\langle\alpha_i::\Gamma_i\rangle.\,\tau, \quad \sigma' = \forall\langle\alpha'_j::\Gamma'_j\rangle.\,\tau' \text{ and } \sigma'' = \forall\langle\alpha''_k::\Gamma''_k\rangle.\,\tau''$$

where the variables α'_j appear only in Γ'_j and τ', and the variables α''_k appear only in Γ''_k and τ''. By definition of conversions:

$$\tau' = [\tau_i/\alpha_i]\,\tau, \quad \mathbf{v}{:}C \oplus \mathbf{u}{:}\langle\alpha'_j::\Gamma'_j\rangle \Vdash \mathbf{d} : [\tau_i/\alpha_i]\,\langle\alpha_i::\Gamma_i\rangle,$$

for some τ_i, $\mathbf{d}$ and $\mathbf{u}$ (disjoint from $\mathbf{v}$) such that $\mathbf{v}{:}C \vdash K = \lambda x.\lambda\mathbf{u}.\,x\mathbf{d}$. In a similar way,

$$\tau'' = [\tau'_j/\alpha'_j]\,\tau', \quad \mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha''_k::\Gamma''_k\rangle \Vdash \mathbf{d}' : [\tau'_j/\alpha'_j]\,\langle\alpha'_j::\Gamma'_j\rangle,$$

for some τ'_j, $\mathbf{d}'$ and $\mathbf{w}$ (disjoint from $\mathbf{v}$) such that $\mathbf{v}{:}C \vdash K' = \lambda x.\lambda\mathbf{w}.\,x\mathbf{d}'$. Since none of the α'_j appear free in τ, we have

$$\tau'' = [\tau'_j/\alpha'_j]\,\tau' = [\tau'_j/\alpha'_j]\,([\tau_i/\alpha_i]\,\tau) = [\tau''_i/\alpha_i]\tau$$

where $\tau''_i = [\tau'_j/\alpha'_j]\tau_i$. Now, in order to apply $[\tau'_j/\alpha'_j]$ to the first of the two augmented instance entailments above, we note that:

$$\mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha''_k::\Gamma''_k\rangle \Vdash \mathbf{v}\mathbf{d}' : [\tau'_j/\alpha'_j]\,(C \oplus \langle\alpha'_j::\Gamma'_j\rangle),$$

since none of the α'_j appear free in C. Thus, by the transitivity under substitution of $\Vdash$, we obtain:

$$\mathbf{v}{:}C \oplus \mathbf{w}{:}\langle\alpha''_k::\Gamma''_k\rangle \Vdash [\mathbf{d}'/\mathbf{u}]\,\mathbf{d} : [\tau'_j/\alpha'_j]\,([\tau_i/\alpha_i]\,\langle\alpha_i::\Gamma_i\rangle).$$

To complete the proof, notice that

$$\begin{aligned}
\mathbf{v}{:}C \vdash K'' &= \lambda x.\lambda \mathbf{w}.\, x\,([\mathbf{d}'/\mathbf{u}]\,\mathbf{d}) \\
&= \lambda x.\lambda \mathbf{w}.\,(\lambda \mathbf{u}.\, x\mathbf{d})\,\mathbf{d}' && (\beta_d) \\
&= \lambda x.\lambda \mathbf{w}.\,(Kx)\,\mathbf{d}' && \text{(property of } K) \\
&= \lambda x.\lambda \mathbf{w}.\, K'(Kx)\,\mathbf{w} && \text{(property of } K') \\
&= \lambda x.\, K'(Kx) && (\eta_d)
\end{aligned}$$

which is the required conversion. ■

Lemma 6.6 *Let C be a context that covers both type τ and type assumption set A. If $gen\,(\tau, A, \mathbf{v}{:}C, \epsilon) = (\sigma', \mathbf{v}'{:}C', \mathbf{w})$, then $\sigma' = \langle \alpha_i :: C\alpha_i \rangle_1^n.\,\tau$ for some $n \geq 0$ such that $\mathbf{v}{:}C \cong \mathbf{w}{:}\langle \alpha_i :: C\alpha_i \rangle \oplus \mathbf{v}'{:}C'$ and $dom(C') = C^*(tv\ A)$.*

Proof: For ease of reference, we include the definition of *gen* below:

$$\begin{aligned}
gen\ (\sigma,\ A,\ \mathbf{v}{:}C,\ \mathbf{w}) \;=\; & \textbf{if } \exists (v{:}(\alpha :: \Gamma)) \in \mathbf{v}{:}C \textbf{ and } \alpha \notin (\ \mathrm{tv}(A) \cup \mathrm{reg}(C)\ ) \\
& \textbf{then } gen\ (\forall \alpha :: \Gamma.\sigma,\ A,\ (\mathbf{v}{:}C)\backslash_{v:(\alpha::\Gamma)},\ v\mathbf{w}) \\
& \textbf{else } (\sigma,\ \mathbf{v}{:}C,\ \mathbf{w})
\end{aligned}$$

The lemma is a direct corollary of the following stronger version:

> If $gen\,(\sigma, A, \mathbf{v}{:}C, \mathbf{w}) = (\sigma', \mathbf{v}'{:}C', \mathbf{w}')$, then $\sigma' = \langle \alpha_i :: C\alpha_i \rangle_1^n.\,\sigma$ for some $n \geq 0$ such that $\mathrm{dom}(C) = \{\alpha_i\} \uplus C^*(\mathrm{tv}\ A)$ and $\mathbf{v} \oplus \mathbf{w} \cong \mathbf{v}' \oplus \mathbf{w}'$.

Without loss of generality we can assume that, initially, $\mathrm{dom}(C) \cap \mathrm{btv}(\sigma) = \emptyset$ and $\mathbf{w}$ is disjoint from $\mathbf{v}$. The stronger version is a consequence of the following two observations:

1. Both $\mathrm{dom}(C) \uplus \mathrm{btv}(\sigma)$ and $\mathbf{v} \oplus \mathbf{w}$ are invariant during the recursion of *gen*.

2. If $\alpha \in C^*(\text{tv } A)$, then $\alpha \in (\text{tv}(A) \cup \text{reg}(C))$. Hence $C^*(\text{tv } A)$ is also invariant during the recursion of *gen*.

Now suppose that $gen\ (\sigma, A, \mathbf{v}{:}C, \mathbf{w}) = (\sigma', \mathbf{v}'{:}C', \mathbf{w}')$, then an immediate consequence of (1) is that $\text{dom}(C) \uplus \text{btv}(\sigma) = \text{dom}(C') \uplus \text{btv}(\sigma')$ and $\mathbf{v} \oplus \mathbf{w} \cong \mathbf{v}' \oplus \mathbf{w}'$.In other words, there exist some $n \geq 0$ such that $\sigma' = \langle \alpha_i :: C\alpha_i \rangle_1^n . \sigma$ and $\text{dom}(C) = \{\alpha_i\} \cup \text{dom}(C')$.

By (2), $C^*(\text{tv } A) \subseteq \text{dom}(C')$. In addition, we claim that $\text{dom}(C') = C^*(\text{tv } A)$, for if $\text{dom}(C') \backslash C^*(\text{tv } A) \neq \emptyset$, then *gen* would not stop at $(\sigma', C', \mathbf{w}')$. To see this, suppose that there exists a type variable $\alpha \in \text{dom}(C') \backslash C^*(\text{tv } A)$. Then there will also be some $\beta \in \text{dom}(C') \backslash \text{reg}(C')$ since C' is acyclic. This in turn implies that β would be generalized by *gen*, which contradicts the assumption that *gen* terminates with $(\sigma', C', \mathbf{w}')$. $\{\alpha_i\} \uplus C^*(\text{tv } A) = \text{dom}(C)$. ■

Lemma 6.7 *If* $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e' : \tau$ *and* $gen(\tau, A, \mathbf{v}{:}C, \epsilon) = (\sigma, \mathbf{v}'{:}C', \mathbf{w})$, *then* $A, \mathbf{v}'{:}C' \vdash e \rightsquigarrow \lambda \mathbf{w}.\, e' : \sigma$.

Proof: By applying ($\forall$-I) to $A, \mathbf{v}{:}C \vdash e \rightsquigarrow e' : \tau$ repeatedly following the order of discharging used in the implementation of *gen*. ■

Lemma 6.8 *Suppose that* $gen(\tau, A, \mathbf{v}{:}C, \epsilon) = (\sigma, \mathbf{v}_1{:}C_1, \mathbf{w})$ *and* $\mathbf{u}{:}D$ *is disjoint from* $\mathbf{v}{:}C$. *Then* $gen(\tau, A, \mathbf{v}{:}C \oplus \mathbf{u}{:}D, \epsilon) = (\sigma'.\, \mathbf{v}'{:}C', \mathbf{w}')$ *for some* σ', $\mathbf{v}'{:}C'$, *and* $\mathbf{w}'$ *such that*

$C' = C_1$, $\mathbf{w}' \cong \mathbf{uw}$, $(\lambda x.\lambda \mathbf{w}'.\, x\, \mathbf{w}) : \sigma' \preceq \sigma$ *and* $(\lambda x.\lambda \mathbf{w}.\, x\mathbf{w}') : \sigma \preceq_{\mathbf{u}:D} \sigma'$.

Proof: By Lemma 6.6,

$$\sigma = \forall \langle \alpha_i :: C\alpha_i \rangle . \tau, \quad \mathbf{v}{:}C \cong \mathbf{w}{:}\langle \alpha_i :: C\alpha_i \rangle \oplus \mathbf{v}_1{:}C_1 \text{ and } C_1 = C^*(\text{tv } A).$$

Let $\mathbf{u}'{:}D' = \mathbf{v}{:}C \oplus \mathbf{u}{:}D$. Since $\mathbf{u}{:}D$ is disjoint from $\mathbf{v}{:}C$, we know that $(D')^*(\mathrm{tv}\ A) = C^*(\mathrm{tv}\ A)$. Hence by Lemma 6.6, $\sigma' = \forall\langle\alpha'_j{::}D'(\alpha'_j)\rangle.\tau$ and

$$\mathbf{u}'{:}D' \cong \mathbf{w}'{:}\langle\alpha'_j{::}D'(\alpha'_j)\rangle \oplus \mathbf{v}'{:}C' \text{ and } C' = C_1\,.$$

It then follows from the definition of $\mathbf{u}'{:}D'$ that $\mathbf{w}' \cong \mathbf{uw}$.

Next, to show that the given conversions are correct, note that clearly none of the α'_j appears free in σ and $\langle\alpha_i{::}C\alpha_i\rangle \sqsubseteq \langle\alpha'_j{::}D'(\alpha_j)\rangle$. So

$$\mathbf{w}'{:}\langle\alpha'_j{::}D'(\alpha_j)\rangle \Vdash \mathbf{w}{:}\langle\alpha_i{::}C\alpha_i\rangle$$

and hence

$$(\lambda x.\lambda\mathbf{w}'.\,x\,\mathbf{w}) : \sigma' \preceq \sigma\,.$$

The other conversion can be similarly derived. ■

Theorem 6.9 *If* $A,\ \mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$, *then* $A,\ \mathbf{v}{:}C \vdash e \rightsquigarrow e' : \tau$.

Proof: By induction on the structure of $A,\ \mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$. The only interesting case is (*let'*):

We have a derivation of the form

$$\frac{A,\ \mathbf{v}'{:}C' \vdash' e_1 \rightsquigarrow e'_1 : \tau_1 \qquad A.x{:}\sigma,\ \mathbf{v}{:}C \vdash' e_2 \rightsquigarrow e'_2 : \tau_2}{A,\ C \vdash' (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\mathtt{let}\ x = \lambda\mathbf{w}.e'_1\ \mathtt{in}\ e'_2) : \tau_2}$$

$$\text{where } (\sigma,\ \mathbf{v}''{:}C'',\ \mathbf{w}) = gen(\tau_1,\ A,\ \mathbf{v}'{:}C', \epsilon) \text{ and } \mathbf{v}''{:}C'' \sqsubseteq \mathbf{v}{:}C,$$

Without loss of generality, we can assume that $\mathrm{dom}(C') \cap \mathrm{dom}(C) = \mathrm{dom}(C'')$, which can be achieved by a suitable renaming of variables in $\mathrm{dom}(C)$. Hence, $\mathrm{dom}(C) \cap \mathrm{btv}(\sigma) = \emptyset$ and we can thus construct the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{A, C' \vdash' e_1 \rightsquigarrow e_1' : \tau_1}{A, C' \vdash e_1 \rightsquigarrow e_1' : \tau_1}\ (induction)}{A, C'' \vdash e_1 \rightsquigarrow \lambda \mathbf{w}.\, e_1' : \sigma}\ (Lemma\ 6.7)}{A, C \vdash e_1 \rightsquigarrow \lambda \mathbf{w}.\, e_1' : \sigma}\ (Lemma\ 3.3) \qquad \dfrac{A.x:\sigma, C \vdash' e_2 \rightsquigarrow e_2' : \tau_2}{A.x:\sigma, C \vdash e_2 \rightsquigarrow e_2' : \tau_2}\ (induction)}{A, C \vdash (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\mathtt{let}\ x = \lambda \mathbf{w}.\, e_1'\ \mathtt{in}\ e_2') : \tau_2}\ (let)$$

This completes the proof. ■

Lemma 6.10 *Let* $(\sigma, C', \mathbf{w}) = gen(\tau, A, \mathbf{v}{:}C, \epsilon)$ *and* $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\tau$. *Suppose that there exist dictionary expressions* $\mathbf{d}$ *such that* $\mathbf{u}{:}D \Vdash \mathbf{d} : SC$. *If* $gen(S\tau, SA, \mathbf{u}{:}D, \epsilon) = (\sigma', \mathbf{u}'{:}D', \mathbf{w}')$, *then*

$$(\lambda x.\lambda \mathbf{w}'.\, x\mathbf{d}') : \sigma' \preceq_{\mathbf{u}':D'} S\sigma.$$

where $\mathbf{d}'$ *is a sublist of* $\mathbf{d}$ *such that* $\mathbf{u}{:}D \Vdash \mathbf{d}' : S\langle\alpha_i{::}\Gamma_i\rangle$.

Proof: Suppose that $\sigma' = \forall\langle\alpha_j'{::}\Gamma_j'\rangle.S\tau$. Then none of the variables α_j' appears free in D' or $S\sigma$. To see this, first apply Lemma 6.6 to σ' and D':

$$\mathrm{dom}(D) = \{\alpha_j'\} \uplus \mathrm{dom}(D') = \{\alpha_j'\} \uplus D^*(\mathrm{tv}\ SA).$$

Hence none of α_j' appears in D'. Next, since $D \Vdash SC$, it follows from Lemma 3.2 that $\mathrm{tv}(S\ C^*(\mathrm{tv}\ A)) \subseteq D^*(\mathrm{tv}\ SA)$. Furthermore, by Lemma 6.6, $\mathrm{dom}(C) = \{\alpha_i\} \uplus C^*(\mathrm{tv}\ A)$. Now since $\mathrm{tv}(\sigma) \subseteq C^*(\mathrm{tv}\ A)$, it then follows that $\mathrm{tv}(S\sigma) \subseteq \mathrm{tv}(S\ C^*(\mathrm{tv}\ A))$. This in turn implies that $\mathrm{tv}(S\sigma) \subseteq D^*(\mathrm{tv}\ SA))$. Therefore, none of α_j' appears free in $S\sigma$, either.

Let β_i be new type variables that are not involved in S. Then

$$S\sigma = \forall\langle\beta_i{::}S[\beta_i/\alpha_i]\Gamma_i\rangle.\, S[\beta_i/\alpha_i]\tau.$$

To show that $\sigma' \preceq_{D'} S\sigma$, we choose the substitution $R = [S\alpha_i/\beta_i]$. Then

$$\begin{aligned} R(S[\beta_i/\alpha_i]\tau) &= [S\alpha_i/\beta_i](S[\beta_i/\alpha_i]\tau) \\ &= ([S\alpha_i/\beta_i]S)([\beta_i/\alpha_i]\tau) \\ &= (S[\alpha_i/\beta_i])[\beta_i/\alpha_i]\tau \\ &= S\tau, \end{aligned}$$

and similarly $R(S[\beta_i/\alpha_i]\Gamma_i) = S\Gamma_i$.

The next thing to show is that there exist dictionaries $\mathbf{d}' \sqsubseteq \mathbf{d}$ such that

$$\mathbf{u}':D' \oplus \mathbf{w}':\langle\alpha'_j::\Gamma'_j\rangle \vdash \mathbf{d}':R\langle\beta_i::S[\beta_i/\alpha_i]\Gamma_i\rangle.$$

But since

$$\begin{aligned} R\langle\beta_i::S[\beta_i\alpha_i]\Gamma_i\rangle &= \langle S\alpha_i::S\Gamma_i\rangle \\ &= S\langle\alpha_i::\Gamma_i\rangle, \end{aligned}$$

and $D = D' \uplus \{\alpha'_j::\Gamma'_j\}$, what we need to show is that $\mathbf{u}:D \vdash \mathbf{d}':S\langle\alpha_i::\Gamma_i\rangle$. This follows directly from the given facts that $\langle\alpha_i::\Gamma_i\rangle \sqsubseteq C$ and $\mathbf{u}:D \vdash \mathbf{d}:SC$. Thus $(\lambda x.\lambda\mathbf{w}'.\, x\ \mathbf{d}') : \sigma' \preceq_{\mathbf{u}':D'} S\sigma$. ■

Lemma 6.11 *Let* $(\sigma, C_0, \mathbf{w}) = gen(\tau, A, \mathbf{v}:C, \epsilon)$. *If* $\mathbf{v}':C' \vdash \mathbf{d} : SC_0$, *then there exist a substitution* R, *an augmented context* $\mathbf{u}:D$ *and dictionary expressions* $\mathbf{d}'$ *such that*

$$RA = SA, \quad \mathbf{u}:D \vdash \mathbf{d}' : RC \quad \textit{and} \quad \mathbf{d}' \cong \mathbf{wd}.$$

Furthermore, if $gen(R\tau, RA, \mathbf{u}:D, \epsilon) = (\sigma', D', \mathbf{w}')$, *then*

$$S\sigma = \sigma', \quad D' \sqsubseteq C' \quad \textit{and} \quad \mathbf{w}' \cong \mathbf{w}.$$

Proof: Write $\sigma = \forall\langle\alpha_i::\Gamma_i\rangle.\,\tau$. Let $R = S[\beta_i/\alpha_i]$ where β_i are new type variables and $\mathbf{u}:D = \mathbf{w}:R\langle\alpha_i::\Gamma_i\rangle \oplus \mathbf{v}'':C''$ where $\mathbf{v}'':C''$ is $\mathbf{v}':C'$ restricted to type variables in SC_0. We show that R and $\mathbf{u}:D$ satisfy the requirements.

First, from Lemma 6.6, we know that none of α_i appears in A, so $RA = SA$. Next, also by Lemma 6.6, $C \cong \langle \alpha_i :: \Gamma_i \rangle \oplus C_0$. Hence $RC \cong R\langle \alpha_i :: \Gamma_i \rangle \oplus SC_0$. Now, since $\mathbf{v}'' : C'' \Vdash \mathbf{d} : SC_0$, it follows that $\mathbf{u} : D \Vdash \mathbf{d}' : RC$ for some dictionaries $\mathbf{d}' \cong \mathbf{wd}$.

Furthermore, since β_i are new, it follows that $S\sigma = \forall \langle \beta_i :: R\Gamma_i \rangle . R\tau$. Now suppose that $\sigma' = \forall \langle \rho_j :: \Gamma'_j \rangle . R\tau$, then $S\sigma = \sigma'$ when $\langle \beta_i :: R\Gamma_i \rangle \cong \langle \rho_j :: \Gamma'_j \rangle$. Given the definition of $\mathbf{u} : D$ and the fact that β_i are new, it suffices to show that $D' \cong C''$. This we prove by considering $\mathrm{tv}(S\alpha)$ for $\alpha \in \mathrm{dom}(C')$.

We know from Lemma 6.6 that $\mathrm{dom}(C_0) = C^*(\mathrm{tv}\ A)$. Moreover, since $D \Vdash RC$ and $RA = SA$, by Lemma 3.2 we have $\mathrm{tv}(S\alpha) \subseteq D^*(\mathrm{tv}\ SA)$ for all $\alpha \in \mathrm{dom}(C_0)$ In other words, none of the type variables in SC_0 can be generalized by $gen(R\tau, RA, \mathbf{u} : D, \epsilon)$. Thus, $D' = C''$ and hence we have $S\sigma = \sigma'$, $D' \sqsubseteq C'$ and $\mathbf{w}' \cong \mathbf{w}$. This completes the proof. ∎

Lemma 6.12 *If* $A,\ \mathbf{v} : C \vdash' e \rightsquigarrow e' : \tau$, *then* $dv(e') \subseteq dom(C)$.

Proof: By induction on the structure of the proof $A, \mathbf{v} : C \vdash' e \rightsquigarrow e' : \tau$. The proof for the cases where the last rule in the derivation is (λ-I') or (λ-E') are straightforward and the proof for (*var'*) follows directly from the property (*dvars*) given in Section 5.2.2.

In the remaining case we have a derivation of the form:

$$\frac{A,\ \mathbf{v}_0 : C_0 \vdash' e_1 \rightsquigarrow e'_1 : \tau \qquad A.x : \sigma,\ \mathbf{v} : C \vdash' e_2 \rightsquigarrow e'_2 : \tau'}{A,\ \mathbf{v} : C \vdash' (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\mathtt{let}\ x = \lambda \mathbf{w}.e'_1\ \mathtt{in}\ e'_2) : \tau'}$$

where $(\sigma, \mathbf{v}_1 : C_1, \mathbf{w}) = gen(\tau, A, \mathbf{v}_0 : C_0, \epsilon)$ and $\mathbf{v}_1 : C_1 \sqsubseteq \mathbf{v} : C$.

By induction $\mathrm{dv}(e'_1) \subseteq \mathbf{v}_0$ and $\mathrm{dv}(e'_2) \subseteq \mathbf{v}$. Also, by Lemma 6.6, $\mathrm{dv}(\lambda \mathbf{w}.\ e'_1) \subseteq \mathbf{v}_1$ and hence $\mathrm{dv}(\lambda \mathbf{w}.\ e'_1) \subseteq \mathbf{v}$. It follows that

$$\mathrm{dv}(\mathtt{let}\ x = \lambda \mathbf{w}.e'_1\ \mathtt{in}\ e'_2) \subseteq \mathbf{v},$$

which completes the proof. ■

Lemma 6.14 *If* A, $\mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$ *and* $\mathbf{v}'{:}C' \Vdash \mathbf{d} : SC$, *then* SA, $\mathbf{v}{:}C' \vdash' e \rightsquigarrow [\mathbf{d}/\mathbf{v}]e' : S\tau$.

Proof: By induction on the structure of the proof $A, \mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$. The only nontrivial cases are (*var'*) and (*let'*).

Case (*var'*) : We have a derivation of the form:

$$\frac{A(x) = \forall\langle\alpha_i{::}\Gamma_i\rangle.\,\tau' \qquad \mathbf{v}{:}C \Vdash \mathbf{d}' : [\tau_i/\alpha_i]\,\langle\alpha_i{::}\Gamma_i\rangle}{A,\ \mathbf{v}{:}C \vdash' x \rightsquigarrow x\mathbf{d}' : [\tau_i/\alpha_i]\,\tau'}$$

Let $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\,\tau'$. Without loss of generality, we can assume that S is *safe* for σ; in other words, when applying S to σ, no name clashes occur. Hence $S\sigma = \forall\langle\alpha_i{::}S\Gamma_i\rangle.\,S\tau'$. Let $\tau = [\tau_i/\alpha_i]\,\tau'$. Our goal is to find a instantiation substitution J such that

$$J(S\tau') = S\tau \text{ and } \mathbf{v}'{:}C' \Vdash [\mathbf{d}/\mathbf{v}]\mathbf{d}' : J\langle\alpha_i{::}S\Gamma_i\rangle.$$

Now define $J = S \circ [\tau_i/\alpha_i]$. We first show that $J(S\tau') = S([\tau_i/\alpha_i]\,\tau')$. This we prove by showing that for each α occurring τ', $J(S\alpha) = S([\tau_i/\alpha_i]\,\alpha)$.

If α is bound in σ, *i.e.*, $\alpha = \alpha_j$ for some j, then

$$J(S\alpha_j) = J\alpha_j = S([\tau_i/\alpha_i]\,\alpha_j).$$

Otherwise α is free in σ, so

$$J(S\alpha) = S([\tau_i/\alpha_i]\,(S\alpha)) = S(S\alpha) = S\alpha = S([\tau_i/\alpha_i]\,\alpha),$$

since S is safe for α_i and α is not in $\{\alpha_i\}$. Hence $J(S\tau') = S\tau$ follows from $\tau = [\tau_i/\alpha_i]\,\tau'$.

By a similar argument we can show that $J(S\Gamma_i) = S([\tau_i/\alpha_i]\,\Gamma_i)$. So

$$J\langle\alpha_i{::}S\Gamma_i\rangle = \langle S([\tau_i/\alpha_i]\,\alpha_i){::}S([\tau_i/\alpha_i]\,\Gamma_i)\rangle = S([\tau_i/\alpha_i]\,\langle\alpha_i{::}\Gamma_i\rangle).$$

Then from $\mathbf{v}{:}C \Vdash \mathbf{d}' : [\tau_i/\alpha_i]\,\langle\alpha_i{::}\Gamma_i\rangle$ and $\mathbf{v}'{:}C' \Vdash \mathbf{d} : SC$, it follows that

$$\mathbf{v}'{:}C' \Vdash [\mathbf{d}/\mathbf{v}]\mathbf{d}' : J\langle\alpha_i{::}S\Gamma_i\rangle$$

by transitivity under substitution. Hence we have

$$SA, \mathbf{v}'{:}C' \;\vdash'\; x \rightsquigarrow x([\mathbf{d}/\mathbf{v}]\mathbf{d}') \;:S\tau,$$

which is the derivation required since $x([\mathbf{d}/\mathbf{v}]\mathbf{d}') \equiv [\mathbf{d}/\mathbf{v}](x\mathbf{d}')$.

Case (*let'*) : We have a derivation of the form:

$$\frac{A,\ \mathbf{v}_0{:}C_0 \;\vdash'\; e_1 \rightsquigarrow e'_1 \;:\tau \qquad A.x{:}\sigma,\ \mathbf{v}{:}C \;\vdash'\; e_2 \rightsquigarrow e'_2 \;:\tau'}{A,\ \mathbf{v}{:}C \;\vdash'\; (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\mathtt{let}\ x = \lambda\mathbf{w}.e'_1\ \mathtt{in}\ e'_2) \;:\tau'}$$

where $(\sigma, \mathbf{v}_1{:}C_1, \mathbf{w}) = gen(\tau, A, \mathbf{v}_0{:}C_0, \epsilon)$ and $\mathbf{v}_1{:}C_1 \sqsubseteq \mathbf{v}{:}C$.

The proof is mainly based on Lemma 6.11. Since $\mathbf{v}{:}C' \Vdash \mathbf{d} : SC$ and $C_1 \sqsubseteq C$, it follows that there exist dictionaries $\mathbf{d}_1 \sqsubseteq \mathbf{d}$ such that $\mathbf{v}'{:}C' \Vdash \mathbf{d}_1 : SC_1$. Then applying Lemma 6.11 to σ, $\mathbf{v}'{:}C'$ and S, we obtain a substitution R and a context $\mathbf{u}{:}D$ and dictionary expressions $\mathbf{d}_0$ such that

$$RA = SA,\ \ \mathbf{u}{:}D \Vdash \mathbf{d}_0 : RC_0,\ \ S\sigma = \sigma',\ \ C'' \sqsubseteq C'\ \ \mathbf{d}_0 \cong \mathbf{w}\mathbf{d}_1 \ \text{ and } \ \mathbf{w}' \cong \mathbf{w}$$

where $(\sigma', C'', \mathbf{w}') = gen(R\tau, SA, \mathbf{u}{:}D, \epsilon)$.

In addition, by Lemma 6.6, $\mathbf{v}_0{:}C_0 \cong \mathbf{w}{:}\langle\alpha_i{::}\Gamma_i\rangle \oplus \mathbf{v}_1{:}C_1$. Then without loss of generality, we can assume that $\mathbf{v}_0 = \mathbf{w}\mathbf{v}_1$ and $\mathbf{d}_0 = \mathbf{w}\mathbf{d}_1$. Hence $[\mathbf{d}_0/\mathbf{v}_0] = [\mathbf{d}_1/\mathbf{v}_1]$ and the required derivation can be constructed:

$$\frac{\dfrac{\dfrac{A,\ \mathbf{v}_0{:}C_0 \;\vdash'\; e_1 \rightsquigarrow e'_1 \;:\tau}{RA,\ \mathbf{u}{:}D \;\vdash'\; e_1 \rightsquigarrow [\mathbf{d}_0/\mathbf{v}_0]e'_1 \;:R\tau}\,(a)}{SA,\ \mathbf{u}{:}D \;\vdash'\; e_1 \rightsquigarrow [\mathbf{d}_1/\mathbf{v}_1]e'_1 \;:R\tau}\,(c) \qquad \dfrac{\dfrac{A.x{:}\sigma,\ \mathbf{v}{:}C \;\vdash'\; e_2 \rightsquigarrow e'_2 \;:\tau'}{SA.x{:}S\sigma,\ \mathbf{v}'{:}C' \;\vdash'\; e_2 \rightsquigarrow [\mathbf{d}/\mathbf{v}]e'_2 \;:S\tau'}\,(b)}{SA.x{:}\sigma',\ \mathbf{v}'{:}C' \;\vdash'\; e_2 \rightsquigarrow [\mathbf{d}/\mathbf{v}]e'_2 \;:S\tau'}\,(d)}{SA,\ \mathbf{v}'{:}C' \;\vdash'\; (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\mathtt{let}\ x = \lambda\mathbf{w}'.[\mathbf{d}_1/\mathbf{v}_1]e'_1\ \mathtt{in}\ [\mathbf{d}/\mathbf{v}]e'_2) \;:S\tau'}$$

where steps (a) and (b) are obtained by induction while (c) and (d) are justified by the equalities $SA = RA$, $S\sigma = \sigma'$ and $[\mathbf{d}_0/\mathbf{v}_0] = [\mathbf{d}_1/\mathbf{v}_1]$.

To complete the proof, note that by Lemma 6.12, $\mathrm{dv}(e'_1) \subseteq \mathbf{v}_0$ and hence $\mathrm{dv}(\lambda\mathbf{w}.\, e'_1) \subseteq \mathbf{v}_1$. So:

$$\begin{aligned}
[\mathbf{d}/\mathbf{v}](\texttt{let}\ x = \lambda\mathbf{w}.e'_1\ \texttt{in}\ e'_2) &\equiv \texttt{let}\ x = [\mathbf{d}/\mathbf{v}](\lambda\mathbf{w}.e'_1)\ \texttt{in}\ [\mathbf{d}/\mathbf{v}]e'_2 \\
&\equiv \texttt{let}\ x = [\mathbf{d}_0/\mathbf{v}_0](\lambda\mathbf{w}.e'_1)\ \texttt{in}\ [\mathbf{d}/\mathbf{v}]e'_2 \\
&\equiv \texttt{let}\ x = [\mathbf{d}_1/\mathbf{v}_1](\lambda\mathbf{w}.e'_1)\ \texttt{in}\ [\mathbf{d}/\mathbf{v}]e'_2 \\
&\equiv \texttt{let}\ x = \lambda\mathbf{w}.[\mathbf{d}_1/\mathbf{v}_1]e'_1\ \texttt{in}\ [\mathbf{d}/\mathbf{v}]e'_2 \\
&= \texttt{let}\ x = \lambda\mathbf{w}'.[\mathbf{d}_1/\mathbf{v}_1]e'_1\ \texttt{in}\ [\mathbf{d}/\mathbf{v}]e'_2
\end{aligned}$$

This completes the proof. ■

Lemma 6.15 *If* A', $\mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$ *and* $K : A' \preceq_{\mathbf{v}:C} A$, *then* A, $\mathbf{v}{:}C \vdash' e \rightsquigarrow e'' : \tau$ *with* A, $\mathbf{v}{:}C \vdash Ke' = e'' : \tau$.

Proof: By induction on the structure of $A', \mathbf{v}{:}C \vdash' e \rightsquigarrow e' : \tau$. The case for ($\lambda$-elim) is trivial and has been omitted.

case (*var'*): We have a derivation of the form

$$\frac{A'(x) = \forall\langle\alpha'_j{::}\Gamma'_j\rangle.\,\nu' \qquad \mathbf{v}{:}C \Vdash \mathbf{d}' : [\tau'_j/\alpha'_j]\,\langle\alpha'_j{::}\Gamma'_j\rangle}{A',\ \mathbf{v}{:}C \vdash' x \rightsquigarrow x\mathbf{d}' : [\tau'_j/\alpha'_j]\,\nu'}$$

Suppose $A(x) = \forall\langle\alpha_i{::}\Gamma_i\rangle.\,\nu$. Given that $K : A' \preceq_{\mathbf{v}:C} A$, we have

$$\lambda x.\, K(x) : \forall\langle\alpha'_j{::}\Gamma'_j\rangle.\,\nu' \preceq_{\mathbf{v}:C} \forall\langle\alpha_i{::}\Gamma_i\rangle.\,\nu\,.$$

Moreover, by hypothesis and definition of conversions we have:

$$(\lambda x.\, x\mathbf{d}') : [\tau'_j/\alpha'_j]\,\nu' \preceq_{\mathbf{v}:C} \forall\langle\alpha'_j{::}\Gamma'_j\rangle.\,\nu'\,.$$

Then we can compose these two conversion using Lemma 6.3:

$$(\lambda x.\,(Kx)\mathbf{d}') : [\tau_j'/\alpha_j']\,\nu' \preceq_{\mathbf{v}:C} \forall\langle\alpha_i{::}\Gamma_i\rangle.\,\nu\,.$$

So there are types τ_i and dictionary expressions $\mathbf{d}$ such that

$$[\tau_i/\alpha_i]\,\nu \;=\; [\tau_j'/\alpha_j']\,\nu', \quad \mathbf{v}{:}C \Vdash \mathbf{d} : [\tau_i/\alpha_i]\,\langle\alpha_i{::}\Gamma_i\rangle,$$

$$\text{and} \quad \mathbf{v}{:}C \vdash (Kx)\mathbf{d}' = x\mathbf{d}.$$

Then using (*var'*), we can construct the derivation:

$$A,\ \mathbf{v}{:}C \vdash' x \rightsquigarrow x\mathbf{d} : [\tau_j'/\alpha_j']\,\nu'\,.$$

Finally, note that

$$\begin{aligned} A,\ \mathbf{v}{:}C \vdash K(x\mathbf{d}') &= (Kx)\mathbf{d}' \\ &= x\mathbf{d} \end{aligned}$$

which establish the required equality.

Case (λ-intro'): We have a derivation of the form:

$$\frac{A'.x:\tau'.\ \mathbf{v}{:}C \vdash' e_1 \rightsquigarrow e_1' : \tau}{A',\ \mathbf{v}{:}C \vdash' \lambda x.e_1 \rightsquigarrow \lambda x.e_1' : \tau' \to \tau}$$

By hypothesis, $K : A' \preceq_{\mathbf{v}:C} A$ and hence by Lemma 6.4(3):

$$K_x : (A'.x:\tau') \preceq_{\mathbf{v}:C} (A.x:\tau')\,.$$

So, by induction, $A.x:\tau',\ \mathbf{v}{:}C \vdash' e_1 \rightsquigarrow e_1'' = K_x e_1' : \tau$ and hence

$$A,\ \mathbf{v}{:}C \vdash' \lambda x.e_1 \rightsquigarrow \lambda x.e_1'' : \tau' \to \tau$$

with

$$\begin{aligned} A, \mathbf{v}{:}C \vdash K(\lambda x.e_1') &= \lambda x.(K_x e_1') && (\textit{Lemma } 6.4(1)) \\ &= \lambda x.e_1'' : \tau' \to \tau && (\vdash K_x e_1' = e_1'') \end{aligned}$$

Case (*let'*): We have a derivation of the form:

$$\frac{A',\ \mathbf{v}'{:}C' \vdash' e_1 \rightsquigarrow e_1' : \tau' \qquad A'.x{:}\sigma',\ \mathbf{v}{:}C \vdash' e_2 \rightsquigarrow e_2' : \tau}{A',\ \mathbf{v}{:}C \vdash' (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\mathtt{let}\ x = \lambda\mathbf{w}.e_1'\ \mathtt{in}\ e_2') : \tau}$$

where $(\sigma', \mathbf{v}''{:}C'', \mathbf{w}) = gen(\tau', A, \mathbf{v}'{:}C', \epsilon)$ and $C'' \preceq C$.

Without loss of generality, we assume that $\mathrm{dom}(C) \cap \mathrm{dom}(C') = \mathrm{dom}(C'')$. By Lemma 6.6, $\sigma' = \forall\langle\alpha_i{::}\Gamma_i\rangle.\,\tau'$ and $C' = \{\alpha_i{::}\Gamma_i\} \uplus C''$. Hence

$$C' \uplus (C \setminus C'') = C \uplus \langle\alpha_i{::}\Gamma_i\rangle.$$

Now let $\mathbf{u}{:}D = \mathbf{v}{:}C \setminus \mathbf{v}''{:}C''$. By Lemma 6.13, we obtain a derivation:

$$A',\ \mathbf{v}'{:}C' \oplus \mathbf{u}{:}D \vdash' e_1 \rightsquigarrow e_1' : \tau'.$$

Furthermore, since $K : A' \preceq_{\mathbf{v}:C} A$ and $\mathbf{v}{:}C \sqsubseteq \mathbf{v}'{:}C' \oplus \mathbf{u}{:}D$, by Lemma 6.5 we have $K : A' \preceq_{\mathbf{v}'\mathbf{u}:C'\oplus D} A$. Thus by induction there is a derivation

$$A,\ \mathbf{v}'{:}C' \oplus \mathbf{u}{:}D \vdash' e_1 \rightsquigarrow e_1'' : \tau'$$

with $A,\ \mathbf{v}{:}C' \oplus \mathbf{u}{:}D \vdash K e_1' = e_1'' : \tau'$.

Next, we consider the derivation:

$$A'.x{:}\sigma',\ \mathbf{v}{:}C \vdash' e_2 \rightsquigarrow e_2' : \tau\,.$$

Let $(\sigma, C_0, \mathbf{w}') = gen(\tau', A', \mathbf{v}'{:}C' \oplus \mathbf{u}{:}D, \epsilon)$. It follows from Lemma 6.8 and $C'' \preceq C$ that

$$(\lambda x.\lambda\mathbf{w}.\,x\mathbf{w}') : \sigma' \preceq_{\mathbf{u}:D} \sigma, \quad \text{and} \quad C_0 \preceq C$$

But $\mathbf{u}{:}D \sqsubseteq \mathbf{v}{:}C$, so by Lemma 6.2 we have

$$(\lambda x.\lambda\mathbf{w}.\,x\mathbf{w}') : \sigma' \preceq_{\mathbf{v}:C} \sigma$$

and hence

$$K_x[\lambda\mathbf{w}.x\mathbf{w}'/x] : A'.x{:}\sigma' \preceq_{\mathbf{v}:C} A.x{:}\sigma\,.$$

Therefore, by induction

$$A.x{:}\sigma,\ \mathbf{v}{:}C \vdash' e_2 \rightsquigarrow e_2'' = (K_x[\lambda \mathbf{w}.x\mathbf{w}'/x])e_2' : \tau.$$

It then follows from (*let'*) that

$$A, \mathbf{v}{:}C \vdash' (\texttt{let } x = e_1 \texttt{ in } e_2) \rightsquigarrow (\texttt{let } x = \lambda \mathbf{w}'.e_1'' \texttt{ in } e_2'') : \tau.$$

Finally, note that

$$\begin{array}{lll}
A,\ \mathbf{v}{:}C \vdash & K(\texttt{let } x = \lambda \mathbf{w}.e_1' \texttt{ in } e_2') & \\
= & \texttt{let } x = \lambda \mathbf{w}.\, K e_1' \texttt{ in } K_x e_2' & (\textit{Lemma } 6.4\,(1)) \\
= & \texttt{let } x = \lambda \mathbf{w}.\, e_1'' \texttt{ in } K_x e_2' & \\
= & \texttt{let } x = \lambda \mathbf{w}.\, (\lambda.\mathbf{w}'.e_1'')\mathbf{w}' \texttt{ in } K_x e_2' & (\beta_d) \\
= & \texttt{let } x = (\lambda x.\lambda \mathbf{w}.\, x\mathbf{w}')(\lambda \mathbf{w}'.e_1'') \texttt{ in } K_x e_2' & (\beta_d) \\
= & \texttt{let } x = [\lambda \mathbf{w}'.e_1''/x](\lambda \mathbf{w}.x\mathbf{w}') \texttt{ in } K_x e_2' & (\text{substitution}) \\
= & \texttt{let } x = \lambda \mathbf{w}'.e_1'' \texttt{ in } [\lambda \mathbf{w}.x\mathbf{w}'/x](K_x e_2') & (\textit{Lemma } 6.1\,(1)) \\
= & \texttt{let } x = \lambda \mathbf{w}'.e_1'' \texttt{ in } (K_x[\lambda \mathbf{w}.x\mathbf{w}'/x])e_2' & (\textit{Lemma } 6.4\,(4)) \\
= & \texttt{let } x = \lambda \mathbf{w}'.e_1'' \texttt{ in } e_2'' &
\end{array}$$

which establish the required equality. ■

Theorem 6.16 *If* $A,\ \mathbf{v}{:}C \vdash e \rightsquigarrow e' : \sigma$, *then there is a context* $\mathbf{v}'{:}C'$, *a type* τ' *and a term* e'' *such that* $\mathbf{v}{:}C \sqsubseteq \mathbf{v}'{:}C'$ *and* $A,\ \mathbf{v}'{:}C' \vdash' e \rightsquigarrow e'' : \tau'$. *Furthermore, if* $gen(\tau', A, \mathbf{v}'{:}C', \epsilon) = (\sigma', C'', \mathbf{w})$, *then* $A,\ \mathbf{v}{:}C \vdash K(\lambda \mathbf{w}.e'') = e' : \sigma$ *where* $K : \sigma \preceq_{\mathbf{v}:C} \sigma'$.

Proof: By induction on the length of the derivation $A,\ \mathbf{v}{:}C \vdash e \rightsquigarrow e' : \sigma$. Consider the last step of the derivation:

Case (*var*): We have a derivation of the form:

$$\frac{A(x) = \sigma}{A,\ \mathbf{v}{:}C \ \vdash\ x \rightsquigarrow x\ :\sigma}$$

Write $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\,\tau$. Define $\mathbf{v}'{:}C'$ and τ' as follows:

$$\mathbf{v}'{:}C' = \mathbf{u}{:}S\langle\alpha_i{::}\Gamma_i\rangle \oplus \mathbf{v}{:}C, \quad \tau' = S\tau$$

with S of the form $[\beta_i/\alpha_i]$ where β_i and $\mathbf{u}$ are new variables. Then by definition $\mathbf{v}{:}C \sqsubseteq \mathbf{v}'{:}C'$ and $\mathbf{v}'{:}C' \Vdash\ \mathbf{u} : S\langle\alpha_i{::}\Gamma_i\rangle$. So by (*var'*),

$$A,\ \mathbf{v}'{:}C' \ \vdash'\ x \rightsquigarrow x\mathbf{u}\ :\tau'.$$

Furthermore, suppose that $gen(\tau', A, \mathbf{v}'{:}C', \epsilon) = (\sigma', C'', \mathbf{w})$. Then by the definition of *gen*, we can write $\sigma' = \forall\langle\alpha''_k{::}\Gamma''_k\rangle.\forall\langle\beta_i{::}S\Gamma_i\rangle.\tau'$, where $\langle\alpha''_k{::}\Gamma''_k\rangle \sqsubseteq C$. This in turn is α-equivalent to $\forall\langle\alpha''_k{::}\Gamma''_k\rangle.\forall\langle\alpha_i{::}\Gamma_i\rangle.\tau'$. Thus $\sigma' = \forall\langle\alpha''_k{::}\Gamma''_k\rangle.\sigma$ and $\mathbf{v}{:}C \oplus \mathbf{u}{:}\langle\alpha_i{::}\Gamma_i\rangle \Vdash\ \mathbf{w} : (\langle\alpha''_k{::}\Gamma''_k\rangle \oplus \langle\alpha_i{::}\Gamma_i\rangle)$. So, $K : \sigma \preceq_{\mathbf{v}:C} \sigma'$ where $K = \lambda x.\lambda\mathbf{u}.x\mathbf{w}$. Finally, we have

$$\begin{aligned}
A,\ \mathbf{v}{:}C \ \vdash\ & K(\lambda\mathbf{w}.x\mathbf{u}) \\
=\ & \lambda\mathbf{u}.(\lambda\mathbf{w}.x\mathbf{u})\mathbf{w} && (\beta),\ (\beta_d) \\
=\ & \lambda\mathbf{u}.x\mathbf{u} && (\beta_d) \\
=\ & x\ :\sigma && (\eta_d)
\end{aligned}$$

which establishes the required equality.

Case ($\forall$-E): we have a derivation of the form:

$$\frac{A,\ \mathbf{v}{:}C \ \vdash\ e \rightsquigarrow e'\ :\forall\alpha{::}\Gamma.\sigma \qquad \mathbf{v}{:}C \Vdash\ d{:}(\tau{::}\Gamma)}{A,\ \mathbf{v}{:}C \ \vdash\ e \rightsquigarrow e'{\cdot}d\ :[\tau/\alpha]\,\sigma}$$

By induction,

$$A,\ \mathbf{v}'{:}C' \vdash' \ e \rightsquigarrow e'' : \tau'$$

with $\mathbf{v}{:}C \sqsubseteq \mathbf{v}'{:}C'$ and

$$A,\ \mathbf{v}{:}C \ \vdash \ K\,(\lambda \mathbf{w}.\ e'') = e' \ : \forall \alpha{::}\Gamma.\sigma$$

where $(\sigma', C'', \mathbf{w}) = gen(\tau', A, \mathbf{v}'{:}C', \epsilon)$ and $K : \forall \alpha{::}\Gamma.\sigma \preceq_{\mathbf{v}:C} \sigma'$.

Using the hypothesis $\mathbf{v}{:}C \Vdash d{:}(\tau{::}\Gamma)$, we have

$$(\lambda x.xd) : [\tau/\alpha]\sigma \preceq_{\mathbf{v}:C} \forall \alpha{::}\Gamma.\sigma,$$

which can be composed with K by Lemma 6.3 to give

$$(\lambda x.(Kx)d) \ : \ [\tau/\alpha]\sigma \preceq_{\mathbf{v}:C} \sigma'.$$

This conversion yields the required equality:

$$A,\ \mathbf{v}{:}C \ \vdash \ (\lambda x.(Kx)d)\,(\lambda \mathbf{w}.\ e'') = (K\,(\lambda \mathbf{w}.\ e''))d = e'd \ : [\tau/\alpha]\sigma.$$

Case ($\forall$-I): we have a derivation of the form:

$$\frac{A,\ \mathbf{u}_1{:}C_1 \oplus v{:}(\alpha{::}\Gamma) \oplus \mathbf{u}_2{:}C_2 \ \vdash \ e \rightsquigarrow e' : \sigma}{A,\ \mathbf{u}_1\mathbf{u}_2{:}C_1C_2 \ \vdash \ e \rightsquigarrow \lambda v.\ e' \ : \forall \alpha{::}\Gamma.\sigma}$$

where $\alpha \notin \mathrm{tv}\ A \cup reg(C_1C_2)$.

Let $\mathbf{u}{:}C_u = \mathbf{u}_1{:}C_1 \oplus v{:}(\alpha{::}\Gamma) \oplus \mathbf{u}_2{:}C_2$. By induction,

$$A,\ \mathbf{v}'{:}C' \vdash' \ e \rightsquigarrow e'' : \tau'$$

with $\mathbf{u}{:}C_u \sqsubseteq \mathbf{v}'{:}C'$ and

$$A,\ \mathbf{u}{:}C_u \ \vdash \ K\,(\lambda \mathbf{w}.\ e'') = e' \ : \sigma$$

where $K : \sigma \preceq_{\mathbf{u}:C_u} \sigma'$ and $(\sigma', C'', \mathbf{w}) = gen(\tau', A, \mathbf{v}'{:}C', \epsilon)$.

Moreover, it is straightforward to show that

$$(\lambda x.\lambda v.Kx) \ : \ \forall\alpha{::}\Gamma.\sigma \preceq_{\mathbf{u}_1\mathbf{u}_2:C_1C_2} \sigma'.$$

Hence we can establish the required equality:

$$\begin{aligned} A,\ \mathbf{u}_1\mathbf{u}_2{:}C_1C_2 \quad \vdash \quad & (\lambda x.\lambda v.Kx)(\lambda \mathbf{w}.\ e'') \\ = \quad & \lambda v.\ K(\lambda \mathbf{w}.\ e'') \\ = \quad & \lambda v.\ e' \ : \forall\alpha{::}\Gamma.\sigma \end{aligned}$$

using (β) and (β_d).

Case (λ-E): we have a derivation of the form:

$$\frac{A,\ \mathbf{v}{:}C \ \vdash \ e_1 \rightsquigarrow e_1' \ : \tau \to \tau' \qquad A,\ \mathbf{v}{:}C \ \vdash \ e_2 \rightsquigarrow e_2' \ : \tau}{A,\ \mathbf{v}{:}C \ \vdash \ e_1\ e_2 \rightsquigarrow e_1'\ e_2' \ : \tau'}$$

By induction,

$$A,\ \mathbf{v}_1{:}C_1 \ \vdash' \ e_1 \rightsquigarrow e_1'' \ : \nu$$

with $\mathbf{v}{:}C \sqsubseteq \mathbf{v}_1{:}C_1$ and

$$A,\ \mathbf{v}{:}C \ \vdash \ K_1(\lambda \mathbf{w}_1.\ e_1'') = e_1' \ : \tau \to \tau'$$

where $K_1 \ : \ \tau \to \tau' \preceq_{\mathbf{v}:C} \sigma_1$ and $(\sigma_1, \mathbf{u}_1{:}C_1', \mathbf{w}_1) = gen(\nu, A, \mathbf{v}_1{:}C_1, \epsilon)$. Write $\sigma_1 = \forall\langle\alpha_i{::}\Gamma_i\rangle.\ \nu$. It follows from the definition of conversions that there are types τ_i to form a substitution $R = [\tau_i/\alpha_i]$ and dictionary expressions $\mathbf{d_1}$ such that

$$R\nu = \tau \to \tau', \quad \mathbf{v}{:}C \Vdash \mathbf{d}_1 : (R\langle\alpha_i{::}\Gamma_i\rangle) \text{ and } \mathbf{v}{:}C \ \vdash \ K_1 = \lambda x.\ x\mathbf{d}_1.$$

To apply R to the syntax-directed derivation above, according to Lemma 6.14, we need to show that there exist dictionaries $\mathbf{d}$ such that $\mathbf{v}{:}C \Vdash \mathbf{d} : RC_1$.

By Lemma 6.6,

$$\mathbf{v}_1{:}C_1 \cong \mathbf{w}_1{:}\langle\alpha_i{::}\Gamma_i\rangle \oplus \mathbf{u}_1{:}C_1' \text{ and } \mathrm{dom}(C_1') = (C_1)^*(\mathrm{tv}\ A).$$

Thus $\mathbf{v}{:}C \Vdash \mathbf{d} : RC_1$ where $\mathbf{d} \cong \mathbf{d}_1\mathbf{u}_1$. So by Lemma 6.14 we obtain:

$$RA,\ \mathbf{v}{:}C \vdash' \ e_1 \rightsquigarrow [\mathbf{d}_1/\mathbf{w}_1]e_1'' : \tau \to \tau'.$$

Furthermore, since none of the α_i appears free in A, this is equivalent to

$$A,\ \mathbf{v}{:}C \vdash' \ e_1 \rightsquigarrow [\mathbf{d}_1/\mathbf{w}_1]e_1'' : \tau \to \tau'.$$

Note also that

$$A,\ \mathbf{v}{:}C \vdash \ e_1' = K_1(\lambda\mathbf{w}_1.\ e_1'') = (\lambda x.\ x\mathbf{d}_1)(\lambda\mathbf{w}_1.\ e_1'') = [\mathbf{d}_1/\mathbf{w}_1]e_1'' : \tau \to \tau'.$$

By a similar argument, $A,\ \mathbf{v}{:}C \vdash' \ e_2 \rightsquigarrow [\mathbf{d}_2/\mathbf{w}_2]e_2'' : \tau$ for some some $\mathbf{d}_2$, S, C_2 and $\mathbf{w}_2$ such that

$$\mathbf{v}{:}C \Vdash \mathbf{d}_2 : SC_2 \quad \text{and} \quad A,\ \mathbf{v}{:}C \vdash \ e_2' = [\mathbf{d}_2/\mathbf{w}_2]e_2'' : \tau.$$

Hence we can construct the derivation:

$$\frac{A,\ \mathbf{v}{:}C \vdash' \ e_1 \rightsquigarrow [\mathbf{d}_1/\mathbf{w}_1]e_1'' : \tau \to \tau'. \qquad A,\ \mathbf{v}{:}C \vdash' \ e_2 \rightsquigarrow [\mathbf{d}_2/\mathbf{w}_2]e_2'' : \tau}{A,\ \mathbf{v}{:}C \vdash' \ e_1 e_2 \rightsquigarrow [\mathbf{d}_1/\mathbf{w}_1]e_1''[\mathbf{d}_2/\mathbf{w}_2]e_2'' : \tau'}$$

Finally, if $gen(\tau', A, \mathbf{v}{:}C, \epsilon) = (\sigma, C', \mathbf{w})$ then obviously $(\lambda x.\ x\mathbf{w}) : \tau' \preceq_{\mathbf{v}:C} \sigma$, and this conversion satisfies the required equality:

$$A,\ \mathbf{v}{:}C \vdash \ (\lambda x.\ x\mathbf{w})(\lambda\mathbf{w}.\ [\mathbf{d}_1/\mathbf{w}_1]e_1''[\mathbf{d}_2/\mathbf{w}_2]e_2'') = [\mathbf{d}_1/\mathbf{w}_1]e_1''[\mathbf{d}_2/\mathbf{w}_2]e_2'' = e_1'\ e_2'$$

using (β) and (β_d).

Case (λ-I): we have a derivation of the form:

$$\frac{A.x{:}\tau',\ \mathbf{v}{:}C \vdash \ e \rightsquigarrow e' : \tau}{A,\ \mathbf{v}{:}C \vdash \ \lambda x.e \rightsquigarrow \lambda x.e' : \tau' \to \tau}$$

By induction,

$$A.x:\tau',\ \mathbf{v}':C' \vdash' \ e \rightsquigarrow e'' : \nu$$

with $\mathbf{v}:C \sqsubseteq \mathbf{v}':C'$ and

$$A.x:\tau',\ \mathbf{v}:C \ \vdash \ K\,(\lambda\mathbf{w}.\, e'') = e' : \tau$$

where $K : \tau \preceq_{\mathbf{v}:C} \sigma$ and $(\sigma, \mathbf{u}:C'', \mathbf{w}) = gen(\nu, A.x:\tau', \mathbf{v}':C', \epsilon)$. Suppose that $\sigma = \forall\langle\alpha_i::\Gamma_i\rangle.\,\nu$, then there are types τ_i to form a substitution $S = [\tau_i/\alpha_i]$ and dictionary expressions $\mathbf{d}$ such that

$$S\nu = \tau \to \tau', \quad \mathbf{v}:C \Vdash \mathbf{d} : (S\langle\alpha_i::\Gamma_i\rangle) \text{ and } \mathbf{v}:C \ \vdash \ K = \lambda x.\, x\mathbf{d}.$$

By Lemma 6.6,

$$\mathbf{v}':C' \cong \mathbf{w}:\langle\alpha_i::\Gamma_i\rangle \oplus \mathbf{u}:C'' \text{ and } \mathrm{dom}(C'') = (C_1)^*(\mathrm{tv}\ A \ \cup\ \mathrm{tv}\ \tau').$$

Thus $\mathbf{v}:C \Vdash \mathbf{d}' : SC'$ where $\mathbf{d}' \cong \mathbf{du}$. Now applying Lemma 6.14 to S and the given derivation, we obtain:

$$S(A.x:\tau'),\ \mathbf{v}:C \ \vdash' \ \epsilon \rightsquigarrow [\mathbf{d}/\mathbf{w}]\epsilon_1'' : \tau.$$

Since none of the α_i appears in $A.x:\tau'$, this is equivalent to

$$A.x:\tau',\ \mathbf{v}:C \ \vdash' \ \epsilon \rightsquigarrow [\mathbf{d}/\mathbf{w}]\epsilon_1'' : \tau$$

and hence

$$A,\ \mathbf{v}:C \ \vdash' \ \lambda x.\epsilon \rightsquigarrow \lambda x.[\mathbf{d}/\mathbf{w}]\epsilon'' : \tau' \to \tau.$$

Note also that

$$\begin{aligned} A.x:\tau',\ \mathbf{v}:C \ \vdash' \ e' &= K\,(\lambda\mathbf{w}.\, e'') \\ &= \lambda x.\, x\mathbf{d}(\lambda\mathbf{w}.\, e'') \\ &= [\mathbf{d}/\mathbf{w}]e'' : \tau' \to \tau \end{aligned}$$

and hence by (λ-intro'),

$$A,\ \mathbf{v}{:}C \ \vdash\ \lambda x.\,[\mathbf{d}/\mathbf{w}]e'' = \lambda x.\,e' \ :\tau' \to \tau.$$

Finally, Suppose that $gen(\tau' \to \tau, A, \mathbf{v}{:}C, \epsilon) = (\sigma, C_0, \mathbf{w})$. Then obviously $(\lambda x.\,x\mathbf{w}) : \tau \to \tau' \preceq_{\mathbf{v}:C} \sigma$, and this conversion satisfies the required equality:

$$A,\ \mathbf{v}{:}C \ \vdash\ (\lambda x.\,x\mathbf{w})(\lambda \mathbf{w}.\,\lambda x.\,[\mathbf{d}/\mathbf{w}]e'') = \lambda x.\,[\mathbf{d}/\mathbf{w}]e'' = \lambda x.\,e'$$

using (β) and (β_d).

Case (*let*): we have a derivation of the form:

$$\frac{A,\ \mathbf{v}{:}C \ \vdash\ e_1 \rightsquigarrow e_1' \ :\sigma \qquad A.x{:}\sigma,\ \mathbf{v}{:}C \ \vdash\ e_2 \rightsquigarrow e_2' \ :\tau}{A,\ \mathbf{v}{:}C \ \vdash\ (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2') \ :\tau}$$

By induction,

$$A,\ \mathbf{v}_1{:}C_1 \ \vdash'\ e_1 \rightsquigarrow e_1'' \ :\tau_1$$

with $\mathbf{v}{:}C \sqsubseteq \mathbf{v}_1{:}C_1$ and

$$A,\ \mathbf{v}{:}C \ \vdash\ K_1(\lambda \mathbf{w}_1.\,e_1'') = e_1' \ :\sigma$$

where $K_1 : \sigma \preceq_{\mathbf{v}:C} \sigma_1$ and $(\sigma_1, \mathbf{u}_1{:}C_1', \mathbf{w}_1) = gen(\tau_1, A, \mathbf{v}_1{:}C_1, \epsilon)$.

Similarly, we have

$$A.x{:}\sigma,\ \mathbf{v}_2{:}C_2 \ \vdash'\ e_2 \rightsquigarrow e_2'' \ :\tau_2$$

with $\mathbf{v}{:}C \sqsubseteq \mathbf{v}_2{:}C_2$ and

$$A.x{:}\sigma,\ \mathbf{v}{:}C \ \vdash\ K_2\,(\lambda \mathbf{w}_2.\,e_2'') = e_2' \ :\tau$$

where $K_2 : \tau \preceq_{\mathbf{v}:C} \sigma_2$ and $(\sigma_2, \mathbf{u}_2{:}C_2', \mathbf{w}_2) = gen(\tau_2,\ A.x{:}\sigma,\ \mathbf{v}_2{:}C_2,\ \epsilon)$.

Our first goal is to construct a derivation for the `let`-expression using A and $\mathbf{v}_2{:}C_2$. Without loss of generality, we can assume that $\mathrm{dom}(C_1) \cap \mathrm{dom}(C_2) =$

$\mathrm{dom}(C)$. Now note that by Lemma 6.6, $\mathrm{dom}(C_1') = (C_1)^*(\mathrm{tv}\ A)$. But obviously $(C_1)^*(\mathrm{tv}\ A) = C^*(\mathrm{tv}\ A)$, so $\mathbf{u}_1{:}C_1' \sqsubseteq \mathbf{v}{:}C$ and hence $\mathbf{u}_1{:}C_1' \sqsubseteq \mathbf{v}_2{:}C_2$. Furthermore, by Lemma 6.2, $K_1 : \sigma \preceq_{\mathbf{v}_2:C_2} \sigma_1$. So, by definition,

$$[K_1 x/x] \ :\ A.x{:}\sigma \preceq_{\mathbf{v}_2:C_2} A.x{:}\sigma_1.$$

Thus, using the conversion $[K_1 x/x]$, we can construct the following derivation:

$$\frac{A,\ \mathbf{v}_1{:}C_1 \ \vdash'\ e_1 \rightsquigarrow e_1'' : \tau_1 \qquad \dfrac{A.x{:}\sigma,\ \mathbf{v}_2{:}C_2 \ \vdash'\ e_2 \rightsquigarrow e_2'' : \tau_2}{A.x{:}\sigma_1,\ \mathbf{v}_2{:}C_2 \ \vdash'\ e_2 \rightsquigarrow e_2''' : \tau_2}}{A,\ \mathbf{v}_2{:}C_2 \ \vdash'\ (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\mathtt{let}\ x = \lambda\mathbf{w}_1.\, e_1''\ \mathtt{in}\ e_2''') : \tau_2}$$

where, by Lemma 6.15, e_2''' is related to e_2'' by the equality

$$A.x{:}\sigma_1,\ \mathbf{v}_2{:}C_2 \ \vdash\ [K_1 x/x]e_2'' = e_2''' : \tau_2,$$

which in turn, after applying rule (*abs*-d) repetitively, gives:

$$A.x{:}\sigma_1,\ \mathbf{v}{:}C \ \vdash\ \lambda\mathbf{w}_2.\, [K_1 x/x]e_2'' = \lambda\mathbf{w}_2.\, e_2''' : \sigma_2$$

Next, we use σ_2 as an intermediate step to construct the required conversion. Recall the definition of σ_2—$(\sigma_2, C_2', \mathbf{w}_2) = \mathit{gen}(\tau_2, A.x : \sigma, \mathbf{v}_2{:}C_2, \epsilon)$. Now let $(\sigma_3, C_2'', \mathbf{w}_3) = \mathit{gen}(\tau_2, A, \mathbf{v}_2{:}C_2, \epsilon)$. A straightforward comparison of these two generalizations based on Lemma 6.6 gives: there exist $\mathbf{u}{:}\langle\alpha_k{::}\Gamma_k\rangle \sqsubseteq \mathbf{v}{:}C$ such that

$$\{\alpha_k\} \subseteq \mathrm{dom}(C) \setminus \mathrm{tv}(\sigma) \ \text{ and } \ (\lambda x.\lambda\mathbf{w}_2.\, x\mathbf{w}_3) : \sigma_2 \preceq_{\mathbf{u}:\langle\alpha_k::\Gamma_k\rangle} \sigma_3.$$

Then composing with K_2 by Lemma 6.3, we obtain the required conversion:

$$(\lambda x.\, K_2\,(\lambda\mathbf{w}_2.\, x\mathbf{w}_3)) : \tau \preceq_{\mathbf{v}:C} \sigma_3.$$

It remains to be shown that this conversion relates the translation of $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ in the original derivation to that in the syntax-directed derivation given above using (*cong*-let):

$$
\begin{aligned}
A,\ \mathbf{v}{:}C \quad \vdash \quad & (\lambda x.\, K_2\,(\lambda \mathbf{w}_2.\, x\, \mathbf{w}_3))\,(\lambda \mathbf{w}_3.\, \mathtt{let}\ x = \lambda \mathbf{w}_1.\, e_1''\ \mathtt{in}\ e_2''') \\
& \quad (\text{by } (\beta) \text{ and } (\beta_d)) \\
= \quad & K_2\,(\lambda \mathbf{w}_2.\, \mathtt{let}\ x = \lambda \mathbf{w}_1.\, e_1''\ \mathtt{in}\ e_2''') \\
& \quad (\textit{using Lemma } 6.1, \textit{parts } (2) \textit{ and } (3)) \\
= \quad & \mathtt{let}\ x = \lambda \mathbf{w}_1.\, e_1''\ \mathtt{in}\ (K_2\,(\lambda \mathbf{w}_2.\, e_2''')) \\
& \quad (\textit{using } A.x{:}\sigma_1,\ \mathbf{v}{:}C \vdash \lambda \mathbf{w}_2.\, [K_1 x/x] e_2'' = \lambda \mathbf{w}_2.\, e_2''' : \sigma_2) \\
= \quad & \mathtt{let}\ x = \lambda \mathbf{w}_1.\, e_1''\ \mathtt{in}\ (K_2\,(\lambda \mathbf{w}_2.\, [K_1 x/x] e_2'')) \\
& \quad (\ x \notin \mathrm{fv}(K_2)\ ) \\
= \quad & \mathtt{let}\ x = \lambda \mathbf{w}_1.\, e_1''\ \mathtt{in}\ [K_1 x/x](K_2(\lambda \mathbf{w}_2.\, e_2'')) \\
& \quad (\textit{using Lemma } 6.1(1)) \\
= \quad & \mathtt{let}\ x = [\lambda \mathbf{w}_1.\, e_1''/x](K_1 x)\ \mathtt{in}\ (K_2(\lambda \mathbf{w}_2.\, e_2'')) \\
& \quad (\textit{substitution}) \\
= \quad & \mathtt{let}\ x = K_1(\lambda \mathbf{w}_1.\, e_1'')\ \mathtt{in}\ (K_2(\lambda \mathbf{w}_2.\, e_2'')) \\
& \quad (\textit{using } A,\ \mathbf{v}{:}C \vdash K_1(\lambda \mathbf{w}_1.\, e_1'') = e_1' : \sigma) \\
= \quad & \mathtt{let}\ x = e_1'\ \mathtt{in}\ (K_2(\lambda \mathbf{w}_2.\, e_2'')) \\
& \quad (\textit{using } A.x{:}\sigma,\ \mathbf{v}{:}C \vdash K_2(\lambda \mathbf{w}_2.\, e_2'') = e_2' : \tau) \\
= \quad & \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2'
\end{aligned}
$$

■

Lemma 6.19 *For any augmented constrained substitution* (S, C, Θ) *and types* τ_1, τ_2, *the invocation* $mgu\ \tau_1\ \tau_2\ (S.\, C, \Theta)$ *either fails or terminates.*

Proof: As mentioned in the main body of the thesis, the augmented unification algorithm maintains the same recursion structure as the original one. Thus, to simplify the presentation, we will use the original version in our proof.

To begin with, we define some metric functions on sets and types. We write $|s|$ for the number of elements in set s, and, overloading the $|\ |$ operation, define $|\tau|$ to be the number of symbols in τ:

$$|\alpha| = 1$$

$$|()| = 1$$

$$|\kappa\ \tau| = 1 + |\tau|$$

$$|(\tau, \tau')| = 1 + |\tau\ | + |\ \tau'|$$

$$|\tau \rightarrow \tau'| = 1 + |\tau\ | + |\ \tau'|$$

We now proceed to prove that call to *mgu* will always terminate. We can think of the four functions, mgu, mgu', mgn, mgn', as mutually recursively defined over the tuple $\mathcal{T} = (\tau_1, \tau_2, \tau_3, \gamma, \Gamma, (S, C))$. In particular, In addition to (S, C), *mgu* and *mgu'* operate on τ_1 and τ_2, *mgn* on τ_3 and Γ, and *mgn'* on τ_3 and γ. The termination of them is proved by associating a *degree* with the parameter tuple:

$$\deg(\mathcal{T}) = (|\mathrm{dom}(C)|, |\tau_3|, |\Gamma|, |\tau_2|, |\tau_1|).$$

In other words, the degree of T is a tuple of natural numbers. We order degrees lexicographically and show that each recursive call reduces the degree.

We know from Lemma 4.2 that C is never enlarged by these functions and whenever S is extended, C will be correspondingly diminished. This fact explains why we put $|\mathrm{dom}(C)|$ as the first component of the degree and is crucial to the following argument.

Calls to *mgu* are unfolded to calls to *mgu'*. In *mgu'*, the recursive call to *mgn* diminishes $|\mathrm{dom}(C)|$ and the recursive calls to *mgu* are supplied with subcomponents

of τ_1, τ_2 and a possibly diminished C. We consider $mgu'\ \bar{\tau_1}\ \bar{\tau_2}\ (S, C)$ and show a few cases here to illustrate the ideas; the others are similar and have been omitted.

Case $(\bar{\tau_1}, \bar{\tau_2})$ of:

- (α, τ): rewrites to $mgn\ \tau\ C\alpha\ [\tau/\alpha]C\backslash_\alpha$. The degree is reduced, since C is diminished.
- $(\kappa\,\tau, \kappa\,\tau')$: rewrites, after unfolding, to $mgu'\ \tau\ \tau'\ (S, C)$. The degree is reduced, since the type arguments are reduced.
- $((\tau_1, \tau_2,)\,(\tau_1', \tau_2'))$: rewrites to $mgu\ \tau_1\ \tau_1'\ (mgu\ \tau_2\ \tau_2'\ (S, C))$. The first call $mgu\ \tau_2\ \tau_2'\ (S, C)$ terminates or fails, since the types are reduced. The interesting case occurs when we obtain from $mgu\ \tau_2\ \tau_2'\ (S, C)$ some (S', C') such that $|S'\tau_1| > |(\tau_1, \tau_2)|$. This seems to be a problem when unfolding $mgu\ \tau_1\ \tau_1'\ (S', C')$ to $mgu'\ S\tau_1\ S\tau_1'\ (S', C')$. But it is not: in that case, we would have $|\mathrm{dom}(C')| < |\mathrm{dom}(C)|$ by Lemma 4.2, and hence the degree would still be reduced.

Function *mgn* recursively calls itself with a smaller class set and calls *mgn'* only when a singleton class set is reached. Note that in the composition of these calls to *mgn*, $|\tau_3|$ may be enlarged as the substitution is extended. But in those situations, $|\mathrm{dom}(C)|$ will be reduced, so the degree is still reduced. The analysis is similar to the one given above for *mgu'*.

In *mgn'*, the calls to *mgu* are unfolded, and, from the preceding argument for *mgu'*, we know that the degree will be reduced. Moreover, in recursively normalizing the context of an instance declaration, the series of recursive calls to *mgn* are passed the subcomponents of τ_3 and a possibly diminished C, thus the degree will be reduced as the normalization process proceeds.

Since there is no infinite decreasing sequence of tuples of natural number, it follows that $mgu\ \tau_1\ \tau_2\ (S, C)$ either terminates or fails. ■

Lemma 6.17

1. If $mgn\ \bar{\tau}_3\ \Gamma\ (S, \mathbf{v}{:}C, \Theta,\ d) = (S', \mathbf{v}'{:}C', \Theta',\ d_1 d)$, *then* $\mathbf{v}'{:}C' \Vdash d_1 : S'(\bar{\tau}_3{::}\Gamma)$ *and* $(S', \mathbf{v}'{:}C', \Theta') \preceq (S, \mathbf{v}{:}C, \Theta)$.

2. If $mgu\ \bar{\tau}_1\ \bar{\tau}_2\ (S, \mathbf{v}{:}C, \Theta) = (S', \mathbf{v}'{:}C', \Theta')$, *then* $S'\bar{\tau}_1 = S'\bar{\tau}_2$ *and* $(S', \mathbf{v}'{:}C', \Theta') \preceq (S, \mathbf{v}{:}C, \Theta)$.

Proof: These two assertions are mutually dependent. The proof is by induction on the degree of the parameters: $(|\mathrm{dom}(C)|,\ |\bar{\tau}_3|,\ |\Gamma|,\ |\bar{\tau}_2|,\ |\bar{\tau}_1|)$.

If $|\mathrm{dom}(C)| = 0$, there are no type variables involved and the lemma is vacuously true. Now consider the case when $|\mathrm{dom}(C)| = n + 1$.

1. $mgn\ \bar{\tau}_3\ \Gamma\ (S, C, \Theta, d)$:

   Induction base $(S\bar{\tau}_3 = \alpha)$,

   (a) $\Gamma = \{\}$: Obvious.

   (b) $S\Gamma = \mathbf{v}{:}\{c\ \tau\}$: Consider $mgn'\ \alpha\ (c\ \tau)\ (S, C, \Theta, d)$, $\alpha \notin \tau$. There are two possibilities:

      i. $\exists \mathbf{w}, \tau'.\ \mathbf{w}{:}(c\ \tau') \in C\alpha$:

         We unfold the call $mgu\ \tau\ \tau'\ (S, \mathbf{v}{:}C, \Theta)$ to $mgu'\ \tau\ \tau'\ (S, \mathbf{v}{:}C, \Theta)$. The interesting cases are:

         A. $\tau = \beta$: Suppose that $v'{:}(\beta{::}\Gamma) \in (\mathbf{v}{:}C)$ and $\beta \notin \mathrm{tv}(\tau')$. Let $(S', C', \Theta', d') = mgn\ \tau'\ (C\beta)\ ([\tau'/\beta]S, [\tau'/\beta]C\backslash_\beta, \Theta, \epsilon)$. Then, since $|\mathrm{dom}(C\backslash_\beta)| = n$, by induction,

$$\mathbf{v}'{:}C' \Vdash d' : S'(\tau'{::}(C\beta)) \text{ and } C' \Vdash \Phi\mathbf{v}_1 : (R'[\tau'/\beta])C\backslash_\beta,$$

         where $S' = R' \circ ([\tau'/\beta] \circ S)$, $\Theta' = \Phi \circ \Theta$ and $\mathbf{v}_1 = \mathbf{v} \setminus v'$. Also note that $(S', \mathbf{v}'{:}C', [d'/v']\Theta)$ is the result we obtain from the call to *mgu*, and $\alpha \notin \mathrm{dom}(S')$ and $\mathbf{w} \notin \mathrm{dom}(\Theta')$.

Now, since $S'\beta = R'\tau'$ and $S'(C\beta) = R'(C\beta)$, combining the above two entailments gives:

$$C' \Vdash ([d'/v]\Theta')\mathbf{v} : (R'[\tau'/\beta])C.$$

Hence $(S', \mathbf{v}':C', [d'/v]\Theta') \preceq (S, \mathbf{v}:C, \Theta)$.

It remains to be shown that $\mathbf{v}':C' \Vdash \mathtt{w} : S'(\alpha :: c\ \beta)$. Since $\mathtt{w}:(c\ \tau') \in C\alpha$ and $S'\alpha = \alpha$, we have $\mathbf{v}':C' \Vdash \mathtt{w} : S'(\alpha :: c\ \tau')$. But $S'\beta = S'\tau'$, so $\mathbf{v}':C' \Vdash \mathtt{w} : S'(\bar{\tau}_3 :: c\ \beta)$.

B. $\tau = (\tau_1, \tau_2)$ and $\tau' = (\tau_1', \tau_2')$:

Let $(S_2, \mathbf{v}_2:C_2, \Theta_2) = mgu\ \tau_2\ \tau_2'\ (S, \mathbf{v}:C, \Theta)$. Since $|\tau_2| < |\tau|$, by induction using (2), we obtain:

$$\mathbf{v}:C_2 \Vdash (\Phi_2\mathbf{v}) : R_2C \text{ and } S_2\tau_2 = S_2\tau_2',$$

where $S_2 = R_2 \circ S$ and $\Theta_2 = \Phi_2 \circ \Theta$. Now consider the second recursive call $mgu\ \tau_1\ \tau_1'\ (S_2, \mathbf{v}_2:C_2, \Theta_2)$. There are two possibilities. If $S_2 = S$ then $C_2 = C$ by Lemma 4.2. So we do induction on $|\tau_1|$ since $|\tau_1| < |\tau|$. Let $(S_1, \mathbf{v}_1:C_1, \Theta_1) = mgu\ \tau_1\ \tau_1'\ (S, \mathbf{v}:C, \Theta)$. By induction:

$$\mathbf{v}_1:C_1 \Vdash (\Phi_1\mathbf{v}) : R_1C \text{ and } S_1\tau_1 = S_1\tau_1'$$

where $S_1 = R_1 \circ S$ and $\Theta_1 = \Phi_1 \circ \Theta$. It is easy to see that the final augmented constrained substitution returned by *mgu* satisfies the requirement: $(S_1, \mathbf{v}_1:C_1, \Theta_1) \preceq (S, \mathbf{v}:C, \Theta)$,

On the other hand, if $S_2 \neq S$, by Lemma 4.2 we have $|\mathrm{dom}(C')| < |\mathrm{dom}(C)|$. So we can do induction on $|\mathrm{dom}(C)|$ and obtain:

$$\mathbf{v}_1:C_1 \Vdash (\Phi_1\mathbf{v}_2) : R_1C_2 \text{ and } S_1\tau_1 = S_1\tau_1'$$

where $S_1 = R_1 \circ S_2$ and $\Theta_1 = \Phi_1 \circ \Theta_2$. It then follows from the transitivity under of substitution of $\Vdash$ that $(S_1, \mathbf{v}_1:C_1, \Theta_1) \preceq (S, \mathbf{v}:C, \Theta)$.

In addition, by an argument similar to the one given in the previous case, we can easily show that for both cases: $\mathbf{v}_1{:}C_1 \Vdash \mathbf{w} : S_1(\alpha{::}\, c\,\tau)$.

ii. If it is not the case that $\mathbf{w}{:}(c\ \tau') \in C\alpha$, then obviously $(S, C[C\alpha \oplus \mathbf{v}{:}(c\ \tau)/\alpha], \Theta, \mathbf{v}d)$ satisfies the requirements.

(c) $\Gamma = \Gamma_1 \cup \Gamma_2$: By two straightforward inductions on $|\Gamma|$ and $|\mathrm{dom}(C)|$. The analysis is similar to the one given for the case of $mgu'\ (\tau_1, \tau_2)\ (\tau_1', \tau_2')$.

Induction step $(S\bar{\tau}_3 = \kappa\,\tau')$,

(a) $\Gamma = \{\}$: Obvious.

(b) $S\Gamma = \mathbf{v}{:}\{(c\,\tau)\}$: Consider $mgn'\ (\kappa\,\tau')\ (c\,\tau)\ (S, \mathbf{v}{:}C, \Theta)$. By our restrictions on instance declarations, we will find only one such (κ, c) declaration. Furthermore, the substitution S' returned by the match operation properly instantiates the declaration. Now consider the call: $mgu\ \tau\ (S'\tilde{\tau})\ (S, C, \Theta)$. Unfolding the call and using an argument similar to the one given above, we obtain:

$$(S'', C'', \Theta'') \preceq (S, C, \Theta) \ \text{ and } \ S''\tau = S''(S'\tilde{\tau}).$$

Also, by Lemma 4.2, $|\mathrm{dom}(C'')| \leq |\mathrm{dom}(C)|$.

Then we proceed to the series of recursive calls to mgn on the list of instance predicates $\langle \tau_i{::}\Gamma_i \rangle_1^n$ derived from the instance declaration. The induction is based on the following observations: First, for all i, $|\tau_i| < |\kappa\,\tau'|$. Second, $|\mathrm{dom}(C)|$ may be reduced in the process.

To begin with, suppose that $mgn\ \tau_n\ \Gamma_n\ (S'', C'', \Theta'', d) = (S_n, C_n, \Theta_n, d_n d)$. Since $|\mathrm{dom}(C'')| < |\mathrm{dom}(C)|$ when $|S''\tau_n| > |\kappa\,\tau'|$, by induction we have

$$(S_n, C_n, \Theta_n) \preceq (S'', C'', \Theta'') \ \text{ and } \ C_n \Vdash d_n : S_n(S''\tau_n{::}\,\Gamma_n).$$

In fact, the same argument applies to all other instance predicates: for all $1 \leq i \leq (n-1)$:

$$(S_i, C_i, \Theta_i) \preceq (S_{i+1}, C_{i+1}, \Theta_{i+1}) \text{ and } C_i \Vdash d_i : S_i(S_{i+1}\tau_i :: \Gamma_i).$$

So, by the transitivity under substitution of $\Vdash$, $(S_1, C_1, \Theta_1) \preceq (S, C, \Theta)$. Finally, let $\mathbf{d_1} = d_1 \ldots d_n$, it then follows from rule (τ) of $\Vdash$ that $C_1 \Vdash (\chi\ \mathbf{d_1}) : S_1(\kappa\, \tau' :: c\, \tau)$.

(c) $\Gamma = \Gamma_1 \cup \Gamma_2$: By a straightforward induction on $|\Gamma|$ and possibly $|\mathrm{dom}(C)|$.

2. *mgu* $\tau_1\ \tau_2\ (S, C)$: By a similar inductive argument using (1).

■

Lemma 6.18

1. Suppose that $(\bar{S}, \bar{C}, \bar{\Theta}) \preceq (S, C, \Theta)$ *and* $\bar{S}\bar{\tau}_1 = \bar{S}\bar{\tau}_2$. *Then mgu* $\bar{\tau}_1\ \bar{\tau}_2\ (S, C, \Theta)$ *succeeds with* (S', C', Θ') *such that* $S'\bar{\tau}_1 = S'\bar{\tau}_2$ *and* $(\bar{S}, \bar{C}, \Theta') \preceq (S', C', \Theta') \preceq (S, C, \Theta)$.

2. Suppose that $(\bar{S}, \bar{C}, \bar{\Theta}) \preceq (S, C, \Theta)$ *and there exist dictionary expressions* $\bar{d}$ *such that* $\bar{C} \Vdash d' : \bar{S}(\bar{\tau}_3 :: \Gamma)$. *Then mgn* $\bar{\tau}_3\ \Gamma\ (S, C, \Theta, d)$ *succeeds with* $(S', C', \Theta', d_1 d)$ *such that* $C' \Vdash d_1 : S'(\bar{\tau}_3 :: \Gamma)$, $(\bar{S}, \bar{C}, \bar{\Theta}) \preceq (S', C', \Theta') \preceq (S, C, \Theta)$ *and* $d' = \Phi d_1$ *for some dictionary substitution* Φ *on variables of* d_1.

Proof: These two assertions are mutually dependent. The proof is by induction on the degree of the parameters: $(|\mathrm{dom}(C)|, |\bar{\tau}_3|, |\Gamma|, |\bar{\tau}_2|, |\bar{\tau}_1|)$.

If $|\mathrm{dom}(C)| = 0$, there are no type variables involved and the lemma is vacuously true. Now consider the case when $|\mathrm{dom}(C)| = n + 1$.

1. *mgu* $\bar{\tau}_1\ \bar{\tau}_2\ (S, C, \Theta)$: Depending on $|S\bar{\tau}_1|$ and $|S\bar{\tau}_2|$, we unfold it to:

(a) $mgu'\ \alpha\ \tau\ (S, C, \Theta)$:

Suppose that $v{:}(\alpha{::}\Gamma) \in (\mathbf{v}{:}C)$. By hypothesis, $(\bar{S}, \bar{C}, \bar{\Theta}) \preceq (S, C, \Theta)$ and $\bar{S}$ unifies α and τ, we know that $\alpha \notin \mathrm{tv}(\tau)$, $\bar{S} = \bar{R} \circ ([\tau/\alpha] \circ S)$ and

$$\bar{C} \Vdash d_\alpha : \bar{R}(\tau{::}C\alpha) \text{ and } \bar{C} \Vdash \mathbf{d}_\alpha : \bar{R}([\tau/\alpha]C\backslash\alpha),$$

where $d_\alpha \mathbf{d}_\alpha \cong \bar{\Theta}\mathbf{v}$.

Now let $(S', \mathbf{v}'{:}C', \Theta', d_1) = mgn\ \tau\ (C\alpha)\ ([\tau/\alpha]S, [\tau/\alpha]C\backslash_\alpha, \Theta, \epsilon)$. Since $|\mathrm{dom}(C\backslash_\alpha)| = n$, by induction using (2), we have

$$(\bar{S}, \bar{C}, \bar{\Theta}) \preceq (S', C', \Theta') \preceq ([\tau/\alpha]S, [\tau/\alpha]C\backslash_\alpha, \Theta)$$

and $d_\alpha = \Phi d_1$ for some dictionary substitution Φ on variables of d_1. Also note that $v \notin \mathrm{dom}(\Theta')$.

Suppose that $S' = R' \circ ([\tau/\alpha]S)$. To complete the proof in this case, we need to show that $\mathbf{v}'{:}C' \Vdash d_1 : R'(\tau{::}C\alpha)$ and $\bar{\Theta} = \Phi'([d_1/v]\Theta')$. The former is true by part (1) of Lemma 6.17, and the latter follows from the facts that $\bar{\Theta}v = d_\alpha$ and $d_\alpha = \Phi d_1$.

So, we have

$$(\bar{S}, \bar{C}, \bar{\Theta}) \preceq (S', C', [d_1/v]\Theta') \preceq (S, C, \Theta),$$

as required by the Lemma.

(b) $mgu'\ (\kappa\ \tau_1)\ (\kappa\ \tau_2)\ (S, C, \Theta)$: By a straightforward induction.

(c) $mgu'\ (\tau_1, \tau_2)\ (\tau_1', \tau_2')\ (S, C, \Theta)$: By two straightforward inductions on $|\tau_1|$, $|\tau_2|$ and possibly $|\mathrm{dom}(C)|$.

2. $mgn\ \bar{\tau_3}\ \Gamma\ (S, C, \Theta)$:

Induction base ($S\bar{\tau_3} = \alpha$): Consider $mgn\ \alpha\ \Gamma\ (S, C)$:

(a) $\Gamma = \{\}$: Obvious.

(b) $S\Gamma = \mathtt{v}{:}\{c\,\tau\}$: In this case d' is just a single dictionary. Consider $mgn'\ \alpha\ (c\,\tau)\ (S, C, \Theta)$. There are two possibilities:

i. $\exists \mathtt{w}, \tau'.\, \mathtt{w}{:}(c\,\tau') \in C\alpha$:

We unfold the call $mgu\ \tau\ \tau'\ (S, C, \Theta)$. The interesting cases are:

- $\tau = \beta$: Assume that $v{:}(\beta{::}\Gamma_\beta) \in C$.

  To induction on the subsequent call to mgn, we need to verify that:

  $$(\bar{S}, \bar{C}, \bar{\Theta}) \preceq ([\tau'/\beta]S,\ [\tau'/\beta]C\backslash_\beta, \Theta) \text{ and } \bar{C} \Vdash d'' : \bar{S}(\tau'{::}\Gamma_\beta)$$

  for some dictionaries d''. Suppose that $\bar{S} = \bar{R} \circ S$, then by hypothesis

  $$\bar{C} \Vdash d' : \bar{R}(\alpha{::}c\,\beta) \text{ and } \bar{C} \Vdash \bar{R}C.$$

  In particular, since $(c\,\tau') \in C\alpha$, we have $\bar{C} \Vdash \bar{R}(\alpha{::}c\,\tau')$. Then, by the at-most-one restriction on instance declarations, $\bar{R}\beta = \bar{R}\tau'$. Moreover, sinces $\beta \in \mathrm{dom}(C)$, we have $\bar{C} \Vdash d_2 : \bar{R}(\beta{::}\Gamma_\beta)$ for some dictionaries d_2. It then follows that $\bar{C} \Vdash d_2 : \bar{S}(\tau'{::}\Gamma_\beta)$ and d_2 is the required dictionaries.

  Now let $\bar{R} = \hat{R} \circ [\tau'/\beta]$. Then the preceding arguments give:

  $$\bar{S} = \hat{R} \circ ([\tau'/\beta]S) \text{ and } \bar{C} \Vdash \hat{R}([\tau'/\beta]C\backslash_\beta).$$

  In other words,

  $$(\bar{S}, \bar{C}, \bar{\Theta}) \preceq ([\tau'/\beta]S,\ [\tau'/\beta]C\backslash_\beta, \Theta).$$

  Next, let $(S', C', \Theta', d_1) = mgn\ \tau'\ \Gamma_\beta\ ([\tau'/\beta]S,\ [\tau'/\beta]C\backslash_\beta, \epsilon)$ and $S' = R' \circ ([\tau'/\beta]S)$. Since $|\mathrm{dom}(C\backslash\beta)| = n$, by induction we have

  $$(\bar{S}, \bar{C}, \bar{\Theta}) \preceq (S', C', \Theta'),\ \ C' \Vdash (R'[\tau'/\beta])C\backslash_\beta,$$

  and

  $$C' \Vdash d_1 : S'(\tau'{::}\Gamma_\beta).$$

such that $d_2 = \Phi' d_1$ for some Φ'. In addition, $(S', C', [d_1/v]\Theta')$ is the final result we obtain from the call to *mgu*.

But $S'\beta = S'\tau'$, so using $[d_1/v]\Theta'$ we obtain $C' \Vdash (R' \circ [\tau'/\beta])C$. Hence $(S', C', [d_1/v]\Theta') \preceq (S, C, \Theta)$. Furthermore, since $d_2 = \Phi' d_1$, we have $(\bar{S}, \bar{C}, \bar{\Theta}) \preceq (S', C', [d_1/v]\Theta')$.

It remains to be shown that $C' \Vdash \mathtt{w} : S'(\alpha :: c\,\beta)$ and $d' = \Phi\mathtt{w}$. Since $\mathtt{w}{:}(c\,\tau') \in C\alpha$ and $S'\alpha = \alpha$, we have $C' \Vdash \mathtt{w} : S'(\alpha :: c\,\tau')$. But $S'\beta = S'\tau'$, so $\mathtt{v}'{:}C' \Vdash \mathtt{w} : S'(\bar{\tau}_3 :: c\,\beta)$. Finally, $\Phi = [d'/\mathtt{w}]$ is the required dictionary substitution.

- $\tau = (\tau_1, \tau_2)$ and $\tau' = (\tau_1', \tau_2')$: Suppose that $\bar{S} = \bar{R} \circ S$. Since $\bar{C} \Vdash \bar{R}(\alpha{::}(c\,\tau))$ and $(c\,\tau') \in C\alpha$, we must have $\bar{R}\tau = \bar{R}\tau'$, Hence the result follows from two straightforward inductions using (1).

ii. If it is not the case that $\mathtt{w}{:}(c\,\tau') \in C\alpha$, then obviously $(S, C[C\alpha \oplus \mathtt{v}{:}(c\,\tau)/\alpha], \Theta, \mathtt{v}d)$ satisfies the requirements.

(c) $\Gamma = \Gamma_1 \cup \Gamma_2$: By two straightforward inductions on $|\Gamma|$ and possibly $|\mathrm{dom}(C)|$.

Induction step ($S\bar{\tau}_3 = \kappa\ \tau'$): Consider $mgn'\ (\kappa\ \tau')\ \Gamma\ (S, C)$.

(a) $\Gamma = \{\}$: Obvious.

(b) $\Gamma = \{(c\,\tau)\}$: Consider $mgn'\ (\kappa\ \tau')\ (c\ \tau)\ (S, C)$.

By hypothesis, $\bar{C} \Vdash d' : \bar{S}((\kappa\,\tau'){::}(c\,\tau))$. But because of the at-most-one restriction, we know that there is a unique instance declaration for (κ, c): *i.e.*, $\ulcorner\ \mathtt{inst}\ C' \Rightarrow \kappa\,\tilde{\tau}'{::}c\,\hat{\tau}\ \urcorner$. Furthermore, since our instance declarations are only templates, the standard match operation works. Suppose that $S' = match\ \tilde{\tau}'\ \tau'$, then $\bar{S}\tau = \bar{S}(S'\tilde{\tau})$.

Therefore, we can do induction on the call to *mgu*, that is, the invocation $mgu\ \tau\ (S'\tilde{\tau})\ (S, C, \Theta)$ succeeds with (S'', C'', Θ'') such that

$$(\bar{S}, \bar{C}, \bar{\Theta}) \preceq (S'', C'', \Theta'') \preceq (S, C, \Theta)$$

and $S''\tau = S''(S'\tilde{\tau})$.

Finally, we need to consider the list of instance predicates, $\langle \tau_i :: \Gamma_i \rangle$, derived from the instance declaration. Since for all i, $|\tau_i'| < |\kappa\,\tau'|$ and $|\mathrm{dom}(C'')| \leq |\mathrm{dom}(C)|$, we can obtain the result by a series of straightforward inductions.

(c) $\Gamma = \Gamma_1 \cup \Gamma_2$: By two straightforward inductions on $|\Gamma|$ and possibly $|\mathrm{dom}(C)|$. .

■

Lemma 6.21 *Let e be an expression, and let $(S, \mathbf{v}{:}C, \Theta)$ be an augmented constrained substitution and A a type assumption set such that C covers SA. If $tp(e, A, S, \mathbf{v}{:}C, \Theta) = (\tau, S', \mathbf{v}'{:}C', \Theta')$, then (S', C', Θ') is an augmented constrained substitution and $(S', \mathbf{v}'{:}C', \Theta') \preceq (S, \mathbf{v}{:}C, \Theta)$.*

Proof: The proof is by induction on the structure of e. For the first part that $(S', \mathbf{v}'{:}C', \Theta')$ is an augmented constrained substitution, the proof is pretty straightforward and has been omitted. The second part is more involved; we need a stronger induction hypothesis. Suppose that $S' = R \circ S$ and $\Theta' = \Phi \circ \Theta$, then the induction hypothesis is the following set of mutually dependent assertions:

1. $\mathrm{tv}(\tau) \subseteq \mathrm{dom}(C')$ and hence $(C')^*(\mathrm{tv}\ \tau) \subseteq \mathrm{dom}(C')$.
2. $\mathrm{tv}(S'A) \subseteq \mathrm{dom}(C')$ and hence $(C')^*(\mathrm{tv}\ S'A) \subseteq \mathrm{dom}(C')$.
3. RC is covered by C'.
4. $(S', \mathbf{v}'{:}C', \Theta') \preceq (S, \mathbf{v}{:}C, \Theta)$.

Case $(e = x)$: Suppose that $A(x) = \forall\langle \alpha_i :: \Gamma_i \rangle.\tau'$ and S is safe for $A(x)$. Now let β_i and $\mathbf{u}$ be new variables. Then

$$\begin{aligned} tp(x, A, S, \mathbf{v}{:}C, \Theta) &= inst(S(Ax), S, \mathbf{v}{:}C, \Theta) \\ &= inst\,(\forall\alpha_i{::}S\Gamma_i.S\tau', S, \mathbf{v}{:}C, \Theta) \\ &= (J(S\tau'), S, \mathbf{v}{:}C \oplus \mathbf{u}{:}D, \Theta) \end{aligned}$$

where $J = [\beta_i/\alpha_i]$ and $D = \{\beta_i{::}J(S\Gamma_i)\}$.

In other words, $\tau = J(S\tau')$, $S' = S$, $\mathbf{v}'{:}C' = \mathbf{v}{:}C \oplus \mathbf{u}{:}D$, and $\Theta' = \Theta$. So, $R = id$ and $\Phi = id$. We can now proceed to prove the assertions:

1. $$\begin{aligned} \mathrm{tv}(\tau) &\subseteq \mathrm{tv}(S\sigma) \cup \{\beta_i\} \\ &\subseteq \mathrm{tv}(SA) \cup \{\beta_i\} \\ &\subseteq \mathrm{dom}(C'). \end{aligned}$$
2. Obvious, since $S'A = SA$.
3. Obvious, since $RC = C$.
4. Obvious, since $\mathbf{v}{:}C \sqsubseteq \mathbf{v}'{:}C'$.

Case ($e = \lambda x.e'$) : By a straightforward induction.

Case ($e = e_1\,e_2$) : We have

$$\begin{aligned} &tp\ (e_1\ e_2, A, S, C) = \\ &\qquad \textbf{let}\quad (\tau_1, e_1', S_1, \mathbf{v}_1{:}C_1, \Theta_1) = tp(e_1, A, S, \mathbf{v}{:}C, \Theta) \\ &\qquad\qquad (\tau_2, e_2', S_2, \mathbf{v}_2{:}C_2, \Theta_2) = tp(e_2, A, S_1, \mathbf{v}_1{:}C_1, \Theta_1) \\ &\qquad\qquad \alpha \text{ be a new type variable} \\ &\qquad\qquad (S_3, \mathbf{v}_3{:}C_3, \Theta_3) = mgu\ \tau_1\ (\tau_2 \to \alpha)\ (S_2, C_2 \oplus (\alpha{::}\langle\rangle), \Theta_2) \\ &\qquad \textbf{in}\quad (S_3\alpha, (e_1'\,e_2'), S_3, C_3, \Theta_3) \end{aligned}$$

In other words, $\tau = S_3\alpha$, $S' = S_3$, $\mathbf{v}'{:}C' = \mathbf{v}_3{:}C_3$ and $\Theta' = \Theta_3$.

By the induction hypothesis (2), $(C_1)^*(\mathrm{tv}\ S_1A) \subseteq \mathrm{dom}(C_1)$. So we can do induction on the second recursive call as well as the first one. In addition, the proof relies on the properties of our unification algorithm. To use those

lemmas for the unification algorithm, we need to show that $\mathrm{tv}(S_2\tau_1) \cup \mathrm{tv}(S_2\tau_2) \subseteq \mathrm{dom}(C_2)$. Suppose that $S_1 = R_1 \circ S$ and $S_2 = R_2 \circ S_1$. By the induction hypothesis (1), $\mathrm{tv}(\tau_1) \subseteq \mathrm{dom}(C_1)$ and $\mathrm{tv}(\tau_2) \subseteq \mathrm{dom}(C_2)$. In addition, by the first part of the lemma, $S_1 C_1 = C_1$ and $S_2 C_2 = C_2$. Hence $S_1\tau_1 = \tau_1$ and $S_2\tau_2 = \tau_2$. Thus it suffices to look into $\mathrm{tv}(R_2\tau_1)$, since $S_2 = R_2 \circ S_1$.

Consider any type variable $\beta \in \mathrm{tv}(\tau_1)$. Since $\mathrm{tv}(\tau_1) \subseteq \mathrm{dom}(C_1)$, we have $\beta \in \mathrm{dom}(C_1)$. But, by the induction hypothesis (3), $\mathrm{tv}(R_2 C_1) \subseteq \mathrm{dom}(C_2)$. So $\mathrm{tv}(R_2\beta) \subseteq \mathrm{dom}(C_2)$. Since this holds for any $\beta \in \mathrm{tv}(\tau_1)$, we have $\mathrm{tv}(R_2\tau_1) \subseteq \mathrm{dom}(C_2)$ as required.

Now applying Lemma 4.1, 4.2 and 4.4 to the call of *mgu*, we obtain a substitution R_3 such that

$$
\begin{aligned}
&S_3 = R_3 \circ S_2, \\
&\mathrm{dom}(R_3) \cup \mathrm{reg}(R_3) \subseteq \mathrm{dom}(C_2.\alpha::\{\}), \\
&\mathrm{dom}(C_2.\alpha::\{\}) \setminus \mathrm{dom}(C_3) = \mathrm{dom}(R_3), \quad \text{and} \\
&C_3 \Vdash R_3(C_2.\alpha::\{\}).
\end{aligned}
$$

These facts will be used frequently in the remainder of the proof for this case.

1. We need to show that $\mathrm{tv}(\tau) \subseteq \mathrm{dom}(C_3)$, where $\tau = R_3\alpha$. If $\alpha \in \mathrm{dom}(R_3)$ then $\mathrm{tv}(R_3\alpha) \subseteq \mathrm{dom}(C_2)$, since in this case we have $\mathrm{reg}(R_3) \subseteq \mathrm{dom}(C_2)$. Furthermore, since $C_3 \Vdash R_3(C_2.\alpha::\{\})$, it follows from $\mathrm{reg}(R_3) \subseteq \mathrm{dom}(C_2)$ that $\mathrm{dom}(C_3) \cap \mathrm{dom}(C_2) \neq \emptyset$. Now, given that $\mathrm{dom}(C_2.\alpha::\{\}) \setminus \mathrm{dom}(C_3) = \mathrm{dom}(R_3)$, it is easy to see that $\mathrm{tv}(R_3\alpha) \subseteq \mathrm{dom}(C_3)$. On the other hand, if $\alpha \notin \mathrm{dom}(R_3)$, then since $C_3 \Vdash R_3(C_2.\alpha::\{\})$, we must have $\alpha \in \mathrm{dom}(C_3)$. In both cases, we have $\mathrm{tv}(R_3\alpha) \subseteq \mathrm{dom}(C_3)$. Therefore, $\mathrm{tv}(\tau) \subseteq \mathrm{dom}(C_3)$.
2. We need to show that $\mathrm{tv}(S_2 A) \subseteq \mathrm{dom}(C_3)$. By induction, $(C_2^*)(\mathrm{tv}\ S_2 A) \subseteq \mathrm{dom}(C_2)$. Since $S_3 A = R_3(S_2 A)$ and $\mathrm{reg}(R_3) \subseteq \mathrm{dom}(C_2.\alpha::\{\})$, we have $\mathrm{tv}(S_3 A) \subseteq \mathrm{dom}(C_2.\alpha::\{\})$. Moreover, since $\mathrm{dom}(R_3) \cap \mathrm{reg}(R_3) = \emptyset$, none

of the variable in $\mathrm{dom}(R_3)$ appears in $\mathrm{tv}(S_3 A)$. But $\mathrm{dom}(C_2\ \alpha::\{\}) \setminus \mathrm{dom}(R_3) = \mathrm{dom}(C_3)$, so $\mathrm{tv}(S_3 A) \subseteq \mathrm{dom}(C_3)$.

3. We show that $\mathrm{tv}(RC) \subseteq \mathrm{dom}(C_3)$, where $R = R_3 R_2 R_1$. By induction,

$$\mathrm{tv}(R_1 C) \subseteq \mathrm{dom}(C_1) \quad \text{and} \quad \mathrm{tv}(R_2 C_1) \subseteq \mathrm{dom}(C_2).$$

So $\mathrm{tv}(R_2 R_1 C) \subseteq \mathrm{dom}(C_2)$. Then it suffices to show that $\mathrm{tv}(R_3 \alpha) \subseteq \mathrm{dom}(C_3)$ for any $\alpha \in \mathrm{tv}(R_2 R_1 C)$. This can be proved by an argument similar to the one given in (1).

4. By a straightforward induction and Lemma 6.17.

Case $e = (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)$: we have

$$\begin{aligned}
&tp\,(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, A, S, \mathbf{v}{:}C, \Theta)\ = \\
&\quad \textbf{let}\ \ \mathbf{v}'{:}C' = \langle\ C\alpha\ \mid\ \alpha \in C^*(\mathrm{fv}\ SA)\ \rangle \\
&\qquad \mathbf{u}{:}D = (\mathbf{v}{:}C)\backslash(\mathbf{v}'{:}C') \\
&\qquad (\tau_1, e_1', S_1, \mathbf{v}_1{:}C_1, \Theta_1)\ =\ tp\ (e_1, A, S, \mathbf{v}'{:}C', \Theta) \\
&\qquad (\sigma, \mathbf{v}_2{:}C_2, \mathbf{w})\ =\ gen\ (\tau_1, S_1 A, \mathbf{v}_1{:}C_1, \epsilon) \\
&\qquad (\tau_2, e_2', S_2, \mathbf{v}_3{:}C_3, \Theta_2)\ =\ tp\ (e_2, A.x{:}\sigma, S_1, \mathbf{v}_2{:}C_2, \Theta_1) \\
&\quad \textbf{in}\ \ (\tau_2, (\texttt{let}\ x = \lambda\mathbf{w}.\Theta_1 e_1'\ \texttt{in}\ e_2'), S_2, \mathbf{v}_3{:}C_3 \oplus \mathbf{u}{:}D, \Theta_2)
\end{aligned}$$

In other words, $\tau = \tau_2$, $S' = S_2$, $\mathbf{v}'{:}C' = \mathbf{v}_3{:}C_3 \oplus \mathbf{u}{:}D$ and $\Theta' = \Theta_3$. Also note that $\mathbf{u}{:}D$ is left intact during the two recursive calls.

Obviously, the first recursive call admits induction. To do induction on the second recursive call, we need to show that C_2 covers $S_1(A.x{:}\sigma)$, or, $(C_2)^*(\mathrm{tv}\ S_1(A.x{:}\sigma)) \subseteq \mathrm{dom}(C_2)$. Suppose that $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\tau_1$. Then by Lemma 6.6,

$$\mathbf{v}_1{:}C_1 \cong \mathbf{w}{:}\langle\alpha_i{::}\Gamma_i\rangle \oplus \mathbf{v}_2{:}C_2 \ \text{ and } \ \mathrm{dom}(C_2) = (C_1)^*(\mathrm{tv}\ S_1 A).$$

It follows that $\mathrm{tv}(\sigma) \subseteq (C_1^*)(\mathrm{tv}\ S_1 A)$ and $\mathrm{tv}(S_1(A.x:\sigma)) = \mathrm{tv}(S_1 A)$ and So $\mathrm{tv}\ S_1(A.x{:}\sigma) \subseteq \mathrm{dom}(C_2)$ and hence $(C_2^*)(\mathrm{tv}\ S_1(A.x{:}\sigma)) \subseteq \mathrm{dom}(C_2)$.

Suppose that $S_1 = R_1 \circ S$ and $S_2 = R_2 \circ S_1$. We can now proceed to prove the assertions.

1. By a straightforward induction.
2. By a straightforward induction.
3. We only need to show that $\mathrm{tv}(R_2 R_1 C') \subseteq \mathrm{dom}(C_3)$. By induction, we have $\mathrm{tv}(R_1 C') \subseteq \mathrm{dom}(C_1)$ and $\mathrm{tv}(R_2 C_2) \subseteq \mathrm{dom}(C_3)$. To complete the proof, we show that $\mathrm{tv}(R_1 C') \subseteq \mathrm{dom}(C_2)$, which is true if none of the α_i occurs in $\mathrm{tv}(R_1 C')$. This in turn can be shown by Lemma 3.2: because $C_1 \Vdash R_1 C'$ and $\mathrm{dom}(C') = (C')^*(\mathrm{tv}\ SA)$, $\mathrm{tv}(R_1\alpha) \subseteq (C_1)^*(\mathrm{tv}\ S_1 A)$ for all $\alpha \in \mathrm{dom}(C')$. Hence none of the α_i appears in $\mathrm{tv}(R_1\alpha)$. Therefore, $\mathrm{tv}(R_1 C) \subseteq \mathrm{dom}(C_2)$ and hence $\mathrm{tv}(R_2 R_1 C') \subseteq \mathrm{dom}(C_3)$.
4. We only need to show that $(S_2, \mathbf{v}_3{:}C_3, \Theta_2) \preceq (S, \mathbf{v}'{:}C', \Theta)$. By induction,

   $$(S_1, \mathbf{v}_1{:}C_1, \Theta_1) \preceq (S, \mathbf{v}'{:}C', \Theta) \text{ and } (S_2, \mathbf{v}_3{:}C_3, \Theta_2) \preceq (S_1, \mathbf{v}_2{:}C_2, \Theta_1).$$

   By definition, $\mathbf{v}_1{:}C_1 \Vdash \Theta_1\mathbf{v}'{:}(R_1 C')$. To complete the proof, we need to show $(S_1, \mathbf{v}{:}C_2, \Theta_1) \preceq (S, \mathbf{v}'{:}C', \Theta)$, or $\mathbf{v}_2{:}C_2 \Vdash \Theta_1\mathbf{v}'{:}(R_1 C')$.

   By the (*strengthen*) property of $\Vdash$ given in Section 5.2.2, it suffices to show that that none of the α_i occurs in $R_1 C'$. This is shown by an argument similar to the one used in (3). So, $(S_1, \mathbf{v}_2{:}C_2, \Theta_1) \preceq (S, \mathbf{v}'{:}C', \Theta)$. It then follows from the transitivity under substitution of $\Vdash$ that $(S_2, \mathbf{v}_3{:}C_3, \Theta_2) \preceq (S, \mathbf{v}'{:}C', \Theta)$.

■

Theorem 6.22 *If* $tp(e, A, S, \mathbf{v}{:}C, \Theta) = (\tau, e', S', \mathbf{v}'{:}C', \Theta')$, *then* $S'A, \mathbf{v}'{:}C' \vdash' e \rightsquigarrow \Theta' e' : \tau$.

Proof: By induction on the structure of e. The nontrivial cases are (*app*) and (*let*).

Case $e = e_1\, e_2$: We have

$$tp\ (e_1\ e_2, A, S, \mathbf{v}{:}C, \Theta) =$$

$$\textbf{let}\ \ (\tau_1, e_1', S_1, \mathbf{v}_1{:}C_1, \Theta_1) = tp\ (e_1, A, S, \mathbf{v}{:}C, \Theta)$$

$$(\tau_2, e_2', S_2, \mathbf{v}_2{:}C_2, \Theta_2) = tp\ (e_2, A, S_1, \mathbf{v}_1{:}C_1, \Theta_1)$$

$$\alpha \text{ be a fresh type variable}$$

$$(S_3, \mathbf{v}_3{:}C_3, \Theta_3) = mgu\ \tau_1\ (\tau_2 \to \alpha)\ (S_2, C_2.\alpha{::}\langle\rangle, \Theta_2)$$

$$\textbf{in}\ \ (S_3\alpha, e_1'\, e_2', S_3, \mathbf{v}_3{:}C_3.\, \Theta_3)$$

By induction,

$$S_1 A,\ \mathbf{v}_1{:}C_1 \vdash'\ e_1 \rightsquigarrow \Theta_1 e_1' :\ \tau_1 \ \text{ and } \ S_2 A,\ \mathbf{v}_2{:}C_2 \vdash'\ e_2 \rightsquigarrow \Theta_2 e_2' :\ \tau_2.$$

We will apply the substitution lemma (6.14) repetitively to combine these two derivations.

Suppose that $S_1 = R_1 \circ S$, $S_2 = R_2 \circ S_1$, and $S_3 = R_2 \circ S_2$.

By Lemma 6.21,

$$\mathbf{v}_1{:}C_1 \Vdash \Theta_1\mathbf{v} : R_1 C \quad \text{and} \quad \mathbf{v}_2{:}C_2 \Vdash \Theta_2\mathbf{v}_1 : R_2 C_1.$$

Furthermore, by Lemma 6.17, $S_3\tau_1 = S_3(\tau_2 \to \alpha)$ and $\mathbf{v}_3{:}C_3 \Vdash \Theta_3\mathbf{v}_2 : R_3(C_2.\alpha{::}\langle\rangle)$. Hence by transitivity (Lemma 5.1), $\mathbf{v}_3{:}C_3 \Vdash \Theta_3\mathbf{v} : R_3(R_2 C_1)$.

Now we can construct the required derivation:

$$\dfrac{\dfrac{\dfrac{S_1 A,\ \mathbf{v}_1{:}C_1 \vdash'\ e_1 \rightsquigarrow \Theta_1 e_1' :\ \tau_1}{S_3 A,\ \mathbf{v}_3{:}C_3 \vdash'\ e_1 \rightsquigarrow \Theta_3 e_1' :\ S_3\tau_1} \qquad \dfrac{S_2 A,\ \mathbf{v}_2{:}C_2 \vdash'\ e_2 \rightsquigarrow \Theta_2 e_2' :\ \tau_2}{S_3 A,\ \mathbf{v}_3{:}C_3 \vdash'\ e_2 \rightsquigarrow \Theta_3 e_2' :\ S_3\tau_2}}{S_3 A,\ \mathbf{v}_3{:}C_3 \vdash'\ e_1 \rightsquigarrow \Theta_3 e_1' :\ S_3\tau_2 \to S_3\alpha}}{S_3 A,\ \mathbf{v}_3{:}C_3 \vdash'\ e_1\, e_2 \rightsquigarrow \Theta_3(e_1'\, e_2') :\ S_3\alpha}$$

Case $e = (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)$: We have:

$$
\begin{array}{l}
tp\,(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, A, S, \mathbf{v}{:}C, \Theta) \ = \\
\qquad \textbf{let}\ \ \mathbf{v}'{:}C' = \langle\ C\alpha \mid \alpha \in C^*(\text{fv}\ SA)\ \rangle \\
\qquad\quad \mathbf{v}_0{:}C_0 = (\mathbf{v}{:}C)\backslash(\mathbf{v}'{:}C') \\
\qquad\quad (\tau_1, e_1', S_1, \mathbf{v}_1{:}C_1, \Theta_1) \ = \ tp\ (e_1, A, S, \mathbf{v}'{:}C', \Theta) \\
\qquad\quad (\sigma, \mathbf{v}_2{:}C_2, \mathbf{w}) \ = \ gen\ (\tau_1, S_1A, \mathbf{v}_1{:}C_1, \epsilon) \\
\qquad\quad (\tau_2, e_2', S_2, \mathbf{v}_3{:}C_3, \Theta_2) \ = \ tp\ (e_2, A.x:\sigma, S_1, \mathbf{v}_2{:}C_2, \Theta_1) \\
\qquad \textbf{in}\ \ (\tau_2, (\texttt{let}\ x = \lambda\mathbf{w}.\Theta_1 e_1'\ \texttt{in}\ e_2'), S_2, \mathbf{v}_3{:}C_3 \oplus \mathbf{v}_0{:}C_0, \Theta_2)
\end{array}
$$

By induction

$$S_1A,\ \mathbf{v}_1{:}C_1\ \vdash'\ e_1 \rightsquigarrow \Theta_1 e_1'\ :\tau_1 \quad \text{and} \quad S_2A.x:S_2\sigma,\ \mathbf{v}_3{:}C_3\ \vdash'\ e_2 \rightsquigarrow \Theta_2 e_2'\ :\tau_2$$

We will apply Lemma 6.11 to combine these two derivations.

Suppose that $S_2 = R_2 \circ S_1$ and $\Theta_2 = \Phi_2 \circ \Theta_1$. By Lemma 6.21, $\mathbf{v}_3{:}C_3 \Vdash \mathbf{d}_2 : R_2C_2$, where $\mathbf{d}_2 = \Phi_2\mathbf{v}_2$. Now, applying Lemma 6.11 to σ, R_2, $\mathbf{v}_3{:}C_3$ and $\mathbf{d}_2$, we obtain a substitution R, a context $\mathbf{u}{:}D$ and dictionary expressions $\mathbf{d}'$ such that

$$R(S_1A) = R_2(S_1A),\ \ \mathbf{u}{:}D \Vdash \mathbf{d}'{:}RC_1,\ \ R_2\sigma = \sigma',\ \ D' \sqsubseteq C_3,\ \ \text{and}\ \ \mathbf{d}' \cong \mathbf{w}\mathbf{d}_2$$

where $(\sigma', D', \mathbf{w}) = gen(R\tau_1, R_2(S_1A), \mathbf{u}{:}D, \epsilon)$.

By Lemma 6.6, $\mathbf{v}_1 \cong \mathbf{v}_2 \oplus \mathbf{w}$. So, using Lemma 6.14, we obtain the following derivation:

$$\dfrac{\dfrac{S_1A,\ \mathbf{v}_1{:}C_1\ \vdash'\ e_1 \rightsquigarrow \Theta_1 e_1'\ :\tau_1}{R(S_1A),\ \mathbf{u}{:}D\ \vdash'\ e_1 \rightsquigarrow [\mathbf{d}'/\mathbf{v}_1]\Theta_1 e_1'\ :R\tau_1}}{S_2A,\ \mathbf{u}{:}D\ \vdash'\ \epsilon_1 \rightsquigarrow [\mathbf{d}_2/\mathbf{v}_2]\Theta_1 e_1'\ :R\tau_1}$$

But clearly $S_2\sigma = R_2\sigma$, since $\text{tv}(\sigma) \subseteq \text{dom}(C_1)$. Hence we can construct the required derivation:

$$\dfrac{\dfrac{S_2A,\ \mathbf{u}{:}D \vdash' e_1 \leadsto [\mathbf{d_2}/\mathbf{v_2}]\Theta_1 e_1' : R\tau_1 \qquad S_2A.x : S_2\sigma,\ \mathbf{v_3}{:}C_3 \vdash' e_2 \leadsto \Theta_2 e_2' : \tau_2}{S_2A,\ \mathbf{u}{:}D \vdash' e_1 \leadsto \Theta_2 e_1' : R\tau_1 \qquad S_2A.x : \sigma',\ \mathbf{v_3}{:}C_3 \vdash' e_2 \leadsto \Theta_2 e_2' : \tau_2}}{S_2A,\ \mathbf{v_3}{:}C_3 \vdash' (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \leadsto (\mathtt{let}\ x = \lambda\mathbf{w}.\Theta_2 e_1'\ \mathtt{in}\ \Theta_2 e_2') : \tau_2}$$

$$\overline{S_2A,\ \mathbf{v_3}{:}C_3 \oplus \mathbf{v_0}{:}C_0 \vdash' (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \leadsto \Theta_2(\mathtt{let}\ x = \lambda\mathbf{w}.\Theta_1 e_1'\ \mathtt{in}\ e_2') : \tau_2}$$

■

Theorem 6.24 *Suppose that* $S'A,\ \mathbf{v}'{:}C' \vdash' e \leadsto e' : \tau'$ *and* $(S', \mathbf{v}'{:}C', \Theta') \preceq (S_0, \mathbf{v_0}{:}C_0, \Theta_0)$. *Then* $tp(e, A, S_0, \mathbf{v_0}{:}C_0, \Theta_0)$ *succeeds with* $(\tau, e'', S, \mathbf{v}{:}C, \Theta)$, *and there is a substitution* R *and dictionary expressions* $\mathbf{d}$ *such that*

1. $S' = RS$, *except possibly on new type variables of* $tp(e, A, S_0, \mathbf{v_0}{:}C_0, \Theta_0)$,
2. $\tau' = R\tau$,
3. $\mathbf{v}'{:}C' \Vdash \mathbf{d} : RC$,
4. $S'A,\ \mathbf{v}'{:}C' \vdash e' = [\mathbf{d}/\mathbf{v}]\Theta e'' : \tau'$.

Proof: By induction on the structure of $S'A.\ C' \vdash' e : \tau'$.

Case (var') : Assume $A(x) = \forall\langle\alpha_i{::}\Gamma_i\rangle.\ \nu$. We have a derivation of the form :

$$\dfrac{S'A(x) = \forall\langle\alpha_i{::}S'\Gamma_i\rangle.\ S'\nu \qquad \mathbf{v}'{:}C' \Vdash \mathbf{d}'{:}([\tau_i/\alpha_i]\langle\alpha_i{::}S'\Gamma_i\rangle)}{S'A,\ \mathbf{v}'{:}C' \vdash' x \leadsto x\mathbf{d}' : [\tau_i/\alpha_i](S'\nu)}$$

Note that here we assume that variables in σ have been suitably renamed so that no name clashes occur in the proof.

Let β_i and $\mathbf{u}$ be new variables, we have

$$tp(x, A, S_0, \mathbf{v_0}{:}C_0, \Theta_0) = ([\beta_i/\alpha_i](S_0\nu), S_0, \mathbf{v_0}{:}C_0 \oplus \mathbf{u}{:}C_1, \Theta_0)$$

where $C_1 = \langle \beta_i :: [\beta_i/\alpha_i](S_0\Gamma_i) \rangle$. In other words, $\tau = [\beta_i/\alpha_i](S_0\nu)$, $e'' = x\,\mathbf{u}$, $S = S_0$, $\mathbf{v}{:}C = \mathbf{v}_0{:}C_0 \oplus \mathbf{u}{:}C_1$ and $\Theta = \Theta_0$.

Let R_0 be the substitution such that $S' = R_0 \circ S_0$ and $C' \Vdash \mathbf{d}'' : R_0C_0$ for some dictionaries $\mathbf{d}''$. Now let $R = [\tau_i/\beta_i]R_0$. Then, clearly $S' = RS$ except possibly on the new type variables β_i, and

$$\begin{aligned} R\tau &= ([\tau_i/\beta_i]R_0)[\beta_i/\alpha_i]S_0\nu \\ &= ([\tau_i/\beta_i][\beta_i/\alpha_i])(R_0S_0)\nu \qquad (R_0 \text{ safe for } \alpha_i) \\ &= [\tau_i/\alpha_i]S'\nu\,. \end{aligned}$$

Furthermore, we have

$$\begin{aligned} RC_1 &= [\tau_i/\beta_i]R_0(\,\langle \beta_i :: [\beta_i/\alpha_i]S_0\Gamma_i \rangle\,) \\ &= \langle \tau_i :: [\tau_i/\alpha_i](R_0S_0\Gamma_i) \rangle \\ &= \langle \tau_i :: [\tau_i/\alpha_i](S'\Gamma_i) \rangle. \end{aligned}$$

Thus, by the uniqueness of dictionary construction (Lemma 5.2), $\mathbf{v}'{:}C' \Vdash \mathbf{d}' : RC_1$ In addition, given that $RC = R_0C_0 \oplus RC_1$ and $\mathbf{v}'{:}C' \Vdash \mathbf{d}'' : R_0C_0$, it follows that $\mathbf{v}'{:}C' \Vdash \mathbf{d} : RC$ where $\mathbf{d} = \mathbf{d}''\mathbf{d}'$.

Finally, note that:

$$S'A,\ \mathbf{v}'{:}C' \vdash [\mathbf{d}/\mathbf{v}](x\,\mathbf{u}) = [\mathbf{d}''/\mathbf{v}_0][\mathbf{d}'/\mathbf{u}](x\,\mathbf{u}) = x\,\mathbf{d}' : [\tau_i/\alpha_i](S'\nu),$$

which establishes the required equality.

Case (λ-elim') : We have a derivation of the form :

$$\frac{S'A,\ \mathbf{v}'{:}C' \vdash' e_1 \rightsquigarrow e_1' : \tau_1 \to \tau_2 \qquad S'A,\ \mathbf{v}'{:}C' \vdash' e_2 \rightsquigarrow e_2' : \tau_1}{S'A,\ \mathbf{v}'{:}C' \vdash' e_1\,e_2 \rightsquigarrow e_1'\,e_2' : \tau_2}$$

By induction, $tp(e_1, A, S_0, \mathbf{v}_0{:}C_0, \Theta_0)$ succeeds with $(\nu_1, e_1'', S_1, \mathbf{v}_1{:}C_1, \Theta_1)$ and there is a substitution R_1 and dictionary expressions $\mathbf{d}_1$ such that

1. $S' = R_1 S_1$, except possibly on Δ_1,
2. $R_1 \nu_1 = \tau_1 \rightarrow \tau_2$,
3. $\mathbf{v}'{:}C' \Vdash \mathbf{d}_1 : R_1 C_1$, and
4. $S'A,\ \mathbf{v}'{:}C' \vdash e_1' = [\mathbf{d}_1/\mathbf{v}_1]e_1'' : \tau_1 \rightarrow \tau_2$

where Δ_1 is the set of new type variables of $tp(e_1, A, S_0, C_0)$.

Besides, by Lemma 6.21, we have $\mathbf{v}_1{:}C_1 \Vdash \mathbf{d}_1' : S_1 C$ where $\mathbf{d}_1' = \Theta_1 \mathbf{v}$.

Clearly, $S'A = (R_1 S_1)A$ and hence $(R_1 S_1)A,\ \mathbf{v}'{:}C' \vdash' e_2 \rightsquigarrow e_2' : \tau_1$. So, by induction , $tp(e_2, A, S_1, \mathbf{v}_1{:}C_1, \Theta_1)$ succeeds with $(\nu_2, e_2'', S_2, \mathbf{v}_2{:}C_2, \Theta_2)$ and there exist a substitution R_2 and dictionary expressions $\mathbf{d}_2$ such that

1. $R_1 S_1 = R_2 S_2$, except possibly on Δ_2,
2. $R_2 \nu_2 = \tau_1$,
3. $\mathbf{v}'{:}C' \Vdash \mathbf{d}_2 : R_2 C_2$, and
4. $S'A,\ \mathbf{v}'{:}C' \vdash e_2' = [\mathbf{d}_2/\mathbf{v}_2]e_2'' : \tau_1$

where Δ_2 is the set of new type variables of $tp(e_2, A, S_1, C_1)$.

Again, by Lemma 6.21, we have $\mathbf{v}_2{:}C_2 \Vdash \mathbf{d}_2' : S_2 C_1$ where $\mathbf{d}_2' = \Theta_2 \mathbf{v}_1$.

Now let α be a new variable and $R' = [\tau_2/\alpha]R_2 S_2$. Then, clearly

$$S' = R' \text{ except possibly on } \Delta_1 \cup \Delta_2 \cup \{\alpha\}$$

and

$$\mathbf{v}'{:}C' \Vdash \mathbf{d}_2 : [\tau_2/\alpha]R_2(C_2.\alpha{::}\langle\rangle).$$

In addition, we show that $(R', \mathbf{v}'{:}C', [\mathbf{d}_2/\mathbf{v}_2])$ is a $(S_2, \mathbf{v}_2{:}C_2.\alpha{::}\langle\rangle, \Theta_2)$-preserving

unifier of ν_1 and $\nu_2 \rightarrow \alpha$:

$$\begin{aligned}
R'\nu_1 &= (R_2S_2)\nu_1 && (\alpha \ new) \\
&= (R_1S_1)\nu_1 && (R_1S_1 = R_2S_2 \ \ except \ on \ \Delta_2) \\
&= R_1\nu_1 && (S_1C_1 = C_1) \\
&= \tau_1 \rightarrow \tau_2 \\
&= R_2\nu_2 \rightarrow R'\alpha \\
&= R_2(S_2\nu_2) \rightarrow R'\alpha && (S_2C_2 = C_2) \\
&= R'\nu_2 \rightarrow R'\alpha && (\alpha \ new) \\
&= R'(\nu_2 \rightarrow \alpha)
\end{aligned}$$

Thus, from Lemma 4.5, $mgu\ \nu_1\ (\nu_2 \rightarrow \alpha)\ (S_2, \mathbf{v}_2{:}C_2.\alpha{::}\langle\rangle, \Theta_2)$ succeeds with $(S_3, \mathbf{v}_3{:}C_3)$ such that $S_3\nu_1 = S_3(\nu_2 \rightarrow \alpha)$, $R' = R \circ S_3$ and $\mathbf{v}'{:}C' \Vdash \mathbf{d}_3 : RC_3$ for some substitution R and dictionary expressions $\mathbf{d}_3$.

Also, by Lemma 6.21, $\mathbf{v}_3{:}C_3 \Vdash \mathbf{d}'_3 : S_3C_2$ where $\mathbf{d}'_3 = \Theta_3\mathbf{v}_2$.

Since *mgu* does not introduce any new type variables, we know that $\Delta_1 \cup \Delta_2 \cup \{\alpha\}$ is the set of new type variables of $tp(e_1\,e_2, A, S_0, C_0)$. Therefore, $tp(e_1\,e_2, A, S_0, C_0, \Theta_0)$ succeeds with $(S_3\alpha, e''_1e''_2, S_3, C_3, \Theta_3)$, and R and $\mathbf{d}_3$ are the required substitution and dictionary expressions respectively:

1. $S' = RS_3$, except possibly on new type variables of $tp(e_1\,e_2, A, S_0, C_0)$,
2. $\tau_2 = R'\alpha = R(S_3\alpha)$.
3. $\mathbf{v}'{:}C' \Vdash \mathbf{d}_3 : RC_3$.

We need to establish the appropriate relationship between $\mathbf{d}_i$ and $\mathbf{d}'_i$ to complete the proof. This we do by using the transitivity under substitution (Lemma 5.1) and the uniqueness of dictionary construction properties (Lemma 5.2) of $\Vdash$.

First, from $\mathbf{v}':C' \Vdash \mathbf{d}_3 : RC_3$ and $\mathbf{v}_3:C_3 \Vdash \mathbf{d}'_3 : S_3C_2$, we have $\mathbf{v}':C' \Vdash [\mathbf{d}_3/\mathbf{v}_3]\mathbf{d}'_3 : R(S_3C_2)$. But,

$$\begin{aligned} R(S_3C_2) &= R'C_2 \\ &= (R_2S_2)C_2 \qquad (\alpha \text{ is new}) \\ &= R_2C_2. \end{aligned}$$

So, by Lemma 5.2, $\mathbf{d}_2 = [\mathbf{d}_3/\mathbf{v}_3]\mathbf{d}'_3$. This implies that we can rewrite $[\mathbf{d}_2/\mathbf{v}_2]e''_2$ to $[\mathbf{d}_3/\mathbf{v}_3][\mathbf{d}'_3/\mathbf{v}2]e''_2$. Furthermore, from $\mathbf{v}_2:C_2 \Vdash \mathbf{d}'_2 : S_2C_1$, we have $\mathbf{v}':C' \Vdash [[\mathbf{d}_3/\mathbf{v}_3]\mathbf{d}'_3/\mathbf{v}_2]\mathbf{d}'_2 : R_2(S_2C_1)$.

Similarly, we have $R_2(S_2C_1) = R_1C_1$ and hence $\mathbf{d}_1 = [[\mathbf{d}_3/\mathbf{v}_3]\mathbf{d}'_3/\mathbf{v}_2]\mathbf{d}'_2$. Therefore, we can rewrite $[\mathbf{d}_1/\mathbf{v}_1]e''_1$ as $[\mathbf{d}_3/\mathbf{v}_3][\mathbf{d}'_3/\mathbf{v}_2][\mathbf{d}'_2/\mathbf{v}_1]e''_1$.

Now we note that:

$$e'_1 = [\mathbf{d}_1/\mathbf{v}_1]e''_1 = [\mathbf{d}_3/\mathbf{v}_3][\mathbf{d}'_3/\mathbf{v}_2][\mathbf{d}'_2/\mathbf{v}_1]e''_1 = [\mathbf{d}_3/\mathbf{v}_3](\Theta_3 e''_1)$$

and

$$e'_2 = [\mathbf{d}_2/\mathbf{v}_2]e''_2 = [\mathbf{d}_3/\mathbf{v}_3][\mathbf{d}'_3/\mathbf{v}2]e''_2 = [\mathbf{d}_3/\mathbf{v}_3](\Theta_3 e''_2).$$

Therefore, we have the required equality:

$$S'A,\ \mathbf{v}':C' \vdash e'_1e'_2 = [\mathbf{d}_3/\mathbf{v}]\Theta_3(e''_1e''_2) : \tau_2$$

Case (λ-intro') : We have a derivation of the form :

$$\frac{S'A.x:\tau_1,\ \mathbf{v}':C' \vdash' e \rightsquigarrow e' : \tau_2}{S'A,\ \mathbf{v}':C' \vdash' \lambda x.e \rightsquigarrow \lambda x.e' : \tau_1 \to \tau_2}$$

Let α be a new type variable and $S'' = [\tau_1/\alpha] \circ S'$. Then,

$$S''(A.x:\alpha), C' \vdash' e : \tau_2 \quad \text{and} \quad (S'', C', \Theta') \preceq (S_0, C_0.\alpha::\langle\rangle, \Theta_0).$$

So, by induction, $tp(e, A.x:\alpha, S_0, \mathbf{v}_0:C_0.\alpha::\langle\rangle, \Theta_0) = (\tau, e''_1, S, \mathbf{v}:C, \Theta)$ succeeds and there exist a substitution R and dictionary expressions $\mathbf{d}$ such that

1. $S'' = RS$ except on the new type variables of $tp(e, A.x:\alpha, S_0, C_0.\alpha::\langle\rangle, \Theta_0)$,
2. $R\tau = \tau_2$,
3. $\mathbf{v}':C' \Vdash \mathbf{d} : RC$ and
4. $S'A.x:\tau_1, \mathbf{v}':C' \vdash e' = [\mathbf{d}/\mathbf{v}](\Theta e'') : \tau_2$.

Therefore, $tp(\lambda x.e, A, S_0, \mathbf{v}_0:C_0, \Theta_0)$ succeeds with $(S\alpha \to \tau, S, \mathbf{v}:C, \Theta)$, and R and $\mathbf{d}$ are the required substitution and dictionary expressions, respectively :

1. $S' = RS$ except possibly on new type variables of $tp(\lambda x.e, A, S_0, C_0)$
2. $R(S\alpha \to \tau) = \tau_1 \to \tau_2$ since $S''\alpha = \tau_1$,
3. $\mathbf{v}':C' \Vdash \mathbf{d} : RC$, and
4. $S'A, \mathbf{v}':C' \vdash \lambda x.\, e' = [\mathbf{d}/\mathbf{v}]\Theta(\lambda x.\, e'') : \tau_1 \to \tau_2$

Case (*let'*) : We have a derivation of the form:

$$\frac{S'A, \mathbf{u}:D \vdash' e_1 \rightsquigarrow e_1' : \tau_1 \qquad S'A.x:\sigma', \mathbf{v}'C' \vdash' e_2 \rightsquigarrow e_2' : \tau_2}{S'A, \mathbf{v}':C' \vdash' (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) \rightsquigarrow (\mathtt{let}\ x = \lambda \mathbf{w}.\, e_1'\ \mathtt{in}\ e_2') : \tau_2}$$

where $(\sigma', \mathbf{u}':D', \mathbf{w}) = gen(\tau_1, S'A, \mathbf{u}:D, \epsilon)$ and $D' \sqsubseteq C'$.

Suppose that $\mathbf{v}_0:C_0 \cong \mathbf{v}_x:C_x \oplus \mathbf{v}_a:C_a$ where $\mathrm{dom}(C_a) = C_0^*(\mathrm{tv}\ S_0A)$. To apply the induction hypothesis on $S'A, \mathbf{u}:D \vdash' e_1 : \tau_1$, we need to show that $D \Vdash S'C_a$. This we prove by showing that all type variables that appear in $S'C_a$ also appear in D.

First, note that by hypothesis $C' \Vdash S'C_0$ and hence $C' \Vdash S'C_a$. Second, by Lemma 3.2, $\bigcup\{\mathrm{tv}(S'\alpha) \mid \alpha \in \mathrm{dom}(C_a)\}$ is contained in $(C')^*(\mathrm{tv}\ S'A)$. Furthermore, by Lemma 6.6, $\mathrm{dom}(D') = D^*(\mathrm{tv}\ S'A)$. But since both D and C' covers $S'A$ and $D' \sqsubseteq C'$, we have $(C')^*(\mathrm{tv}\ S'A) = D^*(\mathrm{tv}\ S'A)$. It then follows from Lemma 3.1 that $D \Vdash S'C_a$

Thus, by induction, $tp(e_1, A, S_0, \mathbf{v}_a:C_a, \Theta_0) = (\nu_1, e_1'', S_1, \mathbf{v}_1:C_1, \Theta_1)$ succeeds and there exist a substitution R_1 and dictionary expressions $\mathbf{d}_1$ such that

1. $S' = R_1 S_1$ except possibly on new type variables of $tp(e_1, A, S_0, C_a, \Theta_0)$,
2. $R_1 \nu_1 = \tau_1$,
3. $\mathbf{u}{:}D \Vdash \mathbf{d}_1 : R_1 C_1$ and
4. $S'A,\ \mathbf{u}{:}D \vdash e_1' = [\mathbf{d}_1/\mathbf{v}_1](\Theta_1 e_1'') : \tau_1$.

Hence $S'A = (R_1 S_1)A$ and $(R_1 S_1)A.x{:}\sigma',\ \mathbf{v}'{:}C' \vdash' e_2 \rightsquigarrow e_2' : \tau_2$.

Before we can proceed to deal with the second recursive call of *tp*, we need to establish a few things. First, we show that $(S', \mathbf{v}'{:}C', \Theta') \preceq (S_1, \mathbf{v}_2{:}C_2, \Theta_1)$, or $\mathbf{v}'{:}C' \Vdash \mathbf{d}'' : R_1 C_2$. Let $(\sigma, \mathbf{v}_2{:}C_2, \mathbf{w}_1) = gen(\nu_1, S_1 A, \mathbf{v}_1{:}C_1, \epsilon)$. By Lemma 6.6, $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\ \nu_1$ and

$$\mathbf{v}_1{:}C_1 \cong \mathbf{w}_1{:}\langle\alpha_i{::}\Gamma_i\rangle \oplus \mathbf{v}_2{:}C_2 \text{ and } \mathrm{dom}(C_2) = (C_1)^*(\mathrm{tv}\ S_1 A).$$

Now consider any type variable $\beta \in \mathrm{dom}(C_2)$. Now since $\mathbf{u}{:}D \Vdash \mathbf{d}_1 : R_1 C_1$, it follows from Lemma 3.2 that $\mathrm{tv}(R_1\beta) \subseteq D^*(\mathrm{tv}\ R_1 S_1 A)$. But we know that $S'A = R_1 S_1 A$, $\mathrm{dom}(D') = D^*(\mathrm{tv}\ S'A)$ and $D' \sqsubseteq C'$. So $\mathrm{tv}(R_1 C_2) \subseteq \mathrm{dom}(C')$. Hence by the (*strengthen*) property given in Section 5.2.2 there exist some dictionaries $\mathbf{d}'' \sqsubseteq \mathbf{d}_1$ such that $\mathbf{v}'{:}C' \Vdash \mathbf{d}'' : R_1 C_2$. Hence it follows that $(S', \mathbf{v}'{:}C', \Theta') \preceq (S_1, \mathbf{v}_2{:}C_2, \Theta_1)$.

Second, we show that $\sigma' \preceq_{\mathbf{v}'{:}C'} R_1\sigma$. The definitions of σ and σ' are related through R_1. Now, since $\mathbf{u}{:}D \Vdash \mathbf{d}_1 : R_1 C_1$, it follows from Lemma 6.10 that $(\lambda\mathbf{w}.\, x\mathbf{d}') : \sigma' \preceq_{\mathbf{u}'{:}D'} R_1\sigma$ for some dictionaries $\mathbf{d}' \sqsubseteq \mathbf{d}_1$. Hence $(\lambda\mathbf{w}.\, x\mathbf{d}') : \sigma' \preceq_{\mathbf{v}'{:}C'} R_1\sigma$ by Lemma 6.2.

Third, we consider the derivation $S'A.x : \sigma',\ C' \vdash' e_2 : \tau_2$ based on the results just established. Clearly, we have

$$R_1 S_1 A.x : \sigma' \preceq_{\mathbf{v}'{:}C'} R_1 S_1 A.x : R_1\sigma$$

through the conversion substitution $[\lambda\mathbf{w}.\, x\mathbf{d}'/x]$. But $S'A.x : \sigma' = R_1 S_1 A.x : \sigma'$ and $R_1 S_1 A.x : R_1\sigma = R_1 S_1 (A.x : \sigma)$, so by Lemma 6.15,

$$R_1 S_1 (A.x : \sigma),\ \mathbf{v}'{:}C' \vdash' e_2 \rightsquigarrow \epsilon'' : \tau_2$$

with $R_1S_1(A.x:\sigma)$, $\mathbf{v}':C' \vdash e'' = [\lambda \mathbf{w}.\, x\mathbf{d}'/x]e_2' : \tau_2$.

Given these results, we can proceed to do induction on the second recursive call of tp: that is, $tp(e_2, A.x:\sigma, S_1, \mathbf{v}_2:C_2, \Theta_1) = (\nu_2, e_2'', S_2, \mathbf{v}_3:C_3, \Theta_2)$ succeeds and there exist a substitution R_2 and dictionary expressions $\mathbf{d}_2$ such that

1. $R_1S_1 = R_2S_2$ except possibly on new type variables of $tp(e_2, A, S_1, C_2, \Theta_1)$,
2. $\tau_2 = R_2\nu_2$,
3. $\mathbf{v}':C' \Vdash \mathbf{d}_2 : R_2C_3$, and
4. $S'A$, $\mathbf{v}':C' \vdash e'' = [\mathbf{d}_2/\mathbf{v}_3](\Theta_2 e_2'') : \tau_2$.

By the definition of tp, we get

$$(\nu_2,\ (\texttt{let}\ x = \lambda \mathbf{w}_1.\, \Theta_1 e_1''\ \texttt{in}\ e_2''),\ S_2,\ \mathbf{v}_3:C_3 \oplus \mathbf{v}_x:C_x,\ \Theta_2)$$

as the final result. In other words, $\tau = \nu_2$, $S = S_2$, $\mathbf{v}:C = \mathbf{v}_3:C_3 \oplus \mathbf{v}_x:C_x$ and $\Theta = \Theta_2$. It remains to show that there exist some dictionaries $\mathbf{d}$ that, together with R_2, satisfy the requirements.

The first three requirements hold obviously. Note that R_2 satisfies the $S' = R_2S_2$ except possibly on new type variables and $\tau_2 = R_2\nu_2$. In addition, by the initial hypothesis, $\mathbf{v}':C' \Vdash \mathbf{d}_x:C_x$, for some dictionaries $\mathbf{d}_x$. Hence $\mathbf{v}':C' \Vdash \mathbf{d}_2\mathbf{d}_x : R_2(C_3 \oplus C_x)$, since $R_2C_x = C_x$. In other words, the required dictionaries $\mathbf{d}$ are $\mathbf{d}_2\mathbf{d}_x$.

We need some more results to establish the last equation required by the theorem. By Lemma 6.21,

$$\mathbf{v}_1:C_1 \Vdash \mathbf{d}_1' : S_1C_a \quad \text{and} \quad \mathbf{v}_3:C_3 \Vdash \mathbf{d}_2' : S_2C_2.$$

In addition, $\mathbf{v}':C' \Vdash \mathbf{d}_2 : R_2C_3$. Hence by Lemma 5.1, $\mathbf{v}':C' \Vdash [\mathbf{d}_2/\mathbf{v}_3]\mathbf{d}_2' : R_2(S_2C_2)$. But $R_2(S_2C_2) = R_1C_2$ and $\mathbf{v}':C' \Vdash \mathbf{d}'':R_1C_2$, so by Lemma 5.2, $\mathbf{d}'' = [\mathbf{d}_2/\mathbf{v}_3]\mathbf{d}_2'$.

Finally, since $\mathbf{v}_x$ do not appear in the translation, it suffices to use the substitution $[\mathbf{d}_2/\mathbf{v}_3]$ to establish the required equation:

$$
\begin{aligned}
S'A,\ \mathbf{v}'{:}C' \ \vdash\ & [\mathbf{d}_2/\mathbf{v}_3]\Theta_2(\texttt{let}\ x = \lambda\mathbf{w}_1.\,\Theta_1 e_1''\ \texttt{in}\ e_2'') \\
& \quad (\,\mathbf{w}_1 \notin \mathrm{dv}(\mathbf{d}_2) \cup \mathrm{reg}(\Theta_2)\,) \\
=\ & \texttt{let}\ x = (\lambda\mathbf{w}_1.\,([\mathbf{d}_2/\mathbf{v}_3]\Theta_2 e_1'')\ \texttt{in}\ [\mathbf{d}_2/\mathbf{v}_3](\Theta_2 e_2'') \\
& \quad (\Theta_2 = [\mathbf{d}_2'/\mathbf{v}_2][\mathbf{d}_1'/\mathbf{v}_a]) \\
=\ & \texttt{let}\ x = \lambda\mathbf{w}_1.\,([\mathbf{d}_2/\mathbf{v}_3][\mathbf{d}_2'/\mathbf{v}_2][\mathbf{d}_1'/\mathbf{v}_a]e_1'')\ \texttt{in}\ e'' \\
& \quad ([\mathbf{d}''/\mathbf{v}_2] = [\mathbf{d}_2/\mathbf{v}_3][\mathbf{d}_2'/\mathbf{v}_2]) \\
=\ & \texttt{let}\ x = \lambda\mathbf{w}_1.\,([\mathbf{d}''/\mathbf{v}_2][\mathbf{d}_1'/\mathbf{v}_a]e_1'')\ \texttt{in}\ [\lambda\mathbf{w}.\,x\mathbf{d}']/x]e_2' \\
& \quad (\textit{using}\ \ R_1 S_1(A.x:\sigma).\ \mathbf{v}'{:}C' \vdash\ e'' = [\lambda\mathbf{w}.\,x\mathbf{d}'/x]e_2' : \tau_2) \\
=\ & \texttt{let}\ x = [\lambda\mathbf{w}_1.\,[\mathbf{d}''/\mathbf{v}_2][\mathbf{d}_1'/\mathbf{v}_a]e_1''/x](\lambda\mathbf{w}.\,x\mathbf{d}')\ \texttt{in}\ \ e_2' \\
& \quad (\textit{Lemma}\ 6.1\,(1)) \\
=\ & \texttt{let}\ x = \lambda\mathbf{w}.\,([\mathbf{d}'/\mathbf{w}_1][\mathbf{d}''/\mathbf{v}_2][\mathbf{d}_1'/\mathbf{v}_a]e_1'')\ \texttt{in}\ e_2' \\
& \quad ([\mathbf{d}_1/\mathbf{v}_1] = [\mathbf{d}'/\mathbf{w}_1][\mathbf{d}''/\mathbf{v}_2],\quad \Theta_1 = [\mathbf{d}_1'/\mathbf{v}_a]) \\
=\ & \texttt{let}\ x = \lambda\mathbf{w}.\,([\mathbf{d}_1/\mathbf{v}_1]\Theta_1 e_1'')\ \texttt{in}\ e_2' \\
& \quad (\textit{using}\ \ S'A,\ \mathbf{v}'{:}C' \vdash\ \lambda\mathbf{w}.\,e_1' = \lambda\mathbf{w}.\,[\mathbf{d}_1/\mathbf{v}_1](\Theta_1 e_1'') : \tau_1) \\
=\ & \texttt{let}\ x = \lambda\mathbf{w}.\,e_1'\ \texttt{in}\ e_2'
\end{aligned}
$$

■

Corollary 6.25 *Suppose that* $S'A,\ \mathbf{v}'{:}C' \vdash e \rightsquigarrow e' : \sigma'$ *and* $(S', \mathbf{v}'{:}C', \Theta') \preceq (S_0, \mathbf{v}_0{:}C_0, \Theta_0)$. *Then* $tp(e, A, S_0, \mathbf{v}_0{:}C_0, \Theta_0)$ *succeeds with* $(\tau, e'', S, \mathbf{v}{:}C, \Theta)$, *and there is a substitution* R, *conversion* K *and dictionary expressions* $\mathbf{d}$ *such that*

1. $S' = RS$, *except possibly on new type variables of* $tp(e, A, S_0, \mathbf{v}_0{:}C_0, \Theta_0)$,
2. $K : \sigma' \preceq_{\mathbf{v}'{:}C'} R\sigma$,
3. $\mathbf{v}'{:}C' \Vdash \mathbf{d} : RD$ *and*
4. $S'A,\ \mathbf{v}'{:}C' \vdash K\,(\lambda \mathbf{w}.\,[\mathbf{d}/\mathbf{u}]\Theta e'') = e' : \sigma'$.

where $(\sigma, \mathbf{u}{:}D, \mathbf{w}) = gen(\tau, SA, \mathbf{v}{:}C, e)$.

Proof: The proof is based on Theorems 6.16 and 6.24. First, by Theorem 6.16, $S'A,\ \mathbf{v}_1{:}C_1 \vdash' e \rightsquigarrow e_1 : \nu$ for some augmented context $\mathbf{v}_1{:}C_1$, type ν, and expression e_1 such that $\mathbf{v}'{:}D \sqsubseteq \mathbf{v}_1{:}C_1$ and

$$S'A,\ \mathbf{v}'{:}C' \vdash K_1\,(\lambda \mathbf{w}_1.\,e_1) = e' : \sigma',$$

where $K_1 : \sigma \preceq_{\mathbf{v}'{:}C'} \sigma_1$ and $(\sigma_1, \mathbf{u}_1{:}C_1', \mathbf{w}_1) = gen(\nu, S'A, \mathbf{v}_1{:}C_1, e)$.

Next, it is clear that $(S', \mathbf{v}_1{:}C_1, \Theta') \preceq (S_0, \mathbf{v}_0{:}C_0, \Theta_0)$. Therefore, by Theorem 6.24, $tp(e, A, S_0, \mathbf{v}_0{:}C_0, \Theta_0) = (\tau, e'', S, \mathbf{v}{:}C, \Theta)$ succeeds and there exist a substitution R and dictionary expressions $\mathbf{d}'$ such that

1. $S' = RS$, except possibly on new type variables of $tp(e, A, S_0, \mathbf{v}_0{:}C_0, \Theta_0)$,
2. $\mathbf{v}_1{:}C_1 \Vdash \mathbf{d}'{:}RC$,
3. $\nu = R\tau$ and
4. $S'A,\ \mathbf{v}_1{:}C_1 \vdash e_1 = [\mathbf{d}'/\mathbf{v}](\Theta e'') : \nu$.

Now suppose that $gen(\tau, SA, \mathbf{v}{:}C, \epsilon) = (\sigma, \mathbf{u}{:}D, \mathbf{w})$. Then by Lemma 6.6, $\sigma = \forall\langle\alpha_i{::}\Gamma_i\rangle.\,\tau$ and $\mathbf{v}{:}C \cong \mathbf{w}{:}\langle\alpha_i{::}\Gamma_i\rangle \oplus \mathbf{u}{:}D$. So by partitioning $\mathbf{d}'$ into $\mathbf{d_1 d_2}$ we can rewrite the last equality to:

$$S'A,\ \mathbf{v}_1{:}C_1 \vdash e_1 = [\mathbf{d}_1/\mathbf{w}][\mathbf{d}_2/\mathbf{u}](\Theta e'') : \nu,$$

from which it follows by Lemma 6.7 that

$$S'A,\ \mathbf{u}_1{:}C_1' \vdash \lambda\mathbf{w}_1.\, e_1 = \lambda\mathbf{w}_1.\,([\mathbf{d}_1/\mathbf{w}][\mathbf{d}_2/\mathbf{u}]\Theta e'') : \sigma_1.$$

Note that σ_1 and σ are related through R. So applying Lemma 6.10 to them, we obtain $(\lambda x.\lambda\mathbf{w}_1.\, x\mathbf{d}_1) : \sigma_1 \preceq_{\mathbf{u}_1:C_1'} R\sigma$. But obviously $\mathbf{u}_1{:}C_1' \sqsubseteq \mathbf{v}'{:}C'$. So by Lemma 6.2, $(\lambda x.\lambda\mathbf{w}_1.\, x\mathbf{d}_1) : \sigma_1 \preceq_{\mathbf{v}':C'} R\sigma$. Then by composing with K_1 using Lemma 6.3, we obtain the required conversion:

$$\lambda x.\, K_1(\lambda\mathbf{w}_1.\, x\mathbf{d}_1) : \sigma' \preceq_{\mathbf{v}':C'} R\sigma.$$

It remains to be shown that $\mathbf{d}_2$ are the required dictionaries and K satisfies the required equation. We first show that $\mathbf{v}'{:}C' \Vdash \mathbf{d}_2 : RD$. By the preceding arguments and (2), $\mathbf{v}_1{:}C_1 \Vdash \mathbf{d}_2 : RD$. So it suffices to show that $\mathrm{tv}(RD) \subseteq \mathrm{dom}(C')$. Since $C_1 \Vdash RD$ and $\mathrm{dom}(D) = C^*(\mathrm{tv}\ SA)$, by Lemma 3.2 we know that for all $\alpha \in \mathrm{dom}(D)$, we have $\mathrm{tv}(R\alpha) \subseteq (C_1)^*(\mathrm{tv}\ S'A)$. Moreover, $S'A = RSA$, C' covers $S'A$ and $C' \sqsubseteq C_1$. So $(C')^*(\mathrm{tv}\ S'A) = (C_1)^*(\mathrm{tv}\ S'A)$, and hence $\mathrm{tv}(RD) \subseteq \mathrm{dom}(C')$. Therefore, $\mathbf{v}'{:}C' \vdash \vdash \mathbf{d}_2 : RD$.

Finally, note that:

$$\begin{array}{lll}
S'A,\ \mathbf{v}'{:}C' \vdash' & (\lambda x.\, K_1(\lambda\mathbf{w}_1.\, x\,\mathbf{d}_1))(\lambda\mathbf{w}.\,[\mathbf{d}_2/\mathbf{u}]\Theta\epsilon'') & \\
= & K_1(\lambda\mathbf{w}_1.\,[\mathbf{d}_1/\mathbf{w}][\mathbf{d}_2/\mathbf{u}]\Theta\epsilon'') & (\beta_d),\ (\beta) \\
= & K_1(\lambda\mathbf{w}_1.\,[\mathbf{d}'/\mathbf{v}]\Theta\epsilon'') & (\mathbf{v} \cong \mathbf{w} \oplus \mathbf{u}) \\
= & K_1(\lambda\mathbf{w}_1.\, e_1) & (4) \\
= & e' : \sigma' &
\end{array}$$

which establishes the required equality. ■

Corollary 6.26 *Suppose that* $dom(C_0) = (C_0)^*(tv\ S_0A)$ *and* $tp(e, A, S_0, \mathbf{v}_0{:}C_0, \Theta_0) = (\tau, e', S, \mathbf{v}{:}C, \Theta)$. *Then* $\lambda\mathbf{w}.\,\Theta e'{:}\sigma$ *is a principal translation for* e *under* SA *and* $\mathbf{v}'{:}C'$ *where* $(\sigma, \mathbf{v}'{:}C', \mathbf{w}) = gen(\tau, SA, \mathbf{v}{:}C, \epsilon)$.

Proof: First, by Lemma 6.23, $SA,\ \mathbf{v}{:}C \vdash e \rightsquigarrow \Theta e' : \tau$. Since $gen(\tau, SA, \mathbf{v}{:}C, \epsilon) = (\sigma, \mathbf{v}'{:}C', \mathbf{w})$, by Lemma 6.7 we have $SA.\ \mathbf{v}'{:}C' \vdash e \rightsquigarrow \lambda\mathbf{w}.\,\Theta e' : \sigma$

Next, suppose that $SA,\ \mathbf{v}'{:}C' \vdash e \rightsquigarrow e'' : \sigma'$ for some type scheme σ'. We apply Lemma 6.25 to complete the proof. To do so, we need to show that $(S, \mathbf{v}'{:}C', \Theta) \preceq (S_0, \mathbf{v}_0{:}C_0, \Theta_0)$.

By Lemma 6.21, $(S, \mathbf{v}{:}C, \Theta) \preceq (S_0, \mathbf{v}_0{:}C_0, \Theta_0)$. In other words, $\mathbf{v}{:}C \Vdash \mathbf{d}_1 : SC_0$ for some dictionary expressions $\mathbf{d}_1 = \Theta\mathbf{v}_0$. Then by Lemma 3.2, $\mathrm{tv}(S\alpha) \subseteq C^*(\mathrm{tv}\ SA)$ for all $\alpha \in \mathrm{dom}(C_0)$. But by Lemma 6.6, $\mathrm{dom}(C') = C^*(\mathrm{tv}\ SA)$. So it follows from Lemma 5.2.2 that $\mathbf{v}'{:}C' \Vdash \mathbf{d}_1 : SC_0$. Hence $(S, \mathbf{v}'{:}C', \Theta) \preceq (S_0, \mathbf{v}_0{:}C_0, \Theta_0)$.

Therefore, by Lemma 6.25, there exist a substitution R and dictionary expressions $\mathbf{d}$ such that

1. $S = RS$ except possibly on new type variables of $tp(e, A, S_0, C_0)$,
2. $K : \sigma' \preceq_{\mathbf{v}'{:}C'} R\sigma$,
3. $\mathbf{v}'{:}C' \Vdash \mathbf{d} : RC'$ and
4. $SA,\ \mathbf{v}'{:}C' \vdash K\,(\lambda\mathbf{w}.\,[\mathbf{d}/\mathbf{v}']\Theta e') = e'' : \sigma'$.

Furthermore, since $\mathrm{dom}(C') = C^*(\mathrm{tv}\ SA)$ and $\mathrm{tv}(\sigma) \subseteq C^*(\mathrm{tv}\ SA)$, it follows from (1) that $R\sigma = \sigma$, $RC' = C'$ and $\mathbf{d} = \mathbf{v}'$. Therefore, $\lambda\mathbf{w}.\,\Theta e' : \sigma$ is a principal translation for e under SA and $\mathbf{v}'{:}C'$. ■