Extensible Records in the System E Framework and a New Approach to Object-Oriented Type Inference

Ryan Heise

A dissertation submitted for the degree of Doctor of Philosophy

Faculty of Information Technology

University of Technology, Sydney

2013

Abstract

Extensible records were proposed by Wand as a foundation for studying object-oriented type inference. One of their key benefits is that they allow for an elegant encoding of object-oriented inheritance, where one class of objects may be defined as an extension of another class of objects. However, every system of type inference designed for extensible records to date has been developed in the Hindley/Milner-style, a consequence being that polymorphism in these systems is not first-class and analysis is not strictly compositional. We argue that both of these features are necessary to retain the modelling and engineering benefits of traditional object-oriented languages such as Java:

1. Object-oriented modelling depends on the treatment of objects as first-class citizens, and this demands a type inference system capable of handling first-class polymorphism.
2. Object-oriented engineering encourages the separate development of software modules, and this should be supported by the type inference system with compositional analysis.

Both of these features are present in a type system for the λ-calculus called System E, which supports first-class polymorphism via intersection types, and compositional type inference via expansion variables. However, research into System E has so far focused on refining and simplifying the formulation of expansion variables and exploring type inference algorithms with various properties. Meanwhile, the system has not yet been extended beyond the terms of the pure λ-calculus and it lacks many features that would be needed in a practical object-oriented language.

In this dissertation, we combine System E with Wand's extensible records resulting in a new approach to type inference for extensible records that better preserves the modelling and engineering benefits of object orientation stated above. The resulting system, called System $\mathrm{E}^{\mathrm{vcr}}$, is significant because previous type inference systems, both for extensible records in particular, and also for object orientation in general, have at best preserved only one or the other of these two benefits, but never both of them simultaneously. System $\mathrm{E}^{\mathrm{vcr}}$ also makes a significant contribution to the work on System E, since it demonstrates for the first time that the System E's expansion variables can be adapted to analyse programs whose term language extends beyond the pure λ-calculus.

To demonstrate the potential use of System $\mathrm{E}^{\mathrm{vcr}}$ in object-oriented type inference, an implementation of our type inference algorithm was created and is shown to succeed on problem examples that previous systems either fail to analyse, or else fail to analyse compositionally.

Declaration

I declare that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text. I also declare that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I declare that all information sources and literature used are indicated in the thesis.

(Ryan Heise)

Acknowledgements

I wish to thank my principal supervisor Barry Jay and co-supervisor Wayne Brooks for their guidance and support, and for teaching me the toolset that I depended upon to develop the ideas within this dissertation. I also wish to thank Pascal Zimmer for helping me to understand the capabilities of existing extensible record type systems, Joe Wells for many vital and informative discussions about expansion variables and Adam Bakewell for assisting me to understand the opus unification algorithm. Many thanks also go to fellow student Quy Tuan Nguyen with whom I have had many stimulating discussions about both his own research and mine.

This work was supported by an Australian Postgraduate Award from the Australian Government Department of Innovation, Industry, Science and Research.

Table of Contents

List of System E^{v} Definitions and Theorems

List of System E^{vc} Definitions and Theorems

List of System $\mathrm{E}^{\mathrm{vcr}}$ Definitions and Theorems

List of Figures

Chapter 1

Introduction

Type inference is a feature of some programming languages that use type information to statically check the validity of programs. Normally, this type information is manually inserted into a program by the programmer, and then an automated type checker uses this information at compile time to check whether the right types of expressions are used in the right types of contexts. The type checker ideally guarantees that accepted programs will not have any type errors at runtime. If a language goes one step further to offer type inference, then the programmer is free to leave out this type information, and a type inferencer will attempt to automatically compute the missing information.

Type inference has long been used in functional programming languages to relieve programmers of the burden of manually inserting type information. However, attempts to bring the benefits of type inference to object-oriented languages have been met with limited success. This is unfortunate since, from the 1990s, object-orientation has been the dominant software development methodology used throughout both industry and education.

Table 1.1 lists the top 11 programming languages in April 2013 listed by the TIOBE Programming Community Index [1], an indicator of the popularity of programming languages. Although such lists should be taken as only rough indicators of language popularity, we can observe from this list a clear pattern of the dominance of the object-oriented paradigm, with only C not supporting object-oriented programming constructs. A more interesting observation, though, is that the languages in this list

Rank	Language	Rating
#1	C	17.862%
#2	Java	17.681%
#3	C++	9.714%
#4	Objective-C	9.598%
#5	C#	6.150%
#6	PHP	5.428%
#7	(Visual) Basic	4.699%
#8	Python	4.442%
#9	Perl	2.335%
#10	Ruby	1.972%
#11	JavaScript	1.509%

Table 1.1: TIOBE Programming Community Index for April 2013

supporting object-orientation adopt two distinct approaches to typing. Java, C++, Objective-C and C# check types at compile time (i.e. "static typing"), and rely on programmer-supplied type annotations to achieve this, while PHP, Python, Perl, Ruby and JavaScript check the types of values at runtime (i.e. "dynamic typing") and do not require any type information to be known at compile time. Visual Basic can operate under either approach, and will use static typing if the programmer supplies type annotations or will use dynamic typing otherwise.

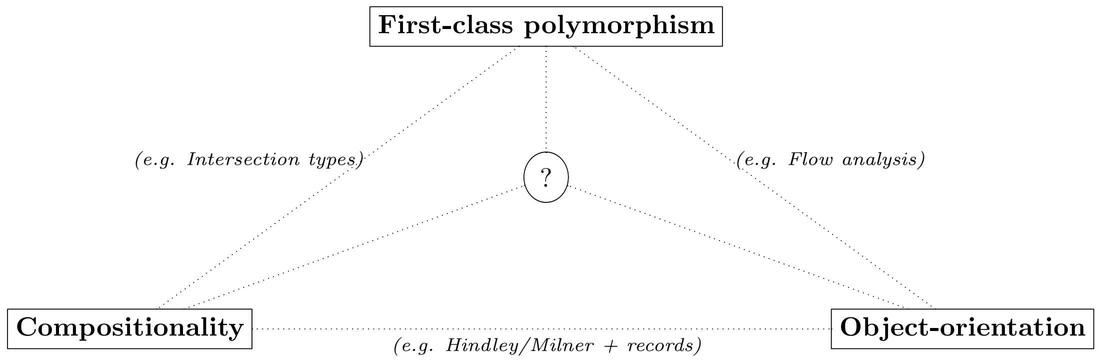
While static typing with explicit type annotations may create more work for the programmer, it also provides the significant benefit of ruling out unsafe programs at compile time and alerting the programmer to faulty code before it is released to users. On the other hand, while dynamic typing may allow for more rapid software development, it also leaves programs vulnerable to type errors at runtime. Missing from Table 1.1, however, are object-oriented languages that are statically typed and require no type annotations through the use of type inference. Such languages could be seen as truly mixing the best of both extremes by allowing programs to be written rapidly without requiring the programmer to supply type information, but to still benefit from static type checking by having this type information automatically inferred by the compiler.

While a number of foundations for object-oriented type inference have been proposed, none has been as successful as those used by functional programming languages, and there has not yet been a proposal that entirely preserves the modelling and engineering benefits of the object-oriented programming style. In particular, previous efforts have not been able to simultaneously address both of the following considerations that are important in the object-oriented paradigm:

1. Object-oriented modelling depends on the treatment of objects as "first-class" citizens, and as such, objects are often passed through parameters. With respect to the type system, objects often also have polymorphic types, meaning that they may be reused in different contexts at different types. Thus, we should expect of any object-oriented type inference system that it be able to infer polymorphic types for parameters, which is referred to as *first-class polymorphism.*

2. Object-oriented engineering encourages the separate development of software modules, and as such, it ought to be possible to analyse and compile a single module on its own without having to re-analyse and recompile the entire system. This kind of analysis is referred to as *compositional analysis* and should be expected of any object-oriented type inference system.

The overall challenge is to create a type inference system that simultaneously supports (1) first-class polymorphism, (2) compositionality and of course (3) object-orientation itself. The current state of affairs is that we have systems that support any two out of these three features (to an extent), but no system that supports all three:



A useful observation is that Hindley/Milner-style type inference [45], despite being widely used in functional languages, is incapable of inferring types that require first-class polymorphism, and that while certain kinds of flow analysis [46, 49, 3] are capable of this, flow analysis is fundamentally incompatible with compositional analysis. However, a workable combination is intersection types which support both first-class polymorphism and compositionality, and records which support object-orientation.

In this dissertation, we pursue object-oriented type inference using a particular kind of record known as an *extensible record* [62, 64], and a particular system of intersection types called System E [17]. The resulting system has been implemented and is demonstrated to successfully infer types for functional, object-oriented programs that make use of first-class polymorphism, while at the same time performing type inference in a strictly compositional manner. While the system presented in this dissertation does not include all features often found in object-oriented programming languages, such as general recursion and state, it instead aims to provide a foundation for studying object-oriented type inference where compositionality and first-class polymorphism are built in from the beginning. Such a foundation is important if type inference is ever to be adopted by mainstream object-oriented languages such as Java where first-class polymorphism and compositional analysis are expected. By focusing on only the core elements of object orientation, this work follows in the same vein as Featherweight Java [27] and Abadi and Cardelli's Object Calculus [2], which also omit all but a few core constructs that are necessary to represent objects themselves, simplifying to an extent that allows complex typing issues to be better understood.

This introduction is organised as follows. Section 1.1 outlines key considerations for object-oriented type inference and presents motivating examples. Section 1.2 reviews the prior work. Section 1.3 provides an overview of our new system based on extensible records and System E. Section 1.4 outlines the chapters of this dissertation. Finally, Section 1.5 provides instructions on how to read this dissertation.

1.1 Key Considerations for Object-Oriented Type Inference

A type inference system that preserves the modelling and engineering benefits of object-orientation should have the following features:

- Object-orientation
- First-class polymorphism
- Compositionality

The following subsections will introduce each of these features, argue for the importance of the last two features with respect to the first, and set out all of the associated problem examples that we intend to handle in our system.

1.1.1 Object-Orientation

Within object-orientation, many alternative paradigms have been explored, including prototypes [59], multimethods [33] and aspects [30] to name just a few. In this dissertation we will limit our analysis to object-oriented concepts that are common to popular, mainstream object-oriented languages such as C++, Java and C#. Shared by all of these languages are: objects, classes, inheritance, method overriding and dynamic dispatch. In this section, we introduce these five features using examples in the Java language.

The fundamental concept in object-oriented programming is that of an *object*. An object is a programming abstraction that allows for the modelling of real-world objects. For example, a banking program might be built from account and customer objects, while a car simulator might contain car, engine and wheel objects. The key point of object-orientation is that an object encapsulates both data and associated operations, meaning that a car object, for instance, might contain position, velocity and direction data, *along with* operations to control the accelerating, braking and steering of the car.

From the program's point of view, an object is essentially a record of *data fields* with associated functions attached to it, called *methods*. Each method is aware of, and can refer to, the object to which it belongs. Most object-oriented languages also classify objects into *classes* so that all objects belonging to a class share the same record structure and methods. Programmers will therefore write code for the classes only and use those classes to spawn like objects. For each class, a special method called a *constructor* defines how new objects of the class are created and initialised.

The example below defines a class of `Rectangle` objects in the Java language, with fields `width` and `height`, a constructor `Rectangle`, and associated methods `area` and `toString`:

```
class Rectangle
{
    double width;
    double height;
    Rectangle(double rwidth, double rheight) { this.width = rwidth; this.height = rheight; }
    double area() { return this.width * this.height; }
    String toString() { this.width + "x" + this.height; }
}
```

The underlined expressions are called *types* and they are used to restrict the set of possible values that may be assigned to fields, parameters and other sorts of variables, as well as the values that must be returned by methods. For example, the fields `width` and `height` may be assigned only values of type `double`, while the method `toString` must return a value of type `String`.

The following program illustrates the creation and use of an object from class `Rectangle`:

```
1  Rectangle r = new Rectangle(2,3);

2  System.out.println(r.width);
3  System.out.println(r.height);
4  System.out.println(r.area());
5  System.out.println(r.toString());
```

Line 1 shows constructor `Rectangle` being invoked via operator `new`. Lines 2-5 illustrate the use of the *dot* operator "`.`" to access both the fields and the methods of the new object `r`. The resulting 4 values are printed using method `println` which itself is accessed from the built-in object `System.out` via the *dot* operator.

Since both the fields and methods are accessed via the dot operator, it is intuitively as though the object `r` were simply a record of 4 fields, altogether containing 2 data values and 2 functions. However, what makes objects different from records is that each method can refer to its containing object. In the Java syntax shown above, this is accomplished via the keyword `this`. Hence, when the `area` method uses the dot operator to access `this.width` and `this.height`, it is referring to the `width` and `height` fields of the containing object.

An important reuse feature of class-based object-oriented languages is the ability for a *sub*-class of objects to be defined as an extension of a *super*-class of objects, a feature known as *inheritance*. Using this feature, we can define a class of `PositionedRectangle` objects as an extension of the class of `Rectangle` objects. The non-constructor members of the superclass are automatically inherited by the subclass, and so the subclass needs only to define a new constructor and any additional members:

```
class PositionedRectangle extends Rectangle
{
    double x;
    double y;

    PositionedRectangle(double rx, double ry, double rwidth, double rheight)
    {
        super(rwidth, rheight); // Invoke the superclass' constructor
        this.x = rx;
        this.y = ry;
    }

    double distanceTo(PositionedRectangle rect)
    {
        return sqrt(sqr(rect.x-this.x)+sqr(rect.y-this.y));
    }

    String toString()
    {
        return super.toString() + "(" + this.x + "," + this.y + ")";
    }
}
```

The subclass adds two fields `x` and `y`, a new constructor `PositionedRectangle`, a new method `distanceTo`, and also *overrides* the existing `toString` method with a new definition. The superclass' version of `toString` may still be accessed from the subclass via `super`, and likewise for the superclass' constructor. Through this mechanism, the subclass can reuse any code from the superclass that was not inherited.

Object-oriented languages typically use *dynamic dispatch* when invoking a method. Given a variable `rect` referring to either a `Rectangle` or a `PositionedRectangle` where the actual type is not known at compile time, the invocation `rect.toString()` will select the appropriate version of the `toString()` method based on the runtime value of `rect`. This is illustrated by the following example:

```
Rectangle larger(Rectangle r1, Rectangle r2)
{
    if (r1.area() > r2.area()) return r1;
    else return r2;
}

String test()
{
    Rectangle r1 = new Rectangle(2, 2);
    PositionedRectangle r2 = new PositionedRectangle(1, 2, 10, 20);
    Rectangle bigRect = larger(r1, r2);
    return bigRect.toString();
}
```

When method `bigRect.toString()` is invoked inside method `test()`, the runtime environment will identify that `bigRect` is currently holding the value `r2` which is a `PositionedRectangle`, and therefore the `PositionedRectangle` version of `toString` will be used to produce the string ”10x20(1,2)”.

1.1.2 First-Class Polymorphism

In this section, we introduce the concepts of polymorphism and first-class polymorphism, and discuss why these are important in the object-oriented programming style. Finally, we illustrate through examples how first-class polymorphism arises in practice in object-oriented programming.

What is polymorphism?

Polymorphism refers to the ability of a program expression to be reused in different contexts at different types. This is achieved by assigning that expression a *polymorphic type* that somehow represents the set of different types at which the expression can be used. In modern object-oriented languages such as Java, C++ and C#, polymorphism can arise in three different situations as illustrated by the following Java example:

```
void printRect(Rectangle r) { System.out.println(r.toString()); }
<X> X identity(X x) { return x; }

void test() {
    Rectangle r = new Rectangle(2,3);
    PositionedRectangle pr = new PositionedRectangle(2,3,2,3);

    System.out.println(r.area());
    System.out.println(pr.area());

    printRect(r);
    printRect(pr);

    Rectangle r1 = identity(r);
    PositionedRectangle pr1 = identity(pr);
}
```

In lines 6-7, *inheritance polymorphism* allows method `area` to work in both class `Rectangle` and its subclass `PositionedRectangle` without being recoded. In lines 8-9, *subtype polymorphism* allows method `printRect` to work on both objects of type `Rectangle` and objects of its subtype `PositionedRectangle` without being recoded. And in lines 10-11, *parametric polymorphism*, allows method `identity` to be used to take a `Rectangle` to a `Rectangle` and a `PositionedRectangle` to a `PositionedRectangle` without being recoded, since method `identity` is defined (line 2) to take any `X` to an `X` for some type parameter `X`, universally quantified by `<X>`.

What is first-class polymorphism?

Objects are treated as *first-class* citizens in the sense that they can be used in expressions, assigned to variables and passed through parameters to methods. This is in contrast to classes which are traditionally not first-class citizens and can be used only in specific situations, such as immediately following the `new` operator while creating a new object from a class. Similarly, polymorphism in mainstream object-oriented languages is also first-class in the sense that objects can be given polymorphic types regardless of whether those objects are used within expressions, assigned to variables or passed through parameters to methods.

Of the three kinds of polymorphism described above, parametric polymorphism has been historically difficult to support in type inference in combination with first-class polymorphism. Milner's Algorithm $\mathcal{W}$ [45] for the Hindley/Milner type system, which is used in the majority of functional programming languages that have type inference, and local type inference [48] which is used in the functional/object-oriented language Scala, both sacrifice first-class polymorphism to make type inference possible[1]. The common realisation in both of these approaches is that it is easy to infer a parametrically polymorphic type for a local variable, but much harder to infer one for a method or function parameter. This can be illustrated by the following example:

```
class Foo {
    void poly(b) {
        let a = new A();
        a.m1(true);
        a.m1(42);
```

[1]Programming languages based on these systems sometimes permit first-class polymorphism, but the programmer is required to manually supply type information in such cases.

```
        b.m2(true);
        b.m2(42);
    }
}
```

In this program, methods `a.m1` and `b.m2` are each used polymorphically since they each need to deal with arguments of type `boolean` and `int`. In the case of `a.m1`, there happens to be a convenient way to infer the complete polymorphic type of `a`, including its method `m1`. Because `a` is initialised locally to an instance of class `A`, it is possible to analyse class `A` to determine the type of `a`.

In contrast to variable `a`, we have no such hint about the variable `b` since its actual value originates outside the context of method `poly`. If parametric polymorphism is used, there are various unrelated possible types for method `b.m2` and no clear choice exists. Some possible alternative types include (see Section 1.1.3 for a discussion on why these types are unrelated):

```
<X> X m2(X x);
<X> Pair<X,X> m2(X x);
<X> Pair<X,Pair<X,X>> m2(X x);
<X> Pair<X,Pair<X,Pair<X,X>>> m2(X x);
...
```

The way to deal with this situation in the widely used Hindley/Milner system is to impose a restricted form of parametric polymorphism called *let-polymorphism* in which only variables introduced via let-declarations, such as variable `a`, may be polymorphic, while parameters such as variable `b` may not be polymorphic. After applying this restriction, the above program is simply ruled out as an invalid program.

It is not an easy task to design a type inference system that supports first-class polymorphism, and some of the challenges appear to be inherent in the use of parametric polymorphism itself, suggesting that alternative approaches to first-class polymorphism should be considered. One of the main difficulties in working with parametric polymorphism is that systems based on it tend to have limited support for compositional analysis, an issue that will be discussed in Section 1.1.3 and Section 1.2.3.

First-class polymorphism in practice

In this section, we examine how first-class polymorphism is used in practice in object-oriented programming. There are fundamental differences between the function-oriented and object-oriented programming styles that influence the use of first-class polymorphism in each style. It is these differences that make the limitations of Hindley/Milner-style type inference, which are relatively more tolerable in the function-oriented programming style, less tolerable in the object-oriented programming style.

In function-oriented programming, functions are declared separately from data, and are often free to be declared globally via let-declarations and act on local data in any part of the program. Because of this, there is a good chance that functions will benefit from let-polymorphism (if they are declared via a let-declaration) and the type inferencer will be able to infer its polymorphic type automatically.

In object-oriented programming, methods (the object-oriented counterpart to functions) are not declared globally. Instead, methods and data are bundled together into objects, making methods just as likely to be accessed from parameters as regular data, and this reduces the opportunity for methods to benefit from let-polymorphism during type inference.

The following pattern describes a general situation in which first-class polymorphism can arise in object-oriented programming:

```
class C
{
    A partA;
    B partB;

    void action(ServiceProvider service)
    {
        service.operateOn(partA);
        service.operateOn(partB);
    }
}
```

Some class `C` is composed of two (or more) parts, each of different types. In one of its methods (here, `action`), a service provider object is passed in as a parameter, and one of its methods (here, `operateOn`) is applied polymorphically to the two parts, `partA` and `partB`, which have different types. This polymorphism is first class since `operateOn` is accessed via an object that came through a parameter.

One application of this pattern is in computer games designed to run on today's multicore CPU architectures. With the increasing prevalence of multicore CPUs on the desktop, it has become increasingly important to consider how best to design games to utilise the parallel processing power available in these CPUs [21]. In a multithreaded game design, artificial intelligence, audio, physics, graphics, etc. can all be processed in parallel by different threads with each thread potentially running on a different CPU core. Thread creation tends to be an expensive operation and so it is useful to share a thread pool between different parts of the application allowing old threads to be recycled and reused rather than recreated.

In Java, a game component can use a shared thread pool to execute two tasks in parallel as follows:

```
1  class GameComponent {
2    Callable<Result1> task1 = ...;
3    Callable<Result2> task2 = ...;

4    void doWork(ExecutorService threadPool) {
5      Future<Result1> future1 = threadPool.submit(task1);
6      Future<Result2> future2 = threadPool.submit(task2);

7      Result1 result1 = future1.get();
8      Result2 result2 = future2.get();

9      process(result1, result2);
10   }
11 }
```

Lines 2 and 3 define the two tasks to be performed in parallel, where `task1` produces a result of type `Result1` and `task2` produces a result of type `Result2`. Here, physics algorithms, audio processing, or other typical game functions could be substituted for `task1` and `task2`, or indeed, a single game function could be further decomposed into threads (see [21] for examples).

Method `doWork` takes a shared `threadPool` as a parameter and uses it to submit the two tasks to be executed. The `threadPool` parameter is an instance of the standard Java class `ExecutorService`, and its `submit` method is *polymorphically* defined with the following method signature:

```
<T> Future<T> submit(Callable<T> task)
```

Here, the type parameter `T` indicates the type of result produced by a given task, and in our case it is either `Result1` or `Result2`. `Callable` represents the generic interface of a task, and `Future` represents a future result that is not immediately ready but will become ready once the thread completes.

Note that the thread pool's `submit` method is used polymorphically. On Line 5, `task1` is submitted to be executed, and a `Future` result of type `Result1` is returned. On Line 6, `task2` is submitted and a `Future` result of type `Result2` is returned. Since the submit method belongs to a thread pool that was shared via a parameter, this polymorphism is also first class.

Finally, lines 7-8 block and wait for the two threads to complete, and then obtain the results from the two futures so that they can be combined and processed on Line 9.

Reflecting on this example, the first-class polymorphism is a direct result of method `submit` being bundled together with the data (i.e. the thread pool) that is passed through a parameter, and this "bundling" is of course a technique that is central to object-oriented programming. In functional programming, the polymorphic method `submit` might instead be declared as a global function where it can benefit from let-polymorphism, and might be defined to take the thread pool data that was once bundled together with it as an additional parameter. While the first-class polymorphism in the above example could possibly be avoided by coding in the functional style, the challenge is to build a type inference system that supports the kinds of polymorphism that arise in the object-oriented programming style.

Next we consider an example taken from the Java implementation of the type inference algorithm presented in this dissertation (for complete source code, see [26]). Type inference algorithms often make use of *type substitutions* which substitute types for type variables in given expressions, although to simplify matters in the following presentation, we shall consider only the substitution of variables for other variables. If `s` is a substitution and `e` is an expression, we intend to use code of the form `s.apply(e)` to apply substitution `s` to expression `e`, and this should produce a new expression with all of the variable substitutions applied. A substitution can work on different kinds of expressions. For example, a substitution could work on a single variable. It could also work on another substitution that contains variables within it. In the full System E (the rules of which are presented in Chapter 2, Figure 2.3), a substitution will also work on other kinds of expressions such as types containing variables and expansions containing variables. Because a substitution can work on different types of expressions, the type of a substitution must be polymorphic. And because substitutions are often passed as parameters to methods, the example will also employ first-class polymorphism.

The example consists of 5 classes:

- Class `Substitution` defines a linked list of extended substitutions with an identity substitution at the root.
- Class `IdentitySub` defines an identity substitution which has no effect when applied to an expression.
- Class `ExtendedSub` defines one particular link in a list of substitutions of one variable for another variable.
- Class `Variable` defines a named variable.
- Class `Expression` defines the superclass of all expressions. `Substitution` and `Variable` are subclasses.

First, we have the class of `Substitution` objects which may be applied to expressions. A substitution is itself a kind of expression:

```
abstract class Substitution extends Expression<Substitution> {
    abstract <X extends Expression<X>> X apply(X expression);
}
```

The polymorphic type for method `apply` states that the method can operate on any type `X` of `Expression` and will produce as its result the same type `X` of `Expression`. For example, if applied to a `Variable`, the return type will be `Variable`, and if applied to a `Substitution`, the return type will be `Substitution`.

There are two kinds of substitution which together are used to build linked lists of variable assignments. The first kind of substitution, represented by class `IdentitySub`, is the identity substitution. It has no effect when applied to an expression, and it is used as the root of a substitution linked list:

```
class IdentitySub extends Substitution {
    <X extends Expression<X>> X apply(X expression)  { return expression; }
    ExtendedSub applySubToSelf(ExtendedSub s)  { return s; }
    String str() { return "{}"; }
}
```

For now, we focus on the implementation of method `apply` which in this case just returns back the original expression without modification.

The second kind of substitution is the `ExtendedSub` which consists of a variable assignment of variable `v` to variable `x`, combined with all of the variable assignments contained in the `tail` substitution:

```
1  class ExtendedSub extends Substitution {
2      Variable x;
3      Variable v;
4      Substitution tail;

5      ExtendedSub(Variable x, Variable v, Substitution tail) {
6          this.x = x;
7          this.v = v;
8          this.tail = tail;
9      }

10     <X extends Expression<X>> X apply(X expression) {
11         return expression.applySubToSelf(this);
12     }

13     Substitution applySubToSelf(ExtendedSub s) {
14         return new ExtendedSub(this.x, s.apply(this.v), s.apply(this.tail));
15     }

16     String str() {
17         return this.x.str() + "=" + this.v.str() + "," + this.tail.str();
18     }
19 }
```

Again we shall focus on the implementation of method `apply`. Unlike an `IdentitySub`, the way in which to apply an `ExtendedSub` to an expression depends upon the type of expression. Hence, Line 11 uses dynamic dispatch to let each expression decide how to handle the application via method `applySubToSelf`. Each kind of expression must provide an implementation of this method to define how an `ExtendedSub` should be applied to that expression. Because the expression on Line 11 could itself be another substitution, even substitutions must define this `applySubToSelf` method. In Line 14 above, an `extendedSub` defines this method by applying the given substitution `s` recursively to each component of `this`.

There are two points to make about substitutions. First, method `applySubToSelf` in class `ExtendedSub` uses its parameter `s` polymorphically, on one occasion to transform the variable `this.v` into another variable, and on another occasion to transform the substitution `this.tail` into another substitution. Because the parameter is used polymorphically, this method requires first-class polymorphism.

The second point is that the polymorphic type for method `apply` in class `Substitution` is arguably quite difficult for a programmer to discover, and automated type inference could help to alleviate this problem. In this case, we have two different mechanisms for polymorphism being used together: parametric polymorphism on the type parameter `X` to ensure that the return type is the same as the parameter type, and subtype polymorphism to allow `X` to be any subtype of `Expression`. An added complication is that class `Expression` must also be parameterised by a type parameter so that the related methods in that class can also produce the correct return types.

Next, we have the class of `Variable` objects. Each `Variable` has a name, and can be substituted for other variables:

```
class Variable extends Expression<Variable> {
    String name;
    Variable(String name) { this.name = name; }
    Variable applySubToSelf(ExtendedSub s) {
        if (this.name.equals(s.x.name)) { return s.v; }
        else { return s.tail.apply(this); }
    }
    String str() { return (this.name); }
}
```

Method `applySubToSelf` performs the actual work of variable substitution. If the head of the substitution matches this variable, then the variable assignment in the head is applied, otherwise the tail of the substitution is applied.

Finally, we have class `Expression` of which all other classes are subclasses. Every expression defines an `applySubToSelf` method, as we have seen, and also a `str` method for producing a string representation of the expression:

```
abstract class Expression<X extends Expression<X>> {
    abstract X applySubToSelf(ExtendedSub s);
    abstract String str();
}
```

Based on the implementation of the `str` methods, the string {} would represent the identity substitution, while the string `x=a,y=b,`{} would represent a substitution that, when applied to any expression, will replace variable `x` by `a`, and variable `y` by `b`.

To show the above classes working together, and to exercise the polymorphism found within them, the following program creates two substitutions, applies one substitution to the other, and then prints a string representation of the result:

```
Variable a = new Variable("a");
Variable b = new Variable("b");
Variable c = new Variable("c");
Variable d = new Variable("d");
IdentitySub id = new IdentitySub();

// Create two substitutions and compose them
Substitution s1 = new ExtendedSub(b, a, id);
Substitution s2 = new ExtendedSub(a, b, new ExtendedSub(c, d, id));
Substitution s1s2 = s1.apply(s2);

// Print out the result
System.out.println("s1 = " + s1.str());
System.out.println("s2 = " + s2.str());
System.out.println("s1 s2 = " + s1s2.str());
```

This produces the following output:

```
s1 = b=a,{}
s2 = a=b,c=d,{}
s1 s2 = a=a,c=d,b=a,{}
```

Reflecting on this example, the key polymorphic code is `s.apply(e)` which applies a substitution to an expression. Since there are various types of expression, `apply` must be polymorphic to support them all. At the same time, there are different kinds of substitution, which are handled by dynamic dispatch. It should be noted that if there were only one kind of substitution, it might be tempting to switch the object and parameter around to read `e.apply(s)` so that `apply` need no longer be polymorphic. If the type inference system of the language did not support first-class polymorphism, then this trick could be used to avoid the type inference system's limitation.

However, there are good object-oriented reasons to define `apply` as a method on substitutions rather than on expressions. At purely the interface level, `apply` is conceptually an operation that is associated with substitution objects and ought to be part of the specification of how a substitution may be used. Getting the interface right conceptually can also help to make the system future proof so that new classes introduced in the future would be more likely to fit the interface. As it turns out, when this example is extended to the full system given in [26], substitutions will be but one of a more general class of objects known as *expansions* that can be applied to any expression, and at this point, the need for first-class polymorphism cannot be avoided.

Once again, the challenge is to offer type inference that allows programmers to make their usual object-oriented design choices without letting limitations of the type inference system dictate their design in any way.

1.1.3 Compositionality

Object-oriented programming encourages a separation of concerns between different software components. At the lowest level, a well-designed class should consist of fields and methods that are closely related and this tends to increase internal dependencies and lower external dependencies. At a higher level, classes can be grouped into packages, again with an aim to increase internal dependencies and lower external dependencies. This separation of concerns is good from a software engineering point of view, especially in large software projects, because it allows different programmers to independently work on different classes, and different teams of programmers to independently work on different packages. As long as the boundaries between different software components are clearly defined, it should be possible for a programmer to recompile the one class being worked on, or for a team of programmers to recompile the package being worked on, without needing to recompile the entire software project. To support such separate compilation, any analysis during compilation should be *compositional*, meaning that it should be possible to analyse different program subcomponents in isolation.

For many object-oriented languages, separate compilation is supported by constructing a dependency graph between classes and checking file modification dates to determine whether any dependencies need to be compiled, and if not, then compiling only the target class. In the case of Java, dependency information can be indicated at the top of each source file in the form of *import* statements which specify other classes that are depended upon[2]. Also, the compiler must be configured with a *class path* specifying the directories where classes to be imported can be found in the file system. The process is roughly illustrated by the following example:

[2]This is a simplification. Java actually treats import statements as shorthand for writing fully qualified class names throughout the body of the class, and analyses the entire class to discover the dependencies

```
1 import package1.A;
2 import package2.B;
3 class C {
4     void mono(B b) {
5         A a = new A();
6         a.m1(true);
7         b.m2(42);
8     }
9 }
```

This Java class specifies dependencies on two external classes `A` and `B`. Because these dependencies are explicitly named, the compiler can, while compiling `C`, efficiently locate definitions for `A` and `B` in the class path under file names corresponding to these class names. Assuming `A` and `B` have not changed since the last compile, the compilation of `C` can be completed using the already-compiled type information for `A` and `B`.

If we consider type inference, care must be taken to preserve the benefits of separate compilation. Before omitting the types, we must have a clear understanding of what is a type and what is a class. In Java, C++, C# and many other object-oriented languages, every class name is a also type name, and this can lead to some confusion about the difference between the two. In Line 5, the occurrence of `A` in "`A a`" indicates the *type* of variable `a`, while the occurrence of `A` in "`new A()`" indicates the *class* from which a new object is to be created. The conceptual difference between the two is that *class* `A` holds all of the code needed to create a fully functioning object, while *type* `A` specifies only what field and methods an object must have, without specifying any code for the methods.

In a hypothetical version of Java supporting type inference, the above program might be written as follows, where parameter "`B b`" is now specified simply as "`b`", and local variable declaration "`A a`" is now replaced by a let declaration without a type:

```
import package1.A;
class C {
    mono(b) {
        let a = new A();
        a.m1(true);
        b.m2(42);
    }
}
```

Notice that the process of omitting types can lead to some import statements being omitted that were previously used for efficient compilation. In particular, while there is still a genuine dependency on class `A`, which is needed to create the new object `a`, there is no longer any dependency on class `B`, and the inferred type of parameter `b` could just as well be any alternative type that also provides a method `m2` that accepts an `int` argument. If the inferred type of `b` must be a class name (such as “`B`”), then the only way to find a matching class would be to search the entire class path for such matching candidates. Setting aside the potential obstacle posed by the sheer number of classes that would need to be searched every time a parameter type must be inferred (roughly 7,000 classes in the standard Java library alone, as of version 1.6), such a whole program analysis will lead to the following problems:

1. In order to make the parameter type as general as possible (so as not to unnecessarily prevent arguments from being passed into it), the inferred parameter type would need to comprehensively describe *all* possible candidate classes within the set of 7,000+ classes. Such a type could potentially become very large.
2. If, at the time of compilation, a complete set of matching classes was found and combined to produce the inferred parameter type, but then after the initial compilation, a new class `B1` was created that would now also match, a recompilation of class `C` would now be required to include `B1` as an acceptable parameter type. Thus, class `C` would need to be regularly recompiled as new classes are invented, despite class `C` having no explicit dependency on those classes. The result is that separate compilation would be problematic.

One way to avoid the problems associated with whole-program analysis when inferring parameter’s type is to use *structural types* [4] where, rather than parameter `b`’s type being a class name, it can instead be a description of the fields and methods that are required of `b`. For example, it is possible, based on the usage of variable `b`, to infer a structural type such as

```
{
    void m1(boolean);
}
```

indicating that `b` must be an object containing a method `m1` that takes a parameter of type `boolean` and returns a result of type `void`.

However, the problem of compositional type inference becomes still more challenging when considering code reuse. This is illustrated by the example of class `Foo` introduced in Section 1.1.2:

```
class Foo {
    poly(b) {
        let a = new A();
        a.m1(true);
        a.m1(42);

        b.m2(true);
        b.m2(42);
    }
}
```

Even when structural types are used, it is not a trivial matter to infer a type for parameter `b` which is reused at two different types, and the success of any type inference approach depends to a large extent on the kind of polymorphism that is offered by the language.

For compositional type inference to work, we should try to find a type for parameter `b` that is general enough to represent all other possible types for `b`, or what is commonly referred to as a *principal type*. This is necessary because it is not known which other parts of the program will use method `poly`, and we need to ensure that no potential caller of this method will be prevented from using it due to the type of parameter `b` being unnecessarily narrow. When using parametric polymorphism, it unfortunately turns out that there exists no principal type for `b`. We can in fact find at least two valid yet unrelated types for `b` (using Java's `interface` syntax for describing types):

```
interface B1<Y> {
    <X> Y m2(X x);
}
interface B2 {
    <X> X m2(X x);
}
```

Note that in `B1`, type variable `X` is not within scope under the universal quantifier `<Y>`, and so it is not possible to instantiate `<Y>` to `X` in `B1` to yield `B2`. Even if we were instead to look for a finite set of types which between them represent all other types for `b`, such a finite set does not exist either because we can find an infinite set of unrelated types for `b`:

```
interface B2 { <X> X m2(X x); }
interface B3 { <X> Pair<X,X> m2(X x); }
interface B4 { <X> Pair<X,Pair<X,X>> m2(X x); }
interface B5 { <X> Pair<X,Pair<X,Pair<X,X>>> m2(X x); }
interface B6 ...
```

The problem can also be viewed from the perspective of the following smaller code fragment:

```
b.m2(true);
b.m2(42);
```

Whether or not this code fragment has a type depends very much on the nature of `b`. For example, in a context where `b` has type `int`, the above code fragment has no type. Whereas in a context where `b` has the type described by `interface B2`, the above code fragment has type `void` (statements in Java do not have values and are therefore of type `void`). The pair of the type and the context in which this type is assigned is together called a *typing*. For compositional type inference to work, it is necessary to find a typing for a given code fragment that is general enough to represent all other possible typings for that code fragment, which is commonly referred to as a *principal typing* [67]. Finding a principal typing means that it is not necessary to know how the term will be used at the moment the term is analysed because it is certain that a principal typing will be general enough to be instantiated into any other valid typing as required. However, by similar reasoning as before, there exists no principal typing for the above code fragment because we can construct an infinite set of unrelated typings, the type of each being `void` and the context of each assigning to `b` one of the types from the set {`B1`, `B2`, ...}.

When dealing with compositional type inference, the notion of typings is more useful than the notion of types, since the goal of compositional type inference is to compute both the type *and* the context for a given program fragment. That is, if a type inference algorithm cannot analyse a program subcomponent without being provided external context as input, then it cannot truly analyse that program subcomponent in isolation. As identified by Wells [67], the lack of principal typings in various systems based on parametric polymorphism is the key factor that makes type inference in those systems challenging. Even the Hindley/Milner type system, which restricts polymorphism to rank 1, lacks the principal typings property. It instead has the weaker *principal types* property which asserts that if the required context is supplied as input, then the type inference algorithm can find a principal type for a given program fragment. This allows type inference to be partially compositional, albeit at the cost of first-class polymorphism. For example, in functional languages such as Haskell, the application term `t1 t2` of function `t1` to argument `t2` (which in Java syntax corresponds to a method invocation `t1(t2)`) can be analysed compositionally by analysing the components `t1` and `t2`, in any order, and then combining the results. However, the let-term `let x = t1 in t2` (which corresponds to the Java syntax `T x = t1; t2;` where `T` is the required type annotation for the variable `x`, and `t2` is any Java expression that may refer to the value of variable `x`) must be analysed non-compositionally by first analysing `t1` and then using the result as context information about `x` while analysing `t2`.

Another indicator of the difficulty of performing type inference for systems based parametric polymorphism is the fact many such systems are now known to have *undecidable* type inference, meaning that any type inference algorithm is forced to be either incomplete or non-terminating. Such systems with undecidable type inference include System F [66], System F+η [65], System $F_\leq$ [47], System F_ω [60], as well as all rank k restrictions ($k > 2$) of System F [35] where *rank* [42] refers to the depth to which polymorphism is allowed on parameters. Specifically, rank 1 refers to a system where a parameter cannot be polymorphic, rank 2 allows a parameter to be polymorphic, rank 3 allows a parameter itself to have a parameter that is polymorphic, and so on.

The difficulties described here suggest a problem inherent in parametric polymorphism itself, and they motivate a consideration of alternative approaches to compositional type inference and first-class polymorphism. Such alternatives will be considered in Section 1.2.3.

1.2 Prior Work

Until now, there has been no type inference system that simultaneously supports object-orientation, first-class polymorphism and compositionality. However, there are systems that can support two out of the three items. This section reviews the prior work for each pair of items.

1.2.1 Object-Orientation & First-Class Polymorphism

Type inference for object-oriented languages with first-class polymorphism has been studied without compositional analysis.

The first and most actively researched approach initiated by Palsberg and Schwartzbach [46] is based on flow analysis. In this approach, types are sets of classes, and subtyping is set inclusion. The analysis begins by tracing method calls throughout the entire program, and generating constraints from these methods calls and also from uses of expressions within each method. Parametric polymorphism is supported by effectively splitting the flow graph for each different call of a method so that a method can be re-analysed with different types for different calls. The generated set of constraints is then solved using a fixed-point computation, and the program is considered typable if this constraint set is solvable. This approach was extended by Plevyak and Chien [49] to support more precise polymorphism in programs that contain deep polymorphic call chains by iteratively splitting the flow graph. A further improvement on Plevyak and Chien's algorithm was the Agesen's Cartesian Product Algorithm [3] which increases efficiency by replacing the iterative splitting procedure with an immediate case analysis of all of the combinations of types that are possible for each call site. All of the above variations, however, require the use of a whole-program analysis, and cannot support separate compilation.

A more recent and quite different approach is Plumicke's type inference system for Java [52, 51]. This system supports first-class parametric polymorphism and also makes use of intersection types in order to compute a principal type for any method. In this approach, the type for a polymorphic parameter is inferred by performing a whole-program search for a set of classes that, after having each been analysed themselves, are found to be compatible with the usage of the parameter. Thus, despite having the principal types property, the type inference procedure is prevented from being compositional due to the required whole-program search.

1.2.2 Object-Orientation & Compositionality

Type inference for object-orientation with a limited form of compositional analysis has been studied using variations of Algorithm $\mathcal{W}$ for the Hindley/Milner type system. With the exception of let-terms, Algorithm $\mathcal{W}$ is compositional. The most general type of a function λ`x.t` taking parameter `x` and returning the value of its body `t` can be analysed by first analysing its body `t` in isolation. The most general type of a function application `s t` can be analysed by first analysing the function `s` and the argument `t` in isolation. The Hindley/Milner type system uses parametric polymorphism so that if the most general types of the function `s` and the argument `t` contain type parameters, a type unification algorithm can be applied to make necessary substitutions for these type parameters so that the type of the parameter of `s` matches the type of the argument `t`. The only kind of term that Algorithm $\mathcal{W}$ does not analyse compositionally is the let-term `let x = s in t`. To infer the type of `x` from its usage within `t` alone would be equivalent to the problem of inferring the type of a parameter `x` from its usage within the function body. As we saw in the previous section, it is especially difficult in parametrically polymorphic type systems to infer the type of such a variable `x` if `x` is used polymorphically. It is for this reason that Algorithm $\mathcal{W}$ analyses let-terms non-compositionally and uses a prior analysis of the term `s` to provide a hint as to the polymorphic nature of the variable `x`. Despite its non-compositional analysis of let-terms, Algorithm $\mathcal{W}$ is compositional to an extent and that is why it is included in this section.

Algorithm $\mathcal{W}$ has been extended to support objects in various ways. Wand produced a variant of Algorithm $\mathcal{W}$ using his extensible records [62, 64] which elegantly encode objects and inheritance by allowing the extension of existing records with additional fields. Wand's original type inference algorithm was incomplete, but complete versions were developed by Jategaonkar and Mitchell [28], and also by Remy [55]. More recently, Boudol developed a type inference system for extensible records that addresses safety issues with integrating recursion and state [9]. Variations of extensible records have also been developed to include first-class record labels [22, 58] which allow the expression of functions that abstract over labels.

A variation on the extensible record idea also explored by Wand [63] and others [25, 54, 44] is *record concatenation* in which two records, each with multiple fields, can be joined to form a new record. Record concatenation has been studied as a way to encode *multiple inheritance*, whereby fields and methods from multiple superclasses can be inherited by a single subclass. An interesting result from Remy [54] demonstrates that any language with record extension possesses record concatenation for free by a translation from the latter into the former. Makholm and Wells [44] proved that type inference for record concatenation is NP-complete, and also presented a restricted form of record concatenation with $O(n^2)$ complexity, or $O(n)$ if bounded in the number of field labels. The simply typed variation of their system has principal typings, and can be extended with let-polymorphism having principal types.

1.2.3 First-Class Polymorphism & Compositionality

The desire to have first-class polymorphism is often discussed in relation to the Hindley/Milner system where first-class polymorphism has been restricted to ease type inference. Although the Hindley/Milner system uses parametric polymorphism, the desire to regain first-class polymorphism should not necessarily be tied to parametric polymorphism. Rather, any form of polymorphism that can support the same kinds of code reuse as parametric polymorphism can be considered as a possible means to this end. In this section, we consider both parametric polymorphism and intersection type polymorphism, and investigate the extent to which they support first-class polymorphism and compositional analysis.

Parametric polymorphism

In Section 1.1.2 and Section 1.1.3, we saw that type inference systems based on parametric polymorphism tend to have difficulty dealing with first-class polymorphism and compositional analysis. Consequently, the prior work in this section can only describe systems that support first-class polymorphism and compositionality partially.

We may consider Algorithm $\mathcal{W}$ to be compositional to an "extent" since it does generally analyse programs from the bottom up, with only the exception of let-terms. To work around the difficulties with first-class polymorphism, there have been efforts to extend Algorithm $\mathcal{W}$ to create "partial" type inference algorithms [39, 41, 31] that will infer types for all terms contained within the Hindley/Milner system, but will fall back on programmer-supplied type annotations when first-class polymorphism is needed.

Currently, the most powerful type inference systems using parametric polymorphism are based on the rank 2 fragment of System F, which contains the typing power of the Hindley/Milner system. However, these type inference systems are not compositional due to the lack of a principal typings property. The first such algorithm was developed by Kfoury and Wells [35] which reduces the type inference problem to an acyclic semi-unification problem. The second such algorithm was developed by Lushman [43] which improves efficiency by instead translating into the R-acyclic semi-unification problem. In an attempt to address the lack of principal typings in System F, Lushman's algorithm also introduces an extended type syntax that can generalise over more types than raw System F types can.

However, the system is not complete and may sometimes infer a type that is not general. In such cases, the programmer may use an annotation to express the desired type.

Beyond rank 2, System F is undecidable, both for unrestricted rank [66] and also for all rank k restrictions for $k > 2$ [35], and no known type inference algorithm exists for these systems.

Intersection type polymorphism

A promising alternative to parametric polymorphism that is able to handle at least the same kinds of code reuse is *intersection type polymorphism*, which was first developed by Coppo and Dezani [19], and at around the same time by Pottinger [53]. In contrast to parametrically polymorphic types, an intersection type does not use type parameters to generalise the set of all possible instances. Instead, an intersection type explicitly enumerates the instance types that are intended to be used within a given program. For example, if a polymorphic function is intended to be used at three different types T_1, T_2 and T_3, then the intersection type to express this polymorphism is $T_1 \sqcap T_2 \sqcap T_3$.

The typing power of these two kinds of polymorphism can be compared by considering the set of terms of the λ-calculus that can be typed in each case. Systems based solely on parametric polymorphism type at most the set of strongly normalising λ-terms [24], but cannot type all such terms [60]. On the other hand, certain intersection type systems will type exactly the set of strongly normalising λ-terms (see [23] for a discussion). By this comparison, intersection type polymorphism allows strictly more programs to be typed than parametric polymorphism.

Regarding first-class polymorphism, type inference is still undecidable for systems with unrestricted intersection types [53]. However, more promising is that type inference has been found decidable for all finite-rank restrictions of some systems of intersection types including System I [36, 37] as well as Boudol and Zimmer's system based on the Klop calculus [10]. This makes it possible to create an always-terminating type inference algorithm by setting a maximum height on the rank of polymorphism.

However, it is not yet clear how such decidable forms of type inference would be used in practice since someone must still decide on a particular rank limit. For example, if the programmer decides to run the analysis with a maximum rank of 10 and type inference fails at that rank limit, it is not clear whether the programmer should accept that the program cannot be typed at this chosen rank limit, or should instead try the analysis again at a higher rank limit. Exploiting the power of intersection types may in practice require a shift in thinking where the programmer is expected to make the decision on what level of precision is required for the analysis. Carlier and Wells [16] suggest a view along these lines when the write: “In fact, there is no reason why one must use the full power of intersection types; for example, one can choose to use principal typings of the rank-k restriction. In the long run, if one wants to use intersection types, it seems best to view them as a flexible framework for typing with a choice of a wide variety of different levels of precision.”

The main relevance of intersection type systems to this thesis is that such systems tend to have the principal typings property, which makes these systems suitable for compositional analysis. A number of different type inference algorithms have been developed to find these principal typings:

- The earliest algorithms by Coppo and Dezani [20] and Ronchi della Rocca and Venneri [56] involved first computing a normal form for a term and then constructing a principal typing for the normal form. An instance is derived from a principal typing via a set of transformation operations that includes the usual substitution of types for type variables, as well as an operation called *expansion* which is unique to intersection type polymorphism, which expands a type T into an intersection of type instances $T1 \sqcap T2$. One problem with this approach is that if given a non-terminating program, the normalisation procedure will also be non-terminating, and there is no simple way to define a subset of the system for which the algorithm always terminates. Another issue is that by deriving an instance from a principal typing via a *set* of different operations rather than a single operation (commonly, a substitution alone), it becomes more difficult to make this operation composable, and to reason about in formal proofs. One more problem is that, without any markers in the type syntax, it is difficult to identify which parts of a type may be subject to the expansion operation (see Chapter 2, Section 2.3.1 for a more detailed review of this original approach to expansions).
- Ronchi della Rocca developed the first approach based on type unification [57]. In this approach, a program is analysed to produce a set of type constraints such that each time a function is applied to an argument, the type of the argument must match the type of the function parameter. Finally, the unification algorithm is invoked to solve this set of constraints. Since the constraint solving is done purely at the type level, this opens the possibility to create an always terminating algorithm

by specifying a maximum rank on polymorphic types. Like the previous approach, however, this approach still uses multiple operations to derive instance typings from principal typings, and uses a complicated expansion operation.

- Kfoury introduced a new unification-based approach to type inference called β-unification [34]. It introduces *expansion variables* whereby the traditional expansion operation now becomes a case of substituting an expansion for an expansion variable in the same way that a type can be substituted for a type variable. Expansion variables make reasoning about expansions easier, and also allow the different operations of type substitution and expansion to be treated in a more uniform way. Kfoury and Wells developed this approach further in System I [36, 37] and showed that every finite-rank restriction of System I has both principal typings and decidable type inference. The type machinery for expansions and expansion variables was later greatly simplified and refined by Carlier, Polakow, Wells and Kfoury in System E [17], which allowed further progress to be made in the design of new type inference algorithms. The first System E type inference algorithm [15] aimed to show that the process of β-unification for a given program has a one-to-one correspondence with β-reduction of the same program. The second System E type inference algorithm [6] makes this correspondence much clearer, and is also parameterised by a choice of evaluation strategy between call-by-value and call-by-need (See Chapter 3, Section 2.4.1 for a more detailed review of this approach).
- Boudol and Zimmer also developed a type inference procedure that finds principal typings using a modified Klop calculus [10]. They present a result similar to that of System I in that their procedure corresponds step for step to reduction in the calculus. The expansion operation is performed by keeping track of the set of type variables that need to be duplicated whenever an argument is used by a function at an intersection of multiple types. However, this set of type variables is not embedded in the type and must be found by analysing the program, which means that the expansion operation cannot work in the realm of types alone. Also, intersection types are permitted to occur only to the left of an arrow. This simplification meets the needs of the λ-calculus and Klop-calculus, but may present difficulties when extending the system with additional term forms (for example, our treatment of extensible records will require the introduction of intersection types to the right of an arrow).

Comparing the approaches to type inference based on parametric polymorphism and intersection types, intersection types clearly have an advantage in relation to compositional analysis due to the principal typings property. Intersection type systems are also beginning to show promise in supporting decidable type inference beyond rank 2 using rank-stratified type systems (System I and Boudol and Zimmer's system). While it may not yet be clear how programmers ought best to exploit these rank-stratified type systems, the overall benefits of intersection type polymorphism over parametric polymorphism suggest that intersection types are a more promising avenue for future research.

1.3 Extensible Records in the System E Framework

This dissertation introduces a new approach to object-oriented type inference that results from integrating extensible records into the System E framework. In the resulting system, objects are represented by extensible records, while first-class polymorphism and compositional type inference are provided by System E.

The fact that this approach to object-oriented type inference supports both first-class polymorphism and compositional type inference is significant because previous type inference systems, both for extensible records in particular, and also for object-orientation in general, have supported only one or the other of these two important features, but never both of them simultaneously. Our system also makes a significant contribution to the work on System E, since it extends System E for the first time beyond the terms of the pure λ-calculus, in this case, to work with extensible record data structures.

This section will give a brief overview of extensible records, System E and the system that results from combining them, called System E^{vcr}.

1.3.1 Extensible Records

A *record* is an unordered collection of labelled values called *fields* where access to individual fields is provided by a *field selection* operator, often indicated by a dot, ".". For example, if we let `john` denote the record { `name="John Smith", age=31` }, then the field selection `john.age` evaluates to 31 while the field selection `john.name` evaluates to `"John Smith"`. Records are a useful construct in a calculus of objects, since an object can be represented simply as a record, where both the methods and fields of the object can be represented by fields of the record.

An *extensible record* is essentially a special kind of record that can be defined as an extension of an existing record. For example, given some record r, then the extensible record { `name = "John",` r } is an extension of r, containing all of the fields of r plus the additional field labelled `name` of value `"John"`, with the additional field potentially overriding any existing field in r with the same name. A record of multiple fields is built from successive record extensions starting with the empty record {}. For example, we may define a record `john` as the following extension:

```
let john   =   { name="John Smith", { age=31, {} } }
```

Field selection is performed by scanning the fields from left to right until the first field with the desired label is found, which is also the mechanism by which field overriding words. For example, selecting the field `age` in the extensible record { `age=41, john` } will result in `41` and not `31`, since the field `age=41` will be encountered first in the left-to-right scanning order. The ability to define one record as an extension of another record while potentially overriding some of its fields is of particular interest to object-orientation since it allows for an elegant encoding of object-oriented inheritance whereby one class of objects may be defined as an extension of another class of objects, potentially overriding some of its methods.

When an extensible record has the empty record {} at its base, it is no more expressive than an ordinary record. The real power of extensible records comes from the ability to define a record with a variable at its base. For example, the following function takes a parameter x representing an unknown record, and returns an extension of x with an additional field labelled `id`:

$$\lambda x.\ \ \{\ \texttt{id=3,}\ x\ \}$$

This is the basic mechanism by which object-oriented inheritance can be encoded into an extensible record calculus. Here, variable x could represent an instance of a superclass that is being extended, in which case variable x would essentially serve the same purpose as Java's `super` keyword. Various encodings of OO into an extensible record calculus will be presented in Section 5.4.2 in Chapter 5.

Typing with row variables

When an extensible record has the empty record {} at its base, it is typed by enumerating the types of the record's fields. For example, the extensible record `{a=3,{b=true,{}}}` has type `{a:Int,b:Bool}`.

However, when an extensible record has a term variable at its base, some means is needed to indicate the types of the unknown fields represented by the term variable. This typing challenge has been traditionally solved using *row variables*. Using this approach, the example function given in the previous section can be typed

$$(1) \qquad \lambda x.\ \ \{\ \texttt{id=3,}\ x\ \} : \{\rho\} \rightarrow \{\texttt{id:Int}, \rho\}$$

where the type $\{\rho\} \rightarrow \{\texttt{id:Int}, \rho\}$ can be read as describing a function that takes as its parameter a record of type $\{\rho\}$ and returns a record of type $\{\texttt{id:Int}, \rho\}$. Here, ρ is a *row variable* representing the types of the unknown fields of the parameter x, and so the resulting record has a type containing at least the fields specified by ρ, plus the additional field `id` of type `Int`.

A row variable is similar to a type variable except that rather than being subject to type substitutions, it is subject to *row* substitutions where a row is a set of pairs of field labels and types that may appear within a record type. An important restriction on row variables is that they may not be used to introduce duplicate field types into a record type. In the above function, ρ may not be used to introduce another `id` field into the record type, but is free to introduce any number of other fields besides `id`.

There have been various treatments of row variables in the literature, and each takes a different approach to example (1) above:

- In Wand's original system, the row variable ρ in example (1) does not itself restrict what field labels may be introduced, and multiple occurrences of a field label may be introduced into a record type. However, when duplicate labels are present, only the leftmost occurrence of a label is effective and all others are ignored. When considering only the effective fields of a record type, this approach prevents effective duplicate fields from being introduced, as desired, although it fails to yield principal types, and it also leads to poor efficiency in the type inference algorithm.

- Jategaonkar and Mitchell [28] defined the substitution operation on record types as a partial function, succeeding only when the set of labels already in the record are disjoint from the set of labels being substituted for the row variable. That is, a substitution σ applied to the type of example (1) is defined only if $\sigma\rho$ does not contain any field labelled `id`. A consequence of this approach is that any substitution for ρ must be aware of the entire record type in which ρ appears.

- Rémy [55] proposed encoding record types in a way that reflects both positive and negative information about which fields are present and which are absent. For example, the equivalent of type { `name:String, age:Int` } in Rémy's system is $\Pi(\texttt{name} : \mathit{pre}(\texttt{String}), \texttt{age} : \mathit{pre}(\texttt{Int}), \texttt{a} : \mathit{abs}, \texttt{b} : \mathit{abs}, \ldots)$ where `a`,`b`,... represent every other possible field label from the complete set of field labels, excluding `name` and `age`. This is abbreviated to just `abs`, and so the record type can be expressed as: $\Pi(\texttt{name} : \mathit{pre}(\texttt{String}), \texttt{age} : \mathit{pre}(\texttt{Int}), \mathit{abs})$. In this system, the function in example (1) above can be given type $\Pi(\texttt{id} : \mathit{abs}, \rho) \rightarrow \Pi(\texttt{id} : \mathit{pre}(\texttt{Int}), \rho)$ where any type substituted for ρ must have $\texttt{id} : \mathit{abs}$. However, it is interesting to note that although Rémy's system does keep track of positive and negative information on record types, the negative information of $\texttt{id} : \mathit{abs}$ is not directly associated with the variable ρ itself. Therefore, any substitution for ρ must still be aware of all of the fields in the entire record type, as in the approach of Jategaonkar and Mitchell, in order to prevent the instantiation of ambiguous types with multiple fields labelled `id`.

- Cardelli and Mitchell [12] developed a type system in which types again express both positive and negative information about present and absent fields, but where a subtype relation is also provided such that a type with more positive and negative information is considered a subtype of a type with less positive and negative information. For example, the record type $\{\}$ is the type of all records, while the subtype $\{\texttt{a} : \texttt{Int}\}\backslash\texttt{b}$ is the type of all records which contain a field labelled `a` of type `Int` and also lack any field labelled `b`. In this system, the function in example (1) above can be assigned the type $\forall(\rho <: \{\}\backslash\texttt{id}).\rho \rightarrow \{\rho|\texttt{id} : \texttt{Int}\}$ which asserts that, given any ρ that is a subtype of the type of all records lacking label `id`, the function will take a record of type ρ and return a record of type ρ modified with the addition of a field labelled `id` of type `Int`. Since negative information is associated directly with the parameter type ρ, any substitution for ρ can be performed on ρ alone and without any knowledge of the record type in which ρ is contained.

This is in contrast to Jategaonkar and Mitchell's solution and also Rémy solution in which a substitution cannot be performed on a row variable without being aware of the entire record type in which the row variable is contained.

- In an approach developed by Harper and Pierce [25], and later refined by Gaster and Jones [22], the idea of *constrained quantification* of row variables is used whereby negative information is associated directly with row variables in the form of a predicate rather than with a subtype relation. For example, the predicate $T\backslash\texttt{id}$ asserts that record type T lacks any field labelled `id`. Using such a predicate, the function in example (1) above can be assigned the type $\forall(\rho\backslash\texttt{id}).\{\rho\} \rightarrow \{\texttt{id:Int}, \rho\}$ which asserts that for any row ρ that lacks a field labelled `id`, the function will take a record of type $\{\rho\}$ and return a record of type $\{\texttt{id:Int}, \rho\}$. Like Cardelli and Mitchell's solution, negative information is associated directly with the row variable ρ, any substitution for ρ can be performed on ρ alone without any knowledge of the record type in which ρ is contained.

It is interesting to note that extensible records and row variables were originally developed by Wand in the context of object-oriented type inference. Ironically, however, Wand's original type inference system, as well as all subsequent type inference systems for extensible records, were developed in the Hindley/Milner style, severely hindering their applicability to object-orientation, a limitation discussed at length in Section 1.1. It is for this reason that we wish to revisit extensible records within a more modern type-theoretical framework that will preserve the modelling and engineering benefits of object-orientation.

1.3.2 System E

This section will give a brief overview of System E, leaving a more detailed review for the next chapter. System E is a type system for the λ-calulus that supports first-class polymorphism via intersection types and type inference via *expansion variables.* In many ways, System E is a typical intersection type system. For example, if the identity function is to be used within a program on both integers and strings, it may be given a polymorphic intersection type such as $\texttt{Int} \rightarrow \texttt{Int} \sqcap \texttt{String} \rightarrow \texttt{String}$.

What distinguishes System E from other intersection type systems, however, is that it uses expansion variables as a means to express principal typings more clearly and more explicitly than has been done in previous systems. In System E, the identity function has $e\ (\alpha \to \alpha)$ as a principal type, where α is a type variable and e is an expansion variable. When it is discovered how the identity function is used by the program during type inference, a so-called *expansion* may be substituted for e which may then act upon $(\alpha \to \alpha)$ to give, for example, the intersection type $\texttt{Int} \to \texttt{Int} \sqcap \texttt{String} \to \texttt{String}$. An expansion can introduce any number of intersection components corresponding to the various ways in which the function is used, with potentially different types substituted for α in each case. This is true even if the function is used zero times, and thus it is also possible to substitute for e an expansion that results in an intersection of zero types. In System E, the intersection of zero types (the nullary case of $\sqcap$) is written ω.

Current type inference algorithms for System E follow a procedure that has a one-to-one correspondence with the evaluation of the program being analysed, a consequence being that the type inference algorithm will terminate exactly whenever evaluation of the analysed program will terminate. For example, such algorithms will run forever if analysing a program whose evaluation would run forever. This behaviour is different from System E's predecessor, System I, which is able to guarantee termination by restricting polymorphism to some specified finite rank. However, it is stated as ongoing future work in the papers on System E type inference [15, 6] to develop type inference algorithms that perform any specified amount of partial evaluation followed by traditional monovariant analysis. Then, in the case of programs whose evaluation exceeds this specified amount, the type inference algorithm would terminate with a type of less precision. Since that is stated as ongoing future work in those papers, we do not pursue the idea in this dissertation.

1.3.3 System $\mathrm{E}^{\mathrm{vcr}}$

This dissertation contributes a new type system, called System $\mathrm{E}^{\mathrm{vcr}}$, that results from integrating extensible records with System E. Combining these two technologies results in a system capable of simultaneously supporting object-orientation, first-class polymorphism and compositional analysis, where previous type inference systems have succeeded only in supporting two out of these three features. This is of particular significance to object-oriented type inference since first-class polymorphism and compositional analysis are crucial to preserving the modelling and engineering benefits of object-orientation.

In this section, we give a brief overview of System $\mathrm{E}^{\mathrm{vcr}}$.

One of the biggest challenges to integrating extensible records into the System E framework is that much of the type machinery of System E was designed around a calculus of functions. It is not obvious, for instance, how expansion variables make sense in the context of extensible records. In System $\mathrm{E}^{\mathrm{vcr}}$, this issue is sidestepped by treating extensible records *as functions* from labels to values, and by treating field selection as the application of an extensible record to a label. To highlight the functional nature of extensible records in our system, extensible records are defined using syntax resembling that of pattern matching functions where labels are the only patterns. The $\rightarrow$ symbol is used to map a particular record label to a particular value, and the $\Cap$ symbol (here used as term operator rather than a type operator) is used to join each case of the function. For example, we may define an extensible record as follows:

$$\texttt{let john = occupation} \rightarrow \texttt{"Student"} \Cap \texttt{name} \rightarrow \texttt{"John"} \Cap \texttt{age} \rightarrow 37$$

Then, the application `john name` evaluates to `"John"` while the application `john age` evaluates to `37`. By treating extensible records as functions, it is possible to reuse all of System E's function-oriented type machinery, almost as is. In particular, it is possible to represent record types as intersections of function types in the manner of Kopylov [38], such that the extensible record `john` can be typed as follows:

$$\begin{array}{rl} \textsf{Record:} & \texttt{occupation} \rightarrow \texttt{"Student"} \Cap \texttt{name} \rightarrow \texttt{"John"} \Cap \texttt{age} \rightarrow 37 \\ \textsf{Type:} & \texttt{occupation} \rightarrow \texttt{String} \quad \Cap \texttt{name} \rightarrow \texttt{String} \Cap \texttt{age} \rightarrow \texttt{Int} \end{array}$$

Using the intersection type constructor $\Cap$, the above type asserts that this record has type `occupation` $\rightarrow$ `String` *and* type `name` $\rightarrow$ `String` *and* type `age` $\rightarrow$ `Int`. Each of these component types is a function type taking a particular field label to a particular type of result. For example, the function type `occupation` $\rightarrow$ `String` asserts that the given record, when applied to the label `occupation`, will return a `String`.

Since the above intersection type asserts that the record has all three function types, it is also possible to derive a number of different intersection types for the same record:

$$
\begin{array}{ll}
(1) & \texttt{occupation} \rightarrow \texttt{String} \sqcap \texttt{name} \rightarrow \texttt{String} \sqcap \texttt{age} \rightarrow \texttt{Int} \\
(2) & \texttt{occupation} \rightarrow \texttt{String} \sqcap \texttt{name} \rightarrow \texttt{String} \\
(3) & \texttt{occupation} \rightarrow \texttt{String}
\end{array}
$$

This results from recombining the component function types in different ways, and achieves very much the same goal as subtype polymorphism where an object of a certain type can also be considered an object of a type with strictly fewer fields. As long as these intersection types are automatically inferred for the programmer, we argue that intersection type polymorphism can effectively serve the purpose of subtype polymorphism and that we need not complicate the system by adding more kinds of polymorphism. Similarly, intersection types can serve the purpose of parametric polymorphism because they can type strictly more terms than are typable using parametric polymorphism (see Section 1.2.3). Finally, intersection types can also serve the purpose of inheritance polymorphism which, as observed in [29], can really be viewed as a special case of parametric polymorphism. It is a strength of our system that only a single brand of polymorphism is required to support 3 different styles of polymorphic programming.

With record types expressed as intersections of function types, the application of expansion variables becomes straightforward. Just as expansion variables were applicable to function types in the original System E, so too are they applicable to function types as they appear in record types in System $\mathrm{E}^{\mathrm{vcr}}$, as shown below:

$$e_1\ (\texttt{occupation} \rightarrow \texttt{String}) \sqcap e_2\ (\texttt{name} \rightarrow \texttt{String}) \sqcap e_3\ (\texttt{age} \rightarrow \texttt{Int})$$

During type inference, if it is discovered that the record `john` is used only for its `name` and `age` fields, an expansion may be substituted for e_1 to reflect that it is used zero times, and expansions may be substituted for e_2 and e_3 to reflect that they are each used once, resulting in the type:

$$\texttt{name} \rightarrow \texttt{String} \Cap \texttt{age} \rightarrow \texttt{Int}$$

One peculiar consequence of using ordinary function types to describe records is that the type syntax is flexible enough to describe even impossible records, such as $\texttt{name} \rightarrow \texttt{String} \Cap \texttt{name} \rightarrow \texttt{Int}$. Since no record can be defined in our language whose `name` field is simultaneously of type `String` and `Int`, this would merely be treated as an empty type, and our typing derivation rules (Definition 3.24) will not permit any record to be assigned this type. Not even the extensible record $\texttt{name} \rightarrow \texttt{"Joe"} \Cap \texttt{name} \rightarrow \texttt{42}$ could be assigned this type, since $\texttt{name} \rightarrow \texttt{"Joe"}$ should be treated as an extension of $\texttt{name} \rightarrow \texttt{42}$, and the former field will override the latter field (see Section 1.3.1).

In traditional type systems for extensible records, the most complicated typing issue is that of typing records that are extensions of unknown records. Given the extensible record (in traditional syntax) of $\{\ \texttt{occupation = "Student",}\ r\ \}$ where r is a variable representing an unknown record, Wand proposed the use of a row variable ρ to describe the type of r so that the entire record might be given the type $\{\ \texttt{occupation : String,}\ \rho\ \}$. As discussed in Section 1.3.1, any substitution for the row variable ρ must be careful not to introduce an additional field called `occupation` into the record type, otherwise the resulting record type would have two fields with the same name.

System $\mathrm{E}^{\mathrm{vcr}}$ does not have row variables, but instead introduces a novel, more fine-grained primitive called a *constrained simple type variable*. From this together with other primitives, it is possible in System $\mathrm{E}^{\mathrm{vcr}}$ to construct a type that serves the same purpose as row variable. For example, the same extensible record can be expressed and typed in System $\mathrm{E}^{\mathrm{vcr}}$ as follows:

$$\begin{array}{rl} \mathsf{Record:} & \texttt{occupation} \rightarrow \texttt{"Student"} \Cap r \\ \mathsf{Type:} & e_1\ (\texttt{occupation} \rightarrow \texttt{String}) \Cap e_2\ (\alpha[\texttt{occupation}] \rightarrow e_3\ \beta) \end{array}$$

Here, the type variable $\alpha[\texttt{occupation}]$ is constrained such that only labels not equal to `occupation` may be substituted for it. Then, $e_2\ (\alpha[\texttt{occupation}] \rightarrow e_3\ \beta)$ effectively acts as a row variable because it can be expanded, via e_2, into an intersection of field types whose labels must not be equal to `occupation`.

Type inference itself is performed by an algorithm called $\mathcal{I}$. This algorithm departs from previous type inference algorithms for System E in that it does not aim to have a direct correspondence with program evaluation. Rather, it follows the compositional style of Algorithm $\mathcal{W}$, analysing each program subterm before working its way outward to the root term of the program. Algorithm $\mathcal{I}$ is parameterised by the choice of a unification algorithm that is tasked with solving type constraints generated by $\mathcal{I}$. However, since $\mathcal{I}$ does not try to simulate program evaluation, the standard β-unification algorithm used in previous type inference algorithms for System E cannot be used here. In particular, β-unification assumes that all functions in the program being analysed are λ-abstractions of the λ-calculus, and this assumption breaks down when we introduce extensible records posing as functions. Instead, $\mathcal{I}$ is designed with *covering unification* in mind, which is particular kind of unification approach that attempts to find a most general solution (or set of solutions) to the constraints given to it, and is exemplified by the opus unification algorithm [7]. At a cost to efficiency, opus is able to find solutions to arbitrary type constraints based solely on the types alone and making no assumptions about the term language or semantics of the underlying calculus. However, opus' loss of efficiency is not minor (see Section 3.6 and Section 5.5) and the algorithm is too inefficient to be used in practice. The loss of efficiency comes from the fact that, since unification in the presence of expansion variables often does not have a *single* most-general solution, in order to find the most general *set* of solutions for a given constraint, opus must branch the search path at potentially each step of unification leading to a combinatorial explosion of alternative solutions.

As an efficient alternative to opus, we also present the new unification algorithm opusβ which derives from opus, but sacrifices the property of covering unifier sets for a gain in efficiency by borrowing ideas from β-unification. Algorithm $\mathcal{I}$ and opusβ have been implemented and are demonstrated successfully on the object-oriented examples presented in this introduction.

1.4 Outline of the Dissertation

The following chapters introduce our system incrementally. An overview of each chapter is given below.

- **Chapter 2** reviews System E and the fundamental concepts of intersection types and expansion variables, and explains how they are used in type inference and type unification.

- **Chapter 3** formally defines our initial calculus called System E^{v} containing only the terms of the pure λ-calculus. System E^{v} differs from System E by the introduction of a value restriction which is necessary to support the typesafe integration of constants and extensible records.
- **Chapter 4** defines an extension of System E^{v} called System E^{vc} that adds the new syntactic category of constants. In this category, data values (e.g. integers), operations (e.g. +, -) and importantly record labels can be included. System E^{vc} depends on the value restriction of System E^{v} to ensure type safety.
- **Chapter 5** defines an extension of System E^{vc} called System $\mathrm{E}^{\mathrm{vcr}}$ that adds extensible records. System $\mathrm{E}^{\mathrm{vcr}}$ depends on the value restriction of System E^{v} to ensure type safety, and depends on the constants of System E^{vc} to provide record labels.
- **Chapter 6** presents conclusions and future work.
- **Appendix A** contains additional proofs related to our type system and type inference algorithm that were not included in the main body of the dissertation.
- **Appendix B** shows the output of our implementation on all examples from the System E Inference Report [13].
- **Appendix C** includes the original opus algorithm [7].

1.5 How to read this dissertation

To help the reader understand the range of typing issues presented in this dissertation, System $\mathrm{E}^{\mathrm{vcr}}$ will be presented incrementally with each increment presented in its own chapter: the first increment System E^{v} presented in Chapter 3 will focus on the pure λ-calculus and also lay the groundwork for studying extended term sets; the second increment System E^{vc} presented in Chapter 4 will extend System E^{v} to support constants; and the third increment System $\mathrm{E}^{\mathrm{vcr}}$ presented in Chapter 5 will extend System E^{vc} to support extensible records with record labels represented by constants.

Although this incremental presentation approach has the benefit of brevity and allows the material to be organised by topic, it may also potentially pose some challenges for the reader since each new increment will present only new or amended definitions, and the entire system will therefore be spread across three chapters. To help guide the reader through this dissertation, this section will outline the conventions used for incremental presentation. For each increment:

- If a definition is amended, the definition will be restated in full with the amended parts underlined.
- If a lemma is needs to be reproved due to amended definitions, the lemma will be restated.
- Theorems will always be restated.
- Proofs of restated lemmas and theorems will often be brief focusing on new cases.

As an example of an incremental definition, this dissertation will present three increments of the definition of term syntax across three chapters with the following labels:

- **In Chapter 3**: Definition 3.1 (Terms for E^{v})
- **In Chapter 4**: Amendment to Definition 3.1 (Terms for E^{vc})
- **In Chapter 5**: Amendment to Definition 3.1 (Terms for $\mathrm{E}^{\mathrm{vcr}}$)

Note that the title of each increment of the definition has a suffix such as "- for $\mathrm{E}^{\mathrm{vcr}}$" which allows the three increments of the definition to be referenced distinctly in discussions that compare one increment to another. When the context is clear, however, the short form reference "Definition 3.1" may also commonly be used to refer to the "current" increment of the definition, particularly in the body of proofs. This is important given that proofs of lemmas and theorems in one increment often carry across to the next increment unchanged, and in such cases, any references made to Definition 3.1 should be interpreted in the context of the current increment. Different increments of a lemma or theorem follow similar naming and referencing conventions except that the prefix "Restatement of -" is used instead of "Amendment to -". For example, "Restatement of Theorem 3.30 (Subject reduction for E^{vc})".

Since each increment focuses only on what is new, it is sometimes important to know how to find definitions, lemmas and theorems from previous increments. This can be done via the index of definitions and theorems, located immediately after the table of contents, which lists the page numbers of all definitions, lemmas, theorems and their amendments or restatements by increment and by number. However, this index is often not needed since each amended definition will restate in full the original definition with amended parts underlined, while each proof of a restated lemma or theorem that focuses only on new cases will make an in-text reference to the page number of the lemma or theorem from the previous increment whose proof is being extended.

To give an example of how to read such a proof, consider Theorem 3.30 (Subject reduction for E^{v}) whose page number can be found in index of definitions and theorems (after the table of contents). The proof of this theorem can be read without reference to any previous increment because it is the first increment. However, this theorem must be reproved for System E^{vc} to take into account new terms and reduction rules related to constants, and so in Chapter 4, this result is restated and reproved under the label "Restatement of Theorem 3.30 (Subject reduction for E^{vc})" (which can also be found in the index of definitions and theorems). When reading the proof of the restated theorem for System E^{vc}, only one additional case for reducing applications of constants is shown, and the cases for all other reduction rules are stated as remaining unchanged. In this case, if the reader wishes to review the other cases, a page number reference is given in the text to the original Theorem 3.30 (Subject reduction for E^{v}) where a proof of all other cases can be found.

As a slightly more complicated example, when reading the proof of "Restatement of Theorem 3.30 (Subject reduction for $\mathrm{E}^{\mathrm{vcr}}$)", only the two additional cases for reducing applications of extensible records are shown, and all other cases are stated as remaining unchanged. In this case, if the reader wishes to review the other cases, the reader is given the page reference to "Restatement of Theorem 3.30 (Subject reduction for E^{vc})" which shows the proof for the cases relating to constants, and from there a page reference is given to "Theorem 3.30 (Subject reduction for E^{v})" which shows the proof for all of the remaining cases related to pure λ-terms.

While this is perhaps an unfortunate consequence of presenting the work incrementally, readers who are interested only in the new cases of the proofs for each chapter will be able to read those cases without referring back to already established cases presented in previous chapters, and it is hoped that this style of presentation will allow the reader to more easily take in a set of complex topics by focusing on one topic at a time.

Chapter 2

Review of System E

System E is a type system for the λ-calculus that provides polymorphism via intersection types and type inference via expansion variables. This chapter reviews the key concepts and ideas of System E that will be used to formulate our own system starting from the next chapter.

Several formulations of System E exist. The original formulation [17] is parameterised on its subtyping relation, and defines special proof terms called *skeletons* which are concise encodings of typing derivations. Since System E permits a reflexive subtyping relation, in which case no typing power is added, and since skeletons do not add typing power, both of these features have sometimes been omitted from some other formulations of System E. For example, all existing type inference algorithms for System E exclude subtyping from analysis by choosing a reflexive subtyping rule [15, 6], while some other formulations of System E have been presented without skeletons [7, 8]. Fortunately, none of these features is necessary to support object-orientation, first-class polymorphism or compositional analysis, and so the formulation of System E presented in this chapter will be simplified accordingly.

This chapter is organised as follows. Section 2.1 introduces the the λ-calculus on which System E is built. Section 2.2 introduces the type language and typing rules of the type system. Section 2.3 defines the expansion and substitution operations of System E which are necessary for performing type inference. Section 2.4 discusses the different approaches to type inference and type unification within System E. Finally, Section 2.5 summarises the main topics of this chapter.

2.1 λ-Calculus

The λ-calculus [18] is a model of computation frequently used to study typing issues in programming languages. The λ-calculus is Turing-complete, yet is remarkably small, and maps closely to concepts found in real programming languages. Programs expressed in the λ-calculus are constructed from *terms*, sometimes called λ-terms, of which there are only three kinds. These terms are defined by the following grammar:

$$s, t, u ::= x \mid \lambda x.t \mid s\ t$$

Metavariables s, t and u range over terms. Metavariables x, y and z range over some countably infinite set of *term variables*, $\lambda x.t$ is called a λ-*abstraction* which is a function taking a parameter x and returning the result t, and $s\ t$ is an *application* of the function s to the argument t.

Within a given term, each occurrence of a term variable can be considered as either *bound* or *free.* A term variable x is bound if it occurs under a λx, and otherwise it is free. For example, in the term $\lambda y.x\ (\lambda x.x)$, the first occurrence of x is free while the second occurrence of x is bound by λx. Based on these notions, the λ-calculus defines the operation of *term substitution* by which it is possible to substitute some term s for all *free* occurrences of some term variable x within another term t. For example, given the term $\lambda y.x\ (\lambda x.x)$, substituting s for x results in $\lambda y.s\ (\lambda x.x)$ because only the first occurrence of x is free, while the second occurrence of x is bound by λx.

In the λ-calculus, the equivalent of executing a program is *evaluating* a λ-term. The evaluation of a λ-term is performed by a sequence of steps called *reductions.* There is only one needed reduction rule in the λ-calculus, called β-*reduction*, which works as follows: an application $(\lambda x.t)\ s$, *reduces* to the term t with s substituted for each free occurrence of x in t. For example, the application $(\lambda x.x\ x)\ (\lambda y.y)$ reduces to $(\lambda y.y)\ (\lambda y.y)$ which in turn reduces to $\lambda y.y$. The term $\lambda y.y$ cannot be reduced any further, and is called a *value.*

A λ-term can be evaluated using a number of different strategies including call-by-value [50], call-by-name [18] and call-by-need [5]. Each evaluation strategy is distinguished by performing sequences of β-reductions in particular orders. Call-by-value reduces a function argument to a value before applying the function. Call-by-name reduces a function argument after applying the function, each time the argument is used. Call-by-need reduces a function argument after applying the function, but only the first time the argument is used, with subsequent uses automatically reusing the result of the first time the argument was reduced.

In this dissertation, we will focus on the call-by-value semantics which is commonly used in object-oriented programming languages such as Java, C++ and C#.

2.2 Type System

The two most important features of System E are *intersection types* and *expansion variables.*

Intersection types are used rather than parametrically polymorphic types to provide polymorphism in System E. The primary difference between these two kinds of polymorphism is that a parametrically polymorphic type represents a possibly *infinite* set of type instances, while an intersection type represents a *finite* set of type instances. Recall from Section 1.1.2 of Chapter 1 that the identity function in Java can be defined with a parametrically polymorphic type as follows:

```
<X> X identity(X x) { return x; }
```

That is, the identity function can take any `X` to an `X` for some type parameter `X`, universally quantified by `<X>`. In the λ-calculus, the identity function can be expressed as $\lambda x.x$ and the equivalent parametrically polymorphic type can be expressed in conventional notation as $\forall\alpha.\alpha \to \alpha$ where $\alpha \to \alpha$ represents the type of a function taking an α to an α, and where $\forall\alpha$ universally quantifies the type parameter α. This polymorphic type represents an *infinite* set of instance types, each resulting from substituting a different type for the type parameter α.

In contrast, an intersection type can represent only a *finite* set of instance types, and is notated by explicitly enumerating the instance types. For example, the identity function may be given the intersection type $\texttt{int} \to \texttt{int} \sqcap \texttt{bool} \to \texttt{bool}$ which represents a set of exactly two instances, delimited by $\sqcap$. Intersection types may appear less powerful than parametrically polymorphic types, but they in fact type strictly more λ-terms than parametrically polymorphic types (see Section 1.2.3 of Chapter 1). The reason that finitary polymorphism suffices is that programs themselves are finite and a function such as the identity function will be used within a program in only a finite number of different contexts.

The second important feature of System E is *expansion variables*, or *E-variables*, which are used to express principal typings and thereby support compositional type inference. A principal type for the identity function is $e\ (\alpha \rightarrow \alpha)$ where e is an expansion variable that marks a position where an *expansion* operation can be performed. During type inference, a finite number of expansions can be inserted at the position of e as the finite number of uses of the identity function are encountered. For example, if the identity function is used on both integers and booleans, then this principal type can, during the type inference process, be expanded to $\mathtt{int} \rightarrow \mathtt{int} \sqcap \mathtt{bool} \rightarrow \mathtt{bool}$.

Note that while the System E type $e\ (\alpha \rightarrow \alpha)$ looks similar in structure to the parametrically polymorphic type $\forall\alpha.\alpha \rightarrow \alpha$, they are very different in nature. $e\ (\alpha \rightarrow \alpha)$ is in fact a monomorphic type, with the expansion variables and type variables serving only as placeholders during type inference to construct true polymorphic types such as $\mathtt{int} \rightarrow \mathtt{int} \sqcap \mathtt{bool} \rightarrow \mathtt{bool}$. That is to say, there is no typing derivation rule that says that a function of type $e\ (\alpha \rightarrow \alpha)$ can be applied to both integers and booleans, whereas there *is* such a rule for a function of type $\mathtt{int} \rightarrow \mathtt{int} \sqcap \mathtt{bool} \rightarrow \mathtt{bool}$. Thus, the most direct counterpart to the infinitary polymorphism in $\forall\alpha.\alpha \rightarrow \alpha$ in System E is the *finitary* polymorphism introduced by intersection types such as $\mathtt{int} \rightarrow \mathtt{int} \sqcap \mathtt{bool} \rightarrow \mathtt{bool}$.

In the remainder of this section, we introduce the type syntax and the typing rules that form the type system of System E. It is important to note that as far as the typing rules alone are concerned, expansion variables have virtually no semantic value. When inserted into types, they act merely as decorations and do not add to the typing power of the type system in any way. Their purpose is solely for use in type inference where substitutions for expansion variables are used to derive more specific types from more general types. Hence, while expansion variables will be introduced now as part of the type syntax, a discussion of their actual use will be postponed until Section 2.3 where the operations of expansion and substitution will be described in detail.

The *types* of System E are defined by the following grammar:

$$S, T, U ::= \omega \mid S \sqcap T \ \mid e\ T \mid \alpha \mid S \rightarrow T$$

ω is the *omega type* which can be assigned to unused terms. $S \sqcap T$ is an *intersection type* which can be assigned to a term that is to be used at both type S and type T. $e\ T$ is an *E-variable application*

$(S \sqcap T) \sqcap U = S \sqcap (T \sqcap U)$	$\sqcap$-associativity
$S \sqcap T = T \sqcap S$	$\sqcap$-commutativity
$T \sqcap \omega = T$	$\sqcap$-unit
$e\ (S \sqcap T) = e\ S \sqcap e\ T$	e distributes over $\sqcap$
$e\ \omega = \omega$	e distributes over ω

Figure 2.1: System E type equalities

type which decorates type T with an *E-variable* e, where metavariables e, f, g range over some countably infinite set of E-variables. Metavariables α, β and γ indicate *type variables* which represent unknown types. And finally, $S \to T$ is a *function type* from parameter type S to result type T.

Intersection types in System E are designed to be used linearly. That is, $S \sqcap T \sqcap U$ is the type of a term that is to be used 3 times, once at type S, once at type T and once at type U. ω should be viewed as the nullary case of $\sqcap$, and so ω is the type of a term that is used zero times.

System E imposes a set of equalities on types, given in Figure 2.1. With ω as the nullary case of $\sqcap$, an E-variable e distributes over ω in conceptually the same way that it distributes over $\sqcap$. Notice that while $\sqcap$ is associative and commutative, it is not also idempotent, meaning that $T \sqcap T$ is not, in general, equal to T. This is due to System E's linear typing model since $T \sqcap T$ is the type of a term that is to be used *twice* at type T, which is distinct from T, the type of a term that is to be used *once* at type T.

A *term context* (metavariable Γ) is a total function from term variables to types, mapping only a finite number of term variables to non-ω types. A term context is written $x_1 : T_1, \ldots, x_n : T_n$ with $y : \omega$ assumed for each variable y not explicitly mentioned. The term context that maps all term variables to ω is denoted Γ_ω.

A *typing judgement* $t : T \lhd \Gamma$ asserts that term t has type T in the term context Γ, or alternatively, that term t has *typing* $T \lhd \Gamma$. For example, the judgement $x\ y : \alpha \lhd x : \beta \to \alpha, y : \beta$ asserts that in a context where x has type $\beta \to \alpha$ and y has type β, the term $x\ y$ has type α.

The rules for deriving typing judgements in System E are presented in Figure 2.2. These rules enforce System E's linear typing model, which is illustrated by the following two examples.

The first example is a typing derivation for the term $\lambda x.x\ x$ in which the variable x is used twice:

$$\text{(var)}\ \frac{}{x : T \lhd x : T} \qquad\qquad \text{(omega)}\ \frac{}{t : \omega \lhd \Gamma_\omega}$$

$$\text{(abs)}\ \frac{t : T \lhd \Gamma, x : S}{\lambda x.t : S \to T \lhd \Gamma} \qquad\qquad \text{(int)}\ \frac{t : S \lhd \Gamma_1 \quad t : T \lhd \Gamma_2}{t : S \mathbin{\dot\cap} T \lhd \Gamma_1 \mathbin{\dot\cap} \Gamma_2}$$

$$\text{(app)}\ \frac{t : S \to T \lhd \Gamma_1 \quad s : S \lhd \Gamma_2}{t\ s : T \lhd \Gamma_1 \mathbin{\dot\cap} \Gamma_2} \qquad\qquad \text{(evar)}\ \frac{t : T \lhd \Gamma}{t : e\ T \lhd e\ \Gamma}$$

$$\text{where :}\quad (\Gamma_1 \mathbin{\dot\cap} \Gamma_2)(x) = \Gamma_1(x) \mathbin{\dot\cap} \Gamma_2(x) \text{ and } (e\ \Gamma)(x) = e\ \Gamma(x)$$

Figure 2.2: System E typing rules

$$\cfrac{\cfrac{\cfrac{}{x : \beta \to \alpha \lhd x : \beta \to \alpha}\,\text{(var)} \quad \cfrac{}{x : \beta \lhd x : \beta}\,\text{(var)}}{x\ x : \alpha \lhd x : (\beta \to \alpha) \mathbin{\dot\cap} \beta}\,\text{(app)}}{\lambda x.x\ x : ((\beta \to \alpha) \mathbin{\dot\cap} \beta) \to \alpha \lhd}\,\text{(abs)}$$

Rule (var) permits only x to be mentioned in the term context, and Rule (app) forces both uses of x to be mentioned, *and nothing more.* Since the individual typing for each x mentions x only once in the term context, Rule (app) must merge the term contexts to reflect both uses of x in the application $x\ x$. When abstracting over x, we obtain a function whose parameter type indicates that the parameter is used twice and at what types the parameter will be used.

The second example is a typing derivation for the term $\lambda x.(\lambda y.y)$ in which the variable x is not used:

$$\cfrac{\cfrac{\cfrac{}{y : \alpha \lhd y : \alpha}\,\text{(var)}}{\lambda y.y : \alpha \to \alpha \lhd}\,\text{(abs)}}{\lambda x.(\lambda y.y) : \omega \to (\alpha \to \alpha) \lhd}\,\text{(abs)}$$

In the first two judgements, x is not mentioned and so it is forced to have type ω in the term context. When abstracting over x in the final judgement, we obtain a function whose parameter type indicates that the parameter is not used.

Intersection types are introduced into parameter types and argument types in different ways. If a parameter is used multiple times by a function thereby requiring an intersection type, it is Rule (app) that is responsible for introducing an intersection type into the term context, and Rule (abs) that transfers the intersection type to the parameter type. However, these rules can ultimately introduce intersection types only to the types of parameters. To introduce an intersection type to the type of an argument that is to be passed into that parameter, Rule (var) can be used if the argument is a variable, or Rule (int) can be used for anything else. The following example illustrates the application of $\lambda x.x\ x$ to the argument $\lambda y.y$. Rule (app) and Rule (abs) are used to derive that the parameter type of $\lambda x.x\ x$ is an intersection type, while Rule (int) is used to give argument $\lambda x.x$ the required intersection type (where T abbreviates $\alpha \to \alpha$):

$$\cfrac{\cfrac{\cfrac{\overline{x : T{\to}T \lhd x : T{\to}T}\;(\text{var}) \qquad \overline{x : T \lhd x : T}\;(\text{var})}{x\ x : T \lhd x : (T{\to}T) \mathbin{\dot\cap} T}\;(\text{app})}{\lambda x.x\ x : ((T{\to}T) \mathbin{\dot\cap} T){\to}T\lhd}\;(\text{abs}) \qquad \cfrac{\cfrac{\overline{y : T \lhd y : T}\;(\text{var})}{\lambda y.y : T{\to}T\lhd}\;(\text{abs}) \qquad \cfrac{\overline{y : \alpha \lhd y : \alpha}\;(\text{var})}{\lambda y.y : T\lhd}\;(\text{abs})}{\lambda y.y : (T{\to}T) \mathbin{\dot\cap} T\lhd}\;(\text{int})}{(\lambda x.x\ x)\ (\lambda y.y) : T\lhd}\;(\text{app})$$

2.3 Expansions and Expansion Variables

The *expansion* operation of intersection type systems was introduced by Coppo, Dezani and Venneri [20] to achieve principal typings, and as such, it is an operation that is crucial to supporting compositional type inference (see Section 1.1.3 for a discussion relating principal typings and compositional analysis). A typing for some term t is said to be *principal* if all other possible typings for t can be derived from it via some specified operation or set of operations. In intersection type systems, this set usually includes the expansion operation along with the standard *substitution* operation which substitutes types for type variables. The main innovation of System I and its successor System E was to introduce *expansion variables*, or "*E-variables*", allowing both normal type substitution and expansion to be done using a single, unified substitution operation. It has been established that System I has the principal typings property for all finite-rank restrictions of the system [37]. System E improves on the work of System I by offering a much more simply defined expansion operation that is also "composable", allowing type inference procedures to be broken into solution steps which can be composed together into a single solution (see Section 2.3.2). A similar principal typings result as the one for System I has not yet been established for System E, although it has been established that the second type inference algorithm for

System E [6] infers typings that are principal with respect to a subset of System E typings that have a correspondence with either call-by-name or call-by-value evaluation.

Before we introduce System E's unified substitution operation based on expansion variables, we will discuss how the separate type substitution and expansion operations originally worked in earlier intersection type systems.

2.3.1 The Traditional Approach

This section reviews the traditional approach to type substitutions and expansions in which they are treated as separate operations.

Let us first review type substitutions. If we already know that the judgement $\lambda x.\lambda y.x\ y : (\alpha \to \beta) \to (\alpha \to \beta) \lhd$ can be derived, then it stands to reason that the judgement $\lambda x.\lambda y.x\ y : (S \to T) \to (S \to T) \lhd$ can also be derived, for any types S and T. This is justified because if we take the typing derivation of the first judgement and substitute type S for every occurrence of the type variable α and type T for every occurrence of the type variable β, we get a valid typing derivation for the second judgement:

Original derivation	After substituting $\alpha := S, \beta := T$
$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha \to \beta \lhd x : \alpha \to \beta}\,(\text{var}) \quad \dfrac{}{y : \alpha \lhd y : \alpha}\,(\text{var})}{x\ y : \beta \lhd x : \alpha \to \beta, y : \alpha}\,(\text{app})}{\lambda y.x\ y : \alpha \to \beta \lhd x : \alpha \to \beta}\,(\text{abs})}{\lambda x.\lambda y.x\ y : (\alpha \to \beta) \to (\alpha \to \beta) \lhd}\,(\text{abs})$	$\dfrac{\dfrac{\dfrac{\dfrac{}{x : S \to T \lhd x : S \to T}\,(\text{var}) \quad \dfrac{}{y : S \lhd y : S}\,(\text{var})}{x\ y : T \lhd x : S \to T, y : S}\,(\text{app})}{\lambda y.x\ y : S \to T \lhd x : S \to T}\,(\text{abs})}{\lambda x.\lambda y.x\ y : (S \to T) \to (S \to T) \lhd}\,(\text{abs})$

Because type variables represent unknown types sitting at the leaves of the type tree, type substitution has the corresponding effect of replacing applications of Rule (var) at the leaves of the typing derivation tree. At least in the absence of intersection types, this allows for the expression of principal typings for terms, from which all other typings can be derived via a type substitution.

However, when considering intersection types, type substitution alone is insufficient to derive all other possible typings, and so the expansion operation was introduced alongside the type substitution operation to obtain the principal typings property in intersection type systems [20]. In contrast to type substitution which has the effect of replacing leaves in the typing derivation tree, expansion has the effect of replacing *nested* nodes in the typing derivation tree, specifically introducing applications of Rule (int). In its original conception, the expansion operation is applied to typings by first selecting a *nucleus* consisting of the segments of the typing to which an expansion operation may be applied. In the following example, a possible nucleus has been underlined:

$$\lambda x.\lambda y.x\ y : (\underline{\alpha \to \beta}) \to (\underline{\alpha \to \beta})\lhd$$

A nucleus must be selected in such a way that the marked segments correspond to a node in the typing derivation tree where it would be permissible to insert an application of Rule (int). Since the marked segments are not explicitly part of the type syntax, the notion of a nucleus is implicit in nature, and the selection of a valid nucleus is governed by a set of rules which are not all straightforward to understand (see [16] for a discussion). This was one of the motivations for using expansion variables in System I and System E which make nuclei explicit in the type syntax. This explicit treatment of nuclei will be discussed in Section 2.3.2.

Next, once the nucleus has been selected, the expansion operation replaces the marked segments of the nucleus with an intersection of copies of the original segment with type variables renamed:

$$\lambda x.\lambda y.x\ y : (\underline{(\alpha_1 \to \beta_1) \mathbin{\dot\cap} (\alpha_2 \to \beta_2)}) \to (\underline{(\alpha_1 \to \beta_1) \mathbin{\dot\cap} (\alpha_2 \to \beta_2)})\lhd$$

This expansion is sound because it corresponds to the insertion of an application of Rule (int) into the derivation of the original typing at the following point:

Original derivation	After expansion
$\dfrac{\dfrac{\vdots}{\lambda y.x\ y : \alpha\to\beta \lhd x : \alpha\to\beta}}{\lambda x.\lambda y.x\ y : (\alpha\to\beta)\to(\alpha\to\beta)\lhd}$	$\dfrac{\dfrac{\dfrac{\vdots}{\lambda y.x\ y : \alpha_1\to\beta_1 \lhd x : \alpha_1\to\beta_1} \quad \dfrac{\vdots}{\lambda y.x\ y : \alpha_2\to\beta_2 \lhd x : \alpha_2\to\beta_2}}{\lambda y.x\ y : (\alpha_1\to\beta_1) \mathbin{\dot\cap} (\alpha_2\to\beta_2) \lhd x : (\alpha_1\to\beta_1) \mathbin{\dot\cap} (\alpha_2\to\beta_2)} \leftarrow \text{(int)}}{\lambda x.\lambda y.x\ y : ((\alpha_1\to\beta_1) \mathbin{\dot\cap} (\alpha_2\to\beta_2))\to((\alpha_1\to\beta_1) \mathbin{\dot\cap} (\alpha_2\to\beta_2))\lhd}$

Like type substitution, expansion allows more specific typings to be derived from more general ones, and together these operations allow the expression of principal typings.

2.3.2 The System E Approach

We now turn to the System E approach which combines both type substitution and expansion into a single substitution operation. The key is to introduce expansion variables which are used to mark positions within a typing where expansions may be applied. Now, the substitution operation can be used to substitute not only types for type variables but also expansions for expansion variables.

Normal type substitutions work as before, so this section will focus on expansion substitutions. First, expansion variables are introduced into typings via Rule (evar) as demonstrated below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{x : \alpha \to \beta \lhd x : \alpha \to \beta} \qquad \overline{y : \alpha \lhd y : \alpha}}{x\ y : \beta \lhd x : \alpha \to \beta, y : \alpha}}{\lambda y.x\ y : \alpha \to \beta \lhd x : \alpha \to \beta}}{\lambda y.x\ y : e\ (\alpha \to \beta) \lhd x : e\ (\alpha \to \beta)} \leftarrow \text{(evar)}}{\lambda x.\lambda y.x\ y : e\ (\alpha \to \beta) \to e\ (\alpha \to \beta) \lhd}$$

Rule (evar) is applied at the point in the derivation where we may wish to insert an expansion, and the E-variable e marks the affected segments of the final typing in much the same way that the underlined nucleus marked the affected segments in the traditional approach. Expansion variables effectively replace the implicit rules for defining nuclei, and also combine with type variables to support a unified substitution operator, and thereby make it easier to compute principal typings. During type inference, an important process is to insert E-variables at appropriate points to give rise to principal typings, and hence support compositional analysis. This will be discussed in Section 2.4.

Expansions themselves are defined more broadly in System E than in traditional systems. Whereas traditional expansions modify typings in a way that corresponds to the introduction of Rule (int) into a typing derivation, System E expansions modify typings in a way that corresponds to the introduction of any of the following 3 rules into a typing derivation: Rule (int), Rule (omega) and Rule (evar).

Formally, the *expansions* form a new syntactic category consisting of

$$E, F, G ::= \omega \mid E \Cap F \mid e\ E \mid \sigma \qquad \textbf{expansions}$$

$$\sigma ::= \boxdot \mid \sigma, \alpha := T \mid \sigma, e := E \qquad \textbf{substitutions}$$

with associated application rules defined in Figure 2.3. Each expansion can be applied to *entities* which collectively refer to both types and expansions, the latter making expansions composable. ω is the *omega-expansion* and transforms any entity into ω (either the type or the expansion depending on context). $E \Cap F$ is an *intersection expansion* and transforms an entity into an intersection of two copies of the original entity with E recursively applied to the left copy and F recursively applied to the right copy. $e\ E$ is an *E-variable application expansion* and transforms an entity into an E-variable application by recursively applying E to the original entity and applying the E-variable e to the result. σ is a *substitution*, inductively defined as either the *identity substitution* $\boxdot$, or the extension of an existing substitution with a *type assignment* $\alpha := T$ or an *expansion assignment* $e := E$. A substitution σ is applied to an entity K by traversing the tree structure of K from its root to its leaves, and replacing variables in the tree according to matching assignments in σ, but without descending into entities nested within E-variable applications. For example, applying the substitution $(\boxdot, \alpha := S, \beta := T)$ to the type $\alpha \to e\ \beta$ results in $S \to e\ \beta$. In order to also substitute for the β variable under e, we can instead use the substitution $(\boxdot, \alpha := S, e := f\ (\boxdot, \beta := T))$ which results in $S \to f\ T$.

To understand how expansion application works in practice, we will consider the following term and typing:

$$(J_1) \qquad \lambda x.\lambda y.x\ y : e\ (\alpha{\to}\beta){\to}e\ (\alpha{\to}\beta)\lhd$$

Let E be the intersection expansion $(\boxdot, \alpha := \alpha_1, \beta := \beta_1) \Cap (\boxdot, \alpha := \alpha_2, \beta := \beta_2)$. Substituting $\boxdot, e := E$ results in the following typing:

$$
\begin{array}{llll}
\omega\ K & = \omega & \sigma\ \omega & = \omega \\
(E \cap F)\ K & = E\ K \cap F\ K & \sigma\ (K_1 \cap K_2) & = \sigma\ K_1 \cap \sigma\ K_2 \\
(e\ E)\ K & = e\ (E\ K) & \sigma\ (e\ K) & = (\sigma\ e)\ K \\
\boxdot\ e & = e\ \boxdot & \sigma\ (T_1 \to T_2) & = \sigma\ T_1 \to \sigma\ T_2 \\
\boxdot\ \alpha & = \alpha & \sigma\ \boxdot & = \sigma \\
(\sigma, X := K)\ X & = K & \sigma\ (\sigma', X := K) & = \sigma\sigma', X := \sigma\ K \\
(\sigma, X := K)\ X' & = \sigma\ X' \text{ if } X \neq X' & &
\end{array}
$$

where: $X ::= \alpha \mid e$ **entity variables**

$K ::= T \mid E$ **entities**

Figure 2.3: System E expansion application rules

$$(J_2) \qquad \lambda x.\lambda y.x\ y : ((\alpha_1 {\to} \beta_1) \cap (\alpha_2 {\to} \beta_2)) {\to} ((\alpha_1 {\to} \beta_1) \cap (\alpha_2 {\to} \beta_2)) \lhd \qquad (\text{applying } \boxdot, e := E)$$

The expansion E introduces an intersection of two copies of the original type under e. In order to allow each branch of the new intersection to be subject to independent substitutions, the expansion E explicitly renames the type variables α and β in each branch. However, explicitly renaming type variables can be a lot of work, particularly when many type variables are involved. Fortunately, an interesting property of expansion variables gives rise to a better alternative. The idea is that rather than substituting the expansion E for e, we can instead substitute a simpler expansion $E' = f\ \boxdot \cap g \boxdot$, giving

$$(J_3) \quad \lambda x.\lambda y.x\ y : (f\ (\alpha {\to} \beta) \cap g\ (\alpha {\to} \beta)) {\to} (f\ (\alpha {\to} \beta) \cap g\ (\alpha {\to} \beta)) \lhd \quad (\text{applying } \boxdot, e := E' \text{ to } J_1)$$

Due to the way substitution application is defined, the E-variables f and g essentially create separate namespaces, allowing each branch of the intersection to be subject to independent substitutions. For example, to apply substitution $\sigma_1 = (\boxdot, \alpha := S_1, \beta := T_1)$ to the left branch and substitution $\sigma_2 = (\boxdot, \alpha := S_2, \beta := T_2)$ to the right branch, we may use the substitution $\sigma = (\boxdot, f := \sigma_1, g := \sigma_2)$ on J_3

resulting in

$$(J_4) \qquad \lambda x.\lambda y.x\ y : ((S_1{\rightarrow}T_1) \cap (S_2{\rightarrow}T_2)){\rightarrow}((S_1{\rightarrow}T_1) \cap (S_2{\rightarrow}T_2))\lhd \qquad \text{(applying } \sigma \text{ to } J_3)$$

Expansion application is defined in such a way that expansions are *composable*. That is, it is a property of System E expansions that an expansion $F\ E$ has the same effect on a given entity as would result from successively applying E and then F to that entity. More precisely, it holds in System E that $(F\ E)\ K = F\ (E\ K)$, and in the specific case of substitutions, that $(\sigma_2\ \sigma_1)\ K = \sigma_2\ (\sigma_1\ K)$. This property is useful in type unification algorithms where a solution to a unification constraint is found by applying a succession of expansions (or substitutions) to reach a solved state, and then composing those expansions (or substitutions) to produce a single unifier.

E-variables make it possible to express principal typings. However, there are two choices in how principal typings can be defined:

1. A principal typing can be defined as one from which all other typings can be derived via a substitution operation. In this case, $f\ (e\ (\alpha{\rightarrow}\beta){\rightarrow}e\ (\alpha{\rightarrow}\beta))\lhd$ is a principal typing for $\lambda x.\lambda y.x\ y$.
2. A principal typing can be defined as one from which all other typings can be derived via an expansion operation, because expansions generalise substitutions. In this case, $e\ (\alpha{\rightarrow}\beta){\rightarrow}e\ (\alpha{\rightarrow}\beta)\lhd$ is a principal typing for $\lambda x.\lambda y.x\ y$. The outer E-variable f is no longer needed since an expansion operation can always apply at the outer level without the assistance of an expansion variable to target a more specific location.

Both definitions have been used in practice: the first type inference algorithm [15] for System E used the expansion operation to define principal typings, while the second type inference algorithm [6] for System E used the substitution operation.

2.4 Type Inference

Type inference in System E is often referred to as *typing* inference because the process requires only the term as input and returns a typing as output. This is in contrast to the Hindley/Milner approach to type inference which requires both a term and a term context as input, and returns a type as output.

The difference can be illustrated by the problematic example of "self application" in which a term variable is applied to itself. This is represented in the λ-calculus by the term $x\ x$. In the Hindley/Milner approach, type inference cannot succeed for this term without being given some "hint" about the variable x in the form of a term context. If the entire program were (let $x = \lambda y.y$ in $x\ x$), the Hindley/Milner approach would be to first infer a polymorphic type for x based on an analysis of $\lambda y.y$, in this case $\forall\alpha.\alpha \to \alpha$. Only after determining that polymorphic type, the type for $x\ x$ can be found by instantiating that polymorphic type twice and setting up and solving a simple first-order unification problem between the type of the second x and the parameter type of the first x. However, if x were a parameter rather than a let-bound variable, as in the abstraction $\lambda x.x\ x$, then the same type inference approach would fail since no such hint could be obtained about the polymorphic nature of x. The dependency on such a hint means that this approach is not strictly compositional because it cannot analyse the term $x\ x$ from just its components.

System E, on the other hand, does not require a term context as input, and in fact computes the term context along with the type. The term $x\ x$ can be analysed compositionally by first inferring typings for each x, say $\alpha \lhd x : \alpha$ and $\beta \lhd x : \beta$. Next, it is necessary to solve the constraint that α (the type of the function) be equal to $\beta \to \gamma$ (a function that takes $x : \beta$ as input), for some arbitrary γ. The solution to this constraint is the substitution $\boxdot, \alpha := \beta \to \gamma$. After applying this substitution, the typings for the two occurrences of x are refined to $\beta \to \gamma \lhd x : \beta \to \gamma$ and $\beta \lhd x : \beta$. The substitution that solved our constraint has essentially transformed the typings for the function x and the argument x so that the type of the argument matches the parameter type of the function, and this means that the inference of a typing for the entire application $x\ x$ immediately follows by Rule (app):

$$\frac{x : \beta \to \gamma \lhd x : \beta \to \gamma \quad x : \beta \lhd x : \beta}{x\ x : \gamma \lhd x : (\beta \to \gamma) \Cap \beta}\ \text{(app)}$$

In the process of applying Rule (app), the two distinct types for the two occurrences of x are intersected together, and thus the following context information is inferred about x: it has the polymorphic type $(\beta \rightarrow \gamma) \sqcap \beta$, being used once at type $\beta \rightarrow \gamma$ and once more at type β. Note that any other typing for $x\ x$ can be derived via an expansion of this typing. If we desire all other typings to be derivable via a substitution, we also need to wrap this typing in an E-variable by Rule (evar), giving:

$$\frac{\vdots}{x\ x : e\ \gamma \lhd x : e\ ((\beta \rightarrow \gamma) \sqcap \beta)}\ \text{(evar)}$$

It is a strength of intersection types that it is possible to infer a principal typing for such terms without requiring any external context. Given the same term in System F or the Hindley/Milner type system, a type inference procedure cannot compute a single context for x that would lead to a principal typing for this application. For example, both $\forall X.X \rightarrow X$ and $\forall X.X \rightarrow T$ are possible types for x but are unrelated to each other.

While the type inference process itself is straightforward in System E, the real work is actually performed by the *unification algorithm* which is invoked to solve the constraints introduced by applications. Given a constraint that type S be the same as type T, the goal of the unification algorithm is to compute a *unifier* which is a substitution σ that satisfies $\sigma S = \sigma T$. Since types in System E may contain both type variables and expansion variables, a unifier may need to substitute not only types for type variables but also expansions for expansion variables.

There are two main approaches to performing type unification with E-variables, called β-unification (reviewed in Section 1.2.3 of Chapter 1) and covering unification [7]. These two approaches are reviewed and compared in the following sections.

2.4.1 β-Unification

β-unification solves constraints generated from a program in a way that exactly corresponds to β-reduction of the same program. A result of this correspondence is that the type inference procedure will terminate on exactly the set of normalising terms, relative to a particular evaluation strategy. Strictly speaking, a type unification algorithm operates at the level of types, and so it would seem that

β-unification could not possibly make any guarantee of a correspondence with β-reduction of terms. To make this guarantee, β-unification must assume that the set of constraints has been generated in just the right way, and that E-variables have been inserted in just the right places. If this is the case, then the unification of those constraints will correspond to β-reduction of the original program.

Constraints are of the form $S \underset{\cdot}{\leq} T$, and an expansion E solves – or "unifies" – such a constraint if $E\ S = E\ T$. A key aspect of β-unification is its use of asymmetric constraints whereby the unification algorithm assumes that, when unifying the type of a function parameter with the type of an argument, the argument type is always on the left side of a constraint and the parameter type is always on the right. Care must be taken when recursively unifying two function types as in constraint $S_1 \to S_2 \underset{\cdot}{\leq} T_1 \to T_2$. This constraint is solved by recursively solving the the constraints $T_1 \underset{\cdot}{\leq} S_1$ and $S_2 \underset{\cdot}{\leq} T_2$, where the parameter types are reversed due to the contravariance of parameter types in function subtyping.

A pre-step for β-unification is to generate for a given term an initial "unsolved" typing and an initial set of constraints, the solution of which would transform the initial unsolved typing into a valid typing for the term. Depending on the placement of the E-variables during this pre-step, the same β-unification can be made to simulate either call-by-name or call-by-value [6]. The procedure for constructing the initial typing and constraints was defined in [6] using the notion of skeletons, but is equivalently restated below directly in terms of typings and constraints. We also present only the simpler of the two procedures, which is call-by-name. The function initial takes a λ-term and returns a pair $(\Delta; \tau)$ of a set of constraints Δ and an initial typing τ.

$$\frac{}{\mathsf{initial}(x) = (\emptyset; \alpha \lhd x : \alpha)}\ \alpha\ \mathsf{fresh} \qquad \frac{\mathsf{initial}(t) = (\Delta; T \lhd \Gamma, x : S)}{\mathsf{initial}(\lambda x.t) = (\Delta; S \to T \lhd \Gamma)}$$

$$\frac{\mathsf{initial}(t) = (\Delta_1; T \lhd \Gamma_1) \quad e\ \mathsf{initial}(s) = (\Delta_2; S \lhd \Gamma_2)}{\mathsf{initial}(t\ s) = (\Delta_1 \cup \Delta_2 \cup T \underset{\cdot}{\leq} S \to \alpha; \alpha \lhd \Gamma_1 \cap \Gamma_2)}\ e, \alpha\ \mathsf{fresh}$$

The initial typing for a term variable corresponds to an application of Rule (var), and this is also a valid typing without any constraint required. The initial typing for an abstraction is defined in a way that corresponds to an application of Rule (abs), and it introduces no new constraints. The initial typing for an application is defined in a way that corresponds to an application of Rule (app), except that the type of the function t's parameter may not yet match the type of the argument s. The unification constraint $T \mathrel{\underline{\lessdot}} S \to \alpha$ is introduced to make these match, with α used as a fresh variable to indicate the return type of the function, and also the result type of the application. Note that argument types are always equipped with an E-variable allowing them to be *expanded* to match the required shape of the parameter type to which they are passed.

Once the initial typing and set of constraints have been generated, β-unification begins. Due to the way in which constraints are generated, constraints of the form $U \to T \mathrel{\underline{\lessdot}} e\ S \to \alpha$ correspond to β-redexes of the form $(\lambda x.t)\ s$ where U corresponds to the type of the parameter x, T corresponds to the type of the body t, S corresponds to the type of the argument s with an E-variable wrapped around it so that it can be expanded to match the required parameter type, and α corresponds to the type of the application result. Solving one constraint of this form corresponds to performing one step of β-reduction on the λ-term.

The correspondence of β-unification to β-reduction can be understood by considering the example of the non-terminating application $(\lambda x.x\ x)\ (\lambda x.x\ x)$ which results in an equivalently non-terminating sequence of β-unification steps. First, the initial typing and set of constraints are generated as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{initial}(x) = (\emptyset; \alpha_1 \lhd x : \alpha_1} \qquad \dfrac{}{\mathsf{initial}(x) = (\emptyset; \alpha_1 \lhd x : \alpha_1}}{\mathsf{initial}(x\ x) = (\alpha_1 \mathrel{\underline{\lessdot}} e_1\alpha_2 {\to} \alpha_3; \alpha_3 \lhd x : \alpha_1 \mathbin{\dot{\cap}} e_1\alpha_2)}}{\mathsf{initial}(\lambda x.x\ x) = (\alpha_1 \mathrel{\underline{\lessdot}} e_1\alpha_2 {\to} \alpha_3; (\alpha_1 \mathbin{\dot{\cap}} e_1\alpha_2) {\to} \alpha_3 \lhd \Gamma_\omega)} \qquad \dfrac{\vdots}{\mathsf{initial}(\lambda x.x\ x) = (\beta_1 \mathrel{\underline{\lessdot}} e_2\beta_2 {\to} \beta_3; (\beta_1 \mathbin{\dot{\cap}} e_2\beta_2) {\to} \beta_3 \lhd \Gamma_\omega)}}{\mathsf{initial}((\lambda x.x\ x)\ (\lambda x.x\ x)) = (\begin{pmatrix} \alpha_1 \mathrel{\underline{\lessdot}} e_1\alpha_2 {\to} \alpha_3, \\ e_3\beta_1 \mathrel{\underline{\lessdot}} e_3(e_2\beta_2 {\to} \beta_3), \\ (\alpha_1 \mathbin{\dot{\cap}} e_1\alpha_2) {\to} \alpha_3 \mathrel{\underline{\lessdot}} e_3((\beta_1 \mathbin{\dot{\cap}} e_2\beta_2) {\to} \beta_3) {\to} \gamma \end{pmatrix}; \gamma \lhd \Gamma_\omega)}$$

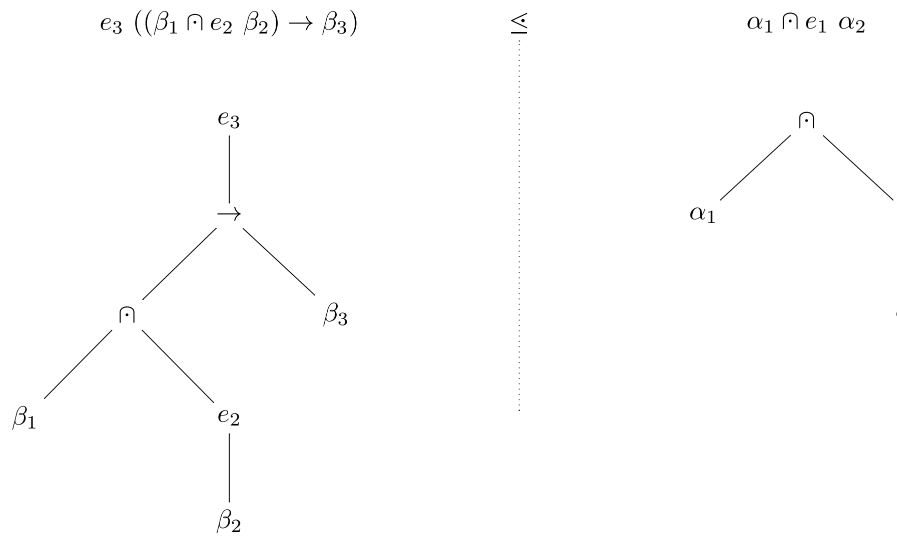
The initial typing is therefore $\gamma \lhd \Gamma_\omega$, and the constraints that must be solved to make this typing correct are:

1.	$\alpha_1 \mathrel{\underline{\lessdot}} e_1\ \alpha_2 \to \alpha_3$	Corresponds to the 1st $x\ x$ application
2.	$e_3\ \beta_1 \mathrel{\underline{\lessdot}} e_3\ (e_2\ \beta_2 \to \beta_3)$	Corresponds to the 2nd $x\ x$ application
3.	$(\alpha_1 \mathbin{\dot{\cap}} e_1\ \alpha_2) \to \alpha_3 \mathrel{\underline{\lessdot}} e_3\ ((\beta_1 \mathbin{\dot{\cap}} e_2\ \beta_2) \to \beta_3) \to \gamma$	Corresponds to $(\lambda x.x\ x)\ (\lambda x.x\ x)$

Each constraint corresponds to an application in the program, although only the third constraint corresponds to a β-redex. Since both sides of the constraint take the form of a function type, the constraint is recursively solved by generating the following constraints, taking care to reverse the parameter types due to the contravariance of parameter types:

$$\begin{array}{ll} \text{3a.} & e_3\ ((\beta_1 \mathbin{\dot{\cap}} e_2\ \beta_2) \to \beta_3) \preceq \alpha_1 \mathbin{\dot{\cap}} e_1\ \alpha_2 \\ \text{3b.} & \alpha_3 \preceq \gamma \end{array}$$

In (3a), the left type $e_3\ ((\beta_1 \mathbin{\dot{\cap}} e_2\ \beta_2) \to \beta_3)$ now represents the type of the argument, and the right type $\alpha_1 \mathbin{\dot{\cap}} e_1\ \alpha_2$ represents the type of the parameter. The strategy to unify these two types is to make the type of the argument expand, via e_3, to match the required type of the parameter. The expansion process on this constraint can be visualised as follows.

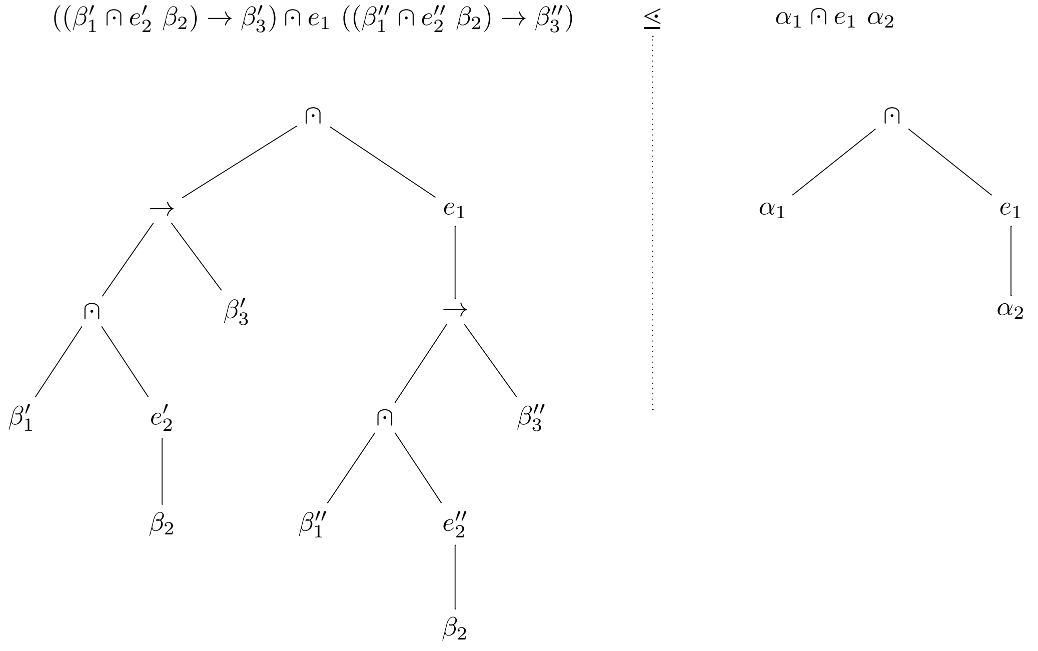

$e_3\ ((\beta_1 \mathbin{\dot{\cap}} e_2\ \beta_2) \to \beta_3)$
$\preceq$
$\alpha_1 \mathbin{\dot{\cap}} e_1\ \alpha_2$
e_3
$\to$
$\dot{\cap}$
β_3
β_1
e_2
β_2
$\dot{\cap}$
α_1
e_1
α_2

By substituting the expansion $\boxdot \mathbin{\dot{\cap}} e_1 \boxdot$ for e_3, it is possible to transform the left type to match the structure required by the right type. To avoid any potential variable name clashes, the type and expansion variables should also be freshly renamed in each branch, and so we use the following substitution where the renaming substitutions are underlined:

$$\sigma_1 = \quad e_3 := \boxdot, (\underline{\boxdot, \beta_1 := \beta_1', e_2 := e_2'\ \boxdot, \beta_3 := \beta_3'}) \mathbin{\dot{\cap}} e_1\ (\underline{\boxdot, \beta_1 := \beta_1'', e_2 := e_2''\ \boxdot, \beta_3 := \beta_3''})$$

After applying this substitution, the constraint now looks as follows:

$$((\beta_1' \mathbin{\dot\cap} e_2'\ \beta_2) \to \beta_3') \mathbin{\dot\cap} e_1\ ((\beta_1'' \mathbin{\dot\cap} e_2''\ \beta_2) \to \beta_3'') \qquad \leq \qquad \alpha_1 \mathbin{\dot\cap} e_1\ \alpha_2$$

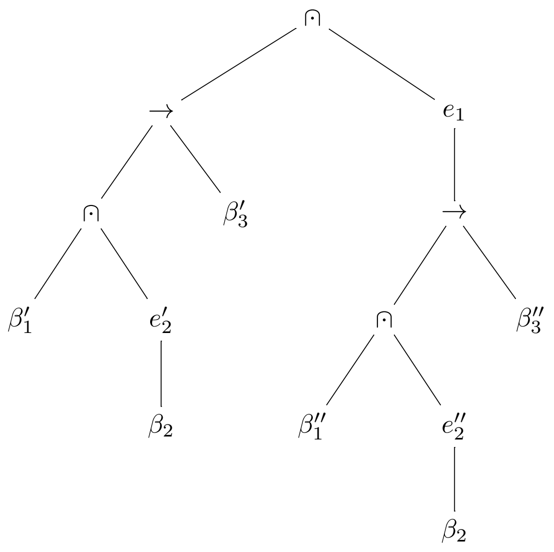


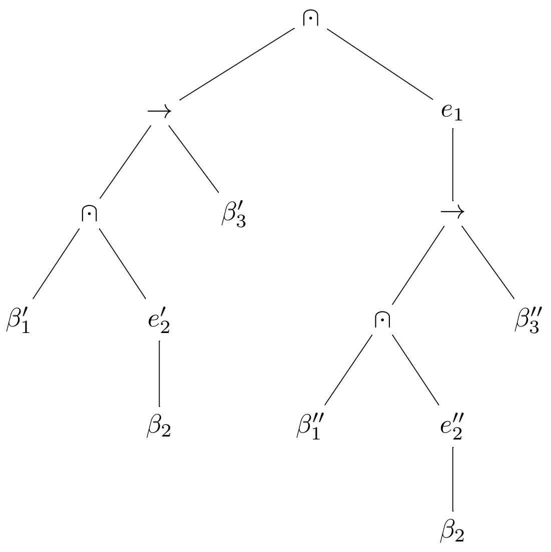
After expansion has matched as much of the tree as possible, the remainder of the unification is performed with plain substitutions for the type variables α_1 and α_2 at the leaves of the tree. This is achieved using the following substitution:

$$\sigma_2 = \quad \alpha_1 := \boxdot, (\beta_1' \mathbin{\dot\cap} e_2'\ \beta_2') \to \beta_3', e_1 := e_1\ (\alpha_2 := (\beta_1'' \mathbin{\dot\cap} e_2''\ \beta_2) \to \beta_3'')$$

Note that since α_2 exists under e_1 and e_1 acts as a namespace (see Section 2.3.2), we cannot directly substitute for α_2 using $\alpha_2 := \ldots$. Instead we substitute for e_1 an expansion that re-inserts e_1 and then applies a substitution for α_2 underneath e_1. This is achieved by the fragment $e_1 := e_1\ (\alpha_2 := \ldots)$. This results in the following unified type tree:

$$((\beta_1' \dot{\cap}\, e_2'\ \beta_2) \rightarrow \beta_3') \dot{\cap}\, e_1\ ((\beta_1'' \dot{\cap}\, e_2''\ \beta_2) \rightarrow \beta_3'') \quad \leq \quad ((\beta_1' \dot{\cap}\, e_2'\ \beta_2) \rightarrow \beta_3') \dot{\cap}\, e_1\ ((\beta_1'' \dot{\cap}\, e_2''\ \beta_2) \rightarrow \beta_3'')$$





Finally, constraint (3b) is easily solved by the following substitution which results in a complete solution for constraint (3):

$$\sigma_3 = \quad \boxdot, \alpha_3 := \gamma$$

Since expansions and substitutions in System E are composable, the substitution $\sigma_3\sigma_2\sigma_1$ has the combined effect of performing all of the above steps. The work accomplished so far by $\sigma_3\sigma_2\sigma_1$ now needs to be applied to the initial typing (which remains as $\gamma \lhd \Gamma_\omega$ since none of the above substitutions decided on a type for γ), and also needs to be applied to the two remaining constraints that are in the process of being solved, namely, (1) and (2). After applying the work of $\sigma_3\sigma_2\sigma_1$, constraints (1) and (2) become (4) and (5) respectively:

4. $(\beta_1' \dot{\cap}\, e_2'\beta_2') \rightarrow \beta_3' \leq e_1\ ((\beta_1'' \dot{\cap}\, e_2''\beta_2) \rightarrow \beta_3'') \rightarrow \gamma$
5. $\beta_1' \dot{\cap}\, e_1\ \beta_1'' \leq (e_2'\ \beta_2 \rightarrow \beta_3') \dot{\cap}\, e_1\ (e_2''\ \beta_2 \rightarrow \beta_3'')$

This now completes one step of β-unification, and this also corresponds to one step of β-reduction on the original term. The correspondence to β-reduction is as follows: given a term t for which $\mathsf{initial}(t)$ computes an initial typing of τ and an initial constraint set of Δ, if one step of β-unification produces the substitution σ, there exists a corresponding step of β-reduction resulting in the term t' where $\mathsf{initial}(t')$ gives the initial typing $\sigma\tau$ and the initial constraint set $\sigma\Delta$. In the above example, this can easily be verified because the term after one step of reduction is identical to the original term. We note that the resulting typing $\gamma \lhd \Gamma_\omega$ after one step of β-unification is identical to the original typing, that constraint (4) is isomorphic to the original constraint (3), and also that once constraint (5) is factored into its parts (5a) and (5b) below, those are also isomorphic to the original constraints (1) and (2):

5a. $\beta'_1 \leq e'_2\ \beta_2 \to \beta'_3$
5b. $e_1\ \beta''_1 \leq e_1\ (e''_2\ \beta_2 \to \beta''_3)$

To understand the correspondence in more detail, constraint (3) corresponds to the application of $(\lambda x.x\ x)$ to $(\lambda x.x\ x)$. One step of β-reduction has the effect of creating two copies of the argument $\lambda x.x\ x$ and substituting them into the two occurrences of x in the body of the left function resulting in $(\lambda x.x\ x) \to (\lambda x.x\ x)$. This corresponds to the unification of constraint (3) where the expansion variable e_3 is wrapped around the argument type allowing it to expand to match the required type of the parameter. In the process, e_3 is expanded into an intersection type thus creating two copies of the argument type.

The substitution produced by this step of β-unification is then applied to constraint (1) which corresponds to the first $x\ x$ occurring as the body of the left function. Just as the body $x\ x$ is transformed into $(\lambda x.x\ x)\ (\lambda x.x\ x)$ by term substitution, the corresponding constraint (1) is transformed into the constraint (4) which is isomorphic to the initial constraint for $(\lambda x.x\ x)\ (\lambda x.x\ x)$.

If the evaluation of a term terminates, then the β-unification procedure will also terminate with a unifier that, when applied to the initial typing, produces a correct (and principal) typing for the term.

2.4.2 Covering Unification

This section reviews the main ideas of covering unification, and presents a walk-through example. A *covering unification* procedure attempts to compute a solution to a given constraint that is in some sense "most general". Rather than being guided by β-reduction, a covering unification procedure finds unifiers purely by analysing type structure. Each constraint $S \doteq T$ is now symmetric since there is no assumption that one side corresponds to a function argument and the other a function parameter.

Ideally, the sort of most general solution to be computed would be a *most general unifier*; that is, a unifier σ such that any other unifier σ' factors through σ by some substitution σ'' (i.e. $\sigma' = \sigma''\sigma$). Unfortunately, constraints in System E do not always have most general unifiers. An example given in [7] is the constraint $e\ \alpha \doteq \alpha \to \alpha$ which has the following unifiers:

$$\sigma_1 = (e := (\alpha := \alpha \to \alpha))$$
$$\sigma_2 = (e := (\alpha := \alpha \to \alpha), \beta := \alpha)$$

σ_1 is the obvious candidate for a most general unifier, although it does not qualify: $\sigma_1\ (e\ \beta \cap \beta) = \beta \cap \beta$ while $\sigma_2\ (e\ \beta \cap \beta) = \beta \cap \alpha$, and hence there is no substitution σ'' such that $\sigma_2 = \sigma''\sigma_1$. The issue is that σ_1 merges the namespace under e with the top-level namespace so that there is no longer any distinction between the β under e and the β at the top level. To fix this, it would be necessary for σ_1 to rename all variables while eliminating e, however the substitution to achieve this would need to have an infinite support.

This problem has led to the notion of a *principal unifier* [7] which is a unifier σ such that for any other unifier σ' there exists a substitution σ'' such that $\sigma'\mathcal{V} = \sigma''\sigma\mathcal{V}$ for some *finite* set of variables $\mathcal{V}$ that at least includes all of the variables in the constraint being solved, and any other variables of consideration (e.g. those appearing in other constraints being solved alongside this one, or type/expansion variables appearing in the term context of an initial typing from which the constraint was generated). Letting $\mathcal{V}$ be the set $\{e, \alpha, \beta\}$, it is now possible to construct σ_3, a replacement for σ_1 that renames all of the variables in $\mathcal{V}$ while eliminating e:

$$\sigma_3 = (e := (\alpha := \alpha \to \alpha, \beta := \beta', e := e' \boxdot))$$

Now, the effect of σ_1 is obtained by composing σ_3 and $(\beta' := \beta, e' := e \boxdot)$, while the effect of σ_2 is obtained by composing σ_3 and $(\beta' := \beta, e' := \boxdot, \beta := \alpha)$.

A second issue is that *single* principal unifiers often do not exist [7]. For example, any solution to the constraint $e\ (\alpha \to \alpha) \sqcap f\ (\alpha \to \alpha) \doteq \alpha \to \alpha$ must choose between substituting $e := \omega$ or $f := \omega$ and so there can be no single principal unifier. In general, therefore, the kind of "most general solution" that can be computed is a set of unifiers called a *covering unifier set* Σ such that for any unifier σ', there exists a $\sigma \in \Sigma$ and a σ'' such that $\sigma'\mathcal{V} = \sigma''\sigma\mathcal{V}$ where $\mathcal{V}$ includes at least all of the variables in the constraint being solved.

The problem of computing covering unifier sets in System E has not been as actively researched as β-unification, however there has been some preliminary work with an algorithm called `opus` [7]. We can illustrate the main ideas of the `opus` algorithm by considering the application of the function $\lambda y.y\ y : ((f\ \gamma \to \beta) \sqcap f\ \gamma) \to \beta\lhd$ to the argument $\lambda x.x : e\ (\alpha \to \alpha)\lhd$. This gives the constraint $e\ (\alpha \to \alpha) \doteq (f\ \gamma \to \beta) \sqcap f\ \gamma$ between the argument type and the parameter type, where no special role is assumed of either the left or the right types.

At each step of unification, the search path may be branched if multiple but unrelated ways to construct a unifier are found. Not all of these branches may lead to a unifier, and only those that do will eventually be added to the covering set. At the very outset, our example constraint can be interpreted in 3 different ways due to System E's type equalities:

1. $e\ (\alpha \to \alpha) \doteq (f\ \gamma \to \beta) \sqcap f\ \gamma$
2. $\omega \sqcap e\ (\alpha \to \alpha) \doteq (f\ \gamma \to \beta) \sqcap f\ \gamma$
3. $e\ (\alpha \to \alpha) \sqcap \omega \doteq (f\ \gamma \to \beta) \sqcap f\ \gamma$

The second interpretation will not lead to a unifier since no substitution will make ω equal to a function type or vice versa. However, the third interpretation will lead to a unifier. Since each step of unification is potentially a branching step, there will of course be too many branches to illustrate all of them, and so we will illustrate only the first branch at each branch point.

Continuing, opus identifies two possible ways to construct a unifier for $e\ (\alpha \to \alpha) \doteq (f\ \gamma \to \beta) \cap f\ \gamma$:

1. $\alpha' \to \alpha' \doteq (f\ \gamma \to \beta) \cap f\ \gamma$ (Using substitution $e := (\alpha := \alpha')$)
2. $e_1\ (\alpha \to \alpha) \cap e_2\ (\alpha \to \alpha) \doteq (f\ \gamma \to \beta) \cap f\ \gamma$ (Using substitution $e := (e_1\ \boxdot \cap e_2\ \boxdot)$)

Note that β-unification never branches the search path and will in the above case always expand e into an intersection type, choosing path (2). opus, aiming to find a most general representation of all possible unifiers, must search both paths. In this walkthrough, we will again follow the first branch. After taking this branch, we find once more that it is necessary to consider different interpretations of the constraint under the type equality rules:

1. $\alpha' \to \alpha' \doteq (f\ \gamma \to \beta) \cap f\ \gamma$
2. $\omega \cap (\alpha' \to \alpha') \doteq (f\ \gamma \to \beta) \cap f\ \gamma$
3. $(\alpha' \to \alpha') \cap \omega \doteq (f\ \gamma \to \beta) \cap f\ \gamma$

The first interpretation leads to a dead end because a function type cannot be unified with an intersection type. The second interpretation also leads to a dead end because ω cannot be unified with the function type $f\ \gamma \to \beta$. But the third interpretation can be unified, and we will follow this branch. The left component of the intersection can be unified as follows:

$$(f\ \gamma \to f\ \gamma) \cap \omega \doteq (f\ \gamma \to f\ \gamma) \cap f\ \gamma \quad \text{(Using substitution } \alpha' := f\ \gamma, \beta := f\ \gamma\text{)}$$

The right component of the intersection can, however, be unified in two different ways:

1. $(\omega \to \omega) \cap \omega \doteq (\omega \to \omega) \cap \omega$ (Using substitution $f := \omega$)
2. $(\omega \to \omega) \cap \omega \doteq (\omega \to \omega) \cap \omega$ (Using substitution $f := (\gamma := \omega)$)

Note that despite resulting in the same typings, neither of the two *unifiers* is more general than the other, and so both are added to the covering set, along with all of the other paths that we skipped (excluding, of course, those paths that would lead to dead ends). opus is not perfect, and may potentially add redundant unifiers to the covering set. However, even if minimal covering unifier sets were produced, it is questionable whether all of the additional unifiers found (such as the two above that produced the exact same result) would be useful in practice. Furthermore, it should be clear that the amount of branching required to search all the necessary paths makes this approach extremely inefficient (see Section 3.6 and Section 5.5).

2.4.3 Comparison of β-Unification and Covering Unification

When we consider only the pure λ-terms, type inference based on β-unification is preferable because it is hardwired to use knowledge of β-reduction to choose a single path of reduction leading to a single unifier, and this has clear efficiency benefits. In comparison, covering unification—lacking any such hint from β-reduction—must explore all possible alternative paths leading to a *covering set* of unifiers, a process that is far less efficient.

However, when we extend the set of terms beyond those of the pure λ-calculus and extend the reduction rules beyond simple β-reduction, there appears to be more incentive to study covering unification. In the case of System $\mathrm{E}^{\mathrm{vcr}}$ presented in Chapter 5, we add support for extensible records by adding new term syntax, new term reduction rules and new typing rules, but using largely the *same* type syntax. The advantage of designing System $\mathrm{E}^{\mathrm{vcr}}$ in this way is that we can use a covering unification algorithm such as opus largely without modification. This is possible because covering unification supports all type constraint forms, regardless of the term language used by the underlying calculus.

The same cannot be said about β-unification because it is hardwired to deal only with the sorts of constraints that are generated by programs in the λ-calculus. In particular, constraints of the form $e\ (S \to T) \leq U$ are supported in β-unification, because the type $e\ (S \to T)$ matches its expectation of what a function type looks like, *assuming* that all functions are λ-abstractions. But if some functions are actually extensible records, as may be the case in System $\mathrm{E}^{\mathrm{vcr}}$, then the unification algorithm needs to expect that function types may also be in the form of intersections of function types, leading to constraints of the form $e_1\ (S_1 \to S_2) \sqcap e_2\ (T_1 \to T_2) \leq U$. Such constraints are not supported by β-unification, but are supported by covering unification for the simple reason that *everything* is supported by covering unification. If a covering unification algorithm is used, each time a new feature is added to the term language that can be typed within the existing type language, no changes should be required for the covering unification algorithm itself.

It is therefore attractive to consider an algorithm like opus when scaling System E to support additional term forms and reduction rules. The issue remains, however, that opus is presently too inefficient to be useful in practice (see Section 3.6 and Section 5.5). According to [7], opus "provides a fairly simple starting point for investigating how to produce finite representations of all unifiers for an arbitrary constraint set" but in relation to developing efficient variants suggests "there is a lack of clear motivation for this strand [. . .] until a wider range of practical instances of β-unification problems are identified". Our efforts to scale System E to support additional term forms has broadened the potential application of unification with expansion variables, and this motivates a further look at the kind of unification approach used in opus.

For the type inference system presented this dissertation, we pursue a hybrid unification approach called opusβ which is, for the most part, based on the opus algorithm, but which also borrows efficiency ideas from β-unification. Like opus, our algorithm handles all constraint forms, not only those that arise when constraints are generated from pure λ-terms, which means that it can unify function types regardless of whether they describe λ-abstractions, or in our case, extensible records. On the other hand, like β-unification, our algorithm prunes the search path by using asymmetric constraints and assuming that the left and right types represent function arguments and parameters respectively. The resulting algorithm lacks the generality of opus, but on the other hand, provides us with a practical algorithm that can be used in our implementation to demonstrate the examples of interest presented in the introduction.

2.5 Summary

This chapter presented a formulation of System E containing only the features that will be used in the remainder of this dissertation. We discussed the two key technologies of intersection types and expansion variables. Intersection types provide polymorphism by listing the finite set of instance types at which a term is to be used. Expansion variables support compositional type inference by enabling the expression of principal typings, and provide a simpler way to reason about the expansion operation than the historical nucleus-based expansion operation. Finally, we discussed type inference and type unification, and presented two existing approaches to type unification with expansion variables: β-unification and covering unification. In the next chapter, we will build on these foundations as we begin to present the new system proposed by this dissertation.

Chapter 3

System E^{v}: The Value Restriction

System E^{v} is the first of three increments which together make up our final system, called E^{vcr}. The main purpose of System E^{v} is to lay the groundwork to support constants, which are added in Chapter 4, and extensible records, which are added in Chapter 5.

To prepare for the safe integration of constants and extensible records, System E^{v} augments System E with a *value restriction*. The idea of a value restriction was first introduced in the functional language ML to rule out unsafe applications in the presence of references [68], and is adapted in our system to rule out unsafe applications in the presence of constants and extensible records. System E^{v} also makes several simplifications to the definitions of System E in order to simplify the proofs and the implementation, and we establish that these simplifications do not in themselves restrict the set of typable terms. However, the value restriction itself does restrict the set of typable terms. For example, the divergent application $(\lambda x.x\ x)\ (\lambda x.x\ x)$ is typable in System E but due to our value restriction is not typable in System E^{v} (see Section 3.1 for a discussion).

This chapter is organised as follows. Section 3.1 gives an overview of our value restriction. Section 3.2 discusses the simplifications that have been made in System E^{v}. Section 3.3 presents System E^{v} and establishes that it is typesafe. Section 3.4 presents the type inference algorithm and establishes properties in relation to termination, correctness and principal typings. Section 3.5 demonstrates our implementation of System E^{v}'s type inference algorithm on a set of examples and compares the inferred typings to the original type inference algorithm for System E. Section 3.6 compares the efficiency of opusβ and opus. Finally, Section 3.7 summarises the contributions of this chapter.

3.1 The Value Restriction

If constants and extensible records are introduced into the term language, it becomes possible to express certain applications that are not reducible, and it is the job of the type system to rule out these applications. For example, the constant application 3 `false` should be rejected since 3 is not a function over booleans. Also, the field selection $(\mathsf{a} \to 3 \sqcap \mathsf{b} \to \mathtt{false})\ \mathsf{c}$ should be rejected since the field c is not present in the record $(\mathsf{a} \to 3 \sqcap \mathsf{b} \to \mathtt{false})$. However, both of these applications would be typable by Rule (omega) if System E were naively extended to support these terms, and therefore such a system would not be typesafe.

Another problem occurs when typing the selection of an unknown field represented by a variable x in an extensible record, such as $(\mathsf{a} \to 3 \sqcap \mathsf{b} \to \mathtt{false})\ x$. Now, there are two unrelated assumptions that we could make about the term variable x: either x is the field labeled a and so the application has type `Int`, or x is the field labeled b and so the application has type `Bool`. By Rule (int), we could conclude that if x is both a and b, then the application has both type `Int` and `Bool`. This typing gives x an empty type that has no permissible values. That is, there is no value that could ever be substituted for x that would be equal to both the label a and the label b, making the typing a meaningless one.

System E^{v} addresses both issues by restricting uses of the expansion typing rules to non-application terms, which we call *values*. This "value restriction" is similar to the value restriction for ML proposed by Wright [68] to maintain type safety in the presence of application terms that manipulate state. Wright's value restriction, also known as *value polymorphism*, applies specifically to the typing derivation rule associated with polymorphism (the "let" rule), while our value restriction applies more generally to the family of rules associated with expansion, namely, rules (omega), (int) and (evar). It is the restriction to Rule (omega) that addresses the first problem of unsafe typings, and the restricton to Rule (int) that addresses the second problem of empty types. The restriction must also apply to Rule (evar) in order for the family of expansion operations to work correctly together. Although this dissertation does not cover state, our value restriction could theoretically be used as a foundation for studying state with expansion variables, since it also restricts polymorphism to values.

The value restriction results in fewer terms being typable compared with System E. For example, in systems that lack Rule (omega), divergent terms with infinite reduction paths, such as $(\lambda x.x\ x)\ (\lambda x.x\ x)$, are not typable [23]. Since System E^v does not allow Rule (omega) to be used with applications, then divergent *applications* are not typable in System E^v. In contrast, System E makes all terms typable via Rule (omega).

3.2 Simplifications

This section provides an overview of the simplifications that have been made in System E^v in order to simplify the proofs and the implementation. None of these simplifications in themselves restrict the set of typable terms.

3.2.1 Type Equivalences

In System E, the set of types is initially defined by a BNF grammar and is subsequently modified by imposing type equality rules so that, for example, the $\sqcap$ operator is associative and commutative with ω as its unit. When examining the grammar without any type equality rules imposed, $T \sqcap \omega$ may be considered as a distinct type from T, whereas after imposing the type equality rules, these are considered to be one and the same type. However, this complicates certain proofs since it means that it is not possible to perform induction directly on the structure of a type.

In System E^v, the set of types is instead defined by a grammar without any type equality rules, and we define separately from this equivalence classes of types which correspond to System E's types. We use types in almost all parts of System E^v except for term contexts where we explicitly use equivalence classes instead. It turns out that this is sufficient to completely preserve the power of System E's type equalities, as we will show in Lemma 3.27 (Equivalent types). One benefit of this change is that proofs by induction on the structure of types are now straightforward.

A study of the types of System E without type equalities was first done by Kfoury and Bakewell [7] while investigating the problem of unification with E-variables, although this was done independently of any type system. System E^v is the first type system based on System E without type equalities for which subject reduction is also proved.

3.2.2 Restrictions to Simple Types

In System E$^{\mathsf{v}}$, we categorise types into two groups. Types of the form α and $S \to T$ are called *simple types* while the remaining types of the form ω, $S \sqcap T$ and $e\ T$ are called expansion types. System E offers redundant rules for introducing expansion types, both via typing derivations and via substitutions. Eliminating this redundancy can make it easier to reason about the type system, and some of this redundancy can be eliminated by restricting certain typing rules, and substitutions of type variables, to simple types.

In the case of typing rules, System E offers two ways to assign, for example, an intersection type to a term variable:

$$\dfrac{}{x : \bar{S} \sqcap \bar{T} \lhd x : \bar{S} \sqcap \bar{T}}\ \text{by (var)} \qquad \dfrac{\dfrac{}{x : \bar{S} \lhd x : \bar{S}} \quad \dfrac{}{x : \bar{T} \lhd x : \bar{T}}}{x : \bar{S} \sqcap \bar{T} \lhd x : \bar{S} \sqcap \bar{T}}\ \text{by (int)}$$

In System E$^{\mathsf{v}}$, we restrict Rule (var) to simple types so that only the second derivation is possible. By Lemma 3.26 (Variable types), no typing power is lost by this restriction since it is always possible to construct a derivation that assigns an arbitrary type to a variable by combining Rule (var) with uses of rules (omega), (int) and (evar).

In the case of substitutions, System E offers two ways to construct a substitution that transforms $e\ \alpha$ into, for example, an intersection type $\bar{S} \sqcap \bar{T}$:

$$\begin{aligned} \sigma_1 = &\quad e := (\alpha := \bar{S} \sqcap \bar{T}) \\ \sigma_2 = &\quad e := (\alpha := \bar{S}) \sqcap (\alpha := \bar{T}) \end{aligned}$$

This redundancy is eliminated in System E$^{\mathsf{v}}$ by restricting type variables to represent only simple types. As such, α is now called a *simple type variable* rather than a type variable. After this modification, only σ_2 is a valid substitution. By Corollary 3.18 (Type substitution), no expressiveness is lost by this restriction since the full power of System E type variables can still be emulated in System E$^{\mathsf{v}}$ by wrapping each simple type variable in an E-variable. Simple type variables stem from the idea of restricted substitutions[7] which were proposed as a way to simplify constraint unification by reducing the number of non-isomorphic solutions to constraints. For example, the constraint $e\ \alpha \doteq \bar{S} \sqcap \bar{T}$ has both σ_1 and σ_2 as solutions in the full System E, but if restricted substitutions are used, then σ_2 is permitted as a solution while σ_1 is not.

3.2.3 Unique Derivations

In System E, typing judgements do not, in general, have unique derivations. For example, since there is no value restriction, we cannot assume applications are always typed using Rule (app). The judgement $s\ t : S \sqcap T \lhd \Gamma_\omega$ may in fact have been derived in either of the following ways:

$$\frac{s : U \to (S \sqcap T) \lhd \Gamma_\omega \quad t : U \lhd \Gamma_\omega}{s\ t : S \sqcap T \lhd \Gamma_\omega}\ \text{by (app)} \qquad \frac{s\ t : S \lhd \Gamma_\omega \quad s\ t : T \lhd \Gamma_\omega}{s\ t : S \sqcap T \lhd \Gamma_\omega}\ \text{by (int)}$$

System E's type equality rules also make it impossible to tell from the syntax of a type which typing rule was used. For example the judgement $\lambda x.t : (S \to T) \sqcap \omega \lhd \Gamma_\omega$ may have been derived in either of the following ways

$$\frac{t : T \lhd x : S}{\lambda x.t : (S \to T) \sqcap \omega \lhd \Gamma_\omega}\ \text{by (abs)} \qquad \frac{\lambda x.t : (S \to T) \sqcap \omega \lhd \Gamma_\omega \quad \lambda x.t : \omega \lhd \Gamma_\omega}{\lambda x.t : (S \to T) \sqcap \omega \lhd \Gamma_\omega}\ \text{by (int)}$$

since the equality $S \to T = (S \to T) \sqcap \omega$ holds. Finally, System E's Rule (var) allows the assignment of any arbitrary type to a term variable, overlapping with a number of other typing rules. For example, the previous section showed that the judgement $x : \bar{S} \sqcap \bar{T} \lhd x : \bar{S} \sqcap \bar{T}$ can be derived in either of the following two ways:

$$\dfrac{}{x : \bar{S} \cap \bar{T} \lhd x : \bar{S} \cap \bar{T}}\ \text{by (var)} \qquad \dfrac{\dfrac{}{x : \bar{S} \lhd x : \bar{S}} \quad \dfrac{}{x : \bar{T} \lhd x : \bar{T}}}{x : \bar{S} \cap \bar{T} \lhd x : \bar{S} \cap \bar{T}}\ \text{by (int)}$$

System E$^{\mathsf{v}}$, on the other hand, has a unique derivation for each derivable judgement. In the first example above, we know that Rule (app) was used and not Rule (int) because the value restriction prevents Rule (int) from being used on applications. In the second example above, we know that Rule (int) was used and not Rule (abs) since $(S \to T) \cap \omega$ is an intersection type which is never equal to the function type $S \to T$ (although the equivalence $(S \to T) \cap \omega \equiv S \to T$ does hold). And in the third example above, we know that Rule (int) was used and not Rule (var) since Rule (var) is restricted to simple types and may not be used to directly assign an intersection type.

This property can be exploited in proofs since it is often useful to infer the form of the premises based on the derived judgement. For an example of this, the reader is referred to propositions (7) to (9) in the (app) case of the proof of Lemma 3.29 (Term substitution).

3.3 Type System

System E$^{\mathsf{v}}$ will now be defined formally, beginning with the term language, then the type language and finally the typing derivation rules.

3.3.1 Terms and Reductions

This section defines the term language and reduction rules of System E$^{\mathsf{v}}$.

Definition 3.1 (Terms for E$^{\mathsf{v}}$)**.** Let metavariables x, y and z range over the countably infinite set of *term variables* (also called *variables*). The syntax for terms and their metavariable conventions are given below.

$$\begin{aligned} s, t, u &::= v \mid s\ t && \textbf{terms} \\ v &::= x \mid \lambda x.t && \textbf{values} \end{aligned}$$

□

A *term* is either a *value* v or an *application* $s\ t$ of some function term s to some argument term t. A value is either a term variable x or an *abstraction* $\lambda x.t$ which is a function with parameter x and body t.

Definition 3.2 (Free variables for E$^{\mathrm{v}}$). The *free variables* $\mathsf{fv}(t)$ of a term t are defined as follows:

$$\begin{aligned} \mathsf{fv}(x) &= \{x\} \\ \mathsf{fv}(\lambda x.t) &= \mathsf{fv}(t)\backslash\{x\} \\ \mathsf{fv}(s\ t) &= \mathsf{fv}(s) \cup \mathsf{fv}(t) \end{aligned}$$

□

Definition 3.3 (Term substitution for E$^{\mathrm{v}}$). A *term substitution* $t[x := v]$ of value v for x in t is defined by the following rules:

$$\begin{aligned} x[x := v] &= v \\ y[x := v] &= y && y \neq x \\ (\lambda x.t)[x := v] &= \lambda x.t \\ (\lambda y.t)[x := v] &= \lambda z.t[y := z][x := v] && y \neq x, z \notin \mathsf{fv}(v) \cup \{x\} \\ (s\ t)[x := v] &= s[x := v]\ t[x := v] \end{aligned}$$

□

Note that a term substitution may substitute only *values* for variables, and not arbitrary terms. This definition is sufficient for the call-by-value semantics that we use, but it will also be required to support the extended definition of term substitution introduced in Chapter 5 for extensible records.

Definition 3.4 (α-conversion for E$^{\mathrm{v}}$)**.** We consider $\lambda x.t$ and $\lambda y.t[x := y]$ to be the same under *α-conversion* if $y \notin \mathsf{fv}(t)$. □

The reduction rules follow a call-by-value evaluation strategy.

Definition 3.5 (Reduction for E$^{\mathrm{v}}$)**.** The *reduction rules* are defined as follows:

$$\begin{array}{ll} (\lambda x.t)\ v > t[x := v] & \\ t\ s > t'\ s & \text{if } t > t' \\ v\ t > v\ t' & \text{if } t > t' \end{array}$$

□

3.3.2 Types and Expansions

This section defines the types, expansions and associated operations.

Definition 3.6 (Syntax for E$^{\mathrm{v}}$)**.** Let metavariables e, f, g range over the countably infinite set of *E-variables* and let metavariables α, β, γ range over the countably infinite set of *simple type variables.* The syntactic categories and their metavariable conventions are given below.

$$\begin{array}{rll} S, T, U ::= & \bar{T} \mid \mathbb{T} & \textbf{types} \\ \bar{S}, \bar{T}, \bar{U} ::= & \alpha \mid S \to T & \textbf{simple types} \\ \mathbb{S}, \mathbb{T}, \mathbb{U} ::= & \omega \mid S \sqcap T \ \mid e\ T & \textbf{expansion types} \\ E, F, G ::= & \omega \mid E \sqcap F \mid e\ E \mid \sigma & \textbf{expansions} \\ \sigma ::= & \boxdot \mid \sigma, \alpha := \bar{T} \mid \sigma, e := E & \textbf{substitutions} \\ X ::= & \alpha \mid e & \textbf{variables} \\ K ::= & T \mid E & \textbf{entities} \end{array}$$

□

A *type* is either a simple type or an expansion type. A *simple type* is either a *simple type variable* α or a *function type* $S \to T$. An *expansion type* is either the *omega type* ω, an *intersection type* $S \sqcap T$ or an *E-variable application type* $e\ T$. An *expansion* is either the *omega expansion* ω, an *intersection expansion* $E \sqcap F$, an *E-variable application expansion* $e\ E$ or a *substitution* σ. A substitution is either the *identity substitution* $\boxdot$, a substitution extended with an assignment $\alpha := \bar{T}$ of simple type $\bar{T}$ to simple type variable α, or a substitution extended an assignment $e := E$ of expansion E to the E-variable e. A *variable* X refers to either a simple type variable or an E-variable. An *entity* K refers to either a type or an expansion.

Note that a simple type may still contain intersection types, E-variable application types and ω in nested positions, but not at the top level. For example, $(S \sqcap T) \to \omega$ is a simple type, while $S \sqcap T$ and ω are themselves not simple types.

Definition 3.7 (Type equivalence for E$^{\mathsf{v}}$)**.** For two types S and T, the proposition $S \equiv T$ asserts that S and T are *equivalent types.* The *equivalence class* $\langle T \rangle$ is the set of all types that are equivalent to T. The type equivalence relation $\equiv$ is defined by the following rules:

Axioms

$$(S \sqcap T) \sqcap U \equiv S \sqcap (T \sqcap U)$$
$$S \sqcap T \equiv T \sqcap S$$
$$T \sqcap \omega \equiv T$$
$$e\ \omega \equiv \omega$$
$$e\ (S \sqcap T) \equiv e\ S \sqcap e\ T$$

Structural congruence

$$S \to T \equiv S' \to T' \qquad \text{if } S \equiv S' \text{ and } T \equiv T'$$
$$S \sqcap T \equiv S' \sqcap T' \qquad \text{if } S \equiv S' \text{ and } T \equiv T'$$
$$e\ T \equiv e\ T' \qquad \text{if } T \equiv T'$$

Equivalence

$$T \equiv T$$
$$T \equiv T' \qquad \text{if } T' \equiv T$$
$$T \equiv T'' \qquad \text{if } T \equiv T' \text{ and } T' \equiv T''$$

□

In Definition 3.7, the 5 axioms correspond to the type equality rules of System E (see Figure 2.1), The structural congruence rules extend the rules to nested types, while the equivalence rules define reflexivity, symmetry and transitivity, giving the required properties for an equivalence relation.

Definition 3.8 (Intersection components for E$^{\mathrm{v}}$)**.** For any simple type $\bar{S}$ and type T, the proposition $\bar{S} \Subset T$ asserts that $\bar{S}$ is an *intersection component* of T, and is defined by the following rules:

$$\begin{array}{ll} \bar{S} \Subset \bar{S} & \\ \bar{S} \Subset T_1 \Cap T_2 & \text{if } \bar{S} \Subset T_1 \text{ or } \bar{S} \Subset T_2 \end{array}$$

$\bar{S} \not\Subset T$ asserts that $\bar{S}$ is *not* an intersection component of T. □

With the exception of function types, for any given simple types $\bar{S}$ and $\bar{T}$ to be equivalent, they must be one and the same type. This is established by Lemma 3.9 and Corollary 3.10 below.

Lemma 3.9 (Intersection components for E$^{\mathrm{v}}$)**.** *Given any S, T, $\bar{S}$ and $\bar{T}$, if $\bar{S}$ is not a function type and $\bar{S} \Subset S$ and $S \equiv T$, then $\bar{S} \Subset T$.*

Proof. The proof is by induction on the structure of the derivation of $S \equiv T$.

case: $S = (T_1 \Cap T_2) \Cap T_3 \equiv T_1 \Cap (T_2 \Cap T_3) = T$. Then $\bar{S} \Subset S$ implies $\bar{S} \Subset T_1$ or $\bar{S} \Subset T_2$ or $\bar{S} \Subset T_3$ which implies $\bar{S} \Subset T$.

case: $S = T_1 \Cap T_2 \equiv T_2 \Cap T_1 = T$. Then $\bar{S} \Subset S$ implies $\bar{S} \Subset T_1$ or $\bar{S} \Subset T_2$ which implies $\bar{S} \Subset T_2 \Cap T_1$.

case: $S = T_1 \Cap \omega \equiv T_1 = T$. Then $\bar{S} \Subset S$ implies $\bar{S} \Subset T_1$ or $\bar{S} \Subset \omega$. By the definition of $\Subset$, it is the case that $\bar{S} \notin \omega$, therefore $\bar{S} \Subset T_1$.

case: $S = e\ \omega \equiv \omega = T$. This case does not occur since $\bar{S} \not\Subset S$ by the definition of $\Subset$.

case: $S = e\ (T_1 \Cap T_2) \equiv e\ T_1 \Cap e\ T_2$. This case does not occur since $\bar{S} \not\Subset S$ by the definition of $\Subset$.

case: $S = S_1 \to S_2 \equiv T_1 \to T_2 = T$, if $S_1 \equiv T_1$ and $S_2 \equiv T_2$. This case does not occur since the assumption is that $\bar{S}$ is not a function type.

case: $S = S_1 \cap S_2 \equiv T_1 \cap T_2$ if $S_1 \equiv T_1$ and $S_2 \equiv T_2$. If $\bar{S} \Subset S$, then $\bar{S} \Subset S_1$ or $\bar{S} \Subset S_2$ which by ind.hyp. implies $\bar{S} \Subset T_1$ or $\bar{S} \Subset T_2$, which by the definition of $\Subset$ implies $\bar{S} \Subset T_1 \cap T_2$.

case: $S = e\ T \equiv e\ T'$ if $T \equiv T'$. This case does not occur since $\bar{S} \not\Subset S$ by the definition of $\Subset$.

case: $S = T_1 \equiv T_1 = T$. Immediate.

case: $S = T_1 \equiv T_2 = T$ if $T_2 \equiv T_1$. Consider the mirrors of the asymmetric rules:

- **case:** $S = T_1 \cap (T_2 \cap T_3) \equiv (T_1 \cap T_2) \cap T_3 = T$. Then $\bar{S} \Subset S$ implies $\bar{S} \Subset T_1$ or $\bar{S} \Subset T_2$ or $\bar{S} \Subset T_3$ which implies $\bar{S} \Subset T$.
- **case:** $S = T_1 \equiv T_1 \cap \omega = T$. If $\bar{S} \Subset T_1$ then $\bar{S} \Subset T_1 \cap \omega$ by the definition $\Subset$.
- **case:** $S = \omega \equiv e\ \omega = T$. This case does not occur since $\bar{S} \not\Subset S$ by the definition of $\Subset$.
- **case:** $S = e\ T_1 \cap e\ T_2 \equiv e\ (T_1 \cap T_2)$. Then $\bar{S} \Subset S$ implies $\bar{S} \Subset e\ T_1$ or $\bar{S} \Subset e\ T_2$, neither of which can be true by the definition of $\Subset$ and so this case does not occur.

case: $S = T_1 \equiv T_3 = T$ if $T_1 \equiv T_2$ and $T_2 \equiv T_3$. If $\bar{S} \Subset T_1$, then by the induction hypothesis, $\bar{S} \Subset T_2$, and then by the induction hypothesis, $\bar{S} \Subset T_3$.

□

Corollary 3.10 (Simple type equivalence for E$^\mathsf{v}$)**.** *Given any $\bar{S}$ and $\bar{T}$, if $\bar{S}$ is not a function type and $\bar{S} \equiv \bar{T}$ then $\bar{S} = \bar{T}$.*

Proof. By Lemma 3.9, since $\bar{S}$ is not a function type and $\bar{S} \Subset \bar{S}$ and $\bar{S} \equiv \bar{T}$, then $\bar{S} \Subset \bar{T}$, which by Definition 3.8 implies that $\bar{S} = \bar{T}$. □

Expansion application is defined in System E$^\mathsf{v}$ by exactly the same rules as those for System E presented in the previous chapter, except that each α should be interpreted as a simple type variable rather than a type variable.

Definition 3.11 (Expansion application for E$^\mathsf{v}$)**.** The rules for applying expansions to expansions, types and E-variables are as follows:

$$
\begin{array}{llll}
\omega\ K & = \omega & \sigma\ \omega & = \omega \\
(E \mathbin{\dot\cap} F)\ K & = E\ K \mathbin{\dot\cap} F\ K & \sigma\ (K_1 \mathbin{\dot\cap} K_2) & = \sigma\ K_1 \mathbin{\dot\cap} \sigma\ K_2 \\
(e\ E)\ K & = e\ (E\ K) & \sigma\ (e\ K) & = (\sigma\ e)\ K \\
\boxdot\ \ e & = e\ \boxdot & \sigma\ (S \to T) & = \sigma\ S \to \sigma\ T \\
\boxdot\ \ \alpha & = \alpha & \sigma\ \boxdot & = \sigma \\
(\sigma, X := K)\ X & = K & \sigma\ (\sigma', X := K) & = \sigma\sigma', X := \sigma\ K \\
(\sigma, X := K)\ X' & = \sigma\ X' \text{ if } X \neq X' & &
\end{array}
$$

□

Lemma 3.12 (Expansion application preserves syntactic categories for E$^{\mathrm{v}}$). *Given any E, F, e, σ_1, σ_2, σ and $\bar{T}$, then (1) $E\ F$ is an expansion, (2) $E\ e$ is an expansion, (3) $E\ T$ is a type, (4) $\sigma_1\ \sigma_2$ is a substitution, and (5) $\sigma\ \bar{T}$ is a simple type.*

Proof. Trivial, by induction on the structure of E then E_1 in (1), by induction on the structure of E in (2) and (3), by induction on the structure of σ_1 then σ_2 in (4), and by induction on the structure of σ then $\bar{T}$ in (5), with the cases proceeding according rules given in Definition 3.11. □

Lemma 3.13 ($\boxdot$ acts as the identity for E$^{\mathrm{v}}$). *Given any K, then $\boxdot\ K = K$.*

Proof. Trivial by induction on the structure of K. □

Lemma 3.14 (Expansion composition for E$^{\mathrm{v}}$). *Given any E, F and K, $E\ (F\ K) = (E\ F)\ K$.*

Proof. The proof is similar to the one in [17], except that since we do not impose equalities on types, induction is directly possible on the structure of K, and there is no need to define a size function and prove an induction principle. A full proof is given in Appendix A. □

Expansion composition is useful in constructing single substitutions that carry out the same work as a sequence of substitutions. For example, the substitution $\sigma_4\sigma_3\sigma_2\sigma_1$ has the same effect as applying the individual substitutions $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ in sequence.

The following table illustrates the application of several different expansions to the type $e\ \alpha \to e\ \alpha$:

Expansion	**Resulting type**
$f\ (e := (\alpha := \bar{T}))$	$f\ (\bar{T} \to \bar{T})$
$e := f\ (\alpha := \bar{T})$	$f\ \bar{T} \to f\ \bar{T}$
$(e{:=}(\alpha{:=}\bar{S})) \cap (e{:=}(\alpha{:=}\bar{T}))$	$(\bar{S} \to \bar{S}) \cap (\bar{T} \to \bar{T})$
$(e{:=}(\alpha{:=}\bar{T})) \cap \omega$	$(\bar{T} \to \bar{T}) \cap \omega$

The last example can be derived using the following steps:

$$\begin{aligned}
((e_1{:=}(\alpha{:=}\bar{T})) \cap \omega)\ (e_1\alpha \to e_1\alpha) &= (e_1{:=}(\alpha{:=}\bar{T}))\ (e_1\alpha \to e_1\alpha) \cap \omega\ (e_1\alpha \to e_1\alpha)\\
&= (e_1{:=}(\alpha{:=}\bar{T}))\ (e_1\alpha \to e_1\alpha) \cap \omega\\
&= ((e_1{:=}(\alpha{:=}\bar{T}))\ (e_1\alpha) \to (e_1{:=}(\alpha{:=}\bar{T}))\ (e_1\alpha)) \cap \omega\\
&= (((e_1{:=}(\alpha{:=}\bar{T}))\ e_1)\ \alpha \to ((e_1{:=}(\alpha{:=}\bar{T}))\ e_1)\ \alpha) \cap \omega\\
&= ((\alpha{:=}\bar{T})\ \alpha \to (\alpha{:=}\bar{T})\ \alpha) \cap \omega\\
&= (\bar{T} \to \bar{T}) \cap \omega
\end{aligned}$$

The following two lemmas show that expansion applications preserve type equivalence.

Lemma 3.15 (Expansion distribution equivalence for E$^{\mathsf{v}}$)**.** *Given any E, S and T, then $E\ (S \cap T) \equiv E\ S \cap E\ T$.*

Proof. The proof is by induction on the structure of E.

case: $E = \omega$.

$$\begin{aligned}
\text{LHS} &= \omega\ (T_1 \cap T_2) &&\\
&= \omega && \text{def 3.11}\\
&\equiv \omega \cap \omega && \text{def 3.7}\\
&= \omega\ T_1 \cap \omega\ T_2 && \text{def 3.11}\\
&= \text{RHS}
\end{aligned}$$

case: $E = E_1 \cap E_2$.

$$\begin{aligned}
\text{LHS} &= (E_1 \cap E_2)\ (T_1 \cap T_2) \\
&= E_1\ (T_1 \cap T_2) \cap E_2\ (T_1 \cap T_2) && \text{def 3.11} \\
&\equiv (E_1\ T_1 \cap E_1\ T_2) \cap (E_2\ T_1 \cap E_2\ T_2) && \text{ind.hyp., def 3.7} \\
&\equiv (E_1\ T_1 \cap E_2\ T_1) \cap (E_1\ T_2 \cap E_2\ T_2) && \text{def 3.7} \\
&= (E_1 \cap E_2)\ T_1 \cap (E_1 \cap E_2)\ T_2 && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $E = e\ E_1$.

$$\begin{aligned}
\text{LHS} &= (e\ E_1)\ (T_1 \cap T_2) \\
&= e\ (E_1\ (T_1 \cap T_2)) && \text{def 3.11} \\
&\equiv e\ (E_1\ T_1 \cap E_1\ T_2) && \text{ind.hyp., def 3.7} \\
&\equiv e\ (E_1\ T_1) \cap e\ (E_1\ T_2) && \text{def 3.7} \\
&= (e\ E_1)\ T_1 \cap (e\ E_1)\ T_2 && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $E = \sigma$.

$$\begin{aligned}
\text{LHS} &= \sigma\ (T_1 \cap T_2) \\
&= \sigma T_1 \cap \sigma T_2 && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

□

Lemma 3.16 (Expansion preserves type equivalence for E$^{\text{v}}$). *Given any T, T' and E, if $T \equiv T'$, then $E\ T \equiv E\ T'$.*

Proof. The proof is by induction on the structure of the derivation of $T \equiv T'$, and then the structure of E, enumerating first over the cases of E and when $E = \sigma$, then enumerating over the cases of $T \equiv T'$.

case: $E = \omega$.

$$\begin{aligned}
\text{LHS} &= \omega\ T \\
&= \omega && \text{def 3.11} \\
&= \omega\ T' && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $E = E_1 \cap E_2$.

$$\begin{aligned}
\text{LHS} &= (E_1 \cap E_2)\ T \\
&= E_1\ T \cap E_2\ T && \text{def 3.11} \\
&\equiv E_1\ T' \cap E_2\ T' && \text{ind.hyp. } (E_1 \text{ smaller}), \text{ ind.hyp. } (E_2 \text{ smaller}), \text{ def 3.7} \\
&= (E_1 \cap E_2)\ T' && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $E = e\ E_1$.

$$\begin{aligned}
\text{LHS} &= (e\ E_1)\ T \\
&= e\ (E_1\ T) && \text{def 3.11} \\
&\equiv e\ (E_1\ T') && \text{ind.hyp. } (E_1 \text{ smaller}), \text{ def 3.7} \\
&= (e\ E_1)\ T' && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $E = \sigma$. We now consider the cases for the derivation of $T \equiv T'$. Here, the cases are straightforward applications of Definition 3.11 and the induction hypotheses. The only tricky case is the one for a substitution applied to an E-variable application:

case: If $T_1 \equiv T_1'$, then $e\ T_1 \equiv e\ T_1'$.

$$\begin{aligned}
\text{LHS} &= \sigma\ (e\ T_1) \\
&= (\sigma\ e)\ T_1 && \text{def 3.11} \\
&\equiv (\sigma\ e)\ T_1' && \text{ind.hyp. } (T_1 \equiv T_1' \text{ smaller}), \text{ def 3.7} \\
&= \sigma\ (e\ T_1') && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

In this case, we do not know that $\sigma\ e$ is smaller than σ, but we rely on the fact that the derivation of $T_1 \equiv T_1'$ is smaller than the derivation of $e\ T_1 \equiv e\ T_1'$, and perform induction *first* on the structure of the derivation of this type equivalence and then *second* on the structure of the expansion E.

A full proof of the other cases is included in Appendix A.

□

The final lemma and corollary in this section establish that, although only simple types may be substituted for simple type variables, the full power of System E type variables can be emulated by wrapping each simple type in an E-variable.

Lemma 3.17 (Type variables for $\mathrm{E^v}$)**.** *Given T and α, there exists an expansion E such that $E\ \alpha = T$.*

Proof. The proof is by induction on the structure of T.

case: $T = \omega$. Then $E = \omega$.

case: $T = T_1 \cap T_2$. Then (1) $\exists E_1.\quad E_1\ \alpha = T_1$ by ind.hyp. (T_1 smaller) and (2) $\exists E_2.\quad E_2\ \alpha = T_2$ by ind.hyp. (T_2 smaller). Then $E = E_1 \cap E_2$ since $(E_1 \cap E_2)\ \alpha = E_1\ \alpha \cap E_2\ \alpha$ by def 3.11 which equals $T_1 \cap T_2$ by (1) and (2).

case: $T = e\ T_1$. Then (1) $\exists E_1.\quad E_1\ \alpha = T_1$ by ind.hyp. (T_1 smaller). Therefore $E = e\ E_1$ since $(e\ E_1)\ \alpha = e\ (E_1\ \alpha)$ by def 3.11 which equals $e\ T_1$ by (1)

case: $T = \bar{T}$. Then $E = (\alpha := \bar{T})$. □

Corollary 3.18 (Type substitution for $\mathrm{E^v}$)**.** *Given any T, e and α, there exists a substitution σ such that $\sigma\ (e\ \alpha) = T$.*

Proof. Let E be the expansion from Lemma 3.17 such that $E\ \alpha = T$. Then $\sigma = (e := E)$. □

3.3.3 Typing Derivations

This section defines the typing derivations of System $\mathrm{E^v}$. These definitions very closely match with their counterparts in System E presented in the previous chapter and so little further explanation will be required, except in relation to the introduction of simple types and the value restriction, and also the replacement of System E's type equalities by explicit type equivalences. In particular, the last change means that the following definitions will make explicit use of type equivalences where needed.

Definition 3.19 (Term contexts for $\mathrm{E^v}$)**.** A *term context* Γ is a total function from term variables to equivalence classes of types, mapping only a finite number of term variables to equivalence classes other than $\langle\omega\rangle$. The notation $x_1 : \langle T_1\rangle, \ldots, x_n : \langle T_n\rangle$ is used to represent a term context where each term variable not explicitly given is assumed to be mapped to $\langle\omega\rangle$. We use Γ_ω to denote the term context in which all term variables are mapped to $\langle\omega\rangle$. □

A key point about the above modified definition of term contexts is that explicit uses of type equivalence classes are needed to give the same meaning as System E.

Definition 3.20 (Operations on term contexts for E$^{\mathrm{v}}$)**.** Intersection introduction, E-variable application and expansion application are extended to term contexts as follows:

$$\begin{aligned}(\Gamma_1 \sqcap \Gamma_2)(x) &= \Gamma_1(x) \sqcap \Gamma_2(x)\\ (e\ \Gamma)(x) &= e\ \Gamma(x)\\ (E\ \Gamma)(x) &= E\ \Gamma(x)\end{aligned}$$

□

Observe that the type equivalences from Definition 3.7 also imply corresponding equalities on term contexts:

$$\begin{aligned}(\Gamma_1 \sqcap \Gamma_2) \sqcap \Gamma_3 &= \Gamma_1 \sqcap (\Gamma_2 \sqcap \Gamma_3)\\ \Gamma_1 \sqcap \Gamma_2 &= \Gamma_2 \sqcap \Gamma_1\\ \Gamma \sqcap \Gamma_\omega &= \Gamma\\ e\ \Gamma_\omega &= \Gamma_\omega\\ e\ (\Gamma_1 \sqcap \Gamma_2) &= e\ \Gamma_1 \sqcap e\ \Gamma_2\end{aligned}$$

Definition 3.21 (Typings for E$^{\mathrm{v}}$)**.** A *typing* $T \lhd \Gamma$ (metavariable τ) is a pair of a type T and term context Γ. □

Definition 3.22 (Operations on typings for E$^{\mathrm{v}}$)**.** Intersection introduction, E-variable application and expansion application are extended to typings as follows:

$$(T_1 \lhd \Gamma_1) \Cap (T_2 \lhd \Gamma_2) = T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2$$

$$e\ (T \lhd \Gamma) = e\ T \lhd e\ \Gamma$$

$$E\ (T \lhd \Gamma) = E\ T \lhd E\ \Gamma$$

□

Definition 3.23 (Typing judgements for E$^\mathsf{v}$). A *typing judgement* $t : T \lhd \Gamma$ (metavariable J) asserts that term t has typing $T \lhd \Gamma$, or equivalently that term t has type T in term context Γ. □

Definition 3.24 (Typing derivations for E$^\mathsf{v}$). A *typing derivation* (metavariable D) is a proof tree of a typing judgement. The rules for deriving valid typing judgements are as follows:

$$\text{(var)}\ \frac{}{x : \bar{T} \lhd x : \langle \bar{T} \rangle} \qquad \text{(omega)}\ \frac{}{v : \omega \lhd \Gamma_\omega}$$

$$\text{(abs)}\ \frac{t : T \lhd \Gamma, x : \langle S \rangle}{\lambda x.t : S \to T \lhd \Gamma} \qquad \text{(int)}\ \frac{v : \tau_1 \quad v : \tau_2}{v : \tau_1 \Cap \tau_2}$$

$$\text{(app)}\ \frac{t : S \to T \lhd \Gamma_1 \quad s : S \lhd \Gamma_2}{t\ s : T \lhd \Gamma_1 \Cap \Gamma_2} \qquad \text{(evar)}\ \frac{v : \tau}{v : e\ \tau}$$

□

These rules differ from System E's rules in three ways.

1. The value restriction is implemented for the rules associated with expansion. These rules are (omega), (int) and (evar).
2. Rule (var) is restricted to simple types. This forces all typings of the form $S \Cap T \lhd \Gamma$ to be derived by Rule (int), all typings of the form $e\ T \lhd \Gamma$ to be derived by Rule (evar) and all typings of the form $\omega \lhd \Gamma$ to be derived by Rule (omega).
3. Equivalence classes of types are used explicitly within term contexts. This usage recovers expressiveness that would otherwise be lost as a consequence of removing System E's type equalities. As an example, the judgement $x : T \Cap S \lhd x : S \Cap T$, which is derivable in System E, would no

longer be derivable without the type equality $S \cap T = T \cap S$. In System E$^{\text{v}}$, the corresponding judgement $x : T \cap S \lhd x : \langle S \cap T \rangle$ is derivable because of the explicit use of the equivalence class $\langle S \cap T \rangle$ in the term context.

Together, these changes give rise to the property that each typing judgement has a unique derivation.

Lemma 3.25 (Typing derivations for E$^{\text{v}}$)**.** *Each typing judgement can be the conclusion of at most one typing rule.*

Proof. Consider any application of a typing rule that concludes with the typing judgement J denoting $t : T \lhd \Gamma$. In each case, the typing rule used to derive J can be determined by examining the syntax of the term t and type T alone, and without examining the term context Γ (whose syntax cannot be examined anyway, due to its use of type equivalence classes).

case: If t is a value,

- **case:** If $T = \omega$, then J was derived by (omega).
- **case:** If $T = T_1 \cap T_2$, then J was derived by (int).
- **case:** If $T = e\ T_1$, then J was derived by (evar).
- **case:** If $T = \bar{T}$, then
  - **case:** If $t = x$, then J was derived by (var).
  - **case:** If $t = \lambda x.t_1$, then J was derived by (abs).

case: If t is an application, then J was derived by (app).

□

With regards to the second difference listed above, the following lemma establishes that $x : T \lhd x : \langle T \rangle$ still holds for any x and T, and therefore no expressiveness is lost by this restriction.

Lemma 3.26 (Variable types for E$^{\text{v}}$)**.** *Given any x and T, then $x : T \lhd x : \langle T \rangle$.*

Proof. The proof is by induction on the structure of T.

case: $T = \omega$. True by Rule (omega).

case: $T = T_1 \Cap T_2$. By the induction hypothesis, $x : T_1 \lhd x : \langle T_1 \rangle$ and $x : T_2 \lhd x : \langle T_2 \rangle$. By Rule (int), $x : T_1 \Cap T_2 \lhd x : \langle T_1 \Cap T_2 \rangle$.

case: $T = e\ T_1$. By the induction hypothesis, $x : T_1 \lhd x : \langle T_1 \rangle$. By Rule (evar), $x : e\ T_1 \lhd x : \langle e\ T_1 \rangle$.

case: $T = \bar{T_1}$. True by Rule (var).

□

With regards to the third difference listed above, the following lemma establishes that our explicit use of equivalence classes preserves the power of System E's type equalities.

Lemma 3.27 (Equivalent types for E$^{\mathrm{v}}$). *Given any t, T, T' and Γ, if $t : T \lhd \Gamma$ and $T \equiv T'$, then $t : T' \lhd \Gamma$.*

Proof. Let D be the derivation of $t : T \lhd \Gamma$. The proof is by induction on the structure of D. The cases proceed first by term form and then by the possible rules that may have been used to conclude $T \equiv T'$. We show some of the more interesting cases below, while a full proof is included in Appendix A.

case: t is v. We now consider the cases for the rule used to conclude $T \equiv T'$. These cases fall into three groups: axioms, structural congruence rules and equivalence rules.

case: Axiom $(T_1 \Cap T_2) \Cap T_3 \equiv T_1 \Cap (T_2 \Cap T_3)$.

Due to Lemma 3.25 (Typing derivations), we know that D ends with

$$\cfrac{\cfrac{(2)\ v : T_1 \lhd \Gamma_1 \quad (3)\ v : T_2 \lhd \Gamma_2}{v : T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2} \qquad (1)\ v : T_3 \lhd \Gamma_3}{v : (T_1 \Cap T_2) \Cap T_3 \lhd (\Gamma_1 \Cap \Gamma_2) \Cap \Gamma_3}$$

Then, (4) $\quad v : T_2 \Cap T_3 \lhd \Gamma_2 \Cap \Gamma_3 \qquad$ (int) with (3) and (1)

$\therefore \quad v : T_1 \Cap (T_2 \Cap T_3) \lhd \Gamma_1 \Cap (\Gamma_2 \Cap \Gamma_3) \qquad$ (int) with (2) and (4)

The cases for the other axioms are similar, using the unique derivations property to discover the premises in D.

case: Structural congruence rule $\dfrac{(1)\ T_1 \equiv T_1' \quad (2)\ T_2 \equiv T_2'}{T_1 \to T_2 \equiv T_1' \to T_2'}$

By Lemma 3.25, there are 2 possible cases for D.

case: D is the following application of Rule (var): $\overline{x : T_1 \to T_2 \lhd x : \langle T_1 \to T_2\rangle}$

$$\begin{array}{lll} \text{Then, (3)} & x : T_1' \to T_2' \lhd x : \langle T_1' \to T_2'\rangle & \text{(var)} \\ \therefore & x : T_1' \to T_2' \lhd x : \langle T_1 \to T_2\rangle & \text{def. 3.7 with (3),(1),(2)} \end{array}$$

case: D ends with the following application of Rule (abs): $\dfrac{(3)\ t_1 : T_2 \lhd \Gamma, x : \langle T_1\rangle}{\lambda x.t_1 : T_1 \to T_2 \lhd \Gamma}$

$$\begin{array}{lll} \text{Then, (4)} & t_1 : T_2' \lhd \Gamma, x : \langle T_1\rangle & \text{ind.hyp with (3) and (2)} \\ (5) & t_1 : T_2' \lhd \Gamma, x : \langle T_1'\rangle & \text{def. 3.7 with (4)} \\ & \lambda x.t_1 : T_1' \to T_2' \lhd \Gamma & \text{(abs) with (5)} \end{array}$$

The cases for the other structural congruence rules are straightforward cases of induction on the structure of D, using the unique derivations property to discover the premises in D.

case: Equivalence rule $T \equiv T$. Done.

case: Equivalence rule $\dfrac{T' \equiv T}{T \equiv T'}$.

For the symmetric rules that have the same meaning when reflected from left to right, the proofs are identical to existing cases. For the non-symmetric rules, the proofs are very similar to their existing reverse cases.

case: $\dfrac{T \equiv T'' \quad T'' \equiv T'}{T \equiv T'}$. Follows from the transitivity of $\Longrightarrow$ in the statement of the lemma.

case: t is $t_1\ t_2$. Then D ends with

$$\frac{(1)\ t_1 : T_1 \to T \lhd \Gamma_1 \quad (2)\ t_2 : T_1 \lhd \Gamma_2}{t_1\ t_2 : T \lhd \Gamma_1 \Cap \Gamma_2}\ \text{(app)}$$

$$\begin{array}{lll} \text{Then, (3)} & T \equiv T' & \text{Assumption} \\ (4) & T_1 \to T \equiv T_1 \to T' & \text{def. 3.7 with (3)} \\ (5) & t_1 : T_1 \to T' \lhd \Gamma_1 & \text{ind.hyp. with (1) and (4)} \\ \therefore & t_1\ t_2 : T' \lhd \Gamma_1 \Cap \Gamma_2 & \text{(app) with (5) and (2)} \end{array}$$

□

It is normally expected in systems with type substitutions that if t has typing τ, then by a type substitution lemma, t also has type $\sigma\tau$ for any substitution σ. This is also true in System E$^{\mathsf{v}}$, however this does not hold true for expansions in general. Due to the value restriction, it is safe for an expansion to be applied to typing judgement only if the term being judged is a value, unless of course that expansion is a substitution, in which case it can always apply. For example, if $t = (\lambda x.x)\ (\lambda x.x)$ and $t : e\alpha \to e\alpha \lhd \Gamma_\omega$, then it does *not* follow by any expansion lemma that $t : e\alpha \to e\alpha ⩀ e\alpha \to e\alpha \lhd \Gamma_\omega$ via the use of expansion $\boxdot ⩀ \boxdot$ because t is not a value.

However, this is generally not a problem if more E-variables are inserted into the original type. In this case, the following typing can be derived for t:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : e\alpha \to e\alpha \lhd e\alpha \to e\alpha}\,(\text{var})}{x : f\ (e\alpha \to e\alpha) \lhd f\ (e\alpha \to e\alpha)}\,(\text{evar})}{\lambda x.x : f\ (e\alpha \to e\alpha) \to f\ (e\alpha \to e\alpha) \lhd \Gamma_\omega}\,(\text{abs}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha \lhd \alpha}\,(\text{var})}{x : e\alpha \lhd e\alpha}\,(\text{evar})}{\lambda x.x : e\alpha \to e\alpha \lhd \Gamma_\omega}\,(\text{abs})}{\lambda x.x : f\ (e\alpha \to e\alpha) \lhd \Gamma_\omega}\,(\text{evar})}{t : f\ (e\alpha \to e\alpha) \lhd \Gamma_\omega}\,(\text{app})$$

The extra E-variable f at the top level effectively allows an expansion to be applied at the top level via a substitution, and so now it *can* follow from a substitution lemma that $t : e\alpha \to e\alpha ⩀ e\alpha \to e\alpha \lhd \Gamma_\omega$ via the substitution $f := \boxdot, (\boxdot ⩀ \boxdot)$.

The following substitution/expansion lemma establishes the applicability of substitutions and expansions to typing judgements, and is a crucial lemma for establishing principal typings for our type inference algorithm.

Lemma 3.28 (Expansion for E$^{\mathsf{v}}$). *Given any t, T, Γ and E, if $t : T \lhd \Gamma$ and if t is a value or E is a substitution, then $t : E\ T \lhd E\ \Gamma$.*

Proof. Let D be the derivation of $t : T \lhd \Gamma$. The proof is by induction on the structure of D then E.

case: E is ω, and so t is v.

(1)	$v : \omega \lhd \Gamma_\omega$	(omega)
$\therefore$	$v : \omega\ T \lhd \omega\ \Gamma$	def. 3.11 with (1)

case: E is $E_1 ⩀ E_2$, and so t is v.

(1)	$v : T \lhd \Gamma$	Assumption
(2)	$v : E_1\ T \lhd E_1\ \Gamma$	ind.hyp. on (1) with E_1 smaller
(3)	$v : E_2\ T \lhd E_2\ \Gamma$	ind.hyp. on (1) with E_2 smaller
(4)	$v : E_1\ T ⩀ E_2\ T \lhd E_1\ \Gamma ⩀ E_2\ \Gamma$	(int) on (2) and (3)
$\therefore$	$v : (E_1 ⩀ E_2)\ T \lhd (E_1 ⩀ E_2)\ \Gamma$	def. 3.11 with (4)

case: E is $e\ E_1$, and so t is v.

(1)	$v : T \lhd \Gamma$	Assumption
(2)	$v : E_1\ T \lhd E_1\ \Gamma$	ind.hyp. on (1)
(3)	$v : e\ (E_1\ T) \lhd e\ (E_1\ \Gamma)$	(evar) on (2)
$\therefore$	$v : (e\ E_1)\ T \lhd (e\ E_1)\ \Gamma$	def. 3.11 with (3)

case: E is σ. Now, t can be any term, and we consider the possibilities for D.

case: D is the following application of Rule (var):

$$\frac{}{x : T \lhd x : \langle T \rangle}$$

Then,

(1)	$x : \sigma\ T \lhd x : \langle \sigma\ T \rangle$	lem 3.26
$\therefore$	$x : \sigma\ T \lhd \sigma\ (x : \langle T \rangle)$	def 3.20 with (1)

case: D ends with an application of Rule (abs) of the form

$$\frac{(1)\ t_1 : T_2 \lhd \Gamma, x : \langle T_1 \rangle}{\lambda x.t_1 : T_1 \to T_2 \lhd \Gamma}$$

Then,

(2)	$t_1 : \sigma\ T_2 \lhd \sigma\ \Gamma, x : \langle \sigma\ T_1 \rangle$	ind.hyp. on (1)
(3)	$\lambda x.t_1 : \sigma\ T_1 \to \sigma\ T_2 \lhd \sigma\ \Gamma$	(abs) with (2)
$\therefore$	$\lambda x.t_1 : \sigma\ (T_1 \to T_2) \lhd \sigma\ \Gamma$	def. 3.11 with (3)

case: D ends with an application of Rule (app) of the form

$$\frac{(1)\ t_1 : S \to T \lhd \Gamma_1 \quad (2)\ t_2 : S \lhd \Gamma_2}{t_1\ t_2 : T \lhd \Gamma_1 \Cap \Gamma_2}$$

Then,

(3)	$t_1 : \sigma S \to \sigma T \lhd \sigma\Gamma_1$	ind.hyp. on (1)
(4)	$t_2 : \sigma S \lhd \sigma\Gamma_2$	ind.hyp. on (2)
(5)	$t_1\ t_2 : \sigma T \lhd \sigma\Gamma_1 \Cap \sigma\Gamma_2.$	(app) with (3) and (4)
$\therefore$	$t_1\ t_2 : \sigma T \lhd \sigma\ (\Gamma_1 \Cap \Gamma_2).$	def 3.11 with (5)

case: D is the following application of Rule (omega):

$$\frac{}{(1)\ v : \omega \lhd \Gamma_\omega}$$

Then,

$v : \sigma\omega \lhd \sigma\Gamma_\omega$ def 3.11 with (1)

case: D ends with an application of Rule (int) of the form

$$\frac{(1)\ v : T_1 \lhd \Gamma_1 \quad (2)\ v : T_2 \lhd \Gamma_2}{v : T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2}$$

Then,

(3)	$v : \sigma T_1 \lhd \sigma\Gamma_1$	ind.hyp. with (1)
(4)	$v : \sigma T_2 \lhd \sigma\Gamma_2$	ind.hyp. with (2)
(5)	$v : \sigma T_1 \Cap \sigma T_2 \lhd \sigma\Gamma_1 \Cap \sigma\Gamma_2$	(int) with (3) and (4)
$\therefore$	$v : \sigma\ (T_1 \Cap T_2) \lhd \sigma\ (\Gamma_1 \Cap \Gamma_2)$	def 3.11 with (5)

case: D ends with an application of Rule (evar) of the form

$$\frac{(2)\ v : T_1 \lhd \Gamma_1}{(1)\ v : e\ T_1 \lhd e\ \Gamma_1}$$

We now consider the cases for σ.

case: $\sigma = \boxdot$. True by Lemma 3.13.

case: $\sigma = \sigma', e := E_1$.

(3)	$v : E_1\ T_1 \lhd E_1\ \Gamma_1$	ind.hyp. with (2) and E_1
(4)	$v : (\sigma\ e)\ T_1 \lhd (\sigma\ e)\ \Gamma_1$	def. 3.11 with (3)
$\therefore$	$v : \sigma\ (e\ T_1) \lhd \sigma\ (e\ \Gamma_1)$	def. 3.11 with (4)

case: $\sigma = \sigma', X := K$, where $X \neq e$.

(3)	$v : \sigma'\ (e\ T_1) \lhd \sigma'\ (e\ \Gamma_1)$	ind.hyp. with (1) and σ'
$\therefore$	$v : \sigma\ (e\ T_1) \lhd \sigma\ (e\ \Gamma_1)$	def. 3.11 with (3)

□

Type safety is established by the combination of progress and subject reduction. Our progress theorem asserts that a typable closed application (with no free variables) can always be reduced by one step. Our subject reduction theorem asserts that the reduced term after one step will have the same typing as the original application. Together, these two theorems establish that evaluation of typable closed application will never become stuck.

The following lemma is crucial for establishing subject reduction and asserts that typings for terms are stable under term substitution.

Lemma 3.29 (Term substitution for E$^{\mathsf{v}}$)**.** *Given any v, t, x, S, T, Γ_1 and Γ_2, if $v : S \lhd \Gamma_2$ and $t : T \lhd \Gamma_1, x : \langle S \rangle$, then $t[x := v] : T \lhd \Gamma_1 \Cap \Gamma_2$.*

Proof. Let D be the derivation of $t : T \lhd \Gamma_1, x : \langle S \rangle$. The proof is by induction on the structure of D.

case: D ends with an application of Rule (var).

case: t is x. Then the application of Rule (var) is of the form

$$\frac{}{(1)\ x : T \lhd x : \langle S \rangle}$$

where (2) $\Gamma_1 = \Gamma_\omega$ and (3) $T = S$.

(4)	$v : S \lhd \Gamma_2$	Assumption
(5)	$v : T \lhd \Gamma_2$	(3),(4)
(6)	$v : T \lhd \Gamma_1 \Cap \Gamma_2$	(5) with (2)
$\therefore$	$x[x := v] : T \lhd \Gamma_1 \Cap \Gamma_2$	(6) with def. 3.3

case: t is y where (1) $y \neq x$. Then the application of Rule (var) is of the form

$$\frac{}{(2)\ y : T \lhd y : \langle T \rangle}$$

where (3) $\Gamma_1 = y : \langle T \rangle$ and (4) $S \equiv \omega$.

(5)	$v : S \lhd \Gamma_2$	Assumption
(6)	$v : \omega \lhd \Gamma_2$	lem. 3.27 with (5) and (4)
(7)	$\Gamma_2 = \Gamma_\omega$	(omega) with (6)
(8)	$y : T \lhd \Gamma_1$	(2) with (3)

(9)	$y : T \lhd \Gamma_1 \Cap \Gamma_2$	(8) with (7)
$\therefore$	$y[x := v] : T \lhd \Gamma_1 \Cap \Gamma_2$	def. 3.3 with (9) and (1)

case: D ends with an application of Rule (abs) of the form

$$\frac{(1)\ t_2 : T_2 \lhd \Gamma_1, x : \langle S\rangle, x_1 : \langle T_1\rangle}{\lambda x_1.t_2 : T_1 \to T_2 \lhd \Gamma_1, x : \langle S\rangle}$$

where $t = \lambda x_1.t_2$ and $T = T_1 \to T_2$.

(2)	$x_1 \neq x$	α-conversion
(3)	$x_1 \notin fv(v)$	”
(4)	$\Gamma_2(x_1) = \langle\omega\rangle$	”
(5)	$t_2[x := v] : T_2 \lhd (\Gamma_1, x_1 : \langle T_1\rangle) \Cap \Gamma_2$	ind.hyp. with (1)
(6)	$t_2[x := v] : T_2 \lhd (\Gamma_1 \Cap \Gamma_2), x_1 : \langle T_1\rangle$	(5) with (4)
(7)	$\lambda x_1.t_2[x := v] : T_1 \to T_2 \lhd \Gamma_1 \Cap \Gamma_2$	(abs) with (6)
$\therefore$	$(\lambda x_1.t_2)[x := v] : T_1 \to T_2 \lhd \Gamma_1 \Cap \Gamma_2$	def. 3.3 with (7), (2), (3)

case: D ends with an application of Rule (app) of the form

$$\frac{(1)\ t_1 : T_2 \to T \lhd \Gamma_1', x : \langle S'\rangle \quad (2)\ t_2 : T_2 \lhd \Gamma_1'', x : \langle S''\rangle}{t_1\ t_2 : T \lhd \Gamma_1, x : \langle S\rangle}$$

where $t = t_1\ t_2$, (3) $\Gamma_1 = \Gamma_1' \Cap \Gamma_1''$ and (4) $S \equiv S' \Cap S''$.

(5)	$v : S \lhd \Gamma_2$	Assumption
(6)	$v : S' \Cap S'' \lhd \Gamma_2$	lem 3.27 with (4) and (5)
	$\exists \Gamma_2', \Gamma_2''$ such that	(int) with (6)
(7)	$v : S' \lhd \Gamma_2'$	”
(8)	$v : S'' \lhd \Gamma_2''$	”
(9)	$\Gamma_2 = \Gamma_2' \Cap \Gamma_2''$	”
(10)	$t_1[x := v] : T_2 \to T \lhd \Gamma_1' \Cap \Gamma_2'$	ind.hyp with (1) and (7)
(11)	$t_2[x := v] : T_2 \lhd \Gamma_1'' \Cap \Gamma_2''$	ind.hyp with (2) and (8)

(12)	$t_1[x := v]\ t_2[x := v] : T \lhd (\Gamma'_1 \Cap \Gamma'_2) \Cap (\Gamma''_1 \Cap \Gamma''_2)$	(app) with (10) and (11)
(13)	$(t_1\ t_2)[x := v] : T \lhd (\Gamma'_1 \Cap \Gamma'_2) \Cap (\Gamma''_1 \Cap \Gamma''_2)$	def. 3.3 with (12)
$\therefore$	$(t_1\ t_2)[x := v] : T \lhd \Gamma_1 \Cap \Gamma_2$	(13) with (3) and (9)

case: D ends with an application of Rule (omega) of the form

$$\frac{}{v_1 : \omega \lhd \Gamma_\omega}$$

where $t = v_1$, $T = \omega$, (1) $\Gamma_1 = \Gamma_\omega$ and (2) $S \equiv \omega$.

(3)	$v : S \lhd \Gamma_2$	Assumption
(4)	$v : \omega \lhd \Gamma_2$	lem. 3.27 with (3) and (2)
(5)	$\Gamma_2 = \Gamma_\omega$	(omega) with (4)
(6)	$v_1[x := v]$ is a value	Observation of def 3.3
(7)	$v_1[x := v] : \omega \lhd \Gamma_\omega$	(omega) with (6)
$\therefore$	$v_1[x := v] : \omega \lhd \Gamma_1 \Cap \Gamma_2$	(7) with (1) and (5)

case: D ends with an application of Rule (int) of the form

$$\frac{(1)\ v_1 : T_1 \lhd \Gamma'_1, x : \langle S' \rangle \quad (2)\ v_1 : T_2 \lhd \Gamma''_1, x : \langle S'' \rangle}{v_1 : T_1 \Cap T_2 \lhd \Gamma_1, x : \langle S \rangle}$$

where $t = v_1$, $T = T_1 \Cap T_2$, (3) $S \equiv S' \Cap S''$ and (4) $\Gamma_1 = \Gamma'_1 \Cap \Gamma''_1$.

(5)	$v : S \lhd \Gamma_2$	Assumption
(6)	$v : S' \Cap S'' \lhd \Gamma_2$	lem. 3.27 with (5) and (3)
	$\exists \Gamma'_2, \Gamma''_2$ such that	(int) with (6)
(7)	$v : S' \lhd \Gamma'_2$	”
(8)	$v : S'' \lhd \Gamma''_2$	”
(9)	$\Gamma_2 = \Gamma'_2 \Cap \Gamma''_2$	”
(10)	$v_1[x := v] : T_1 \lhd \Gamma'_1 \Cap \Gamma'_2$	ind.hyp. with (1) and (7)
(11)	$v_1[x := v] : T_2 \lhd \Gamma''_1 \Cap \Gamma''_2$	ind.hyp. with (2) and (8)

(12)	$v_1[x := v] : T_1 \Cap T_2 \lhd (\Gamma_1' \Cap \Gamma_2') \Cap (\Gamma_1'' \Cap \Gamma_2'')$	(int) with (10) and (11)
$\therefore$	$v_1[x := v] : T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2$	(12) with (4) and (9)

case: D ends with an application of Rule (evar) of the form

$$\frac{(1)\ v_1 : T' \lhd \Gamma_1', x : \langle S' \rangle}{v_1 : e\ T' \lhd e\ \Gamma_1', x : \langle e\ S' \rangle}$$

where $t = v_1$, (2) $T = e\ T'$, (3) $S \equiv e\ S'$ and (4) $\Gamma_1 = e\ \Gamma_1'$.

(5)	$v : S \lhd \Gamma_2$	Assumption
(6)	$v : e\ S' \lhd \Gamma_2$	lem. 3.27 with (5) and (3)
	$\exists \Gamma_2'$ such that	(evar) with (6)
(7)	$v : S' \lhd \Gamma_2'$	"
(8)	$\Gamma_2 = e\ \Gamma_2'$	"
(9)	$v_1[x := v] : T' \lhd \Gamma_1' \Cap \Gamma_2'$	ind.hyp. with (1) and (7)
(10)	$v_1[x := v] : e\ T' \lhd e\ \Gamma_1' \Cap e\ \Gamma_2'$	(evar) and def 3.11 with (9)
$\therefore$	$v_1[x := v] : e\ T' \lhd \Gamma_1 \Cap \Gamma_2$	(10) with (4), (8)

□

Theorem 3.30 (Subject reduction for E$^{\mathrm{v}}$). *If $t : T \lhd \Gamma$ and $t > t'$, then $t' : T \lhd \Gamma$.*

Proof. The proof is by induction on the structure of the reduction $t > t'$.

case: $(\lambda x.t_1)\ v > t_1[x := v]$.

The derivation of $(\lambda x.t_1)\ v : T \lhd \Gamma$ ends with

$$\frac{\dfrac{(2)\ t_1 : T \lhd \Gamma_1, x : \langle S \rangle}{\lambda x.t_1 : S \to T \lhd \Gamma_1} \qquad (1)\ v : S \lhd \Gamma_2}{(\lambda x.t_1)\ v : T \lhd \Gamma_1 \Cap \Gamma_2}$$

where $\Gamma = \Gamma_1 \Cap \Gamma_2$.

Then $t_1[x := v] : T \lhd \Gamma_1 \Cap \Gamma_2$ by Lemma 3.29 with (1) and (2).

case: $t_1\ t_2 > t_1'\ t_2$ if (1) $t_1 > t_1'$.

The derivation of $t_1\ t_2 : T \lhd \Gamma$ ends with

$$\frac{(2)\ t_1 : S \to T \lhd \Gamma_1 \quad (3)\ t_2 : S \lhd \Gamma_2}{t_1\ t_2 : T \lhd \Gamma_1 \Cap \Gamma_2}$$

where $\Gamma = \Gamma_1 \Cap \Gamma_2$.

(4)	$t_1' : S \to T \lhd \Gamma_1$	ind.hyp. with (1),(2)
$\therefore$	$t_1'\ t_2 : T \lhd \Gamma_1 \Cap \Gamma_2$	(app) with (4) and (3)

case: $v_1\ t_2 > v_1\ t_2'$ if (1) $t_2 > t_2'$.

The derivation of $v_1\ t_2 : T \lhd \Gamma$ ends with

$$\frac{(2)\ v_1 : S \to T \lhd \Gamma_1 \quad (3)\ t_2 : S \lhd \Gamma_2}{v_1\ t_2 : T \lhd \Gamma_1 \Cap \Gamma_2}$$

where $\Gamma = \Gamma_1 \Cap \Gamma_2$.

(4)	$t_2' : S \lhd \Gamma_2$	ind.hyp with (1),(3)
$\therefore$	$v_1\ t_2' : T \lhd \Gamma_1 \Cap \Gamma_2$	(app) with (2) and (4)

□

Theorem 3.31 (Progress for E^v). *If $t\ s : T \lhd \Gamma_\omega$, then there exists a term t' such that $s\ t > t'$.*

Proof. The proof is by induction on the structure of $t\ s$.

The derivation of $t\ s : T \lhd \Gamma_\omega$ ends with

$$\frac{(1)\ t : S \to T \lhd \Gamma_\omega \quad s : S \lhd \Gamma_\omega}{t\ s : T \lhd \Gamma_\omega}$$

case: t is v_1.

case: s is v_2.

case: v_1 is x. This case cannot occur since, by (var) with (1), $\Gamma_\omega(x) = \langle S \to T \rangle$, which is a contradiction.

case: v_1 is $\lambda x.t$. Then $(\lambda x.t)\ v_2 > t[x := v_2]$.

case: s is $s_1\ s_2$. By the induction hypothesis, there exists an s' such that $s_1\ s_2 > s'$. Therefore, $v_1\ (s_1\ s_2) > v_1\ s'$.

case: t is $t_1\ t_2$. By the induction hypothesis, there exists a t' such that $t_1\ t_2 > t'$. Therefore, $(t_1\ t_2)\ s > t'\ s$. $\square$

3.4 Type Inference

This section defines a type inference algorithm for System E$^\mathsf{v}$ called Algorithm $\mathcal{I}$. Algorithm $\mathcal{I}$ is parameterised by the choice of type unification algorithm $\mathcal{U}$. $\mathcal{I}$ will terminate whenever the chosen $\mathcal{U}$ terminates, will produce correct results whenever the chosen $\mathcal{U}$ produces correct results, and will find principal typing sets whenever the chosen $\mathcal{U}$ finds covering unifier sets. While these three properties could be combined, we establish them independently so that a different subset of properties might be established for each unification algorithm that is used with Algorithm $\mathcal{I}$.

Our type inference algorithm departs from the previous algorithms for System E based on β-unification [15, 6] in two respects. First, despite E-variables supporting compositional type inference, β-unification itself is a form of whole-program analysis since it solves constraints in a way that corresponds to evaluation of the whole program. Our type inference algorithm instead analyses programs strictly compositionally by analysing each subterm in isolation before analysing the parent terms containing those subterms. Second, β-unification's correspondence to β-reduction means that adding new terms and reduction rules may require corresponding changes to the type unification algorithm. Our type inference algorithm is instead designed to work with unification algorithms that solve type constraints independently of the underlying term language from which those constraints were generated. This means that if new terms are introduced that require no changes to the type syntax, no changes will be needed to the unification algorithm.

Currently, two unification algorithms are suitable for use with Algorithm $\mathcal{I}$ in this respect. The first is the existing algorithm opus [7] which terminates when a solution exists, produces correct results and finds covering unifier sets. The entire opus algorithm is included in Appendix C for reference, along with notes on how it can be adapted to our system. The second is our new algorithm, called opusβ, which derives from opus, but sacrifices the property of covering unifier sets to gain efficiency by borrowing ideas from β-unification. opusβ is established only to produce correct results, and thus its properties are weaker than those of the original opus. However, a result of sacrificing the property of finding covering unifier sets is that opusβ is more efficient than opus, which will be shown in Section 3.6.

This section is structured as follows. Section 3.4.1 defines what classifies as a type unification algorithm and defines the property of covering unification. Section 3.4.2 presents the opusβ unification algorithm. Finally, Section 3.4.3 presents our type inference algorithm called $\mathcal{I}$.

3.4.1 Preliminary Definitions

Definition 3.32 (Distinct sequences for E$^{\mathsf{v}}$)**.** A *distinct sequence* is a sequence of distinct elements. Informally, we will sometimes treat a distinct sequence as a set where the order of elements after applying the $\cap$ and $\cup$ operations is left unspecified. For any metavariable, the superscript s indicates a distinct sequence of that sort of item. $\square$

As examples of our metavariable convention for distinct sequences, τ^s ranges over distinct sequences of typings while σ^s ranges over distinct sequences of substitutions.

Definition 3.33 (Unification constraints for E$^{\mathsf{v}}$)**.** A *unification constraint* $S \,\underline{\lessdot}\, T$ (metavariable δ) is a pair of types. The meta-constructor $\doteq$ is defined such that $S \doteq T$ matches unification constraints $S \,\underline{\lessdot}\, T$ and $T \,\underline{\lessdot}\, S$. A *unification constraint set* (metavariable Δ) is a set of unification constraints. $\square$

Definition 3.34 (Operations on unification constraints for E$^{\mathsf{v}}$)**.** E-variable application and expansion application are extended to unification constraints and unification constraint sets as follows:

$$\begin{aligned} e\ (S \,\underline{\lessdot}\, T) &= e\ S \,\underline{\lessdot}\, e\ T \\ E\ (S \,\underline{\lessdot}\, T) &= E\ S \,\underline{\lessdot}\, E\ T \\ e\ \Delta &= \{e\ \delta \mid \delta \in \Delta\} \\ E\ \Delta &= \{E\ \delta \mid \delta \in \Delta\} \end{aligned}$$

The equivalence relation is extended to unification constraints as follows:

$$(S \,\underline{\lessdot}\, T) \equiv (S' \,\underline{\lessdot}\, T') \qquad\qquad \text{if } S \equiv S' \text{ and } T \equiv T'$$

□

Definition 3.35 (Solved unification constraints and constraint sets for E$^{\mathrm{v}}$). A unification constraint or constraint set is defined to be *solved* by the following rules:

$$\begin{array}{ll} \mathsf{solved}(S \stackrel{.}{\leq} T) & \text{iff } S \equiv T \\ \mathsf{solved}(\Delta) & \text{iff } \delta \in \Delta \implies \mathsf{solved}(\delta) \end{array}$$

□

Definition 3.36 (Unifiers for E$^{\mathrm{v}}$). A *unifier* for unification constraint set Δ is a substitution σ such that $\mathsf{solved}(\sigma\Delta)$.□

Definition 3.37 (Unification algorithm for E$^{\mathrm{v}}$). A unification algorithm $\mathcal{U}$ is an algorithm that takes a unification constraint set Δ and computes a distinct sequence of unifiers for Δ. □

Definition 3.38 (Covering unifier sets for E$^{\mathrm{v}}$). A *covering unifier set* for unification constraint set Δ is a distinct sequence of unifiers σ^s for Δ such that for any unifier σ for Δ, there exists a unifier $\sigma' \in \sigma^s$ and a substitution σ'' such that $\sigma\Delta \equiv \sigma''\sigma'\Delta$. □

Definition 3.39 (Covering unification algorithm for E$^{\mathrm{v}}$). $\mathsf{covering}(\mathcal{U})$ asserts that $\mathcal{U}$ is a *covering unification algorithm* in the sense that for any given Δ, if $\mathcal{U}(\Delta)$ terminates with the result σ^s, then σ^s is a covering unifier set for Δ. □

3.4.2 Algorithm opusβ

This section defines our new unification algorithm, called opusβ. This algorithm is derived from the original opus algorithm and retains the ability to recognise all of the constraint forms necessary to eventually handle extensible records. However, like β-unification, opusβ assumes that the left type of a constraint is an argument type while the right type is a parameter type. This assumption allows opusβ to choose a single search path to what is likely to be the most useful unifier rather than taking all possible search paths in parallel. Consider, for example, the opus constraint $e_1\ T_1 \doteq e_2\ T_2$. In order to solve this constraint, opus must try substituting an expansion for e_1 to make the left type the same shape as the right type, and then also try substituting an expansion for e_2 to make the right type the same shape as the left type, thus branching the search path. Based on ideas from β-unification, $\mathsf{opus}\beta$ assumes that the left type represents an argument type and tries to substitute an expansion only for e_1 to make the argument expand to match the required type of the parameter. This improves the efficiency of $\mathsf{opus}\beta$ over opus, although the resulting algorithm is therefore not a covering unification algorithm.

The $\mathsf{opus}\beta$ algorithm consists of three main relations: $\overrightarrow{\mathsf{rfactor}_\beta}$, which factors initial constraints, $\overrightarrow{\mathsf{opus}_\beta}$, which applies one reduction step of the constraint solving process, and $\Rightarrow^{\sigma}_{*}$, which performs a sequence of alternating $\overrightarrow{\mathsf{rfactor}_\beta}$ and $\overrightarrow{\mathsf{opus}_\beta}$ steps until a solution is found.

Definition 3.40 ($\mathsf{rfactor}_\beta$ for E$^{\vee}$)**.** The $\overrightarrow{\mathsf{rfactor}_\beta}$ relation on constraints and constraint sets factors out common structure and is defined *non-deterministically* by the following rules:

$$\begin{array}{lll}
\Delta & \overrightarrow{\mathsf{rfactor}_\beta} & \bigcup\{\Delta' \mid \delta \in \Delta, \delta \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta'\} \\
\delta & \overrightarrow{\mathsf{rfactor}_\beta} & \Delta \text{ if } \delta \equiv (S_1{\to}S_2 \mathrel{\dot{\leq}} T_1{\to}T_2) \text{ and } \{T_1 \mathrel{\dot{\leq}} S_1, S_2 \mathrel{\dot{\leq}} T_2\} \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta \\
\delta & \overrightarrow{\mathsf{rfactor}_\beta} & \Delta \text{ if } \delta \equiv (S_1 \dot{\cap} S_2 \mathrel{\dot{\leq}} T_1 \dot{\cap} T_2) \text{ and } \{S_1 \mathrel{\dot{\leq}} T_1, S_2 \mathrel{\dot{\leq}} T_2\} \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta \text{ and } (S_1 \not\equiv \omega \wedge S_2 \not\equiv \omega) \vee (T_1 \not\equiv \omega \wedge T_2 \not\equiv \omega) \\
\delta & \overrightarrow{\mathsf{rfactor}_\beta} & \Delta \text{ if } \delta \equiv (e\ S \mathrel{\dot{\leq}} e\ T) \text{ and } \Delta = \{e\ \delta' \mid (S \mathrel{\dot{\leq}} T) \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta', \delta' \in \Delta'\} \\
\delta & \overrightarrow{\mathsf{rfactor}_\beta} & \{\delta\} \text{ if no other rule applies}
\end{array}$$

□

Lemma 3.41 (Correctness of $\overrightarrow{\mathsf{rfactor}_\beta}$ for E$^{\vee}$)**.** *For any E, Δ and any unification constraint or unification constraint set Y, if $Y \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta$ then* $\mathsf{solved}(E\ \Delta) \implies \mathsf{solved}(E\ Y)$*.*

Proof. The proof is by induction on the structure of the derivation of $Y \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta$.

case: $\Delta_1 \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta$ if $\Delta = \bigcup\{\Delta' \mid \delta \in \Delta_1, \delta \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta'\}$

By the induction hypothesis, for each δ and Δ' such that $\delta \in \Delta_1$ and $\delta \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta'$, $\mathsf{solved}(E\ \Delta') \implies \mathsf{solved}(E\ \delta)$. Therefore, by Definition 3.35 and Definition 3.34, $\mathsf{solved}(E\ \Delta) \implies \mathsf{solved}(E\ \Delta_1)$.

case: $\delta \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta$ if $\delta \equiv (S_1{\to}S_2 \mathrel{\dot{\leq}} T_1{\to}T_2)$ and $\{T_1 \mathrel{\dot{\leq}} S_1, S_2 \mathrel{\dot{\leq}} T_2\} \ \overrightarrow{\mathsf{rfactor}_\beta}\ \Delta$.

If $\mathsf{solved}(E\ \Delta)$, then:

(1)	$\mathsf{solved}(\{E\ T_1 \leq E\ S_1, E\ S_2 \leq E\ T_2\})$	ind.hyp.
(2)	$E\ T_1 \equiv E\ S_1$ and $E\ S_2 \equiv E\ T_2$	def 3.35
(3)	$E\ S_1 \rightarrow E\ S_2 \equiv E\ T_1 \rightarrow E\ T_2$	def 3.7
$\therefore$	$\mathsf{solved}(E\ (S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2))$	def 3.35

case: $\delta \xrightarrow{\mathsf{rfactor}_\beta} \Delta$ if $\delta \equiv (S_1 \cap S_2 \leq T_1 \cap T_2)$ and $\{S_1 \leq T_1, S_2 \leq T_2\} \xrightarrow{\mathsf{rfactor}_\beta} \Delta$ and $(S_1 \not\equiv \omega \wedge S_2 \not\equiv \omega) \vee (T_1 \not\equiv \omega \wedge T_2 \not\equiv \omega)$.

Similar to the previous case.

case: $\delta \xrightarrow{\mathsf{rfactor}_\beta} \Delta$ if $\delta \equiv (e\ S \leq e\ T)$ and $\Delta = \{e\ \delta' \mid (S \leq T) \xrightarrow{\mathsf{rfactor}_\beta} \Delta', \delta' \in \Delta'\}$.

If $\mathsf{solved}(E\ \Delta)$, then for each $e\ \delta'$ in Δ:

$\mathsf{solved}(E\ e\ \delta')$	def 3.35
$\mathsf{solved}((E\ e\ \boxdot)\ \delta')$	def 3.11

Therefore, $\mathsf{solved}((E\ e\ \boxdot)\ \delta')$ for each $\delta' \in \Delta'$, and so by Definition 3.35 $\mathsf{solved}((E\ e\ \boxdot)\ \Delta')$. By the induction hypothesis $\mathsf{solved}((E\ e\ \boxdot)\ (S \leq T))$. By Definition 3.11, $\mathsf{solved}(E\ (e\ S \leq e\ T))$.

case: $\delta \xrightarrow{\mathsf{rfactor}_\beta} \{\delta\}$ if no other rule applies . True by Definition 3.35.

□

$\xrightarrow{\mathsf{rfactor}_\beta}$ is based largely on $\xrightarrow{\mathsf{rfactor}}$ from the original opus algorithm (see Appendix C). The main difference from the original algorithm is that constraints are now asymmetric rather than symmetric, with the left type representing the type of an argument and the right type representing the type of a parameter. Each rule works as follows:

- The first rule factors a constraint set by recursively factoring each constraint within the set.
- The second rule factors a constraint between types that are equivalent to function types. The original constraint is factored into two constraints; the first is between the function parameter types (with the order reversed to preserve the roles of the left and right types of a constraint) and the second is between the function return types.
- The third rule factors a constraint between types that are equivalent to intersection types by generating a constraint between the pair of left intersection components and another constraint between the pair of right intersection components. The side condition ensures that this rule applies

only when at least one of the two intersection types is a “true” intersection type, by which we mean that neither its left nor right component is ω. This rule makes it possible, for example, to attempt to unify an intersection type $e\ (S_1 \to S_2) \cap f\ (T_1 \to T_2)$ with a function type $U_1 \to U_2$. Non-deterministically, the constraint can be factored in two ways. The first is by interpreting the constraint as $e\ (S_1 \to S_2) \cap f\ (T_1 \to T_2) \lessdot (U_1 \to U_2) \cap \omega$ and the second is by interpreting the constraint as $e\ (S_1 \to S_2) \cap f\ (T_1 \to T_2) \lessdot \omega \cap (U_1 \to U_2)$.

- The fourth rule factors a constraint between types that are equivalent to E-variable applications. This is done by recursively applying $\overrightarrow{\mathsf{rfactor}_\beta}$ to the constraint nested within e and wrapping the result of this recursive call back into e.
- The fifth rule applies as the default case and simply returns the original constraint as is.

Another point of difference with the original opus algorithm is the manner in which type equivalences are handled. The original algorithm made use of a limited equivalence relation $\overset{\text{unit}}{\equiv}$ that has only the one rule: T is equivalent to $T \cap \omega$. Since System E$^{\mathsf{v}}$ defines a more complete equivalence relation $\equiv$ that corresponds to the full set of type equalities present in System E (see Figure 2.1 in Chapter 2), in $\overrightarrow{\mathsf{rfactor}_\beta}$ we replace $\overset{\text{unit}}{\equiv}$ by $\equiv$. A consequence of this change being that some of the rules of the $\overrightarrow{\mathsf{opus}_\beta}$ relation can be simplified (see the discussion following Definition 3.46 ($\overrightarrow{\mathsf{opus}_\beta}$ relation)).

Note that an implementation of $\overrightarrow{\mathsf{rfactor}_\beta}$ should *not* expand out every possible type equivalence for each use of $\equiv$ because there can be infinitely many. For example, given the constraint $(S_1 \to S_2)\cap\omega\lessdot T_1 \to T_2$, it suffices to consider only the equivalent constraint $S_1 \to S_2 \lessdot T_1 \to T_2$ so that it matches the second rule of $\overrightarrow{\mathsf{rfactor}_\beta}$. It is not necessary to also consider constraints of the form $S_1' \to S_2 \lessdot T_1 \to T_2$ for each $S_1' \equiv S_1$, because all of these would be handled in the same way by the same rule of $\overrightarrow{\mathsf{rfactor}_\beta}$.

Our implementation of $\overrightarrow{\mathsf{rfactor}_\beta}$ [26] uses the following strategy to enumerate equivalent constraints. First, all occurrences of $\omega \cap T$ and $T \cap \omega$ within the constraint δ are rewritten to T, forming a simplified constraint δ' which is clearly equivalent to δ. Then, only the equivalences listed in Figure 3.1 are considered.

Definition 3.42 (Variable structures for E$^{\mathsf{v}}$). A *variable structure* (metavariable $\mathcal{V}$) is a partial function from variables to variable structures, with $\emptyset$ denoting the variable structure with an empty domain. For any variable structure $\mathcal{V}$ and variable X, we define $\mathcal{V}/X$ as

$$\begin{array}{lll}
\delta' = (S_1 \to S_2 \leq T_1 \to T_2) & \equiv (S_1 \to S_2 \leq T_1 \to T_2) & \\
\delta' = (S_1 \cap S_2 \leq T_1 \cap T_2) & \equiv (S_1 \cap S_2 \leq T_1 \cap T_2) & \\
\delta' = (S_1 \cap S_2 \leq T) & \equiv (S_1 \cap S_2 \leq T \cap \omega) & \text{where } T \text{ is a simple type or } \omega \\
\delta' = (S_1 \cap S_2 \leq T) & \equiv (S_1 \cap S_2 \leq \omega \cap T) & \text{where } T \text{ is a simple type or } \omega \\
\delta' = (S \leq T_1 \cap T_2) & \equiv (S \cap \omega \leq T_1 \cap T_2) & \text{where } S \text{ is a simple type or } \omega \\
\delta' = (S \leq T_1 \cap T_2) & \equiv (\omega \cap S \leq T_1 \cap T_2) & \text{where } S \text{ is a simple type or } \omega \\
\delta' = (e\ S \leq e\ T) & \equiv (e\ S \leq e\ T) & \\
\delta' = (e\ S \leq \omega) & \equiv (e\ S \leq e\ \omega) & \\
\delta' = (\omega \leq e\ T) & \equiv (e\ \omega \leq e\ T) &
\end{array}$$

Figure 3.1: Equivalences considered by the implementation of $\mathsf{rfactor}_\beta$

$$\mathcal{V}/X = \begin{cases} \mathcal{V}(X) & \text{if } \mathcal{V}(X) \text{ is defined} \\ \emptyset & \text{otherwise} \end{cases}$$

□

For example, if $\mathcal{V}$ is the variable structure $\{e \mapsto \{f \mapsto \{\alpha \mapsto \emptyset\}, g \mapsto \{\alpha \mapsto \emptyset\}\}\}$, then $\mathcal{V}/e$ is the variable structure $\{f \mapsto \{\alpha \mapsto \emptyset\}, g \mapsto \{\alpha \mapsto \emptyset\}\}$, and $\mathcal{V}/e/f$ is the variable structure $\{\alpha \mapsto \emptyset\}$.

Variable structures are used to describe the structural relationships between E-variables and type variables within a type or a collection of types, and the following definition gives a function that derives a variable structure for any given type, unification constraint or unification constraint set.

Definition 3.43 (varstruct for E$^{\mathsf{v}}$)**.** The total function varstruct takes any type, unification constraint or unification constraint set and returns a variable structure. It is defined by the following rules:

$$\begin{aligned}
&\mathsf{varstruct}(\alpha) && = \alpha \mapsto \emptyset \\
&\mathsf{varstruct}(\omega) && = \emptyset \\
&\mathsf{varstruct}(S \to T) = \mathsf{varstruct}(S \cap T) && = \mathsf{varstruct}(S) \sqcup \mathsf{varstruct}(T) \\
&\mathsf{varstruct}(e\ T) && = e \mapsto \mathsf{varstruct}(T) \\
&\mathsf{varstruct}(S \leq T) && = \mathsf{varstruct}(S) \sqcup \mathsf{varstruct}(T) \\
&\mathsf{varstruct}(\Delta) && = \bigsqcup\{\mathsf{varstruct}(\delta) \mid \delta \in \Delta\}
\end{aligned}$$

where $\mathcal{V}_1 \sqcup \mathcal{V}_2 = \{e \mapsto (\mathcal{V}_1/e) \sqcup (\mathcal{V}_2/e) \mid e \in \mathsf{Dom}(\mathcal{V}_1) \cup \mathsf{Dom}(\mathcal{V}_2)\}$

□

To better understand the relationship between types and variable structures, some examples are shown below of types and the corresponding variable structures that would be computed by varstruct.

Type	**Variable structure**
$e\ (f\ \alpha \to f\ \alpha)$	$\{e \mapsto \{f \mapsto \{\alpha \mapsto \emptyset\}\}\}$
$e\ (f\ \alpha \to g\ \alpha)$	$\{e \mapsto \{f \mapsto \{\alpha \mapsto \emptyset\}, g \mapsto \{\alpha \mapsto \emptyset\}\}\}$
$f\ (e\ \alpha \to g\ \omega) \to f\ (e\ \beta \to g\ \alpha)$	$\{f \mapsto \{e \mapsto \{\alpha \mapsto \emptyset, \beta \mapsto \emptyset\}, g \mapsto \{\alpha \mapsto \emptyset\}\}\}$

The last example can be read, "under f, there are two variables, e and g. Under e there are two variables, α and β. Under g there is one variable, α."

Definition 3.44 (ovars for E^{v})**.** The set of *outer variables* of a type T, written $\mathsf{ovars}(T)$, is given by:

$$\mathsf{ovars}(T) = \mathsf{Dom}(\mathsf{varstruct}(T))$$

□

The outer variables of a type are the variables at the top level of the variable structure of that type.

For example, for the type $f\ (e\ \alpha \to g\ \omega) \to f\ (e\ \beta \to g\ \alpha)$, the only outer variable is f since all other variables are beneath f in the variable structure. The outer variables of a type are those variables that can be directly addressed by a substitution.

Definition 3.45 (Fresh renamings for E$^{\mathsf{v}}$)**.** A *fresh renaming* for some finite set of variables X^s is a substitution $(\boxdot, X_1 := K_1, \ldots X_n := K_n)$ where $\{X_1 := K_1, \ldots X_n := K_n\} = \{e := e'\ \boxdot \mid e \in X^s, e'\ \mathsf{fresh}\} \cup \{\alpha := \alpha' \mid \alpha \in X^s, \alpha'\ \mathsf{fresh}\}$. □

Definition 3.46 ($\overrightarrow{\mathsf{opus}_\beta}$ relation for E$^{\mathsf{v}}$)**.** The $\overrightarrow{\mathsf{opus}_\beta}$ relation performs one step of unification on a factored constraint and is defined by the following rules:

$$
\begin{array}{llll r}
(\alpha \doteq \bar{T}, \mathcal{V}) & \overrightarrow{\mathsf{opus}_\beta} & (\alpha := \bar{T}) & \text{if } \alpha \notin \mathsf{ovars}(\bar{T}) & \text{(T-unify)} \\
(e\ T \leq \bar{T}, \mathcal{V}) & \overrightarrow{\mathsf{opus}_\beta} & (e := \mathsf{ren}(\mathcal{V}, e)) & & \text{(EA-unify)} \\
(e\ S \leq T \cap U, \mathcal{V}) & \overrightarrow{\mathsf{opus}_\beta} & (e := e_1\ \boxdot \cap e_2\ \boxdot) & & \text{(EI-unify)} \\
(e\ S \leq e\ T, \mathcal{V}) & \overrightarrow{\mathsf{opus}_\beta} & (e := e\ \sigma) & \text{if } (S \leq T, \mathcal{V}/e)\ \overrightarrow{\mathsf{opus}_\beta}\ \sigma & \text{(E-unify)} \\
 & & (e := e\ \omega) & \text{otherwise} & \\
(e\ S \leq f\ T, \mathcal{V}) & \overrightarrow{\mathsf{opus}_\beta} & (e := f\ g\ \boxdot) & \text{if } S \text{ is } \alpha \text{ or } T \text{ is } \mathbb{T},\ g\ \mathsf{fresh} & \text{(EE-unify)} \\
 & & (f := e\ g\ \boxdot) & \text{otherwise, } g\ \mathsf{fresh} &
\end{array}
$$

where $\mathsf{ren}(\mathcal{V}, e)$ is a fresh renaming of the variables in $\mathsf{Dom}(\mathcal{V}/e)$. □

$\overrightarrow{\mathsf{opus}_\beta}$ is based largely on $\overrightarrow{\mathsf{opus}}$ from the original opus algorithm (see Appendix C). Each rule of $\overrightarrow{\mathsf{opus}_\beta}$ can be understood as follows.

- Rule (T-unify) unifies a simple type variable with a simple type and performs an occurs check. This rule matches when the simple type variable appears on either the left or right side of a constraint, since the meta-constructor $\doteq$ is used. While the corresponding rule of the original opus algorithm could handle constraints of the form $\alpha \doteq T_1 \cap T_2$, this is not possible in System E$^{\mathsf{v}}$ since a simple type variable can only be unified with a simple type. However, it is still possible in System E$^{\mathsf{v}}$ to unify $e\ \alpha$ with $T_1 \cap T_2$ via Rule (EI-unify).
- Rule (EA-unify) unifies an E-variable application with a simple type by eliminating the E-variable and renaming the variables beneath it to avoid capture. This rule is the same as the corresponding

rule in the original algorithm except that it has now been generalised to support not just function types but any simple type (including yet-to-be-defined type constants).

- Rule (EI-unify) unifies an E-variable application type with an intersection type. Following the strategy of β-unification, Rule (EI-unify) assumes that the argument type on the left should expand to match the shape of the parameter type on the right. Another path of unification that could have been tried is to eliminate e by a substitution, although this could potentially fail if S is is a simple type or ω. This additional path is searched in the original algorithm to obtain a covering unifier set, while in our algorithm this rule has been made deterministic and chooses only one search path. For example, unifying $e\ (e_1\ \texttt{Int} \sqcap e_2\ \texttt{Str})$ and $\texttt{Int} \sqcap \texttt{Str}$ will now lead only to $e := ((e_1 := \boxdot, e_2 := \omega) \sqcap (e_1 := \omega, e_2 := \boxdot))$ and not also $e := (e_1 := \boxdot, e_2 := \boxdot)$.
- Rule (E-unify) recursively solves one step of the nested constraint under e, taking care to also descend one level deeper into the given variable structure (see below for a more detailed discussion).
- Rule (EE-unify) unifies two E-variable application types choosing either for the left type to expand to match the shape of the right type, or vice versa. If S is a simple type variable or if T is an expansion type, then $e\ S$ expands to shape of $f\ T$. The rationale for this rule is that, by Corollary 3.18 (Type substitution), a type of the form $e\ \alpha$ has the power of a System E type variable and is therefore always capable of transforming via substitution to type $f\ T$. Hence, if S is a simple type variable, then $e\ S$ can expand to the exact shape of the right type. Otherwise, if T is an "expansion" type, then we assume that $e\ S$ should expand to match its shape. That is, in this case, we assume $e\ S$ is an argument type and $f\ T$ is a parameter type. If neither of these conditions hold, then the assumption is reversed and $f\ T$ is assumed to be the argument type while $e\ S$ is assumed to be the parameter type. This rule essentially uses a heuristic to guess which search path will lead to the most useful solution, and this allows the rule to be deterministic. By comparison, the original algorithm opus will unify $e\ S$ and $f\ T$ by symmetrically trying to expand $e\ S$ to match the shape of T and also trying to expand $f\ T$ to match the shape of S, thus branching the search path.

When examining these rules, the main difference from the original opus algorithm is that each rule has been made deterministic, resulting in increased efficiency. Another way in which efficiency has been increased is the use of variable structures to generate renamings. All rules now take a variable structure as a parameter which is assumed to be a variable structure for the constraint set being solved. Rule (E-unify) plays an important part since, while recursively solving one step of the nested constraint, it also passes the nested variable structure corresponding to that nested constraint. Rule (EA-unify) then renames only the E-variables directly visible at the current level of nesting. This is in contrast to the original rules of $\xrightarrow{\mathsf{opus}}$ which blindly rename *all* E-variables involved in the constraint. Our renamings are therefore smaller and more efficient to deal with, especially when many substitutions are composed together in sequence.

Another important difference is that in the original opus algorithm, constraints of the form $e\,\bar{T}\underline{\leq}\omega$ are handled by a special rule called (C EO-unify) which substitutes ω for e, However, in our own algorithm, this rule is no longer needed. This is because $\xrightarrow{\mathsf{rfactor}\beta}$ will first factor this constraint to $e\,(\bar{T}\underline{\leq}\omega)$ due to the full equivalence relation $\equiv$ replacing the more limited relation $\overset{\text{unit}}{\equiv}$. After factorisation, this constraint will now be handled by Rule (E-unify) of $\xrightarrow{\mathsf{opus}\beta}$ which, since the inner constraint $\bar{T}\ \underline{\leq}\ \omega$ is unsolvable, will substitute $e\ \omega$ for e.

The main relation, $\Rightarrow^{\sigma}_{*}$, is now defined which performs a sequence of alternating $\xrightarrow{\mathsf{rfactor}\beta}$ and $\xrightarrow{\mathsf{opus}\beta}$ steps until a solution is found. This relation is then used to construct the final Algorithm opusβ.

Definition 3.47 (opus$_\beta$ reduction for E$^{\mathsf{v}}$). The relation $\Rightarrow^{\sigma}$ is defined by: $(\Delta,\mathcal{V})\Rightarrow^{\sigma}(\Delta_2,\sigma\mathcal{V})$ if $\Delta\xrightarrow{\mathsf{rfactor}\beta}\Delta_1$ and $\delta\in\Delta_1$ and not $\mathsf{solved}(\delta)$ and $(\delta,\mathcal{V})\xrightarrow{\mathsf{opus}\beta}\sigma$ and $\sigma\Delta_1\xrightarrow{\mathsf{rfactor}\beta}\Delta_2$. The relation $\Rightarrow^{\sigma}_{*}$ is defined by $(\Delta,\mathcal{V})\Rightarrow^{\boxdot}_{*}(\Delta,\mathcal{V})$ and $(\Delta_1,\mathcal{V}_1)\Rightarrow^{\sigma_2\sigma_1}_{*}(\Delta_3,\mathcal{V}_3)$ if $(\Delta_1,\mathcal{V}_1)\Rightarrow^{\sigma_1}(\Delta_2,\mathcal{V}_2)$ and $(\Delta_2,\mathcal{V}_2)\Rightarrow^{\sigma_2}_{*}(\Delta_3,\mathcal{V}_3)$. □

The following lemma is key to establishing the correctness of the opusβ algorithm.

Lemma 3.48 (Correctness of $\Rightarrow^{\sigma}_{*}$ for E$^{\mathsf{v}}$). *If* $(\Delta,\mathcal{V})\Rightarrow^{\sigma}_{*}(\Delta',\mathcal{V}')$ *and* $\mathsf{solved}(\Delta')$ *then* $\mathsf{solved}(\sigma\Delta)$.

Proof. The proof is by induction on the structure of the reduction of $(\Delta,\mathcal{V})\Rightarrow^{\sigma}_{*}(\Delta',\mathcal{V}')$.

case: $(\Delta,\mathcal{V})\Rightarrow^{\boxdot}_{*}(\Delta,\mathcal{V})$. Immediate.

case: $(\Delta,\mathcal{V})\Rightarrow^{\sigma_2\sigma_1}_{*}(\Delta',\mathcal{V}')$ if $(\Delta,\mathcal{V})\Rightarrow^{\sigma_1}(\Delta_1,\mathcal{V}_1)$ and $(\Delta_1,\mathcal{V}_1)\Rightarrow^{\sigma_2}_{*}(\Delta',\mathcal{V}')$.

(1)	$\mathsf{solved}(\sigma_2\Delta_1)$	ind.hyp.

(2)	$\exists \Delta_2, \delta.\quad \Delta \overrightarrow{\mathsf{rfactor}\beta}\ \Delta_2$	def. 3.47 with $(\Delta, \mathcal{V}) \Rightarrow^{\sigma_1} (\Delta_1, \mathcal{V}_1)$
(3)	$\delta \in \Delta_2$	”
(4)	not $\mathsf{solved}(\delta)$	”
(5)	$(\delta, \mathcal{V}) \overrightarrow{\mathsf{opus}\beta}\ \sigma_1$	”
(6)	$\sigma_1 \Delta_2 \overrightarrow{\mathsf{rfactor}\beta}\ \Delta_1$	”
(7)	$\mathcal{V}_1 = \sigma_1 \mathcal{V}$	”
(8)	$\mathsf{solved}(\sigma_2\ (\sigma_1 \Delta_2))$	lem 3.41 with (6),(1)
(9)	$\mathsf{solved}((\sigma_2 \sigma_1)\ \Delta_2))$	lem 3.14
$\therefore$	$\mathsf{solved}((\sigma_2 \sigma_1)\ \Delta))$	lem 3.41 with (2),(9)

□

Definition 3.49 (Algorithm $\mathsf{opus}\beta$ for E$^{\mathsf{v}}$)**.** Algorithm $\mathsf{opus}\beta$ is defined by:

$$\mathsf{opus}\beta(\Delta) = \{\sigma \mid (\Delta, \mathsf{varstruct}(\Delta)) \Rightarrow^{\sigma}_{*} (\Delta', \mathcal{V}), \mathsf{solved}(\Delta')\}$$

□

Theorem 3.50 (Correctness of $\mathsf{opus}\beta$ for E$^{\mathsf{v}}$)**.** *Given any Δ and σ, then $\sigma \in \mathsf{opus}\beta(\Delta) \implies \mathsf{solved}(\sigma\Delta)$.*

Proof. Follows immediately from Lemma 3.48. □

3.4.3 Algorithm $\mathcal{I}$

This section will now present Algorithm $\mathcal{I}$, a compositional type inference algorithm for System E$^{\mathsf{v}}$. Algorithm $\mathcal{I}$ analyses terms strictly compositionally by analysing each subterm in isolation before analysing the parent terms containing those subterms.

Definition 3.51 (estrip for E$^{\mathsf{v}}$)**.** estrip is a total function from typings to typings, defined as follows:

$$\mathsf{estrip}(\tau) = \begin{cases} \mathsf{estrip}(\tau_1) & \text{if } \tau = e\ \tau_1 \\ \tau & \text{otherwise} \end{cases}$$

□

estrip has the effect of stripping any E-variables applied to a typing. For example, $\mathsf{estrip}(e_1\ e_2\ e_3\ \tau)$ returns the typing τ.

Lemma 3.52 (Correctness of estrip for E^v). *Given any t and τ, if $t : \tau$ then $t : \mathsf{estrip}(\tau)$.*

Proof. The proof is by induction on the structure of τ.

case: τ is $e\ \tau_1$.

	let $\sigma = (e := id)$	
(1)	$t : \tau$	assumption
(2)	$t : \sigma\ \tau$	lem 3.28 with (1)
(3)	$t : \tau_1$	def 3.11 with (2)
(4)	$t : \mathsf{estrip}(\tau_1)$	ind.hyp. with (3)
$\therefore$	$t : \mathsf{estrip}(\tau)$	def 3.51 with (4)

case: τ is not $e\ \tau_1$.

(1)	$t : \tau$	assumption
(2)	$\mathsf{estrip}(\tau) = \tau$	def 3.51
$\therefore$	$t : \mathsf{estrip}(\tau)$	(1),(2)

□

During type inference, estrip is used to remove superfluous E-variables from typings before re-wrapping them in fresh E-variables. To show that $\mathcal{I}$ satisfies the principal typings property, the following lemma will be used to show that the original typing can be recovered from a stripped typing.

Lemma 3.53 (estrip reversion for E^v). *For any typing τ there exists an expansion E such that $E\ \mathsf{estrip}(\tau){=}\tau$.*

Proof. The proof is by induction on the structure of τ.

case: τ is $e\ \tau_1$.

(1) $\exists E_1.\quad E_1\ \mathsf{estrip}(\tau_1) = \tau_1$ ind.hyp.

(2) $(e\ E_1)\ \mathsf{estrip}(e\ \tau_1)$

$= (e\ E_1)\ \mathsf{estrip}(\tau_1)$ def 3.51

$= e\ (E_1\ \mathsf{estrip}(\tau_1))$ def 3.11

$= e\ \tau_1$ (1)

$\therefore\quad E = e\ E_1$ (2)

case: τ is not $e\ \tau_1$.

(1) $\mathsf{estrip}(\tau) = \tau$ def 3.51

$\therefore\quad E = \boxdot$ lem 3.13 with (1)

□

Definition 3.54 (isect for E^{v}). isect is a total function from distinct sequence of typings to typings, defined by the following rules:

$$\mathsf{isect}(\emptyset) = \omega \lhd \Gamma_\omega$$

$$\mathsf{isect}(\tau, \tau^s) = \begin{cases} \mathsf{isect}(\tau^s) & \text{if } \mathsf{estrip}(\tau) = (\omega \lhd \Gamma_\omega) \\ \mathsf{estrip}(\tau) & \text{otherwise if } \forall \tau'.\tau' \in \tau^s \implies \mathsf{estrip}(\tau') = (\omega \lhd \Gamma_\omega) \\ e_1\ \mathsf{estrip}(\tau) \sqcap e_2\ \mathsf{isect}(\tau^s) & \text{otherwise, where } e_1, e_2\ \mathsf{fresh} \end{cases}$$

□

Effectively, isect takes a distinct sequence of typings and intersects them together into a single typing with superfluous $\omega \lhd \Gamma_\omega$ typings omitted and with fresh E-variables wrapped around different intersection components. The case of a single typing is handled when $\tau^s = \emptyset$.

Lemma 3.55 (Correctness of isect for E^{v}). *Given any v, τ and τ_s, if $\tau \in \tau^s \implies v : \tau$, then* $v : \mathsf{isect}(\tau^s)$.

Proof. The proof is by induction on the size of τ^s.

case: $\tau^s = \emptyset$.

(1) $v : \omega \lhd \Gamma_\omega$ (omega)

$\therefore\quad v : \mathsf{isect}(\emptyset)$ def 3.54 with (1)

case: $\tau^s = \tau_1, \tau_1^s$ where $\mathsf{estrip}(\tau_1) = \omega \lhd \Gamma_\omega$.

(1)	$v : \mathsf{isect}(\tau_1^s)$	ind.hyp.
$\therefore$	$v : \mathsf{isect}(\tau_1, \tau_1^s)$	def 3.54 with (1)

case: $\tau^s = \tau_1, \tau_1^s$ where $\mathsf{estrip}(\tau_1) \neq \omega \lhd \Gamma_\omega$ and $\forall \tau'.\tau' \in \tau_1^s \implies \mathsf{estrip}(\tau') = (\omega \lhd \Gamma_\omega)$.

(1)	$v : \tau_1$	assumption
(2)	$v : \mathsf{estrip}(\tau_1)$	lem 3.52 with (1)
$\therefore$	$v : \mathsf{isect}(\tau_1, \tau_1^s)$	def 3.54 with (2)

case: $\tau^s = \tau_1, \tau_1^s$ where $\mathsf{estrip}(\tau_1) \neq \omega \lhd \Gamma_\omega$ and $\exists \tau' \in \tau^s.\mathsf{estrip}(\tau') \neq (\omega \lhd \Gamma_\omega)$

(1)	$\exists e_1, e_2.\ \mathsf{isect}(\tau_1, \tau_1^s) = e_1\ \mathsf{estrip}(\tau_1) \sqcap e_2\ \mathsf{isect}(\tau_1^s)$	def 3.54
(2)	$v : \tau_1$	assumption
(3)	$v : \mathsf{estrip}(\tau_1)$	lem 3.52 with (2)
(4)	$v : e_1\ \mathsf{estrip}(\tau_1)$	(evar) with (3)
(5)	$v : \mathsf{isect}(\tau_1^s)$	ind.hyp.
(6)	$v : e_2\ \mathsf{isect}(\tau_1^s)$	(evar) with (5)
(7)	$v : e_1\ \mathsf{estrip}(\tau_1) \sqcap e_2\ \mathsf{isect}(\tau_1^s)$	(int) with (4),(6)
$\therefore$	$v : \mathsf{isect}(\tau_1, \tau_1^s)$	(7),(1)

□

During type inference, isect is used to intersect multiple possible typings for a value into a single typing. To show that $\mathcal{I}$ satisfies the principal typings property, the following lemma will be used to show that any of the original typings given to isect can be recovered from the result of isect.

Lemma 3.56 (isect reversion for E^{v}). *Given any τ, τ^s and e, if $\tau \in \tau^s$, then there exists an expansion E such that $\tau \equiv E\ (\mathsf{isect}(\tau^s))$.*

Proof. The proof is by induction on the size of τ^s.

case: $\tau^s = \emptyset$. The premise $\tau \in \emptyset$ is false, and therefore the implication is true.

case: $\tau^s = \tau_1, \tau_1^s$ where (1) $\mathsf{estrip}(\tau_1) = \omega \lhd \Gamma_\omega$.

case: $\tau = \tau_1$.

$$\begin{aligned} \exists E'.\tau &= E'\ (\omega \lhd \Gamma_\omega) && \text{lem 3.53 with (1)} \\ &= E'\ (\omega\ \mathsf{isect}(\tau_1, \tau_1^s)) && \text{def 3.11} \\ &= (E'\ \omega)\ (\mathsf{isect}(\tau_1, \tau_1^s)) && \text{def 3.11} \\ \therefore\quad & E = E'\ \omega \end{aligned}$$

case: $\tau \in \tau_1^s$.

$$\begin{aligned} \exists E.\tau &\equiv E\ \mathsf{isect}(\tau_1^s) && \text{ind.hyp.} \\ &= E\ \mathsf{isect}(\tau_1, \tau_1^s) && \text{def 3.54} \\ \therefore\quad & \sigma = \sigma' \end{aligned}$$

case: $\tau^s = \tau_1, \tau_1^s$ where $\mathsf{estrip}(\tau_1) \neq \omega \lhd \Gamma_\omega$ and (1) $\forall \tau'.\tau' \in \tau_1^s \implies \mathsf{estrip}(\tau') = (\omega \lhd \Gamma_\omega)$.

case: $\tau = \tau_1$.

$$\begin{aligned} \exists E'.\tau_1 &= E'\ \mathsf{estrip}(\tau_1) && \text{lem 3.53} \\ &= E'\ \mathsf{isect}(\tau_1, \tau_1^s)) && \text{def 3.54} \\ \therefore\quad & E = E' \end{aligned}$$

case: $\tau \in \tau_1^s$.

$$\begin{aligned} \exists E'.\tau &= E'\ (\omega \lhd \Gamma_\omega) && \text{lem 3.53 with (1)} \\ &= E'\ (\omega\ \mathsf{isect}(\tau_1, \tau_1^s)) && \text{def 3.11} \\ &= (E'\ \omega)\ \mathsf{isect}(\tau_1, \tau_1^s) && \text{def 3.11} \\ \therefore\quad & E = E'\ \omega \end{aligned}$$

case: $\tau^s = \tau_1, \tau_1^s$ where $\mathsf{estrip}(\tau_1) \neq \omega \lhd \Gamma_\omega$ and $\exists \tau' \in \tau^s.\mathsf{estrip}(\tau') \neq (\omega \lhd \Gamma_\omega)$

case: $\tau = \tau_1$.

$$\begin{aligned} (1)\quad & \mathsf{isect}(\tau_1, \tau_1^s) = e_1\ \mathsf{estrip}(\tau_1) \Cap e_2\ \mathsf{isect}(\tau_1^s) && \text{def 3.54} \\ (2)\quad & \text{where } e_1 \neq e_2 && \text{''} \\ \exists E'.\tau_1 &= E'\ \mathsf{estrip}(\tau_1) && \text{lem 3.53} \\ &\equiv E'\ \mathsf{estrip}(\tau_1) \Cap (\omega \lhd \Gamma_\omega) && \text{def 3.7} \\ &= E'\ \mathsf{estrip}(\tau_1) \Cap \omega\ \mathsf{isect}(\tau_1^s) && \text{def 3.11} \\ &= (e_1 := E', e_2 := \omega)\ (e_1\ \mathsf{estrip}(\tau_1) \Cap e_2\ \mathsf{isect}(\tau_1^s)) && \text{def 3.11, (2)} \\ &= (e_1 := E', e_2 := \omega)\ \mathsf{isect}(\tau_1, \tau_1^s) && \text{(1)} \\ \therefore\quad & E = (e_1 := E', e_2 := \omega) \end{aligned}$$

case: $\tau \in \tau_1^s$.

(1)	$\mathsf{isect}(\tau_1, \tau_1^s) = e_1\ \mathsf{estrip}(\tau_1) \cap e_2\ \mathsf{isect}(\tau_1^s)$	def 3.54
(2)	where $e_1 \neq e_2$	”
$\exists E'.\tau$	$\equiv E'\ \mathsf{isect}(\tau_1^s)$	ind.hyp.
	$= (e_2 := E')\ (e_2\ \mathsf{isect}(\tau_1^s))$	def 3.11
	$= (e_2 := E', e_1 := \omega)\ (e_2\ \mathsf{isect}(\tau_1^s))$	(2)
	$\equiv (\omega \lhd \Gamma_\omega) \cap (e_2 := E', e_1 := \omega)\ (e_2\ \mathsf{isect}(\tau_1^s))$	def 3.7
	$= \omega\ \mathsf{estrip}(\tau_1) \cap (e_2 := E', e_1 := \omega)\ (e_2\ \mathsf{isect}(\tau_1^s))$	def 3.11
	$= (e_2 := E', e_1 := \omega)\ (e_1\ \mathsf{estrip}(\tau_1)) \cap (e_2 := E', e_1 := \omega)\ (e_2\ \mathsf{isect}(\tau_1^s))$	def 3.11
	$= (e_2 := E', e_1 := \omega)\ (e_1\ \mathsf{estrip}(\tau_1) \cap e_2\ \mathsf{isect}(\tau_1^s))$	def 3.11
	$= (e_2 := E', e_1 := \omega)\ \mathsf{isect}(\tau_1, \tau_1^s)$	(1)
$\therefore$	$E = (e_2 := E', e_1 := \omega)$	

□

Definition 3.57 (Algorithm $\mathcal{I}$ for E^{v})**.** The type inference algorithm $\mathcal{I}$ takes a unification algorithm and a term and returns a distinct sequence of typings. It is defined by the following rules:

Terms

$\mathcal{I}(\mathcal{U}, v) = \quad \{e\ \mathcal{I}_v(\mathcal{U}, v)\} \quad e$ fresh $\quad (\mathcal{I}.\text{val})$

$\mathcal{I}(\mathcal{U}, t\ s) = \quad \{\sigma\ (e\ \alpha \lhd \Gamma_1 \cap \Gamma_2) \quad (\mathcal{I}.\text{app})$
$\quad \mid (T \lhd \Gamma_1) \in \mathcal{I}(\mathcal{U}, t), (S \lhd \Gamma_2) \in \mathcal{I}(\mathcal{U}, s)$
$\quad \sigma \in \mathcal{U}(\{T \leq S \to e\ \alpha\} \cup \{U \leq U \mid x : U \in \Gamma_1 \cap \Gamma_2\}), e, \alpha$ fresh$\}$

Values

$\mathcal{I}_v(\mathcal{U}, x) = \quad \alpha \lhd x : \langle\alpha\rangle \quad \alpha$ fresh $\quad (\mathcal{I}.\text{var})$

$\mathcal{I}_v(\mathcal{U}, \lambda x.t) = \quad \mathsf{isect}(\{(S \to T \lhd \Gamma) \mid (T \lhd \Gamma, x : \langle S\rangle) \in \mathcal{I}(\mathcal{U}, t)\}) \quad (\mathcal{I}.\text{abs})$

□

Algorithm $\mathcal{I}$ has only two rules at the top level: ($\mathcal{I}$.val) for values and ($\mathcal{I}$.app) for applications. Typings for values are wrapped in an E-variable, while typings for applications are not due to the value restriction. Applications of the form $t\ s$ are handled by recursively inferring typings for t and s and then for each typing $T \lhd \Gamma_1$ for t and $S \lhd \Gamma_2$ for s, a unification constraint $T \mathrel{\underline{\lessdot}} S \to e\ \alpha$ is generated and solved via $\mathcal{U}$ to make the type of argument s match the parameter type of function t. If it is intended for $\mathcal{I}$ to produce a principal typing for $t\ s$, it is necessary for the result of $\mathcal{U}$ to be most general with respect to all of the variables that appear within the typings for s and t (see Definition 3.38). To achieve this, the additional dummy constraints $U \mathrel{\underline{\lessdot}} U$ are introduced into the set of constraints solved by $\mathcal{U}$. While these constraints are trivially satisfied, their presence does have an effect on how $\mathcal{U}$ performs variable renamings, and if $\mathcal{U}$ is a covering unification algorithm, these dummy constraints will force the result of $\mathcal{U}$ to be most general with respect to all of the relevant variables. This procedure is repeated for each of the different typings for s and t. After unification is applied in each case, each return type of t is counted as an inferred type for $t\ s$. These results are not intersected together due to the value restriction.

The specific cases for values are handled by $\mathcal{I}_v$. ($\mathcal{I}$.var) assigns a simple type variable, which is then wrapped in an E-variable by ($\mathcal{I}$.val). ($\mathcal{I}$.abs) recursively infers typings for the function body and then constructs a function type in a way that corresponds to an application of Rule (abs). All typings found are intersected together, and then wrapped in an E-variable by ($\mathcal{I}$.val).

Algorithm $\mathcal{I}$ satisfies the following properties.

Theorem 3.58 (Termination of $\mathcal{I}$ for E$^{\text{v}}$). *Given any $\mathcal{U}$ and t, if each use of $\mathcal{U}$ by Algorithm $\mathcal{I}$ in the course of computing $\mathcal{I}(\mathcal{U}, t)$ terminates, then $\mathcal{I}(\mathcal{U}, t)$ terminates.*

Proof. Straightforward by induction on the structure of t. □

Theorem 3.59 (Correctness of $\mathcal{I}$ for E$^{\text{v}}$). *Given any $\mathcal{U}$, τ and t, if $\tau \in \mathcal{I}(\mathcal{U}, t)$, then $t : \tau$.*

Proof. The proof is by induction on the structure of t.

case: $t = x$.

(1)	$\exists e, \alpha.\quad \mathcal{I}(\mathcal{U}, x) = \{e\ \mathcal{I}_v(\mathcal{U}, x)\} = \{e\ \alpha \lhd x : \langle e\ \alpha\rangle\}$	($\mathcal{I}$.val)/($\mathcal{I}$.var)
(2)	$x : \alpha \lhd x : \langle \alpha \rangle$	(var)
(3)	$x : e\ \alpha \lhd x : \langle e\ \alpha \rangle$	(evar) with (2)
$\therefore$	if $\tau \in \mathcal{I}(\mathcal{U}, x)$, then $x : \tau$	(1),(3)

case: $t = \lambda x.t_1$.

let $\tau^s = \{S{\to}T \lhd \Gamma \mid (T \lhd \Gamma, x : \langle S\rangle) \in \mathcal{I}(\mathcal{U}, t_1)\}$

(1)	$\exists e.\ \ \mathcal{I}(\mathcal{U}, \lambda x.t_1) = \{e\ \mathcal{I}_v(\mathcal{U}, \lambda x.t_1)\} = \{e\ \mathsf{isect}(\tau^s)\}$	$(\mathcal{I}.\text{val})/(\mathcal{I}.\text{abs})$
(2)	$\forall S, T, \Gamma.$	
	$(S{\to}T \lhd \Gamma) \in \tau^s$	
	$\Longrightarrow\ (T \lhd \Gamma, x : \langle S\rangle) \in \mathcal{I}(\mathcal{U}, t_1)$	(1)
	$\Longrightarrow\ t_1 : T \lhd \Gamma, x : \langle S\rangle$	ind.hyp.
	$\Longrightarrow\ \lambda x.t_1 : S{\to}T \lhd \Gamma$	(abs)
(3)	$\forall \tau'.\tau' \in \tau^s \implies \lambda x.t_1 : \tau'$	$\Longrightarrow$ transitivity with (2)
(4)	$\lambda x.t_1 : \mathsf{isect}(\tau^s)$	lem 3.55 with (3)
(5)	$\lambda x.t_1 : e\ \mathsf{isect}(\tau^s)$	(evar) with (4)
$\therefore$	if $\tau \in \mathcal{I}(\mathcal{U}, \lambda x.t_1)$, then $\lambda x.t_1 : \tau$	(1),(5)

case: $t = t_1\ t_2$.

(1)	$\mathcal{I}(\mathcal{U}, t_1\ t_2) = \{\sigma\ (e\ \alpha) \lhd \sigma\ (\Gamma_1 \Cap \Gamma_2) \mid$	$(\mathcal{I}.\text{app})$
(2)	$(T_1 \lhd \Gamma_1) \in \mathcal{I}(\mathcal{U}, t_1),$	"
(3)	$(T_2 \lhd \Gamma_2) \in \mathcal{I}(\mathcal{U}, t_2),$	"
(4)	$\sigma \in \mathcal{U}(\{T_1 \leq T_2{\to}e\ \alpha\} \cup \{U \leq U \mid x : U \in \Gamma_1 \Cap \Gamma_2\}), e, \alpha\ \mathsf{fresh}\}$	"
(6)	For each $(\sigma\ (e\ \alpha) \lhd \sigma\ (\Gamma_1 \Cap \Gamma_2)) \in \mathcal{I}(\mathcal{U}, t_1\ t_2)$,	
(6a)	$\sigma\ T_1 \equiv \sigma\ T_2 \to \sigma\ (e\ \alpha)$	def 3.37 with (4)
(6b)	$t_1 : T_1 \lhd \Gamma_1$	ind.hyp. with (2)
(6c)	$t_1 : \sigma T_1 \lhd \sigma\Gamma_1$	lem 3.28 with (6b)
(6d)	$t_1 : \sigma T_2 \to \sigma\ (e\ \alpha) \lhd \sigma\Gamma_1$	lem 3.27 with (6c),(6a)
(6e)	$t_2 : T_2 \lhd \Gamma_2$	ind.hyp. with (3)
(6f)	$t_2 : \sigma T_2 \lhd \sigma\Gamma_2$	lem 3.28 with (6e)
(6g)	$t_1\ t_2 : \sigma\ (e\ \alpha) \lhd \sigma\Gamma_1 \Cap \sigma\Gamma_2$	(app) with (6d),(6f)
(6h)	$t_1\ t_2 : \sigma\ (e\ \alpha) \lhd \sigma\ (\Gamma_1 \Cap \Gamma_2)$	lem 3.15 with (6g)
$\therefore$	if $\tau \in \mathcal{I}(\mathcal{U}, t_1\ t_2)$, then $t_1\ t_2 : \tau$	(6)-(6h)

□

Theorem 3.60 (Principality of $\mathcal{I}$ for E$^{\mathrm{v}}$)**.** *Given any $\mathcal{U}$, t and τ', if* $\mathsf{covering}(\mathcal{U})$ *and $t : \tau'$ and $\mathcal{I}(\mathcal{U}, t)$ terminates, then there exists a substitution σ and a typing $\tau \in \mathcal{I}(\mathcal{U}, t)$ such that $\tau' \equiv \sigma\tau$.*

Proof. Let D be the derivation of $t : \tau'$. The proof is by induction on the structure of D.

case: D is the following application of Rule (var):

$$\frac{}{(1)\ x : \bar{T} \lhd x : \langle \bar{T} \rangle}$$

(2)	$\exists e, \alpha.\quad \mathcal{I}(\mathcal{U}, x) = \{e\ \mathcal{I}_v(\mathcal{U}, x)\} = \{e\ \alpha \lhd x : \langle e\ \alpha \rangle\}$	$(\mathcal{I}.\mathrm{val})/(\mathcal{I}.\mathrm{var})$
	let $\sigma_1 = (e := (\alpha := \bar{T}))$	
(3)	$\sigma_1\ (e\ \alpha \lhd x : \langle e\ \alpha \rangle) = \tau'$	def 3.11
$\therefore$	σ is σ_1	(3),(2)

case: D ends with the following application of Rule (abs):

$$\frac{(2)\ t_1 : T_2 \lhd \Gamma, x : \langle T_1 \rangle}{(1)\ \lambda x.t_1 : T_1 \to T_2 \lhd \Gamma}$$

(3)	$\mathcal{I}(\mathcal{U}, t)$ terminates	assumption
(4)	$\exists e.\quad \mathcal{I}(\mathcal{U}, t) = \{e\ \mathcal{I}_v(\mathcal{U}, \lambda x.t_1)\} = \{e\ \mathsf{isect}(\tau_1^s)\}$ where	$(\mathcal{I}.\mathrm{val})/(\mathcal{I}.\mathrm{abs})$ with (3)
	$\tau_1^s = \{S \to T \lhd \Gamma \mid (T \lhd \Gamma, x : \langle S \rangle) \in \mathcal{I}(\mathcal{U}, t_1)\}$	
(5)	$\mathcal{I}(\mathcal{U}, t_1)$ terminates	(3),(4)
(6)	$\exists \sigma'', \Gamma'', T_1', T_2'$ such that	ind.hyp with (5),(2)
(7)	$(T_2' \lhd \Gamma'', x : \langle T_1' \rangle) \in \mathcal{I}(\mathcal{U}, t_1)$	"
(8)	$\sigma''(T_2' \lhd \Gamma'', x : \langle T_1' \rangle) \equiv (T_2 \lhd \Gamma, x : \langle T_1 \rangle)$	"
(9)	$\sigma''(T_1' \to T_2' \lhd \Gamma'') \equiv (T_1 \to T_2 \lhd \Gamma)$	(8)
(10)	$(T_1' \to T_2' \lhd \Gamma'') \in \tau_1^s$	(7),(4)
(11)	$\exists E.(T_1' \to T_2' \lhd \Gamma'') \equiv E\ \mathsf{isect}(\tau_1^s)$	lem 3.56 with (10)
	$= (e := E)\ (e\ \mathsf{isect}(\tau_1^s))$	def 3.11

(12)	$(T_1 \to T_2 \lhd \Gamma) \equiv \sigma'' \ (T_1' \to T_2' \lhd \Gamma'')$	(9)
	$\equiv \sigma'' \ ((e := E) \ (e \ \mathsf{isect}(\tau_1^s)))$	lem 3.16 with (11)
	$\equiv (\sigma''(e := E)) \ (e \ \mathsf{isect}(\tau_1^s))$	lem 3.14
$\therefore$	σ is $\sigma''(e := E)$	(12),(4)

case: D ends with an application of Rule (app) of the form

$$\frac{(2)\ t_1 : T_2 \to T_1 \lhd \Gamma_1 \quad (3)\ t_2 : T_2 \lhd \Gamma_2}{(1)\ t_1\ t_2 : T_1 \lhd \Gamma_1 \Cap \Gamma_2}$$

(4)	$\mathcal{I}(\mathcal{U}, t_1\ t_2)$ terminates	assumption
(5)	$\mathcal{I}(\mathcal{U}, t_1)$ terminates	(4)
(6)	$\exists \sigma_1, T_1', \Gamma_1'$ such that	ind.hyp. with (5), (2)
(7)	$(T_1' \lhd \Gamma_1') \in \mathcal{I}(\mathcal{U}, t_1)$	"
(8)	$\sigma_1(T_1' \lhd \Gamma_1') \equiv (T_2 \to T_1 \lhd \Gamma_1)$	"
(9)	$\mathcal{I}(\mathcal{U}, t_2)$ terminates	(4)
(10)	$\exists \sigma_2, T_2', \Gamma_2'$ such that	ind.hyp. with (9), (3)
(11)	$(T_2' \lhd \Gamma_2') \in \mathcal{I}(\mathcal{U}, t_2)$	"
(12)	$\sigma_2(T_2' \lhd \Gamma_2') \equiv (T_2 \lhd \Gamma_2)$	"
(13)	$\mathsf{Dom}(\sigma_1) \cap \mathsf{Dom}(\sigma_2) = \emptyset$	def 3.57
(14)	$\exists e, \alpha, \Delta$ such that	def 3.57 with (4),(7),(11)
(15)	$\Delta = \{T_1' \stackrel{.}{\leq} T_2' \to e\ \alpha\} \cup \{S \stackrel{.}{\leq} S \mid x : S \in \Gamma_1' \Cap \Gamma_2'\}$	"
(16)	e does not occur in $T_1' \lhd \Gamma_1'$ or $T_2' \lhd \Gamma_2'$	"
(17)	$\{\sigma\ (e\alpha \lhd \Gamma_1' \Cap \Gamma_2') \mid \sigma \in \mathcal{U}(\Delta)\} \subseteq \mathcal{I}(\mathcal{U}, t_1\ t_2)$	"
	let $\sigma_3 = \sigma_2\sigma_1, e := E$ where $E\ \alpha = T_1$	E exists due to lem 3.17
(18)	$\sigma_3 T_1' = \sigma_1 T_1' \equiv T_2 \to T_1$ and $\sigma_3 \Gamma_1' = \sigma_1 \Gamma_1' = \Gamma_1$	(13),(16),(8)
(19)	$\sigma_3 T_2' = \sigma_2 T_2' \equiv T_2$ and $\sigma_3 \Gamma_2' = \sigma_2 \Gamma_2' = \Gamma_2$	(13),(16),(12)
(20)	$\sigma_3\ (e\ \alpha) = T_1$	def 3.11
(21)	$\sigma_3(T_2' \to e\ \alpha) \equiv T_2 \to T_1$	(19), (20)

(22)	σ_3 is a unifier for Δ	(18), (21)
(23)	$\mathcal{U}(\Delta)$ terminates	(4)
(24)	$\exists \sigma_4, \sigma_5.\quad \sigma_4 \in \mathcal{U}(\Delta)$ and $\sigma_3\Delta \equiv \sigma_5\sigma_4\Delta$	$\mathsf{covering}(\mathcal{U})$ with (22),(23)
(25)	$\sigma_3\ (\Gamma_1' \Cap \Gamma_2') = \Gamma_1 \Cap \Gamma_2$	(18), (19)
(26)	$\sigma_3(e\alpha) \equiv (\sigma_5\sigma_4)(e\alpha)$ and $\sigma_3(\Gamma_1' \Cap \Gamma_2') = (\sigma_5\sigma_4)(\Gamma_1' \Cap \Gamma_2')$	(24)
(27)	$(\sigma_4\ (e\ \alpha \lhd \Gamma_1' \Cap \Gamma_2')) \in \mathcal{I}(\mathcal{U}, t_1\ t_2)$	(24),(17)

(28)	$T_1 \lhd \Gamma_1 \Cap \Gamma_2 \equiv \quad \sigma_3\ (e\ \alpha \lhd \Gamma_1' \Cap \Gamma_2')$	(20),(25)
	$\equiv \quad (\sigma_5\sigma_4)\ (e\ \alpha \lhd \Gamma_1' \Cap \Gamma_2')$	(26)
	$\equiv \quad \sigma_5\ (\sigma_4\ (e\ \alpha \lhd \Gamma_1' \Cap \Gamma_2'))$	lem 3.14
$\therefore$	$\sigma = \sigma_5$	(27),(28)

case: D ends with an application of Rule (omega) of the form

$$\frac{}{(1)\ v : \omega \lhd \Gamma_\omega}$$

(2)	$\mathcal{I}(\mathcal{U}, v)$ terminates	assumption
(3)	$\exists e, \tau_1.\quad \mathcal{I}(\mathcal{U}, v) = \{e\ \tau_1\}$	($\mathcal{I}$.val) with (2)
	let $\sigma_1 = (e := \omega)$	
(4)	$\sigma_1\ (e\ \tau_1) = (\omega \lhd \Gamma_\omega)$	def 3.11
$\therefore$	$\sigma = \sigma_1$	(4),(3)

case: D ends with an application of Rule (int) of the form

$$\frac{(2)\ v : T_1 \lhd \Gamma_1 \quad (3)\ v : T_2 \lhd \Gamma_2}{(1)\ v : T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2}$$

(4)	$\mathcal{I}(\mathcal{U}, v)$ terminates	assumption
(5)	$\exists \sigma_1, e, T, \Gamma$ such that	ind.hyp./($\mathcal{I}$.val) with (4),(2)
(6)	$\mathcal{I}(\mathcal{U}, v) = \{e\ (T \lhd \Gamma)\}$	"
(7)	$\sigma_1\ (e\ (T \lhd \Gamma)) \equiv (T_1 \lhd \Gamma_1)$	"
(8)	$\exists \sigma_2.\quad \sigma_2\ (e\ (T \lhd \Gamma)) \equiv (T_2 \lhd \Gamma_2)$	ind.hyp. with (6),(3)
	let $\sigma_3 = (e := \sigma_1(e\ \boxdot) \Cap \sigma_2(e\ \boxdot))$	

$$
\begin{array}{lrll}
(9) & \sigma_3\ (e\ (T \lhd \Gamma)) = & (\sigma_1(e\ \boxdot))\ (T \lhd \Gamma) \Cap (\sigma_2(e\ \boxdot))\ (T \lhd \Gamma) & \\
 & = & \sigma_1((e\ \boxdot)\ (T \lhd \Gamma)) \Cap \sigma_2((e\ \boxdot)\ (T \lhd \Gamma) & \text{lem 3.14} \\
 & = & \sigma_1(e\ (T \lhd \Gamma)) \Cap \sigma_2(e\ (T \lhd \Gamma)) & \text{def 3.11} \\
 & \equiv & (T_1 \lhd \Gamma_1) \Cap (T_2 \lhd \Gamma_2) & (7),(8) \\
 & = & T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2 & \text{def 3.22}
\end{array}
$$

$\therefore \quad \sigma = \sigma_3 \quad$ (9),(6)

case: D ends with an application of Rule (evar) of the form

$$\frac{(2)\ v : T \lhd \Gamma}{(1)\ v : e\ T \lhd e\ \Gamma}$$

(3)	$\mathcal{I}(\mathcal{U}, v)$ terminates	assumption
(4)	$\exists \sigma_1, e_1, \Gamma_1, T_1$ such that	ind.hyp./($\mathcal{I}$.val) with (3),(2)
(5)	$\mathcal{I}(\mathcal{U}, v) = \{e_1\ (T_1 \lhd \Gamma_1)\}$	"
(6)	$\sigma_1\ (e_1\ (T_1 \lhd \Gamma_1)) \equiv (T \lhd \Gamma)$	"
	let $\sigma_2 = (e_1 := e\ (\sigma_1(e_1\ \boxdot))$	

$$
\begin{array}{lrll}
(7) & \sigma_2\ (e_1\ (T_1 \lhd \Gamma_1)) = & (\sigma_2\ e_1)\ (T_1 \lhd \Gamma_1) & \text{def 3.11} \\
 & = & (e\ (\sigma_1(e_1\ \boxdot)))\ (T_1 \lhd \Gamma_1) & \text{def 3.11} \\
 & = & e\ ((\sigma_1(e_1\ \boxdot))\ (T_1 \lhd \Gamma_1)) & \text{def 3.11} \\
 & = & e\ (\sigma_1\ ((e_1\ \boxdot)\ (T_1 \lhd \Gamma_1))) & \text{lem 3.14} \\
 & = & e\ (\sigma_1\ (e_1\ (T_1 \lhd \Gamma_1))) & \text{def 3.11} \\
 & \equiv & e\ (T \lhd \Gamma) & \text{def 3.7 with (6)} \\
 & = & e\ T \lhd e\ \Gamma & \text{def 3.22}
\end{array}
$$

$\therefore \quad \sigma = \sigma_2 \quad$ (7),(5)

$\square$

3.5 Examples

This section presents a set of examples which are used to compare the behaviour of our type inference algorithm $\mathcal{I}$ with the original type inference algorithm for System E based on β-unification [15]. Although a newer β-unification algorithm was later published [6], an implementation is available only for the original algorithm, and so we use that one for comparison purposes. This implementation is called the System E type inference tool and is available at [14]. Algorithm $\mathcal{I}$ was implemented by this author and is called `evcr` [26].

We demonstrate and compare the output of both implementations on selected examples from the System E Inference Report [13], a collection of 61 terms that were used to demonstrate the original System E type inference algorithm. Compared to the System E type inference tool, `evcr` succeeds and fails on exactly the same set of terms with the exception of Term 8 (discussed below) which is supported by the System E type inference tool but not by `evcr`. The output of `evcr` on all 61 terms from this report is included in Appendix B.

Term 1 in the System E Inference Report is a single variable `x`:

```
$ x;;
: a[] <| x : a[]
```

In the inferred typing `a[] <| x :  a[]`, the letter `a` represents an E-variable, while `[]` represents a simple type variable. Because Algorithm $\mathcal{I}$ wraps each simple type variable in an E-variable, only one simple type variable is needed in practice and so its name is irrelevant. For example, given E-variables `a` and `b`, the types `a[]` and `b[]` can for all intents and purposes function as distinct type variables, where $\boxdot, \texttt{a} := (\texttt{[]} := \bar{S}, \boxdot), \texttt{b} := (\boxdot, \texttt{[]} := \bar{T})$ will substitute the simple type $\bar{S}$ for the first `[]` and the simple type $\bar{T}$ for the second `[]`. By Lemma 3.17 (Type variables), it is also possible to construct substitutions that will substitute any two types (not just simple types) for `a []` and `b []`.

The System E type inference tool infers a similar typing of $a_0 \lhd x : a_0$, where a_0 indicates a type variable. Similarly to `evcr`, the System E type inference tool also requires only a single type variable and names it a_0. The only difference between these two typings is that `evcr` represents the type variable by a simple type variable `[]` wrapped in an E-variable `a`.

Term 2 is the identity function:

```
$ \x.x;;
: a (b[] -> b[])
```

In the inferred typing, all lowercase letters (i.e. `a` and `b`) are E-variables. The System E type inference tool infers $e_0\ a_0 \rightarrow e_0\ a_0$ for this term, however both typings are principal in their respective systems. In `evcr`, principal typings are instantiated via substitutions, and so the outer expansion variable `a` is required to allow the substitution of an outer expansion. In the System E type inference tool, principal typings are instantiated via expansions, and so no outer expansion variable is needed.

Term 3 is the application of one term variable to another:

```
$ x y;;
: a[] <| y : b[],  x : b[] -> a[]
```

In this typing, `a[]` and `b[]` effectively represent distinct type variables. The System E type inference tool infers $a_0 \lhd y : e_2\ a_0, x : e_2\ a_0 \rightarrow a_0$ for this term. In this typing, a_0 and $e_2\ a_0$ effectively represent distinct type variables, corresponding respectively to `a[]` and `b[]` in the `evcr` typing. Just as `evcr` uses only one simple type variable `[]`, the System E type inference tool also uses only one type variable α_0, and strategically applies E-variables so that each distinct type variable occurs in a different namespace.

Term 4 is an example of "self application":

```
$ x x;;
: a[] <| x : (b[] -> a[]) ^ b[]
```

The ability to infer typings for programs involving self application is crucial to our encoding of object-orientation which is based on the self-application semantics (see Section 5.4 of Chapter 5). This example is not possible in Hindley/Milner-based type inference because x is used polymorphically and no context information is provided about x. In contrast, `evcr` is able to infer a typing for this term compositionally without any context information for x, inferring an intersection of the two types at which x is used. The System E type inference tool infers the similar typing $a_0 \lhd x : (e_2 a_0 \to a_0) \cap e_2 a_0$ which again differs only in the approach to using E-variable namespaces to name distinct type variables.

Term 8 uses the Y-combinator to compute the factorial of 2 using Church Numerals:

```
$ (\h.(\x.h (x x)) (\x.h (x x))) (\f.\n.n (\v.\x.\y.y)
  (\x.\y.x) (\f.\x.f x) (\g.\x.n (f (n (\p.\s.s
  (p (\x.\y.y)) (\f.\x.f (p (\x.\y.y) f x))) (\s.s
  (\f.\x.x) (\f.\x.x)) (\x.\y.x)) g) x)) (\f.\x.f (f x))
(No output due to non-termination)
```

Although this term is not strongly normalising, the System E type inference tool still manages to find a type for it: $(e_0 e_0 (e_2 a_0 \to a_0) \cap e_0 e_0 e_2 (e_2 a_0 \to a_0)) \to e_0 (e_0 e_2 e_2 a_0 \to e_0 a_0)$. This type describes a function taking a first parameter that is used twice, and a second parameter that is used once, which is what we find when inspecting the Church Numeral `\f.\x.f (f x)` for 2 that is computed by this example. The first parameter `f` is used twice, first at type $e_0 e_0 (e_2 a_0 \to a_0)$ and second at type $e_0 e_0 e_2 (e_2 a_0 \to a_0)$, then under the E-variable e_0, the second parameter `x` is used once at type $e_0 e_2 e_2 a_0$, and then also under the same E-variable e_0, this function gives a result of type $e_0 a_0$ which is also the same as the result type of the first use of parameter `f`.

Interestingly, if given the Y-combinator subterm alone, the System E type inference tool will fail to terminate just as `evcr` would. This example highlights an important difference between our type inference approach and the standard β-unification approach used in the System E type inference tool. β-unification is effectively a form of whole-program analysis. The only reason it can succeed for the larger term, and fail for the Y-combinator subterm is that it *needs* the context of the larger program to know how many times the Y-combinator will be unfolded. In contrast, the approach used by `evcr` is strictly compositional and will attempt to analyse the Y-combinator subterm on its own, and lacking the required context of how many times the Y-combinator will be unfolded, it will continue unfolding forever. Although our type inference algorithm fails to analyse the Y-combinator term, it should be noted that the Hindley/Milner type inference system also cannot analyse the Y-combinator term and solves the problem by introducing a primitive fixed-point operator with a special typing rule. Adding a fixed-point operator is important future work for System E$^\mathsf{v}$, and may require using System E's non-linear types to account for an unknown (or infinite) number of uses of a given term.

Term 14 applies a function that discards its argument:

```
$ (\z.\x.x x) (\y.y);;
: a (((b[] -> c[]) ^ b[]) -> c[])

> \x.x x
```

The System E type inference algorithm infers the typing $(e_0(e_2a_0 \to a_0) \cap e_0e_2a_0) \to e_0a_0$, which is almost the same in structure as the one inferred by `evcr` except that it inserts an additional E-variable e_0. This additional E-variable is not inserted into the `evcr` typing due to our use of the value restriction, which we can see when we look at the System E$^\mathsf{v}$ typing derivation for the subterm `\x.x x` (where `evcr` syntax is used in the derivation):

$$\dfrac{\dfrac{\dfrac{}{\texttt{x} : \texttt{b[] -> c[]} \lhd \texttt{x} : \langle \texttt{b[] -> c[]} \rangle}\,(\text{var}) \qquad \dfrac{}{\texttt{x} : \texttt{b[]} \lhd \texttt{x} : \langle \texttt{b[]} \rangle}\,(\text{var})}{\texttt{x x} : \texttt{c[]} \lhd \texttt{x} : \langle \texttt{((b[] -> c[]) \^{} b[])} \rangle}\,(\text{app})}{\texttt{\textbackslash x.x x} : \texttt{((b[] -> c[]) \^{} b[]) -> c[]} \lhd}\,(\text{abs})$$

In order to insert an E-variable corresponding to e_0 into the `evcr` typing, it would need to be introduced into the derivation when the subject is the term `x x`, and this is specifically prevented by the value restriction because `x x` is not a value.

3.6 Efficiency

As discussed in Chapter 2, opus creates a significant amount of branching in the search path in order to produce a covering unifier set, and this branching makes the algorithm too inefficient to be useful in practice. In this chapter, we compare the efficiency of opusβ and opus by running our implementation of both algorithms side by side on the 58 convergent terms from the System E Inference Report.

Figure 3.2 shows the number of $\overrightarrow{\mathsf{opus}_\beta}$ steps required by opusβ and the number of $\overrightarrow{\mathsf{opus}}$ steps required by opus to find a typing for each of these 58 terms. In many cases, opus ran too long and was forceably stopped after 5 minutes (indicated by "Timed out" in the figure). Of the cases in which opus successfully terminated, Term 41 was the worst case at 954,239 $\overrightarrow{\mathsf{opus}}$ steps. By comparison, opus successfully terminated on all 58 terms with the worst case being Term 11 at 1,104 $\overrightarrow{\mathsf{opus}_\beta}$ steps. The cumulative 3,481 $\overrightarrow{\mathsf{opus}_\beta}$ steps used by opusβ took 9 seconds to complete on an Intel® Core$^{\text{TM}}$ i5-2400 CPU @ 3.10GHz

Term	$\overrightarrow{\mathsf{opus}_\beta}$ steps	$\overrightarrow{\mathsf{opus}}$ steps
01	0	0
02	0	0
03	2	2
04	2	2
05	4	4
06	11	213
07	16	30
09	33	Timed out
10	33	Timed out
11	1104	Timed out
12	275	Timed out
13	245	Timed out
14	10	14
15	8	12
16	8	12
17	25	231
18	8	12
19	8	12
20	148	Timed out
21	106	Timed out
22	92	Timed out
23	78	Timed out
24	58	Timed out
25	64	Timed out
26	127	Timed out
27	41	2156
29	26	292
30	16	218
31	17	25
32	8	12
33	17	375
34	15	225
35	13	215
36	13	215
37	11	213
38	16	24
39	8	12
40	57	9609
41	65	954239
43	56	102857
44	13	289
45	50	Timed out
46	36	Timed out
47	30	89663
48	38	1313
49	46	52391
50	10	14
51	16	30
52	13	21
53	11	213
54	49	Timed out
55	118	Timed out
56	118	Timed out
57	6	6
58	26	60
59	6	6
60	26	40
61	25	Timed out

Figure 3.2: Performance of $\mathsf{opus}\beta$ vs opus with functions

with a single-threaded implementation. By comparison, the cumulative 1,215,249 $\overrightarrow{\mathsf{opus}}$ steps (of the cases that terminated without timing out) took 203 seconds to complete on the same hardware.

As we shall see in Chapter 5, the difference in efficiency becomes even more pronounced when analysing object-oriented programs, to the extent that records or objects with more than 2 fields cannot in practice be analysed without some optimisations, such as those adopted by $\mathsf{opus}\beta$.

3.7 Summary

This chapter presented System E$^{\text{v}}$, the first of three increments of System E$^{\text{vcr}}$. The main contribution of System E$^{\text{v}}$ is to introduce the value restriction, which is necessary to safely integrate constants and extensible records in the following chapters. This system is established to be typesafe under the call-by-value semantics. System E$^{\text{v}}$ also simplified some definitions and removed some redundancies of the original System E in order to simplify the proofs and implementation. Then, we presented a type inference algorithm, called $\mathcal{I}$, that is parameterised by the choice of unification algorithm, and established that it terminates, is correct and finds principal typing sets whenever the chosen unification algorithm terminates, is correct and finds covering unifier sets respectively. Algorithm $\mathcal{I}$ is currently compatible with both the existing opus algorithm, which finds covering unifier sets, and with our own algorithm opusβ, which sacrifices the covering unifier sets property for an increase in efficiency. Next, we compared our implementation of Algorithm $\mathcal{I}$ with the System E type inference tool and found that it matched the results of the System E type inference tool on all but one of the 61 λ-calculus examples from the System E Inference Report, failing, where it did, in a way that was predictable and consistent with the compositional approach of our algorithm. Finally, we examined the efficiency issues of opus that motivated our desire to create a derivative of opus called opusβ that adopts efficiency ideas from β-unification.

Chapter 4

System E$^{\text{vc}}$: Constants

System E$^{\text{vc}}$ extends System E$^{\text{v}}$ with *constants.* Constants are the built-in values in a language, including both data values (e.g. 1, 2, 3, `true`, `false`) and functions (e.g. $+$, $-$, $*$, $/$, `not`, `or`). Constants are desirable in any practical language because they allow for efficient implementations and effective type checking. However, they also play a critical role in our approach to extensible records since field labels will be treated as constants.

This chapter is organised as follows. Section 4.1 gives motivations for introducing constants. Section 4.2 discusses how the value restriction is used to preserve type safety in the presence of constants. Section 4.3 formally presents System E$^{\text{vc}}$ and establishes that it is typesafe. Section 4.4 adapts the type inference and unification algorithms to support constants. Finally, Section 4.6 summarises the contributions of this chapter.

4.1 Motivations

Integers, booleans and associated operations can in theory be represented in the terms of the pure λ-calculus using Church encodings [18], but in practice it is desirable to include these as built-ins of a programming language.

One reason to use built-in constants is of course that they allow for faster processing than Church encodings. But another reason is that they allow better type checking. For example, in many programming languages, we would expect the application `not` 3 to generate a type error since the argument 3 is not boolean. However, the Church encoding of this application, $(\lambda m.\lambda a.\lambda b.m\ b\ a)\ (\lambda s.\lambda z.s\ s\ s\ z)$, is still a valid λ term in its own right, and is even reducible and typable in System E$^{\text{v}}$ (reducing to $\lambda a.\lambda b.(\lambda s.\lambda z.s\ s\ s\ z)\ b\ a$ with type $e_1(e_2e_3\alpha \to e_2((((e_4\alpha \to e_5\alpha \to e_3\alpha \to e_6\alpha) \sqcap e_4\alpha) \sqcap e_5\alpha) \to e_6\alpha)))$, although it results in a term that has no clear meaning in either a numeric or boolean interpretation. If we wish to classify the application `not` 3 as having a type error, then it would be beneficial to define `not` and 3 as built-in constants with special typing rules requiring the argument of `not` to have a boolean type.

Apart from the usual reasons for desiring constants, another reason that is especially important for this dissertation is that we would like to interpret extensible records as functions from field labels to field values. Since it will then be possible to pass labels as arguments to functions, just like integers and booleans, it makes sense to treat labels as just another kind of constant and reuse all of the type machinery for constants.

4.2 Integrating Constants with E-Variables

If we were to naively add constants to System E, then nonsense applications such as `not` 3 would be typable via Rule (omega):

$$\frac{}{\texttt{not}\ 3 : \omega \lhd \Gamma_\omega}\ \text{(omega)}$$

This is not desirable since applications that are unable to be reduced to values should be flagged as type errors. In general, Rule (omega) allows any application to be typed regardless of whether or not a reduction rule has been defined for it. This rule is at least justified for the pure λ-terms where any function may be safely applied to any argument. However, once constants are introduced, functions can be defined that require a certain type of argument and must reject all others, just as we would like the `not` function to accept only boolean arguments and reject all others. Also, constants allow for the introduction of values that are non-functional and cannot be applied to anything. For example, in many languages the application 3 `false` should not be typable since 3 is not a function.

The solution adopted by System E^{vc} is to make use of the value restriction of System E^{v}. Under the value restriction, Rule (omega) is restricted to values and hence may not be used on application terms. Since values are never reduced in the call-by-value semantics, we are assured that evaluation will never get stuck for any terms assigned type ω by Rule (omega).

It is worth noticing that even with the value restriction in place, an application of a function *can* still be assigned type ω if it can be derived that the function has a return type of ω. For example, the application (`not true`) can still be assigned type ω in System E^{vc} via the following derivation:

$$\frac{\texttt{not} : \texttt{Bool} \to \omega \lhd \Gamma_\omega \quad \texttt{true} : \texttt{Bool} \lhd \Gamma_\omega}{\texttt{not true} : \omega \lhd \Gamma_\omega}\ \text{(app)}$$

To support this, function `not` would be given the principal type $\texttt{Bool} \to e\, \texttt{Bool}$ requiring a parameter of type `Bool` and returning a value of a type that is any expansion of type `Bool` (including ω).

4.3 Type System

System E^{vc} introduces two new elements: *constants* (such as 3 and `false`), and *type constants* (such as `Int` and `Bool`). This section will define System E^{vc} as an extension of System E^{v}, following the conventions set forth in Section 1.5 of Chapter 1. That is, any amended definitions, lemmas and theorems given in this chapter will override their respective original definitions, lemmas and theorems given in the previous chapter.

4.3.1 Terms and Reductions

System E^{vc} introduces a single new term form into the term language: the *type constant.* The amended term language, free variables, term substitution and reduction rule definitions are presented below, with the amendments underlined.

Amendment to Definition 3.1 (Terms for E^{vc})**.** Let metavariables x, y and z range over the countably infinite set of *term variables* (also called *variables*). The syntax for terms and their metavariable conventions are given below.

$$\begin{array}{rll}
s, t, u ::= v \mid s\ t & & \textbf{terms} \\
v ::= x \mid \underline{c} \mid \lambda x.t & & \textbf{values} \\
\underline{c} ::= \ldots & & \textbf{constants}
\end{array}$$

□

The set of constants is left unspecified, and the relevant definitions of System E$^{\text{vc}}$ that follow are parameterised by the choice of constants.

Amendment to Definition 3.2 (Free variables for E$^{\text{vc}}$)**.** The *free variables* $\mathsf{fv}(t)$ of a term t are defined by the following rules:

$$\begin{aligned}
\mathsf{fv}(x) &= \{x\} \\
\underline{\mathsf{fv}(c)} &= \emptyset \\
\mathsf{fv}(\lambda x.t) &= \mathsf{fv}(t)\backslash\{x\} \\
\mathsf{fv}(s\ t) &= \mathsf{fv}(s) \cup \mathsf{fv}(t)
\end{aligned}$$

□

Amendment to Definition 3.3 (Term substitution for E$^{\text{vc}}$)**.** A *term substitution* $t[x := v]$ of value v for x in t is defined by the following rules:

$$\begin{array}{rll}
x[x := v] = v & & \\
y[x := v] = y & & y \neq x \\
\underline{c[x := v]} = c & & \\
(\lambda x.t)[x := v] = \lambda x.t & & \\
(\lambda y.t)[x := v] = \lambda z.t[y := z][x := v] & & y \neq x, z \notin \mathsf{fv}(v) \cup \{x\} \\
(s\ t)[x := v] = s[x := v]\ t[x := v] & &
\end{array}$$

□

Amendment to Definition 3.5 (Reduction for E^{vc}). The *reduction rules* are defined as follows:

$$\begin{array}{rl}
\underline{c\ v} > \phi(c\ v) & \text{if } \phi(c\ v) \text{ is defined} \\
(\lambda x.t)\ v > t[x := v] & \\
t\ s > t'\ s & \text{if } t > t' \\
v\ t > v\ t' & \text{if } t > t'
\end{array}$$

where ϕ is a partial function that takes an application of the form $(c\ v)$ and returns a value. □

The above definition allows for built-in operations to take a single argument only, although binary operations can be supported by bundling the two arguments into a Church pair. For example, the + operation can be defined as follows:

$$\begin{array}{rr}
\phi(+\ v) = \mathsf{sum}(a, b) & \text{if } v\ (\lambda x.\lambda y.x) > \ldots > a \text{ and } v\ (\lambda x.\lambda y.y) > \ldots > b \\
& \text{and } a,\ b \text{ are integer constants} \\
& \text{and } \mathsf{sum}(a, b) \text{ calculates the sum of } a \text{ and } b
\end{array}$$

4.3.2 Types and Expansions

Typically, constants such as 3, `false` and `"Hello World"` have corresponding types `Int`, `Bool` and `Str` which are introduced into the type system as *type constants.* Supporting type constants requires additions to the type syntax and the rules for expansion application.

Amendment to Definition 3.6 (Syntax for E^{vc}). Let metavariables e, f, g range over the countably infinite set of *E-variables* and let metavariables α, β, γ range over the countably infinite set of *simple type variables.* The syntactic categories and their metavariable conventions are given below.

$S, T, U ::= \bar{T} \mid \mathbb{T}$		**types**
$\bar{S}, \bar{T}, \bar{U} ::= \alpha \mid \underline{C} \mid S \rightarrow T$		**simple types**
$\underline{C} ::= \ldots$		**type constants**
$\mathbb{S}, \mathbb{T}, \mathbb{U} ::= \omega \mid S \sqcap T \mid e\, T$		**expansion types**
$E, F, G ::= \omega \mid E \sqcap F \mid e\, E \mid \sigma$		**expansions**
$\sigma ::= \boxdot \mid \sigma, \alpha := \bar{T} \mid \sigma, e := E$		**substitutions**
$X ::= \alpha \mid e$		**variables**
$K ::= T \mid E$		**entities**

□

The set of type constants is left unspecified, and the relevant definitions of System E$^{\mathrm{vc}}$ that follow are parameterised by the choice of type constants.

Amendment to Definition 3.11 (Expansion application for E$^{\mathrm{vc}}$). The rules for applying expansions to expansions, types and E-variables are defined as follows:

$\omega\ K$	$= \omega$	$\sigma\ \omega$	$= \omega$
$(E \sqcap F)\ K$	$= E\ K \sqcap F\ K$	$\sigma\ (K_1 \sqcap K_2)$	$= \sigma\ K_1 \sqcap \sigma\ K_2$
$(e\ E)\ K$	$= e\ (E\ K)$	$\sigma\ (e\ K)$	$= (\sigma\ e)\ K$
$\boxdot\ e$	$= e\ \boxdot$	$\sigma\ (S \rightarrow T)$	$= \sigma\ S \rightarrow \sigma\ T$
$\boxdot\ \alpha$	$= \alpha$	$\sigma\ \boxdot$	$= \sigma$
$(\sigma, X := K)\ X$	$= K$	$\sigma\ (\sigma', X := K)$	$= \sigma\sigma', X := \sigma\ K$
$(\sigma, X := K)\ X'$	$= \sigma\ X'$ if $X \neq X'$	$\underline{\sigma\ C}$	$= C$

□

4.3.3 Typing Derivations

The typing rule for constants should consider both data constants and function constants. In System E^{vc}, a data constant such as `true` would have the simple type `Bool` or any expansion of `Bool` including, for example, `Bool` $\sqcap$ `Bool` and ω. For data constants, it therefore suffices to have a typing rule that assigns a simple type to each data constant with all other types derivable by subsequent applications of the expansion typing rules (int), (omega) and (evar).

Functional constants are more complicated. A functional constant such as `not` would have the simple type `Bool` $\rightarrow e$ `Bool`. That is, it requires a `Bool` argument and returns a result whose type can be any expansion of `Bool`. To allow the return type to be any expansion of `Bool`, the typing rule for constants should incorporate the ability to apply substitutions (in this case, for e). The function `not` also ought to have the equivalent type (`Bool` $\sqcap\, \omega$) $\rightarrow e$ `Bool` for Lemma 3.27 (Equivalent types) to hold, and this type is not derivable via the expansion rules or via a substitution. Therefore, we need a typing rule for constants that also incorporates type equivalence.

In summary, the process for deriving a type for a constant is:

1. Begin with a *raw* type (e.g. `Bool` for `true` and `Bool` $\rightarrow e$ `Bool` for `not`)
2. Apply the desired substitution
3. Derive the desired equivalent type

For step 1, we define the raw types of constants as follows.

Definition 4.1 (typeof for E^{vc})**.** typeof is a total function from constants to type constants and function types. For any constant c, $\mathsf{typeof}(c)$ is called the *raw type* of c. $\square$

For steps 2 and 3, we introduce the new typing derivation rule for constants as follows.

Amendment to Definition 3.24 (Typing derivations for E^{vc})**.** A *typing derivation* (metavariable D) is a proof tree of a typing judgement. The rules for deriving valid typing judgements are given below.

$$\text{(var)}\ \frac{}{x : \bar{T} \lhd x : \langle \bar{T} \rangle} \qquad\qquad \text{(omega)}\ \frac{}{v : \omega \lhd \Gamma_\omega}$$

$$\underline{\text{(con)}}\ \frac{}{c : \bar{T} \lhd \Gamma_\omega}\ \bar{T} \equiv \sigma\ \mathsf{typeof}(c) \qquad\qquad \text{(int)}\ \frac{v : \tau_1 \quad v : \tau_2}{v : \tau_1 \Cap \tau_2}$$

$$\text{(abs)}\ \frac{t : T \lhd \Gamma, x : \langle S \rangle}{\lambda x.t : S \to T \lhd \Gamma} \qquad\qquad \text{(evar)}\ \frac{v : \tau}{v : e\ \tau}$$

$$\text{(app)}\ \frac{t : S \to T \lhd \Gamma_1 \quad s : S \lhd \Gamma_2}{t\ s : T \lhd \Gamma_1 \Cap \Gamma_2}$$

□

The new rule (con) assigns to a constant any simple type that is equivalent to any substitution of the constant's raw type. Invoking the equivalence relation in Rule (con) ensures that Lemma 3.27 (Equivalent types) holds, while applying the substitution σ ensures that Lemma 3.28 (Expansion) holds. Also, restricting the type to a simple type ensures that Lemma 3.25 (Typing derivations) holds, forcing all intersection types, E-variable application types and the ω type to be introduced via rules (int), (evar) and (omega) respectively.

Since the actual set of constants is left unspecified, the following requirement ensures that for any given set of constants, subject reduction and progress still hold in the presence of constant applications such as `not true`.

Requirement 4.2 (ϕ-typability for E$^{\text{vc}}$)**.** *If $c\ v : T \lhd \Gamma_\omega$, then $\phi(c\ v)$ is defined and $\phi(c\ v) : T \lhd \Gamma$.* □

Two additional lemmas hold for types in System E$^{\text{vc}}$. The first lemma establishes that no expressiveness is lost by the restriction of Rule (con) to simple types, and the second lemma asserts that the term context of a typing of a constant is always empty.

Lemma 4.3 (Constant types for E$^{\text{vc}}$)**.** *Given any T, σ and c, if $T \equiv \sigma\ \mathsf{typeof}(c)$, then $c : T \lhd \Gamma_\omega$.*

Proof.

(1)	$\sigma\ \mathsf{typeof}(c)$ is a simple type	lem 3.12
(2)	$c : \sigma\ \mathsf{typeof}(c) \lhd \Gamma_\omega$	(con) with (1)
(3)	$\sigma\ \mathsf{typeof}(c) \equiv T$	assumption
$\therefore$	$c : T \lhd \Gamma_\omega$	lem 3.27 with (2) and (3)

□

Lemma 4.4 (Constant term contexts for E$^{\mathrm{vc}}$). *Given any c, T and Γ, if $c : T \lhd \Gamma$, then $\Gamma = \Gamma_\omega$.*

Proof. Let D be the derivation of $c : T \lhd \Gamma$. The proof is by induction on the structure of D. There are only four cases of typing rules where the subject can be a constant.

case: D is the following application of Rule (con):

$$\frac{}{c : \bar{T} \lhd \Gamma_\omega}\ \bar{T} \equiv \sigma\ \mathsf{typeof}(c)$$

where $T = \bar{T}$ and (1) $\Gamma = \Gamma_\omega$. The result is (1).

case: D is the following application of Rule (omega):

$$\frac{}{c : \omega \lhd \Gamma_\omega}$$

where $T = \omega$ and (1) $\Gamma = \Gamma_\omega$. The result is (1).

case: D ends with the following application of Rule (int):

$$\frac{c : T_1 \lhd \Gamma_1 \quad c : T_2 \lhd \Gamma_2}{c : T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2}$$

where $T = T_1 \Cap T_2$ and $\Gamma = \Gamma_1 \Cap \Gamma_2$. By the induction hypothesis, $\Gamma_1 = \Gamma_\omega$ and $\Gamma_2 = \Gamma_\omega$. Therefore $\Gamma = \Gamma_\omega$.

case: D ends with the following application of Rule (evar):

$$\frac{c : T_1 \lhd \Gamma_1}{c : e\ T_1 \lhd e\ \Gamma_1}$$

where $T = e\ T_1$ and (1) $\Gamma = e\ \Gamma_1$. By the induction hypothesis, $\Gamma_1 = \Gamma_\omega$ and so from (1) we get (2) $\Gamma = e\ \Gamma_\omega$. By Definition 3.7 and Definition 3.20, we have $e\ \Gamma_\omega = \Gamma_\omega$, and so from (2) we get $\Gamma = \Gamma_\omega$.

□

Finally, we extend the proofs of lemmas and theorems from the previous chapter that are affected by the introduction of constants and type constants.

Restatement of Lemma 3.25 (Typing derivations for E$^{\mathrm{vc}}$)**.** *Each typing judgement can be the conclusion of at most one typing rule.*

Proof. Consider any application of a typing rule that concludes with the typing judgement J denoting $t : T \lhd \Gamma$. The cases are the same as before (p.88), with one new case for constants.

case: If t is a value,

case: If $T = \bar{T}$

case: If $t = c$, then J was derived by (con).

□

Restatement of Lemma 3.27 (Equivalent types for E$^{\mathrm{vc}}$)**.** *Given any t, T, T' and Γ, if $t : T \lhd \Gamma$ and $T \equiv T'$, then $t : T' \lhd \Gamma$.*

Proof. Let D be the derivation of $t : T \lhd \Gamma$. The proof is by induction on the structure of D. The cases are the same as before (p.89), with one new case for Rule (con).

case: t is v.

case: $T \equiv T'$ was derived by structural congruence rule $\dfrac{(1)\ T_1 \equiv T_1' \quad (2)\ T_2 \equiv T_2'}{T_1 \to T_2 \equiv T_1' \to T_2'}$

By Lemma 3.25, D can end with one of three possible rules; as before, Rule (var) and Rule (con), but now also Rule (con) whose case is shown below:

case: D is the following application of Rule (con):

$$\frac{}{(3)\ c : T_1 \to T_2 \lhd \Gamma_\omega}\ (4)\ T_1 \to T_2 \equiv \sigma\ \mathsf{typeof}(c)$$

Then,

(5)	$T_1 \to T_2 \equiv T_1' \to T_2'$	Assumption
(6)	$T_1' \to T_2' \equiv \sigma\ \mathsf{typeof}(c)$	def. 3.7 with (4) and (5)
$\therefore$	$c : T_1' \to T_2' \lhd \Gamma_\omega$	(con) with (6)

□

Restatement of Lemma 3.28 (Expansion for E$^{\text{vc}}$). *Given any t, T, Γ and E, if $t : T \lhd \Gamma$ and if t is a value or E is a substitution, then $t : E\ T \lhd E\ \Gamma$.*

Proof. Let D be the derivation of $t : T \lhd \Gamma$. The cases are the same as before (p.91), with one new case for Rule (con).

case: E is σ.

case: D is the following application of Rule (con)

$$\frac{}{(1)\ c : \bar{T_1} \lhd \Gamma_\omega}\ (2)\ \bar{T_1} \equiv \sigma_1\ \mathsf{typeof}(c)$$

Then,

(3)	$\sigma\ \bar{T_1} \equiv \sigma\ (\sigma_1\ \mathsf{typeof}(c))$	lem. 3.16 with (2)
(4)	$\sigma\ \bar{T_1} \equiv (\sigma\sigma_1)\ \mathsf{typeof}(c)$	lem. 3.14 with (3)
$\therefore$	$c : \sigma\ \bar{T_1} \lhd \Gamma_\omega$	lem 4.3 with (4)

□

Restatement of Lemma 3.29 (Term substitution for E$^{\text{vc}}$). *Given any v, t, x, S, T, Γ_1 and Γ_2, if $v : S \lhd \Gamma_2$ and $t : T \lhd \Gamma_1, x : \langle S\rangle$, then $t[x := v] : T \lhd \Gamma_1 \Cap \Gamma_2$.*

Proof. Let D be the derivation of $t : T \lhd \Gamma_1, x : \langle S\rangle$. The cases are the same as before (p.94), with one new case for Rule (con).

case: D is the following application of Rule (con):

$$\frac{}{(1)\ c : \bar{T_1} \lhd \Gamma_\omega}\ (2)\ \bar{T} \equiv \sigma\ \mathsf{typeof}(c)$$

where (3) $t = c$, (4) $T = \bar{T_1}$, (5) $\Gamma_1 = \Gamma_\omega$ and (6) $S \equiv \omega$.

(7)	$v : S \lhd \Gamma_2$	Assumption
(8)	$v : \omega \lhd \Gamma_2$	lem. 3.27 with (7) and (6)
(9)	$\Gamma_2 = \Gamma_\omega$	(omega) with (8)
(10)	$c : \bar{T_1} \lhd \Gamma_1 \Cap \Gamma_2$	(1) with (5) and (9)
$\therefore$	$c[x := v] : \bar{T_1} \lhd \Gamma_1 \Cap \Gamma_2$	def. 3.3 with (10)

□

Restatement of Theorem 3.30 (Subject reduction for E^{vc}). *If $t : T \lhd \Gamma$ and $t > t'$, then $t' : T \lhd \Gamma$.*

Proof. The cases are the same as before (p.97), but with one new case for constant applications.

case: $c\ v > \phi(c\ v)$ if $\phi(c\ v)$ is defined.

By Requirement 4.2, $\phi(c\ v) : T \lhd \Gamma$.

□

Restatement of Theorem 3.31 (Progress for E^{vc}). *If $t\ s : T \lhd \Gamma_\omega$, then there exists a term t' such that $t\ s > t'$.*

Proof. The proof is by induction on the structure of $t\ s$.

The derivation of $t\ s : T \lhd \Gamma_\omega$ ends with

$$\frac{(1)\ t : S \to T \lhd \Gamma_\omega \quad s : S \lhd \Gamma_\omega}{t\ s : T \lhd \Gamma_\omega}$$

The cases are the same as before (p.98), but with one new case for constants.

case: t is v_1.

case: s is v_2.

case: v_1 is c. By the assumption, $c\ v_2 : T \lhd \Gamma_\omega$. Therefore, $\phi(c\ v_2)$ is defined by Requirement 4.2. Hence, $c\ v_2 > \phi(c\ v_2)$. □

4.4 Type Inference

This section extends our type inference algorithm $\mathcal{I}$ and unification algorithm $\mathsf{opus}\beta$ to support constants, and re-establishes theorems and lemmas stated in the previous chapter that are affected by the introduction of constants and type constants.

4.4.1 Algorithm $\mathsf{opus}\beta$

This section extends our type unification algorithm $\mathsf{opus}\beta$ to support type constants. A type constant can potentially be unified with a simple type variable, an E-variable application or itself. However, no new rules need to be added to the $\xrightarrow{\mathsf{opus}\beta}$ relation to support these cases:

- The unification of a type constant with a simple type variable is already handled by Rule (T-unify) because type constants are included in the category of simple types.
- The unification of a type constant with an E-variable application is already handled by Rule (EA-unify), also for the reason that type constants are included in the category of simple types.
- The unification of a type constant with itself is not needed because the $\xrightarrow{\mathsf{opus}\beta}$ relation is invoked only on unsolved constraints, by Definition 3.47 (opus_β reduction).

All that is required to support type constants is a small change to the definition of $\mathsf{varstruct}$ which is used by ovars, which in turn is used by the occurs check of Rule (T-unify).

Amendment to Definition 3.43 ($\mathsf{varstruct}$ for E^{vc})**.** The total function $\mathsf{varstruct}$ takes any type, unification constraint or unification constraint set and returns a variable structure. It is defined by the following rules:

$$
\begin{aligned}
&\mathsf{varstruct}(\alpha) && = \alpha \mapsto \emptyset \\
&\underline{\mathsf{varstruct}(C)} && = \emptyset \\
&\mathsf{varstruct}(\omega) && = \emptyset \\
&\mathsf{varstruct}(S \to T) = \mathsf{varstruct}(S \cap T) && = \mathsf{varstruct}(S) \sqcup \mathsf{varstruct}(T) \\
&\mathsf{varstruct}(e\ T) && = e \mapsto \mathsf{varstruct}(T) \\
&\mathsf{varstruct}(S \leq T) && = \mathsf{varstruct}(S) \sqcup \mathsf{varstruct}(T) \\
&\mathsf{varstruct}(\Delta) && = \bigsqcup \{\mathsf{varstruct}(\delta) \mid \delta \in \Delta\}
\end{aligned}
$$

where $\mathcal{V}_1 \sqcup \mathcal{V}_2 = \{e \mapsto (\mathcal{V}_1/e) \sqcup (\mathcal{V}_2/e) \mid e \in \mathsf{Dom}(\mathcal{V}_1) \cup \mathsf{Dom}(\mathcal{V}_2)\}$ □

Restatement of Theorem 3.50 (Correctness of $\mathsf{opus}\beta$ for E$^{\mathrm{vc}}$). *Given any Δ and σ, then $\sigma \in \mathsf{opus}\beta(\Delta) \implies \mathsf{solved}(\sigma\Delta)$.*

Proof. The original proof (p.110) remains unchanged since $\mathsf{opus}\beta$ is defined to stop as soon as all constraints are solved, independently of what rules are used to solve the constraint. □

4.4.2 Algorithm $\mathcal{I}$

Algorithm $\mathcal{I}$, too, requires only a very minor change to support constants.

Amendment to Definition 3.57 (Algorithm $\mathcal{I}$ for E$^{\mathrm{vc}}$). The type inference algorithm $\mathcal{I}$ takes a unification algorithm and a term and returns a distinct sequence of typings. It is defined by the following rules:

Terms

$\mathcal{I}(\mathcal{U}, v) =$	$\{e\ \mathcal{I}_v(\mathcal{U}, v)\}$ $\quad e$ fresh	$(\mathcal{I}.\text{val})$
$\mathcal{I}(\mathcal{U}, t\ s) =$	$\{\sigma\ (e\ \alpha \lhd \Gamma_1 \Cap \Gamma_2)$ $\mid (T \lhd \Gamma_1) \in \mathcal{I}(\mathcal{U}, t), (S \lhd \Gamma_2) \in \mathcal{I}(\mathcal{U}, s)$ $\sigma \in \mathcal{U}(\{T \leq S \to e\ \alpha\} \cup \{U \leq U \mid x : U \in \Gamma_1 \Cap \Gamma_2\}), e, \alpha$ fresh$\}$	$(\mathcal{I}.\text{app})$

Values

$\mathcal{I}_v(\mathcal{U}, x) =$	$\alpha \lhd x : \langle\alpha\rangle$ $\quad \alpha$ fresh	$(\mathcal{I}.\text{var})$
$\underline{\mathcal{I}_v(\mathcal{U}, c)} =$	$\mathsf{typeof}(c) \lhd \Gamma_\omega$	$(\mathcal{I}.\text{con})$
$\mathcal{I}_v(\mathcal{U}, \lambda x.t) =$	$\mathsf{isect}(\{(S \to T \lhd \Gamma) \mid (T \lhd \Gamma, x : \langle S\rangle) \in \mathcal{I}(\mathcal{U}, t)\})$	$(\mathcal{I}.\text{abs})$

□

The new rule ($\mathcal{I}$.con) simply uses $\mathsf{typeof}(c)$ to supply the type of the constant c. The rest of the rules remain as before, and in particular, Rule ($\mathcal{I}$.val) will be used to wrap the inferred type of a constant in a fresh E-variable.

As before, Algorithm $\mathcal{I}$ satisfies three properties.

Restatement of Theorem 3.58 (Termination of $\mathcal{I}$ for E$^{\text{vc}}$)**.** *Given any $\mathcal{U}$ and t, if each use of $\mathcal{U}$ by $\mathcal{I}(\mathcal{U}, t)$ terminates, then $\mathcal{I}(\mathcal{U}, t)$ terminates.*

Proof. Straightforward by induction on the structure of t. □

Restatement of Theorem 3.59 (Correctness of $\mathcal{I}$ for E$^{\text{vc}}$)**.** *Given any $\mathcal{U}$, τ and t, if $\tau \in \mathcal{I}(\mathcal{U}, t)$, then $t : \tau$.*

Proof. The cases are the same as before (p.116), with one new case for constants.

case: $t = c$.

(1)	$\exists e.\quad \mathcal{I}(\mathcal{U}, c) = \{e\ \mathcal{I}_v(\mathcal{U}, c)\} = \{e\ \mathsf{typeof}(c) \lhd \Gamma_\omega\}$	$(\mathcal{I}.\text{val})/(\mathcal{I}.\text{con})$
(2)	$c : \mathsf{typeof}(c) \lhd \Gamma_\omega$	(con)
(3)	$c : e\ \mathsf{typeof}(c) \lhd \Gamma_\omega$	(evar) with (2)
$\therefore$	if $\tau \in \mathcal{I}(\mathcal{U}, c)$, then $c : \tau$	(1),(3)

□

Restatement of Theorem 3.60 (Principality of $\mathcal{I}$ for E$^{\mathrm{vc}}$). *Given any $\mathcal{U}$, t and τ', if* covering($\mathcal{U}$) *and $t : \tau'$ and $\mathcal{I}(\mathcal{U}, t)$ terminates, then there exists a substitution σ and a typing $\tau \in \mathcal{I}(\mathcal{U}, t)$ such that $\tau' \equiv \sigma\tau$.*

Proof. Let D be the derivation of $t : \tau'$. The cases are the same as before (p.118), with one new case for Rule (con).

case: D is the following application of Rule (con):

$$\frac{}{(1)\ c : \bar{T} \lhd \Gamma_\omega}\ (2)\ \bar{T} \equiv \sigma_1\ \mathsf{typeof}(c)$$

$$\begin{array}{lll}
(3) & \exists e.\quad \mathcal{I}(\mathcal{U}, c) = \{e\ \mathcal{I}_v(\mathcal{U}, c)\} = \{e\ \mathsf{typeof}(c) \lhd \Gamma_\omega\} & (\mathcal{I}.\text{val})/(\mathcal{I}.\text{con}) \\
(4) & (e := \sigma_1)\ (e\ \mathsf{typeof}(c) \lhd \Gamma_\omega) = (\sigma_1\ \mathsf{typeof}(c) \lhd \Gamma_\omega) & \text{def } 3.11 \\
 & \qquad \equiv \tau' & (2) \\
\therefore & \sigma \text{ is } (e := \sigma_1) & (4),(3)
\end{array}$$

□

4.5 Examples

In this section, we demonstrate our implementation `evcr` on a variety of examples and examine the behaviour of Algorithm $\mathcal{I}$ in practice with respect to constants. Although constants represent only a small addition to our system, some interesting issues turn up in practice, and these will be discussed.

Inferred types for data constants such as `3` are always wrapped in E-variables for generality:

```
$ 3;;
: a Int
```

The E-variable `a` acts as a placeholder for the insertion of an expansion during type inference when it is discovered how `3` is to be used. For example, consider the abstraction:

```
$ \x. x + x;;
: a ((Int ^ Int) -> b Int)
```

The parameter is used twice at type `Int`, hence its intersection type `Int ^ Int`. In order to apply this function to the argument `3`, the type inference algorithm will unify the parameter type `Int ^ Int` with the argument type `a Int` and produce a substitution that replaces `a` by an intersection expansion.

Since System E^{vc} does not directly allow functional constants to take more than one parameter, binary operators can be defined to accept Church pairs:

```
1  $ (2,3);;
2  : a ((b Int -> c Int -> d[]) -> d[])
3  = (\x.\y.\f.f x y) 2 3
4  > (\y.\f.f 2 y) 3
5  > \f.f 2 3

6  $ add;;
7  : a (((b (c d[] -> c (w -> d[])) -> Int) ^ (e (w -> f (g[] -> g[])) -> Int)) -> h Int)

8  $ add (2,3);;
9  : a Int
10 = + ((\x.\y.\f.f x y) 2 3)
11 > + ((\y.\f.f 2 y) 3)
12 > + (\f.f 2 3)
13 > 5
```

The surface syntax `(2,3)` on Line 1 is automatically translated to the Church pair
`f.f 2 3` which has type `a ((b Int -> c Int -> d[]) -> d[])`. The predefined functional constant `add` on Line 6 has a parameter type compatible with Church pairs. It is an intersection of two types, indicating that `add` uses its parameter twice. In the first use, the given Church pair is used at type `b (c d[] -> c (w -> d[])) -> Int` where `w` (which represents type omega[1]) is used to indicate that the second value of the pair is unused. In the second use, the given Church pair is used at type `e (w -> f (g[] -> g[])) -> Int`, this time indicating that the first value of the pair is not used. In other words, this parameter type reveals that the given Church pair will be used twice, once for its first value, and once for its second value. The result of `add` is an `Int`, wrapped in an E-variable for generality.

Note that the infix notation `2 + 3` is simply translated into an ordinary application of `add`:

[1]The lowercase letter `w` is reserved to mean ω, while all other lowercase letters indicate E-variables.

```
$ 2 + 3;;
: a Int
= add ((\x.\y.\f.f x y) 2 3)
> add ((\y.\f.f 2 y) 3)
> add (\f.f 2 3)
> 5
```

As we should expect in a system with constants, it is possible for there to be no typings found for a given application if a function is applied to the wrong types of arguments:

```
$ 3 + false;;
: No typings found
```

This type error is only caught due to the value restriction, without which this application could have been assigned type ω.

Intersection type polymorphism also works well with constants. The following program requires as input a function that can take both integers to integers and booleans to booleans, and tests whether that function acts as the identity on two particular values:

```
$ \f.f 3 == 3 && f true;;
: a (((b Int -> Int) ^ (c Bool -> Bool)) -> d Bool)
```

Here, `&&` is the logical-AND operator, and `==` is the equality operator. This example is not supported by Hindley/Milner-style type inference since polymorphic function parameters are not allowed. The application of the above program to the identity function produces the expected answer `true` of the expected type `a Bool`:

```
$ (\f.f 3 == 3 && f true) (\x.x);;
: a Bool
= (\f.and ((\x.\y.\f.f x y) (== ((\x.\y.\f.f x y) (f 3) 3)) (f true))) (\x.x)
> and ((\x.\y.\f.f x y) (== ((\x.\y.\f.f x y) ((\x.x) 3) 3)) ((\x.x) true))
> and ((\x.\y.\f.f x y) (== ((\x.\y.\f.f x y) 3 3)) ((\x.x) true))
> and ((\x.\y.\f.f x y) (== ((\y.\f.f 3 y) 3)) ((\x.x) true))
> and ((\x.\y.\f.f x y) (== (\f.f 3 3)) ((\x.x) true))
> and ((\x.\y.\f.f x y) true ((\x.x) true))
> and ((\y.\f.f true y) ((\x.x) true))
> and ((\y.\f.f true y) true)
> and (\f.f true true)
> true
```

4.5.1 Issues

Although constants are our most trivial addition to System E, they are not always well behaved. This section describes some of the issues.

Both opusβ and β-unification use asymmetric unification constraints of the form $S \stackrel{.}{\leq} T$ where S is assumed to be an argument type and T a parameter type. However, it is possible to define functional constants that break this assumption. An example is the `if/else` ternary function which takes a boolean argument followed by two arbitrary arguments which must have the same type as each other. This is problematic because it will cause constraints to be generated between two argument types rather than between a parameter type and an argument type. Hence, when generating unification constraints for applications of the `if/else` ternary function, it may make sense to fall back to the full Algorithm opus in that specific case.

But even if we use the full Algorithm opus when unifying argument types with argument types, it can sometimes happen that the types of abstractions and the types of functional constants do not unify very well, as illustrated by the following example:

```
$ not;;
: a (Bool -> b Bool)

$ \x.x;;
: c (d [] -> d [])

$ if (true) not else (\x.x);;
: e (Bool -> Bool)
```

On their own, the types for `not` and `\x.x` appear to be satisfactory. One important feature that both types have is that the result types are wrapped in an E-variable which allows, via an expansion, the result to have type ω if needed. But when these two types are unified with each other, this E-variable is eliminated because the identity function must return exactly the same type as its parameter which, in unification with the type of `not`, must be exactly type `Bool`. The inability of the return type to be ω causes a problem for programs such as the one below:

```
$ let ignore = \x.3;;
: a (w -> b Int)

$ let myfalse = (if (true) not else \x.x) true;;
```

```
: Bool

$ ignore myfalse
: No typings found
```

Since `ignore` ignores its argument, it requires its argument to have type ω. And since `myfalse` cannot be given type ω, it is impossible to apply `ignore` to `myfalse`, even though this application should be typesafe.

4.6 Summary

This chapter presented System E^{vc}, an extension of System E^{v} with constants. Despite being a relatively simple extension, we exposed some of the problems associated with introducing constants. First, a naive attempt to introduce constants would allow unsafe applications to be typable via Rule (omega), and type safety would be lost. In System E^{vc}, type safety was preserved by adopting a value restriction to rule out unsafe applications of constants. Second, while constants appear to be well behaved in most normal usages, an issue can arise when two argument types are unified, and this will require further study.

Chapter 5

System $\mathrm{E}^{\mathrm{vcr}}$: Extensible Records

System $\mathrm{E}^{\mathrm{vcr}}$ is the final system presented in this dissertation. It builds on the intersection types and expansion variables of System E, the value restriction of System E^{v} and the constants of System E^{vc} to develop a system of extensible records that supports first-class polymorphism, compositional type inference and object-orientation.

This chapter is organised as follows. Section 5.1 gives an overview of the approach System $\mathrm{E}^{\mathrm{vcr}}$ takes to integrate extensible records with expansion variables. Section 5.2 formally defines System $\mathrm{E}^{\mathrm{vcr}}$ and establishes its properties. Section 5.3 formally defines the type inference and unification algorithms and establishes their properties. Section 5.4 shows how all of the problem examples from the introductory chapter can be encoded into our calculus and demonstrates our type inference algorithm on each of them. Section 5.5 revisits the efficiency differences between opus and opusβ in the context of extensible records. Finally, Section 5.6 summarises the contributions of this chapter.

5.1 Integrating Extensible Records with E-Variables

In System $\mathrm{E}^{\mathrm{vcr}}$, extensible records are treated as functions from labels to terms, and this allows us to reuse all of the existing type machinery of System E that was designed around functions. The syntax for defining an extensible record resembles that for defining a pattern matching function in which labels are the only patterns, and the default case is the empty record $\{\}$:

$$\texttt{let john} \quad = \quad \texttt{name} \rightarrow \texttt{"John Smith"} \sqcap \texttt{age} \rightarrow 31 \sqcap \{\}$$

The field selection `john.age` now becomes a straightforward function application, `john age`, which conveniently has a similar syntax. As is the case with field selection for traditional extensible records, the application `john age` is evaluated by scanning the cases of the function from left to right until a case with a matching label is found.

Following Kopylov [38], we interpret record types as intersections of function types such that `john` now has the type:

$$\texttt{name} \rightarrow \texttt{String} \sqcap \texttt{age} \rightarrow \texttt{Int}$$

In a way that resembles subtyping, our type system also allows `john` to be assigned weaker types with fields omitted, such as

$$\texttt{name} \rightarrow \texttt{String}$$

which happens to be the function type that is needed in order to apply `john` to the label `name`. Subtype polymorphism demands that the record `john` be able to have both of the above types *simultaneously.* System E^{vcr} provides the same effect without subtype polymorphism by using an intersection type to intersect both of the above types together:

$$(\texttt{name} \rightarrow \texttt{String} \sqcap \texttt{age} \rightarrow \texttt{Int}) \sqcap (\texttt{name} \rightarrow \texttt{String})$$

This type is *linear*, indicating that `john` is to be used once at type $\texttt{name} \rightarrow \texttt{String} \sqcap \texttt{age} \rightarrow \texttt{Int}$ and once at type $\texttt{name} \rightarrow \texttt{String}$.

One consequence of treating extensible records as functions from labels to values is that labels are now first-class citizens. In System E^{vcr}, it is possible for a term variable to represent an unknown label, and so the function $\lambda x.(\texttt{name} \rightarrow \texttt{"John"} \sqcap \texttt{employed} \rightarrow \texttt{true} \sqcap \{\})\ x$ is possible, which takes an unknown label as a parameter and then selects a field of a record matching that unknown label. First-class labels are an interesting consequence of our treatment of records as functions over labels, although this was not a specific design goal of System $\mathrm{E}^{\mathrm{vcr}}$. We do not push this idea to the heights reached in the system by Leijen [40] where term variables can appear in place of the labels of actual fields. For example, Leijen's system allows functions such as $\lambda x.\lambda y.(x \to \texttt{"John"} \Cap y \to \texttt{true} \Cap \{\})$ which takes two unknown labels and constructs a record using those two labels, but this is not supported by System $\mathrm{E}^{\mathrm{vcr}}$.

The most significant departure of System $\mathrm{E}^{\mathrm{vcr}}$ from the traditional extensible record type systems is its treatment of row variables. In fact, System $\mathrm{E}^{\mathrm{vcr}}$ does not include row variables in its type language, although types that serve the same purpose as row variables can be constructed from more fine-grained primitives. Before we show how row variables can be constructed, we will illustrate how row variables work at the coarse-grained level by assuming the existence of pre-built row variables represented by metavariable ρ. The purpose of beginning the explanation at the coarse-grained level is not only to begin at a point of familiarity, but also to show why the coarse-grained level is insufficient in System $\mathrm{E}^{\mathrm{vcr}}$ and why, ultimately, we must build row variables out of more fine-grained primitives.

At the coarse-grained level, our approach follows that of constrained quantification introduced by Harper and Pierce [25] (see Section 1.3.1 in Chapter 1 for a review) in that we use predicates directly on row variables. One difference is that we use predicates directly on each occurrence of a row variable rather than once globally, which can be illustrated by the following example:

$$\lambda x.\texttt{id} \to 3 \Cap x : \rho\backslash\texttt{id} \to (\texttt{id} \to \texttt{Int} \Cap \rho\backslash\texttt{id})$$

The row variable $\rho\backslash\texttt{id}$ represents an unknown record type lacking a field labelled `id`, and the list of labels following $\backslash$ is called a *label constraint*. Note that each row variable is uniquely identified by the combination of its name and its label constraint, and it is only because both occurrences of $\rho\backslash\texttt{id}$ in the above type have the same name *and* label constraint that these refer to the same row variable. By comparison, the row variables $\rho\backslash\texttt{id}$ and $\rho\backslash\texttt{age}$ are *not* the same row variable despite having the same name ρ. This simplifies the theory since we do not need to add a third component to typings that carries around information about each row variable.

This coarse-grained view of row variables is, however, insufficient when attempting to solve unification constraints. For example, consider the constraint

$$\alpha \to \bar{T} \leq \rho\backslash\mathsf{age}$$

in which the left type is a record type with one field whose label is unknown (of type α), and the right type is a row variable representing an unknown record type lacking a field labelled `age`. Without some way to refine the simple type variable α so that it may prevent any later substitution of the label `age`, the best that can be done to solve this constraint is to pick some concrete label not equal to `age` to substitute for α. The choice would however be arbitrary, and so this constraint of course has no single most-general unifier. It does not even have a finite covering unifier set because there are infinitely many labels not equal to `age` that could be substituted for α and that are unrelated to one another.

The example above illustrates a need to place label constraints directly on simple type variables. We call $\alpha[L]$ a *constrained simple type variable*, and it represents an unknown label that must not be equal to any of the labels in the label constraint L. Once we have constrained simple type variables, row variables can now be constructed from this and other primitives. For example, a row variable $\rho\backslash L$, for some given label constraint L, can be encoded as:

$$e\ (\beta[L] \to f\ \gamma)$$

This type behaves as a row variable because it can be expanded via e to any intersection of function types, the domain of which can be any label besides those listed in L. That is, it describes record types that do not use any of the field names in L. The unification constraint that caused earlier difficulty can now be expressed as:

$$\alpha \to \bar{T} \leq e\ (\beta[\mathsf{age}] \to f\ \gamma)$$

This unification constraint now has a most general unifier $(\alpha := \beta[\mathsf{age}], e := (f := (\gamma := \bar{T})))$ which results in the following solved constraint:

$$\beta[\mathsf{age}] \to \bar{T} \leq \beta[\mathsf{age}] \to \bar{T}$$

Based on this encoding of row variables, we return to the example function used in the literature review on row variables (Section 1.3.1), which can now be assigned the following type in System E$^{\mathrm{vcr}}$:

$$\lambda x.(\texttt{id} \to 3 ⩀ x) : e\ (\alpha[\texttt{id}] \to f\ \alpha) \to (\texttt{id}\texttt{:}\texttt{Int}, e\ (\alpha[\texttt{id}] \to f\ \alpha))$$

Here, the encoded row variable $e\ (\alpha[\texttt{id}] \to f\ \alpha)$ indicates that the type of x must be a record type not already containing the label `id`, or more precisely, that the type of x must be composed of function types where each parameter type must be a label not equal to `id`. A possible argument that satisfies this constraint is the following extensible record:

$$\texttt{a} \to 3 ⩀ \texttt{b} \to 4 ⩀ \{\} : \texttt{a} \to \texttt{Int} ⩀ \texttt{b} \to \texttt{Int}$$

Substituting this extensible record for the parameter x and substituting its type for the encoded row variable gives the following extensible record:

$$\texttt{id} \to 3 ⩀ \texttt{a} \to 3 ⩀ \texttt{b} \to 4 ⩀ \{\} : \texttt{id} \to \texttt{Int} ⩀ \texttt{a} \to \texttt{Int} ⩀ \texttt{b} \to \texttt{Int}$$

It is important to note that the constraint on parameter x prevents only the *type* of the argument from containing an `id` field, but does not prevent the argument itself from containing an `id` field. For example, the following possible argument also satisfies the constraint:

$$\texttt{id} \to \texttt{"hidden"} ⩀ \texttt{a} \to 3 ⩀ \texttt{b} \to 4 ⩀ \{\} : \texttt{a} \to \texttt{Int} ⩀ \texttt{b} \to \texttt{Int}$$

This is possible because System E$^{\mathrm{vcr}}$ allows an extensible record to be assigned a type with fields omitted (in this case, field `id`) in a way that resembles subtyping. Substituting this argument for the parameter x and substituting its type for the encoded row variable gives the following extensible record:

$$\texttt{id} \to 3 ⩀ \texttt{id} \to \texttt{"hidden"} ⩀ \texttt{a} \to 3 ⩀ \texttt{b} \to 4 ⩀ \{\} : \texttt{id} \to \texttt{Int} ⩀ \texttt{a} \to \texttt{Int} ⩀ \texttt{b} \to \texttt{Int}$$

This type is valid because the new field $\texttt{id} \to 3$ overrides the old field $\texttt{id} \to \texttt{"hidden"}$ so that the resulting record effectively has only one `id` field of value 3.

Finally, we turn to the use of expansion variables in record types. Expansion variables should be inserted into typings by the type inference algorithm in such a way that the inferred typing represents the range of the different typings that are possible for a given term. In the case of extensible records, this must include types where some fields have been omitted, and must also include intersections of two alternative types for the same record. To address the first requirement, expansion variables are placed around each field type, and by substituting the ω expansion for one of these expansion variables, it is possible to obtain a type with that field omitted. For example, if `john` has type e_1 `(name` $\rightarrow$ `String)` $\cap$ e_2 `(age` $\rightarrow$ `Int)`, we can derive another type with field `age` omitted by substituting ω for e_2 giving e_1 `(name` $\rightarrow$ `String)` $\cap$ ω, which is equivalent to e_1 `(name` $\rightarrow$ `String)`.

To address the second requirement, expansion variables are also placed around whole record types containing multiple fields, and by substituting an intersection expansion for one of these expansion variables, it is possible to obtain an intersection type with each branch having a different alternative type. For example, if `john` has type e_3 `(`e_1 `(name` $\rightarrow$ `String)` $\cap$ e_2 `(age` $\rightarrow$ `Int))`, we can derive another type for `john` that intersects together several other derivable types by substituting for e_3 an intersection expansion $E_1 \cap E_2$ where E_1 and E_2 derive alternative types for the left and right branches of the intersection.

5.2 Type System

This section formally defines System E$^{\mathrm{vcr}}$ as an extension of System E$^{\mathrm{vc}}$.

5.2.1 Terms and Reductions

To support extensible records, we need to extend the term language as well as the rules for free variables, term substitution and reduction.

Amendment to Definition 3.1 (Terms for E$^{\mathrm{vcr}}$)**.** Let metavariables x, y and z range over the countably infinite set of *term variables* (also called *variables*). Let metavariable l range over the countably infinite set of *labels*. The syntax for terms and their metavariable conventions are given below.

$$\begin{aligned} s,t,u &::= v \mid s\ t && \textbf{terms} \\ v &::= x \mid c \mid \lambda x.t \mid \underline{l \to t \Cap v} && \textbf{values} \\ c &::= \ldots \mid \underline{l} \mid \underline{\{\}} && \textbf{constants} \end{aligned}$$

□

The values are extended to include *extensions*, where the extension $l \to t \Cap v$ is a function that takes label l to term t and otherwise acts as value v. The constants are extended to include labels and the *empty record* $\{\}$.

Extensions resemble pattern matching functions where labels are the only patterns that can be matched. Being functions, extensions are subtly different from extensible records. A traditional extensible record $\{$`field=2+3,` $\{\}\}$ would be reducible to $\{$`field=5,` $\{\}\}$, but the extension (`field2` $\to 2+3 \Cap \{\}$) is not reducible at all since it is already a value. The subterm $2+3$ is not reduced until the function is applied to the label `field2`. However, the standard behaviour of extensible records can be encoded by using the syntactic sugar $\{\ \ l = t_1, t_2\ \ \}$ to denote $(\lambda x.\lambda y.(l \to x \Cap y))\ t_1\ t_2$.

Amendment to Definition 3.2 (Free variables for E$^{\text{vcr}}$)**.** The *free variables* $\mathsf{fv}(t)$ of a term t are defined by the following rules:

$$\begin{aligned} \mathsf{fv}(x) &= \{x\} \\ \mathsf{fv}(c) &= \emptyset \\ \mathsf{fv}(\lambda x.t) &= \mathsf{fv}(t)\backslash\{x\} \\ \underline{\mathsf{fv}(l \to t \Cap v)} &= \mathsf{fv}(t) \cup \mathsf{fv}(v) \\ \mathsf{fv}(s\ t) &= \mathsf{fv}(s) \cup \mathsf{fv}(t) \end{aligned}$$

□

Amendment to Definition 3.3 (Term substitution for E$^{\text{vcr}}$)**.** A *term substitution* $t[x := v]$ of value

v for x in t is defined by the following rules:

$$\begin{array}{rll}
x[x := v] &= v & \\
y[x := v] &= y & y \neq x \\
c[x := v] &= c & \\
(\lambda x.t)[x := v] &= \lambda x.t & \\
(\lambda y.t)[x := v] &= \lambda z.t[y := z][x := v] & y \neq x, z \notin \mathsf{fv}(v) \cup \{x\} \\
\underline{(l \to t \Cap v)[x := v_1]} &= l \to t[x := v_1] \Cap v[x := v_1] & \\
(s\ t)[x := v] &= s[x := v]\ t[x := v] &
\end{array}$$

□

Amendment to Definition 3.5 (Reduction for E$^{\text{vcr}}$)**.** The *reduction rules* are defined as follows:

$$\begin{array}{rll}
c\ v &> \phi(c\ v) & \text{if } \phi(c\ v) \text{ is defined} \\
(\lambda x.t)\ v &> t[x := v] & \\
\underline{(l \to t \Cap v)\ l} &> t & \\
\underline{(l \to t \Cap v)\ l_1} &> v\ l_1 & \text{if } l_1 \neq l \\
t\ s &> t'\ s & \text{if } t > t' \\
v\ t &> v\ t' & \text{if } t > t'
\end{array}$$

where ϕ is a partial function that takes an application of the form $(c\ v)$ and returns a value. □

For example, the field selection `john age` is reduced to 31 via the following reduction path:

$$\begin{array}{ll}
& (\texttt{name} \to \texttt{"John Smith"} \Cap \texttt{age} \to 31 \Cap \{\})\ \texttt{age} \\
> & (\texttt{age} \to 31 \Cap \{\})\ \texttt{age} \\
> & 31
\end{array}$$

These reduction rules work only if the selected field actually exists in the record. If selection of the non-existent field `address` is tried, reduction will become stuck at the application $\{\}$ `address` for which no reduction rule exists:

$$\begin{aligned} & (\texttt{name} \to \texttt{"John Smith"} \capdot \texttt{age} \to 31 \capdot \{\})\ \texttt{address} \\ > {} & (\texttt{age} \to 31 \capdot \{\})\ \texttt{address} \\ > {} & \{\}\ \texttt{address} \end{aligned}$$

By the progress theorem, the typing rules ensure that this cannot happen.

5.2.2 Types and Expansions

This section extends the definitions of types and expansions to account for constrained simple type variables, labels and the empty record type.

Amendment to Definition 3.6 (Syntax for E$^{\mathrm{vcr}}$)**.** Let metavariables e, f, g range over the countably infinite set of *E-variables* and let metavariables α, β, γ range over the countably infinite set of *simple type variables.* The syntactic categories and their metavariable conventions are given below.

$S, T, U ::= \bar{T} \mid \mathbb{T}$	**types**
$\bar{S}, \bar{T}, \bar{U} ::= \underline{\alpha[L]} \mid C \mid S \to T$	**simple types**
$C ::= \ldots \mid \underline{l} \mid \underline{\{\}}$	**type constants**
$\mathbb{S}, \mathbb{T}, \mathbb{U} ::= \omega \mid S \capdot T \mid e\ T$	**expansion types**
$E, F, G ::= \omega \mid E \capdot F \mid e\ E \mid \sigma$	**expansions**
$\sigma ::= \boxdot \mid \sigma, \alpha := \bar{T} \mid \sigma, e := E$	**substitutions**
$X ::= \underline{\alpha[L]} \mid e$	**variables**
$K ::= T \mid E$	**entities**

□

The simple types are extended with *constrained simple type variables* $\alpha[L]$ where L is a *label con-*

straint specifying a finite set of labels written as a comma-delimited list. If L is empty, then $\alpha[L]$ is a simple type variable and is written α. The type constants are extended with labels and the *empty record type* $\{\}$. The variables are extended with constrained simple type variables.

Note that the definition of constrained simple type variables subsumes the definition of simple type variables and thus relevant proofs need only consider the general case of constrained simple type variables. Also note that it is now the combination of variable name and label constraint that uniquely identifies a constrained simple type variable such that if $L_1 \neq L_2$, then $\alpha[L_1] \neq \alpha[L_2]$. This means that $\alpha[L_1]$ and $\alpha[L_2]$ can be independently assigned by substitutions despite having the same name.

Given that label constraints are now attached to simple type variables, it is necessary to restrict substitutions so that only simple types that meet constraint L can be substituted for some $\alpha[L]$. The following definitions treat this formally.

Definition 5.1 (Meets judgement for E$^{\text{vcr}}$)**.** The *meets judgement* $\bar{T}\upharpoonright L$ asserts that simple type $\bar{T}$ satisfies the label constraint L and is defined as follows:

$$\bar{T}\upharpoonright L \text{ iff } L \neq \emptyset \implies ((\bar{T} = l \text{ and } l \notin L) \text{ or } (\bar{T} = \alpha[L'] \text{ and } L' \supseteq L))$$

□

The definition of the meets judgement can be understood as follows. If the label constraint L is empty, then it is met by all simple types. If L is not empty, then it is met only if $\bar{T}$ is a label that is not in L, or $\bar{T}$ is a constrained simple type variable whose label constraint contains at least all of the labels in L.

Definition 5.2 (Valid substitutions for E$^{\text{vcr}}$)**.** A substitution σ is valid if and only if, for each assignment $\alpha[L] := \bar{T}$ in σ, it is the case that $\bar{T}\upharpoonright L$. □

Convention 5.3 (Valid substitutions for E$^{\text{vcr}}$)**.** *Only valid substitutions are used.*

The following lemma establishes that if a simple type meets a constraint L, then it also meets any weaker constraint (i.e. that is a subset of L).

Lemma 5.4 (Meets subsumption for E$^{\text{vcr}}$)**.** *Given any $\bar{T}$, L and L', if $\bar{T}\upharpoonright L$ and $L \supseteq L'$, then $\bar{T}\upharpoonright L'$.*

Proof. By Definition 5.1 with (1) we have the following cases:

case: (3) L is empty.

(4)	L' is empty	(3),(2)
$\therefore$	$\bar{T}\restriction L'$	def 5.1 with (4)

case: L is not empty. Then by Definition 5.1 with (1) we have the following cases:

case: (3) $\bar{T} = l$ and (4) $l \notin L$.

(5)	$l \notin L'$	(4),(2)
$\therefore$	$\bar{T}\restriction L'$	def 5.1 with (3),(5)

case: (3) $\bar{T} = \alpha[L'']$ and (4) $L'' \supseteq L$.

(5)	$L'' \supseteq L'$	(4),(2)
$\therefore$	$\bar{T}\restriction L'$	def 5.1 with (3),(5)

□

The following lemma establishes that meets judgements are stable under type substitution.

Lemma 5.5 (Meets substitution for $\mathrm{E}^{\mathrm{vcr}}$). *Given any $\bar{T}$, L and σ, if $\bar{T}\restriction L$, then $(\sigma\ \bar{T})\restriction L$.*

Proof. The proof is by induction on the structure of σ. Following Definition 5.1, we have the following cases:

case: L is empty. Then $\sigma\ \bar{T}$ is a simple type by Lemma 3.12 and $(\sigma\ \bar{T})\restriction L$ by Definition 5.1.

case: L is not empty. Then by Definition 5.1 with (1) we have the following cases:

case: (2) $\bar{T} = l$ and (3) $l \notin L$.

(4)	$l\restriction L$	def 5.1 with (3)
$\therefore$	$(\sigma\ l)\restriction L$	def 3.11 with (4)

case: (2) $\bar{T} = \alpha[L']$ and (3) $L' \supseteq L$. There are three cases for σ:

case: $\sigma = \boxdot$. Immediate.

case: $\sigma = (\sigma', \alpha[L'] := \bar{T}_1)$ where (4) $\bar{T}_1\restriction L'$.

(5)	$\sigma\ \alpha[L'] = \bar{T}_1$	def 3.11
(6)	$\sigma\ \alpha[L']\restriction L'$	(4),(5)
$\therefore$	$\sigma\ \alpha[L']\restriction L$	lem 5.4 with (6),(3)

case: $\sigma = (\sigma', X := K)$ where $X \neq \alpha[L']$.

(4)	$(\sigma'\ \alpha[L'])\lceil L$	ind.hyp. (σ' smaller)
$\therefore$	$(\sigma\ \alpha[L'])\lceil L$	def 3.11 with (4)

□

Label constraints operate at the level of individual labels. However, at the level of extensible records, it will also be useful to determine whether the type of a particular record lacks a certain label. This is handled by the next judgement.

Definition 5.6 (Lacks judgement for $\mathrm{E}^{\mathrm{vcr}}$)**.** The *lacks judgement* T::l is defined by the following rules:

$$\begin{array}{rr}
\omega::l & \\
(S \cap T)::l & \text{if } S::l \text{ and } T::l \\
(e\ T)::l & \text{if } T::l \\
(S \to T)::l & \text{if } S \equiv \bar{S} \text{ and } \bar{S}\lceil l
\end{array}$$

□

Effectively, the lacks judgement T::l asserts that T is composed of function types of the form $T_1 \to T_2$ where in each case, T_1 is equivalent to a simple type that meets the label constraint $\{l\}$.

Lemma 5.7 (Lacks equivalence for $\mathrm{E}^{\mathrm{vcr}}$)**.** *Given any T, T' and l, if T::l and $T \equiv T'$, then T'::l.*

Proof. The proof is by induction on the structure of the derivation of $T \equiv T'$.

case: $(T_1 \cap T_2) \cap T_3 \equiv T_1 \cap (T_2 \cap T_3)$. By Definition 5.6, $((T_1 \cap T_2) \cap T_3)$::$l$ is derived from (1) T_1::l and (2) T_2::l and (3) T_3::l. By Definition 5.6 with (1), (2) and (3), we can also derive $(T_1 \cap (T_2 \cap T_3))$::$l$.

case: $T_1 \cap T_2 \equiv T_2 \cap T_1$. By Definition 5.6, $(T_1 \cap T_2)$::l is derived from (1) T_1::l and (2) T_2::l. By Definition 5.6 with (1) and (2), we can also derive $(T_2 \cap T_1)$::l.

case: $T_1 \cap \omega \equiv T_1$. By Definition 5.6, $(T_1 \cap \omega)$::l is derived from T_1::l, which is the desired result.

case: $e\ \omega \equiv \omega$. By Definition 5.6, ω::l.

case: $e\ (T_1 \capdot T_2) \equiv e\ T_1 \capdot e\ T_2$. By Definition 5.6, $(e\ (T_1 \capdot T_2))::l$ is derived from $(T_1 \capdot T_2)::l$ which in turn is derived from (1) $T_1::l$ and (2) $T_2::l$. By definition 5.6 with (1) and (2), we can derive (3) $(e\ T_1)::l$ and (4) $(e\ T_2)::l$, and from (3) and (4) we can derive $(e\ T_1 \capdot e\ T_2)::l$.

case: $T_1 \to T_2 \equiv T_1' \to T_2'$, if (1) $T_1 \equiv T_1'$ and $T_2 \equiv T_2'$. By Definition 5.6, $(T_1 \to T_2)::l$ is derived from (2) $T_1 \equiv \bar{T}_1$ and (3) $\bar{T}_1 \upharpoonright l$. By (1) and (2), we have (4) $T_1' \equiv \bar{T}_1$. By Definition 5.6 with (4) and (3) we can derive $(T_1' \to T_2')::l$.

case: $T_1 \capdot T_2 \equiv T_1' \capdot T_2'$, if $T_1 \equiv T_1'$ and $T_2 \equiv T_2'$. By Definition 5.6, $(T_1 \capdot T_2)::l$ is derived from (1) $T_1::l$ and (2) $T_2::l$. By the induction hypothesis with (1) we have (3) $T_1'::l$ and by the induction hypothesis with (2) we have (4) $T_2'::l$. Now by Definition 5.6 with (3) and (4), we can derive $(T_1' \capdot T_2')::l$.

case: $e\ T_1 \equiv e\ T_1'$, if $T_1 \equiv T_1'$. By Definition 5.6, $(e\ T_1)::l$ is derived from (1) $T_1::l$. By the induction hypothesis with (1), we have (2) $T_1'::l$. By Definition 5.6 with (2) we can derive $(e\ T_1')::l$.

case: $T \equiv T$. Done.

case: $T \equiv T'$ if $T' \equiv T$. For the symmetric rules that mean the same when reflected from left to right, these cases have already been proved. For the non-symmetric rules, we have the following cases:

- **case:** $T_1 \capdot (T_2 \capdot T_3) \equiv (T_1 \capdot T_2) \capdot T_3$. Similar to the reverse case.
- **case:** $T \equiv T \capdot \omega$. By Definition 5.6, (1) $\omega::l$. By Definition 5.6 with the assumption $T::l$ and (1), we have $(T \capdot \omega)::l$.
- **case:** $\omega \equiv e\ \omega$. By Definition 5.6, (1) $\omega::l$. By Definition 5.6 with (1), we have $(e\ \omega)::l$.
- **case:** $e\ T_1 \capdot e\ T_2 \equiv e\ (T_1 \capdot T_2)$. By Definition 5.6, $(e\ T_1 \capdot e\ T_2)::l$ is derived from (1) $(e\ T_1)::l$ and (2) $(e\ T_2)::l$ which are in turn derived from (3) $T_1::l$ and (4) $T_2::l$. By Definition 5.6, from (3) and (4) we can derive the judgement (5) $(T_1 \capdot T_2)::l$ and from (5) we can derive the judgement $(e\ (T_1 \capdot T_2))::l$.

case: $T \equiv T''$, if $T \equiv T'$ and $T' \equiv T''$. This case follows from the transitivity of $\Longrightarrow$ in the statement of the lemma.

$\square$

The lacks judgement $T::l$ is used when typing extensions to ensure that the type of the record being extended does not already contain a function type of the form $l \to T'$. From the rules of Definition 5.6

(Lacks judgement), it follows that $(l_1 \to T_1 \Cap l_2 \to T_2)::l_3$ since the function domain lacks l_3. Also, we have $(\alpha[l_3] \to T_1 \Cap l_2 \to T_2)::l_3$ since the constraint on $\alpha[l_3]$ ensures that the domain cannot be made to include l_3 even after a substitution. We also permit $((l_1 \Cap \omega) \to T_1)::l_3$ where the parameter type is *equivalent* to a label, so that Lemma 3.27 (Equivalent types) holds. By comparison, the judgement $(l_1 \to T_1 \Cap l_2 \to T_2)::l_1$ cannot be derived since the function domain does contain the label l_1.

Amendment to Definition 3.11 (Expansion application for E$^{\text{vcr}}$)**.** The rules for applying expansions to expansions, types and E-variables are defined as follows:

$\omega\ K$	$= \omega$	$\sigma\ \omega$	$= \omega$
$(E \Cap F)\ K$	$= E\ K \Cap F\ K$	$\sigma\ (K_1 \Cap K_2)$	$= \sigma\ K_1 \Cap \sigma\ K_2$
$(e\ E)\ K$	$= e\ (E\ K)$	$\sigma\ (e\ K)$	$= (\sigma\ e)\ K$
$\boxdot\ e$	$= e\ \boxdot$	$\sigma\ (S \to T)$	$= \sigma\ S \to \sigma\ T$
$\underline{\boxdot\ \alpha[L]}$	$= \alpha[L]$	$\sigma\ \boxdot$	$= \sigma$
$(\sigma, X := K)\ X$	$= K$	$\sigma\ (\sigma', X := K)$	$= \sigma\sigma', X := \sigma\ K$
$(\sigma, X := K)\ X'$	$= \sigma\ X'$ if $X \neq X'$	$\sigma\ C$	$= C$

□

Again, it should be noted that new case for constrained simple type variables subsumes the original case for simple type variables. That is, if the constraint L is empty, the case becomes $\boxdot\ \alpha = \alpha$ which is the same as the original case.

The following lemma asserts that lacks judgements are stable under expansion.

Lemma 5.8 (Lacks expansion for E$^{\text{vcr}}$)**.** *Given any T, l and E, if $T::l$, then $(E\ T)::l$.*

Proof. The proof is by induction on the structure of E and T.

case: $E = \omega$.

(1)	$\omega::l$	def 5.6
$\therefore$	$(\omega\ T)::l$	def 3.11 with (1)

case: $E = E_1 \Cap E_2$.

(1)	$(E_1\ T)::l$	ind.hyp. (E_1 smaller)
(2)	$(E_2\ T)::l$	ind.hyp. (E_2 smaller)
(3)	$(E_1\ T \Cap E_2\ T)::l$	def 5.6 with (1),(2)
$\therefore$	$((E_1 \Cap E_2)\ T)::l$	def 3.11 with (3)

case: $E = e\ E_1$.

(1)	$(E_1\ T)::l$	ind.hyp. (E_1 smaller)
(2)	$(e\ (E_1\ T))::l$	def 5.6 with (1)
$\therefore$	$((e\ E_1)\ T)::l$	def 3.11 with (2)

case: $E = \sigma$. We now consider the cases for T.

case: $T = \omega$.

(1)	$\omega::l$	def 5.6
$\therefore$	$(\sigma\ \omega)::l$	def 3.11 with (1)

case: $T = T_1 \Cap T_2$.

(1)	$(T_1 \Cap T_2)::l$	assumption
(2)	$T_1::l$	def 5.6 with (1)
(3)	$T_2::l$	”
(4)	$(\sigma T_1)::l$	ind.hyp. with (2)
(5)	$(\sigma T_2)::l$	ind.hyp. with (3)
(6)	$(\sigma T_1 \Cap \sigma T_2)::l$	def 5.6 with (4),(5)
$\therefore$	$(\sigma\ (T_1 \Cap T_2))::l$	def 3.11 with (6)

case: $T = e\ T_1$. We now consider the three cases for σ.

case: $\sigma = \boxdot$.

(1)	$(e\ T_1)::l$	assumption
$\therefore$	$(\boxdot\ (e\ T_1))::l$	lem 3.13 with (1)

case: $\sigma = (\sigma', e := E_1)$.

(1)	$(E_1\ T_1)::l$	ind.hyp. (E_1 and T_1 smaller)
(2)	$(((\sigma', e := E_1)\ e)\ T_1)::l$	def 3.11
$\therefore$	$((\sigma', e := E_1)\ (e\ T_1))::l$	def 3.11

case: $\sigma = (\sigma', X := K)$ where $X \neq e$.

(1)	$(\sigma' \ (e \ T_1))::l$	ind.hyp. (σ' smaller)
(2)	$((\sigma' \ e) \ T_1)::l$	def 3.11
(3)	$(((\sigma', X := K) \ e) \ T_1)::l$	def 3.11
$\therefore$	$((\sigma', X := K) \ (e \ T_1))::l$	def 3.11

case: $T = \alpha[L']$. This case cannot occur by Definition 5.6.

case: $T = C$. This case cannot occur by Definition 5.6.

case: $T = T_1 \to T_2$.

(1)	$(T_1 \to T_2)::l$	assumption
(2)	$\exists \bar{T_1}. \quad T_1 \equiv \bar{T_1}$	def 5.6 with (1)
(3)	$\bar{T_1} \upharpoonright l$	"
	let $\bar{S} = \sigma \ \bar{T_1}$	
(4)	$\bar{S} \upharpoonright l$	lem 5.5 with (3)
(5)	$\sigma \ T_1 \equiv \bar{S}$	lem 3.16 with (2)
(6)	$(\sigma \ T_1 \to \sigma \ T_2)::l$	def 5.6 with (5),(4)
$\therefore$	$(\sigma \ (T_1 \to T_2))::l$	def 3.11 with (6)

□

5.2.3 Typing Derivations

This section extends typeof with the raw types for labels and the empty record, and defines the new rules used to derive typings for extensions.

Amendment to Definition 4.1 (typeof for E$^{\text{vcr}}$)**.** typeof is a total function from constants to type constants and function types, defined as:

$$\underline{\mathsf{typeof}(c)} = l \qquad \text{iff } c = l$$

$$\underline{\mathsf{typeof}(\{\})} = \{\}$$

For any constant c, typeof(c) is called the *raw type* of c. □

Amendment to Definition 3.24 (Typing derivations for E$^{\text{vcr}}$)**.** A *typing derivation* (metavariable D) is a proof tree of a typing judgement. The rules for deriving valid typing judgements are given below.

$$\text{(var)}\ \dfrac{}{x : \bar{T} \lhd x : \langle \bar{T} \rangle} \qquad\qquad \text{(omega)}\ \dfrac{}{v : \omega \lhd \Gamma_\omega}$$

$$\text{(con)}\ \dfrac{}{c : \bar{T} \lhd \Gamma_\omega}\ \bar{T} \equiv \sigma\ \mathsf{typeof}(c) \qquad\qquad \text{(int)}\ \dfrac{v : \tau_1 \quad v : \tau_2}{v : \tau_1 \Cap \tau_2}$$

$$\text{(abs)}\ \dfrac{t : T \lhd \Gamma, x : \langle S \rangle}{\lambda x.t : S \to T \lhd \Gamma} \qquad\qquad \text{(evar)}\ \dfrac{v : \tau}{v : e\ \tau}$$

$$\underline{\text{(extl)}}\ \dfrac{t : T \lhd \Gamma}{l \to t \Cap v : S \to T \lhd \Gamma}\ S \equiv l \qquad\qquad \underline{\text{(extr)}}\ \dfrac{v : \bar{T} \lhd \Gamma}{l \to t \Cap v : \bar{T} \lhd \Gamma}\ \bar{T}::l$$

$$\text{(app)}\ \dfrac{t : S \to T \lhd \Gamma_1 \quad s : S \lhd \Gamma_2}{t\ s : T \lhd \Gamma_1 \Cap \Gamma_2}$$

□

The new rules (extl) and (extr) assign typings individually to the left and right branches of an extension, which provides the necessary flexibility to apply expansions at any nested branch. These individual typings may be combined by Rule (int) to produce a type that mentions all of an extensible record's fields. The side condition $\bar{T}::l$ in (extr) ensures that the new label l being introduced in the extension is not already present in the extension type. The following example illustrates a typing derivation for the record `john` which is defined as `name` $\to$ `"John Smith"` $\Cap$ `age` $\to 31 \Cap \{\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{"John Smith"} : \texttt{Str} \lhd \Gamma_\omega}\text{(con)}}{\texttt{"John Smith"} : e_1\ \texttt{Str} \lhd \Gamma_\omega}\text{(evar)}}{\texttt{john} : \texttt{name} \to e_1\ \texttt{Str} \lhd \Gamma_\omega}\text{(extl)}}{\texttt{john} : e\ (\texttt{name} \to e_1\ \texttt{Str}) \lhd \Gamma_\omega}\text{(evar)} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{31 : \texttt{Int} \lhd \Gamma_\omega}\text{(con)}}{31 : f_1\ \texttt{Int} \lhd \Gamma_\omega}\text{(evar)}}{\texttt{age} \to 31 \Cap \{\} : \texttt{age} \to f_1\ \texttt{Int} \lhd \Gamma_\omega}\text{(extl)}}{\texttt{john} : \texttt{age} \to f_1\ \texttt{Int} \lhd \Gamma_\omega}\text{(extr)}}{\texttt{john} : f\ (\texttt{age} \to f_1\ \texttt{Int}) \lhd \Gamma_\omega}\text{(evar)}}{\texttt{john} : e\ (\texttt{name} \to e_1\ \texttt{Str}) \Cap f\ (\texttt{age} \to f_1\ \texttt{Int}) \lhd \Gamma_\omega}\text{(int)}$$

Note that Definition 4.1 defines `typeof` with the restriction that label l is the only constant that can have raw type l. This is necessary for progress to hold since otherwise non-label constants would be accepted by the type system as valid arguments to extensions. For example, if it were possible to define a constant `&` such that $\mathsf{typeof}(\texttt{\&}) = l$ for some label l, then the application $(l \rightarrow 3 \cap \{\})$ `&` would be typable despite being irreducible.

The following lemma asserts that label l is the only value that can have type l in an empty term context:

Lemma 5.9 (Label identity for E$^{\text{vcr}}$). *Given any v, T and l, if $v : T \lhd \Gamma_\omega$ and $T \equiv l$, then $v = l$.*

Proof. Given $v : T \lhd \Gamma_\omega$ and $T \equiv l$ we have (1) $v : l \lhd \Gamma_\omega$ by Lemma 3.27. By Lemma 3.25 (1) was derived by either (var) or (con). (1) could not have been derived by (var) because that would require a non-empty term context. Therefore (1) was derived by (con) as follows:

$$\frac{}{v : l \lhd \Gamma_\omega}\ (2)\ l \equiv \sigma\ \mathsf{typeof}(v)$$

Since $\mathsf{typeof}(v)$ is a simple type by Definition 4.1, then by Lemma 3.12, (3) $\sigma\ \mathsf{typeof}(v)$ is also a simple type. By Corollary 3.10 with (2) and (3), we have (4) $l = \sigma\ \mathsf{typeof}(v)$. By Definition 4.1, $\mathsf{typeof}(v)$ is either some type constant C or some function type $T_1 \rightarrow T_2$, and so by Definition 3.11, one of the following is true:

(A) $\sigma\ \mathsf{typeof}(v) = \sigma\ C = C$

(B) $\sigma\ \mathsf{typeof}(v) = \sigma\ (T_1 \rightarrow T_2) = \sigma T_1 \rightarrow \sigma T_2$

Due to (4), we know that (A) must be true, and so $C = l$, which also means that $\mathsf{typeof}(v) = l$. By Definition 4.1, $\mathsf{typeof}(v) = l$ iff $v = l$, and so $v = l$.

□

The next lemma asserts that if an extension at type $T_1 \rightarrow T_2$ is applicable to a value v at type T_1 in an empty term context, then T_1 is equivalent to a label and v is the same label:

Lemma 5.10 (Extension parameter type for E$^{\text{vcr}}$). *Given any l, s, v, v_1, Γ, T_1 and T_2, if $l \rightarrow s \cap v : T_1 \rightarrow T_2 \lhd \Gamma$ and $v_1 : T_1 \lhd \Gamma_\omega$, then there exists a label l' such that $v_1 = l'$ and $T_1 \equiv l'$.*

Proof. Let D be the derivation of $l \rightarrow s \cap v : T_1 \rightarrow T_2 \lhd \Gamma$. The proof is by case analysis of D. There are only two possible typing rules that assign typings to extensions.

case: D ends with the following application of Rule (extl)

$$\frac{(2)\ s : T_2 \lhd \Gamma}{(1)\ l \to s \Cap v : T_1 \to T_2 \lhd \Gamma}\ (3)\ T_1 \equiv l$$

By Lemma 5.9 with (3) and the assumption $v_1 : T_1 \lhd \Gamma_\omega$ we have (4) $v = l$. The result is (4) and (3).

case: D ends with the following application of Rule (extr)

$$\frac{(2)\ v : T_1 \to T_2 \lhd \Gamma}{(1)\ l \to s \Cap v : T_1 \to T_2 \lhd \Gamma}\ (3)\ (T_1 \to T_2)::l$$

By Definition 5.6 with (3), we have (4) $T_1 \equiv \bar{S}$ and (5) $\bar{S}\restriction l$. By Definition 5.1 with (5) we have the following two cases:

case: $\bar{S} = l''$ and $l'' \neq l$. Then by Lemma 5.9 with (4) and the assumption $v_1 : T_1 \lhd \Gamma_\omega$ we have (6) $v = l''$. The result is (6) and (4).

case: $\bar{S} = \alpha[L]$ and $l \in L$. This case cannot occur which we will prove by contradiction. By Lemma 3.27 with (4) and the assumption $v_1 : T_1 \lhd \Gamma_\omega$, we have (6) $v_1 : \bar{S} \lhd \Gamma_\omega$. If we suppose that $\bar{S} = \alpha[L]$, then (6) becomes $v_1 : \alpha[L] \lhd \Gamma_\omega$ however there is no typing rule that can be used to assign a constrained simple type variable $\alpha[L]$ to a value in an empty context. Therefore $\bar{S} \neq \alpha[L]$.

$\square$

Finally, we extend the proofs of lemmas and theorems that are affected by the introduction of extensions and constrained simple type variables.

Restatement of Lemma 3.25 (Typing derivations for E$^{\mathrm{vcr}}$)**.** *Each typing judgement can be the conclusion of at most one typing rule.*

Proof. Consider any application of a typing rule that concludes with the typing judgement J denoting $t : T \lhd \Gamma$. The cases are the same as before (p.137), with the following new cases for extensions.

- If t is a value,

– If $T = \bar{T}$

  ∗ If $t = l \to t_1 \Cap v$, then

    · If $T = T_1 \to T_2$ and $T_1 \equiv l$, then J was derived by (extl).

    · Otherwise J was derived by (extr) □.

Restatement of Lemma 3.27 (Equivalent types for E$^{\text{vcr}}$). *Given any t, T, T' and Γ, if $t : T \lhd \Gamma$ and $T \equiv T'$, then $t : T' \lhd \Gamma$.*

Proof. Let D be the derivation of $t : T \lhd \Gamma$. The proof is by induction on the structure of D. The cases are the same as before (p.137), with two new cases for Rule (extl) and Rule (extr).

case: t is v.

case: $T \equiv T'$ was derived by structural congruence rule $\dfrac{(1)\ T_1 \equiv T_1' \quad (2)\ T_2 \equiv T_2'}{T_1 \to T_2 \equiv T_1' \to T_2'}$

By Lemma 3.25, D can end with one of five possible rules; as before, Rule (var), Rule (con) and Rule (con), but now also Rule (extl) and Rule (extr) whose cases are shown below:

case: D ends with the following application of Rule (extl)

$$\frac{(4)\ t_1 : T_2 \lhd \Gamma}{(3)\ l \to t_1 \Cap v : T_1 \to T_2 \lhd \Gamma}\ (5)\ T_1 \equiv l$$

Then,

(6)	$t_1 : T_2' \lhd \Gamma$	ind.hyp. with (4) and (2)
(7)	$T_1' \equiv l$	def 3.7 with (5) and (1)
$\therefore$	$l \to t_1 \Cap v : T_1' \to T_2' \lhd \Gamma$	(extl) with (6) and (7)

case: D ends with the following application of Rule (extr)

$$\frac{(4)\ v : T_1 \to T_2 \lhd \Gamma}{(3)\ l \to t_1 \Cap v : T_1 \to T_2 \lhd \Gamma}\ (5)\ (T_1 \to T_2)::l$$

where

(6)	$T_1 \to T_2 \equiv T_1' \to T_2'$	Assumption
(7)	$v : T_1' \to T_2' \lhd \Gamma$	ind.hyp. with (4) and (6)
(8)	$\exists \bar{T}_1.\quad T_1 \equiv \bar{T}_1$	def 5.6 with (5)
(9)	$\bar{T}_1 \upharpoonright l$	"
(10)	$T_1' \equiv \bar{T}_1$	(1),(8)
(11)	$(T_1' \to T_2') :: l$	def. 5.6 with (10),(9)
$\therefore$	$l \to t_1 \Cap v : T_1' \to T_2' \lhd \Gamma$	(extr) with (7) and (11)

□

Restatement of Lemma 3.28 (Expansion for E$^{\text{vcr}}$). *Given any t, T, Γ and E, if $t : T \lhd \Gamma$ and if t is a value or E is a substitution, then $t : E\ T \lhd E\ \Gamma$.*

Proof. Let D be the derivation of $t : T \lhd \Gamma$. The cases are the same as before (p.138), with two new cases for Rule (extl) and Rule (extr).

case: E is σ.

case: D ends with the following application of Rule (extl):

$$\frac{(2)\ t_1 : T_1 \lhd \Gamma}{(1)\ l \to t_1 \Cap v : S \to T_1 \lhd \Gamma}\ (3)\ S \equiv l$$

Then,

(4)	$t_1 : \sigma T_1 \lhd \sigma\Gamma$	ind.hyp. with (2)
(5)	$\sigma S \equiv \sigma l$	lem. 3.16 with (3)
(6)	$\sigma S \equiv l$	def. 3.11 with (5)
(7)	$l \to t_1 \Cap v : \sigma S \to \sigma T_1 \lhd \sigma\Gamma$	(extl) with (4) and (6)
$\therefore$	$l \to t_1 \Cap v : \sigma\ (S \to T_1) \lhd \sigma\Gamma$	def. 3.11 with (7)

case: D ends with the following application of Rule (extr):

$$\frac{(2)\ v : \bar{T}_1 \lhd \Gamma}{(1)\ l \to t_1 \Cap v : \bar{T}_1 \lhd \Gamma}\ (3)\ \bar{T}_1 :: l$$

Then,

(4)	$v : \sigma\bar{T}_1 \lhd \sigma\Gamma$	ind.hyp. with (2)
(5)	$(\sigma\bar{T}_1){::}l$	lem. 5.8 with (3)
$\therefore$	$l \to t_1 \Cap v : \sigma\bar{T}_1 \lhd \sigma\Gamma$	(extr) with (4) and (5)

□

Restatement of Lemma 3.29 (Term substitution for $\mathrm{E}^{\mathrm{vcr}}$). *Given any v, t, x, S, T, Γ_1 and Γ_2, if $v : S \lhd \Gamma_2$ and $t : T \lhd \Gamma_1, x : \langle S\rangle$, then $t[x := v] : T \lhd \Gamma_1 \Cap \Gamma_2$.*

Proof. Let D be the derivation of $t : T \lhd \Gamma_1, x : \langle S\rangle$. The cases are the same as before (p.138), with two new cases for Rule (extl) and Rule (extr).

case: D ends with an application of Rule (extl) of the form

$$\frac{(2)\ t_1 : T_1 \lhd \Gamma_1, x : \langle S\rangle}{(1)\ l \to t_1 \Cap v_1 : S_1 \to T_1 \lhd \Gamma_1, x : \langle S\rangle}\ (3)\ S_1 \equiv l$$

where (4) $t = l \to t_1 \Cap v_1$ and (5) $T = S_1 \to T_1$.

(6)	$t_1[x := v] : T_1 \lhd \Gamma_1 \Cap \Gamma_2$	ind.hyp. with (2)
(7)	$l \to t_1[x := v] \Cap v_1[x := v] : S_1 \to T_1 \lhd \Gamma_1 \Cap \Gamma_2$	(extl) with (6) and (3).
$\therefore$	$(l \to t_1 \Cap v_1)[x := v] : S_1 \to T_1 \lhd \Gamma_1 \Cap \Gamma_2$	def. 3.3 with (7)

case: D ends with an application of Rule (extr) of the form

$$\frac{(2)\ v_1 : \bar{T}_1 \lhd \Gamma_1, x : \langle S\rangle}{(1)\ l \to t_1 \Cap v_1 : \bar{T}_1 \lhd \Gamma_1, x : \langle S\rangle}\ (3)\ \bar{T}_1{::}l$$

where (4) $t = l \to t_1 \Cap v_1$ and (5) $T = \bar{T}_1$.

(6)	$v_1[x := v] : \bar{T}_1 \lhd \Gamma_1 \Cap \Gamma_2$	ind.hyp. with (2)
(7)	$l \to t_1[x := v] \Cap v_1[x := v] : \bar{T}_1 \lhd \Gamma_1 \Cap \Gamma_2$	(extr) with (6) and (3)
$\therefore$	$(l \to t_1 \Cap v_1)[x := v] : \bar{T}_1 \lhd \Gamma_1 \Cap \Gamma_2$	def. 3.3 with (7)

□

Restatement of Theorem 3.30 (Subject reduction for $\mathrm{E}^{\mathrm{vcr}}$). *If $t : T \lhd \Gamma$ and $t > t'$, then $t' : T \lhd \Gamma$.*

Proof. The cases are the same as before (p.139), with two new cases for the reduction rules of extension applications.

case: $(l \to t' \Cap v)\ l > t'$.

The derivation of $(l \to t' \Cap v)\ l : T \lhd \Gamma$ ends with

$$\frac{(2)\ l \to t' \Cap v : S \to T \lhd \Gamma_1 \quad (3)\ l : S \lhd \Gamma_2}{(1)\ (l \to t' \Cap v)\ l : T \lhd \Gamma_1 \Cap \Gamma_2}$$

where $\Gamma = \Gamma_1 \Cap \Gamma_2$.

By Lemma 5.10 with (2) and (3), we have (4) $S \equiv l$. Following the proof of Lemma 3.25 in the case of (4), the derivation of $l \to t' \Cap v : S \to T \lhd \Gamma_1$ ends with the following application of Rule (extl):

$$\frac{(5)\ t' : T \lhd \Gamma_1}{l \to t' \Cap v : S \to T \lhd \Gamma_1}\ S \equiv l$$

By Lemma 4.4 with (3), we have (6) $\Gamma_2 = \Gamma_\omega$. Finally, by (6) and (5), we have $t' : T \lhd \Gamma_1 \Cap \Gamma_2$.

case: $(l \to t_1 \Cap v)\ l_1 > v\ l_1$ if (1) $l_1 \not\equiv l$.

The derivation of $(l \to t_1 \Cap v)\ l_1 : T \lhd \Gamma$ ends with

$$\frac{(2)\ l \to t_1 \Cap v : S \to T \lhd \Gamma_1 \quad (3)\ l_1 : S \lhd \Gamma_2}{(l \to t_1 \Cap v)\ l_1 : T \lhd \Gamma_1 \Cap \Gamma_2}$$

where $\Gamma = \Gamma_1 \Cap \Gamma_2$.

By Lemma 5.10 with (2) and (3), we have $S \equiv l_1$ and hence (4) $S \not\equiv l$ due to (1). Following the proof of Lemma 3.25 in the case of (4), the derivation of $l \to t' \Cap v : S \to T \lhd \Gamma_1$ ends with the following application of Rule (extr):

$$\frac{(5)\ v : S \to T \lhd \Gamma_1}{l \to t_1 \Cap v : S \to T \lhd \Gamma_1}\ (S \to T)::l$$

By Rule (app) with (5) and (3) we have $v\ l_1 : T \lhd \Gamma_1 \Cap \Gamma_2$.

□

Restatement of Theorem 3.31 (Progress for E$^{\text{vcr}}$). *If $t\ s : T \lhd \Gamma_\omega$, then there exists a term t' such that $t\ s > t'$.*

Proof. The proof is by induction on the structure of $t\ s$.

The derivation of $t\ s : T \lhd \Gamma_\omega$ ends with

$$\frac{(1)\ t : S \to T \lhd \Gamma_\omega \quad s : S \lhd \Gamma_\omega}{t\ s : T \lhd \Gamma_\omega}$$

The cases are the same as before (p.139), but with one new case for extensions.

case: t is v_1.

case: s is v_2.

case: v_1 is $l \to t_1 \Cap v$.

case: $v_2 = l$. Then $(l \to t_1 \Cap v)\ l > t_1$.

case: $v_2 \neq l$.

(1)	$(l \to t_1 \Cap v)\ v_2 : T \lhd \Gamma_\omega$	assumption
(2)	$\exists S.\quad l \to t_1 \Cap v : S \to T \lhd \Gamma_\omega$	rule (app) with (1)
(3)	$v_2 : S \lhd \Gamma_\omega$	”
(4)	$\exists l'.v_2 = l'$	lem 5.10 with (2),(3)
$\therefore$	$(l \to t_1 \Cap v)\ v_2 > v\ v_2$	(4) and $v_2 \neq l$

□

5.3 Type Inference

This section extends our type inference algorithm $\mathcal{I}$ and unification algorithm opusβ to support extensible records, and re-establishes all theorems and lemmas from the previous chapter.

5.3.1 Algorithm opusβ

This section extends the unification algorithm opusβ to support constrained simple type variables.

Variable Structures and Renamings

The definitions of variable structures and renamings are updated by replacing each mention of a simple type variable by a constrained simple type variable. The logic behind these rules does not change, however, and the new definitions subsume the old definitions.

Amendment to Definition 3.43 (varstruct for $\mathrm{E}^{\mathrm{vcr}}$)**.** The total function varstruct takes any type, unification constraint or unification constraint set and returns a variable structure. It is defined by the following rules:

$$\begin{aligned}
&\underline{\mathsf{varstruct}(\alpha[L])} && = \alpha[L] \mapsto \emptyset \\
&\mathsf{varstruct}(C) && = \emptyset \\
&\mathsf{varstruct}(\omega) && = \emptyset \\
&\mathsf{varstruct}(S \to T) = \mathsf{varstruct}(S \Cap T) && = \mathsf{varstruct}(S) \sqcup \mathsf{varstruct}(T) \\
&\mathsf{varstruct}(e\ T) && = e \mapsto \mathsf{varstruct}(T) \\
&\mathsf{varstruct}(S \leq T) && = \mathsf{varstruct}(S) \sqcup \mathsf{varstruct}(T) \\
&\mathsf{varstruct}(\Delta) && = \bigsqcup \{\mathsf{varstruct}(\delta) \mid \delta \in \Delta\}
\end{aligned}$$

where $\mathcal{V}_1 \sqcup \mathcal{V}_2 = \{e \mapsto (\mathcal{V}_1/e) \sqcup (\mathcal{V}_2/e) \mid e \in \mathsf{Dom}(\mathcal{V}_1) \cup \mathsf{Dom}(\mathcal{V}_2)\}$ $\square$

Amendment to Definition 3.45 (Fresh renamings for $\mathrm{E}^{\mathrm{vcr}}$)**.** A *fresh renaming* for some finite set of variables X^s is a substitution $(\boxdot, X_1 := K_1, \ldots X_n := K_n)$ where $\{X_1 := K_1, \ldots X_n := K_n\} = \{e := e' \ \boxdot \mid e \in X^s, e' \ \mathsf{fresh}\} \cup \{\underline{\alpha[L]} := \underline{\alpha'[L]} \mid \underline{\alpha[L]} \in X^s, \alpha' \ \mathsf{fresh}\}$. $\square$

Unification with Constrained Simple Type Variables

We isolate the rules for unification with constrained simple type variables into a separate function called hunify.

Definition 5.11 (hunify for $\mathrm{E}^{\mathrm{vcr}}$)**.** The hunify function unifies a constrained simple type variable with a simple type, and is defined by the rules below.

$$\begin{array}{rll}
\mathsf{hunify}(\alpha[L_1] \doteq \beta[L_2]) = (\boxdot, \alpha[L_1] := \gamma[L_1 \cup L_2],\ \beta[L_2] := \gamma[L_1 \cup L_2]) & \gamma\ \mathsf{fresh} & \text{(TT-unify)} \\
\mathsf{hunify}(\alpha[L] \doteq C) = (\boxdot, \alpha[L] := C) & \text{if } C \restriction L & \text{(TC-unify)} \\
\mathsf{hunify}(\alpha \doteq T_1 \to T_2) = (\boxdot, \alpha := T_1 \to T_2) & \text{if } \alpha \notin \mathsf{ovars}(T_1 \to T_2) & \text{(TA-unify)}
\end{array}$$

□

Lemma 5.12 (hunify correctness for E$^{\text{vcr}}$). *Let $\delta = (\alpha[L] \doteq \bar{T})$. If* $\mathsf{hunify}(\delta) = \sigma$ *is defined, then* $\sigma\alpha[L] \equiv \sigma\bar{T}$.

Proof. The proof is by case analysis of the rules for $\mathsf{hunify}(\delta)$.

case: $\sigma = (\boxdot, \alpha[L] := \gamma[L \cup L_2], \beta[L_2] := \gamma[L \cup L_2])$ by Rule (TT-unify)where γ is fresh and $\bar{T} = \beta[L_2]$. Then $\sigma\ \alpha[L] = \gamma[L \cup L_2] = \sigma\ \beta[L_2]$ by Definition 3.11.

case: $\sigma = (\boxdot, \alpha[L] := C)$ by Rule (TC-unify)where $\bar{T} = C$ and $C{\restriction}L$. Then $\sigma\alpha[L] = C = \sigma C$ by Definition 3.11.

case: $\sigma = (\boxdot, \alpha[L] := T_1 \to T_2)$ by Rule (TA-unify)where $\bar{T} = T_1 \to T_2$ and $L = \emptyset$ and $\alpha[L] \notin \mathsf{ovars}(T_1 \to T_2)$. Then $\sigma\alpha[L] = T_1 \to T_2$ by Definition 3.11 and $T_1 \to T_2 = \sigma(T_1 \to T_2)$ since $\alpha[L] \notin \mathsf{ovars}(T_1 \to T_2)$.

□

The function hunify encapsulates all that is needed from a unification algorithm to support extensible records, and the purpose of extracting these rules into a separate function is to make the job simpler for other unification algorithms to borrow this functionality (such as the opus algorithm given in Appendix C). While the opusβ unification algorithm does not in general aim to find covering unifier sets, the hunify fragment of the algorithm *does* find covering unifier sets, which are always of size one (i.e. single, principal unifiers).

Lemma 5.13 (hunify is complete and principal for E$^{\text{vcr}}$). *Let $\delta = (\alpha[L] \doteq \bar{T})$. If there exists a unifier σ for δ, then*

1. $\mathsf{hunify}(\delta) = \sigma_1$ *is defined, and*
2. *for any set of variables X^s that excludes the* fresh *variables introduced by* $\mathsf{hunify}(\delta)$*, there exists a substitution σ_2 such that $\forall X \in X^s.\ \sigma X \equiv \sigma_2\sigma_1 X$.*

Proof. The proof is by case analysis of $\bar{T}$.

case: $\bar{T}$ is $\beta[L_2]$.

1. $\sigma_1 = (\boxdot, \alpha[L] := \gamma[L \cup L_2], \beta[L_2] := \gamma[L \cup L_2])$ by (TT-unify).

2. Let $\sigma' = (\sigma, \gamma[L \cup L_2] := \sigma\alpha[L])$. Then,

$$\begin{aligned}
(1)\ \sigma'\sigma_1 &= \sigma'\ (\boxdot, \alpha[L] := \gamma[L \cup L_2], \beta[L_2] := \gamma[L \cup L_2]) \\
&= \sigma', \alpha[L] := \sigma'\gamma[L \cup L_2], \beta[L_2] := \sigma'\gamma[L \cup L_2] \\
&= \sigma', \alpha[L] := \sigma\alpha[L], \beta[L_2] := \sigma\alpha[L] \\
&= \sigma, \gamma[L \cup L_2] := \sigma\alpha[L], \alpha[L] := \sigma\alpha[L], \beta[L_2] := \sigma\alpha[L]
\end{aligned}$$

We next test if σ' is a candidate for σ_2 by checking that $\forall X \in X^s.\ \sigma'\sigma_1 X \equiv \sigma X$. Because γ is fresh, $\gamma[L \cup L_2]$ is not in X^s and so there are only the following 3 possibilities for each $X \in X^s$:

case: $X = \alpha[L]$. Then $\sigma'\sigma_1\alpha[L]$ equals $\sigma\alpha L$ by (1).

case: $X = \beta[L_2]$. Then $\sigma'\sigma_1\beta[L_2]$ equals $\sigma\alpha L$ by (1) which is equivalent to $\sigma\beta[L_2]$ by the assumption that σ is a unifier.

case: $X \notin \{\alpha[L], \beta[L_2], \gamma[L \cup L_2]\}$. Then $\sigma'\sigma_1 X = \sigma X$ by Definition 3.11 with (1).

Therefore, $\forall X \in X^s.\ \sigma'\sigma_1 X \equiv \sigma X$, and so σ_2 is σ'.

case: $\bar{T}$ is C.

1. We consider the cases for L.

case: L is empty. Then $\sigma_1 = (\alpha[L] := C)$ by (TC-unify).

case: L is not empty.

(1)	$\sigma\ \alpha[L] \equiv \sigma\ C$	assumption
(2)	$\sigma\ \alpha[L] \equiv C$	def 3.11 with (1)
(3)	$\sigma\ \alpha[L] = C$	cor 3.10 with (2)
(4)	$\alpha[L]\restriction L$	def 5.1
(5)	$(\sigma\ \alpha[L])\restriction L$	lem 5.5 with (4)
(6)	$C\restriction L$	(5),(3)
(7)	$\sigma_1 = (\alpha[L] := C)$	(TC-unify) with (6)

In both cases, $\sigma_1 = (\alpha[L] := C)$.

2. By Definition 3.11, (8) $\sigma\sigma_1 = (\sigma, \alpha[L] := C)$. Next, we consider σ as a candidate for σ_2. There are two possibilities for each $X \in X^s$:

case: $X = \alpha[L]$. Then by (8) and (3), it follows that $\sigma\sigma_1\alpha[L] = \sigma\alpha[L]$.

case: $X \neq \alpha[L]$. Then by Definition 3.11 with (8) and the assumption that $X \neq \alpha[L]$, it follows that $\sigma\sigma_1 X = \sigma X$.

Therefore, $\forall X \in X^s.\ \sigma\sigma_1 X \equiv \sigma X$, and so σ_2 is σ.

case: $\bar{T}$ is $T_1 \to T_2$.

1.

(1)	$\sigma\ \alpha[L] \equiv \sigma\ (T_1 \to T_2)$	assumption
(2)	$\sigma\ \alpha[L] \equiv \sigma\ T_1 \to \sigma\ T_2$	def 3.11 with (1)
(3)	$\sigma\ \alpha[L] = \sigma\ T_1 \to \sigma\ T_2$	cor 3.10 with (2)
(4)	$\alpha[L]\restriction L$	def 5.1
(5)	$(\sigma\ \alpha[L])\restriction L$	lem 5.5 with (4)
(6)	$\sigma\ \alpha[L]$ is a function type	(3)
(7)	$L = \emptyset$	def 5.1 with (5),(6)
(8)	$\alpha[L] \notin \mathsf{ovars}(T_1 \to T_2)$	(1)
(9)	$\sigma_1 = (\alpha[L] := T_1 \to T_2)$	(TA-unify) with (7),(8)

2. By Definition 3.11, (10) $\sigma\sigma_1 = (\sigma, \alpha[L] := \sigma\ (T_1 \to T_2))$. Next, we consider σ as a candidate for σ_2. There are two possibilities for each $X \in X^s$:

case: $X = \alpha[L]$. Then $\sigma\sigma_1\ \alpha[L]$ equals $\sigma\ (T_1 \to T_2)$ by (10) which is equivalent to $\sigma\ \alpha[L]$ by (1).

case: $X \neq \alpha[L]$. Then $\sigma\sigma_1\ X = \sigma\ X$ by Definition 3.11 and (10).

Therefore, $\forall X \in X^s.\ \sigma\sigma_1 X \equiv \sigma X$, and so σ_2 is σ.

□

Based on the definition of hunify, two unification rules for the $\overrightarrow{\mathsf{opus}_\beta}$ relation are changed to account for constrained simple type variables.

Amendment to Definition 3.46 ($\overrightarrow{\mathsf{opus}_\beta}$ relation for $\mathrm{E}^{\mathrm{vcr}}$)**.** The $\overrightarrow{\mathsf{opus}_\beta}$ relation performs one step of unification on a factored constraint and is defined by the following rules:

$$
\begin{array}{llll r}
\underline{(\alpha[L] \doteq \bar{T}, \mathcal{V})} & \overrightarrow{\mathsf{opus}_\beta} & \mathsf{hunify}(\alpha[L] \doteq \bar{T}) & & \text{(T-unify)} \\
(e\ T \mathrel{\underline{\lessdot}} \bar{T}, \mathcal{V}) & \overrightarrow{\mathsf{opus}_\beta} & (e := \mathsf{ren}(\mathcal{V}, e)) & & \text{(EA-unify)} \\
(e\ S \mathrel{\underline{\lessdot}} T \mathbin{\dot{\cap}} U, \mathcal{V}) & \overrightarrow{\mathsf{opus}_\beta} & (e := e_1\ \boxdot \mathbin{\dot{\cap}} e_2\ \boxdot) & & \text{(EI-unify)} \\
(e\ S \mathrel{\underline{\lessdot}} e\ T, \mathcal{V}) & \overrightarrow{\mathsf{opus}_\beta} & (e := e\ \sigma) & \text{if } (S \mathrel{\underline{\lessdot}} T, \mathcal{V}/e)\ \overrightarrow{\mathsf{opus}_\beta}\ \sigma & \text{(E-unify)} \\
 & & (e := e\ \omega) & \text{otherwise} & \\
\underline{(e\ S \mathrel{\underline{\lessdot}} f\ T, \mathcal{V})} & \overrightarrow{\mathsf{opus}_\beta} & (e := f\ g\ \boxdot) & \text{if } S \text{ is } \alpha[L] \text{ or } T \text{ is } \mathbb{T},\ g\ \mathsf{fresh} & \text{(EE-unify)} \\
 & & (f := e\ g\ \boxdot) & \text{otherwise, } g\ \mathsf{fresh} &
\end{array}
$$

where $\mathsf{ren}(\mathcal{V}, e)$ is a fresh renaming of the variables in $\mathsf{Dom}(\mathcal{V}/e)$. □

Note that the logic of Rule (EE-unify) has not changed, and the simple type variables have merely been replaced by constrained simple type variables. The real change is to Rule (T-unify) which has been updated to unify a constrained simple type variable with a simple type by invoking hunify.

Restatement of Theorem 3.50 (Correctness of $\mathsf{opus}\beta$ for E$^{\mathrm{vcr}}$)**.** *Given any* Δ *and* σ*, then* $\sigma \in \mathsf{opus}\beta(\Delta) \implies \mathsf{solved}(\sigma\Delta)$.

Proof. The proof remains unchanged from the one for System E$^{\mathrm{vc}}$ (p.141). □

5.3.2 Algorithm $\mathcal{I}$

Algorithm $\mathcal{I}$ requires the addition of one new rule to support extensible records.

Amendment to Definition 3.57 (Algorithm $\mathcal{I}$ for E$^{\mathrm{vcr}}$)**.** The type inference algorithm $\mathcal{I}$ takes a unification algorithm and a term and returns a distinct sequence of typings. It is defined by the following rules:

Terms

$$\begin{array}{lll}
\mathcal{I}(\mathcal{U}, v) = & \{e\ \mathcal{I}_v(\mathcal{U}, v)\} \qquad e \textsf{ fresh} & (\mathcal{I}.\text{val}) \\
\mathcal{I}(\mathcal{U}, t\ s) = & \{\sigma\ (e\ \alpha \lhd \Gamma_1 \Cap \Gamma_2) & (\mathcal{I}.\text{app}) \\
 & \mid (T \lhd \Gamma_1) \in \mathcal{I}(\mathcal{U}, t), (S \lhd \Gamma_2) \in \mathcal{I}(\mathcal{U}, s) & \\
 & \sigma \in \mathcal{U}(\{T \leq S \to e\ \alpha\} \cup \{U \leq U \mid x : U \in \Gamma_1 \Cap \Gamma_2\}), e, \alpha \textsf{ fresh}\} &
\end{array}$$

Values

$$\begin{array}{lll}
\mathcal{I}_v(\mathcal{U}, x) = & \alpha \lhd x : \langle\alpha\rangle \qquad \alpha \textsf{ fresh} & (\mathcal{I}.\text{var}) \\
\mathcal{I}_v(\mathcal{U}, c) = & \textsf{typeof}(c) \lhd \Gamma_\omega & (\mathcal{I}.\text{con}) \\
\mathcal{I}_v(\mathcal{U}, \lambda x.t) = & \textsf{isect}(\{(S {\to} T \lhd \Gamma) \mid (T \lhd \Gamma, x : \langle S\rangle) \in \mathcal{I}(\mathcal{U}, t)\}) & (\mathcal{I}.\text{abs}) \\
\underline{\mathcal{I}_v(\mathcal{U}, l \to t \Cap v)} = & \textsf{let } \tau_1^s = \{(l \to T_1 \lhd \Gamma_1) \mid (T_1 \lhd \Gamma_1) \in \mathcal{I}(\mathcal{U}, t)\} \textsf{ in} & (\mathcal{I}.\text{ext}) \\
 & \textsf{let } \tau_2^s = \{\sigma(T_2 \lhd \Gamma_2) \mid (T_2 \lhd \Gamma_2) \in \mathcal{I}(\mathcal{U}, v), & \\
 & \quad \sigma \in \mathcal{U}(\{e(\alpha[l] \to f\ \beta) \leq T_2\} \cup \{U \leq U \mid x : U \in \Gamma_2\}), e, f, \alpha, \beta \textsf{ fresh}\} \textsf{ in} & \\
 & \textsf{isect}(\tau_1^s \cup \tau_2^s) &
\end{array}$$

□

In words, the new rule ($\mathcal{I}$.ext) states that to infer a typing for an extension $l \to t \Cap v$ we must first infer typings for the subterms t and v. Typings for $l \to t \Cap v$ are constructed from the typings for t in a way that corresponds to the (extl) typing rule and these are placed in the set τ_1^s. The inferred typings for v on the other hand may not be immediately suitable as typings for $l \to t \Cap v$ because by Rule (extr) they must "lack" the label l in their domain. To fix this, the unification algorithm is invoked with a constraint (explained below) to ensure that in each case the type of v lacks the label l. The typings for v after unification are then placed in the set τ_2^s. Finally, all of the typings from both τ_1^s and τ_2^s are then intersected together into a single typing using the function isect which wraps each intersection component in a fresh E-variable for generality.

The novel part of this rule is the way that E-variables are used in combination with constrained simple type variables to describe the constraint that the type of v must lack the label l in its domain. The constraint $e(\alpha[l]{\to}f\ \beta) \leq T_2$ ensures that T_2 will lack the label l after the unifier is applied, and is also designed so that e can be expanded into an intersection of function types in cases where T_2

represents the type of an extension with multiple fields. The type $e\ (\alpha[l]{\to}f\ \beta))$ essentially corresponds to a row variable that would be found in traditional extensible record systems, but is instead built up from more primitive constructs, most of which are inherited from the System E type system.

The following lemma is used to show that our encoding of row variables is general enough to describe any unknown record type.

Lemma 5.14 (Row variable substitution for E$^{\mathrm{vcr}}$)**.** *Given any T, $\alpha[l]$, β and e, if $T{::}l$ then there exists an expansion E such that $E\ (\alpha[l] \to e\ \beta) \equiv T$.*

Proof. The proof is by induction on the structure of the derivation of $T{::}l$.

case: $\omega{::}l$. Then E is ω.

case: $(T_1 \cap T_2){::}l$ if $T_1{::}l$ and $T_2{::}l$.

(1)	$\exists E_1.\quad E_1\ (\alpha[l] \to e\ \beta) \equiv T_1$	ind.hyp.
(2)	$\exists E_2.\quad E_2\ (\alpha[l] \to e\ \beta) \equiv T_2$	ind.hyp.
(3)	$(E_1 \cap E_2)\ (\alpha[l] \to e\ \beta) = E_1\ (\alpha[l] \to e\ \beta) \cap E_2\ (\alpha[l] \to e\ \beta)$	def 3.11
	$\equiv T_1 \cap T_2$	(1),(2)
$\therefore$	E is $E_1 \cap E_2$	(3)

case: $(e_1\ T_1){::}l$ if $T_1{::}l$.

(1)	$\exists E_1.\quad E_1\ (\alpha[l] \to e\ \beta) \equiv T_1$	ind.hyp.
(2)	$(e_1\ E_1)\ (\alpha[l] \to e\ \beta) = e_1\ (E_1\ (\alpha[l] \to e\ \beta))$	def 3.11
(3)	$\equiv e_1\ T_1$	(1)
$\therefore$	E is $e_1\ E_1$	(3)

case: $(T_1 \to T_2){::}l$ if (1) $T_1 \equiv \bar{T_1}$ and (2) $\bar{T_1}{\restriction}l$.

(3)	$\exists E_1.\quad E_1\ \beta = T_2$	lem 3.17
(4)	$(\alpha[l] := \bar{T_1}, e := E_1)\ (\alpha[l] \to e\ \beta) = \bar{T_1} \to T_2$	def 3.11 with (2),(3)
	$\equiv T_1 \to T_2$	def 3.7 with (1)
$\therefore$	E is $(\alpha[l] := \bar{T_1}, e := E_1)$	(4)

□

Finally, we extend the proofs of lemmas and theorems from previous chapters that are effected by the introduction of extensions constrained simple types variables.

Restatement of Theorem 3.58 (Termination of $\mathcal{I}$ for E$^{\text{vcr}}$). *Given any $\mathcal{U}$ and t, if each use of $\mathcal{U}$ by $\mathcal{I}(\mathcal{U}, t)$ terminates, then $\mathcal{I}(\mathcal{U}, t)$ terminates.*

Proof. Straightforward by induction on the structure of t. □

Restatement of Theorem 3.59 (Correctness of $\mathcal{I}$ for E$^{\text{vcr}}$). *Given any $\mathcal{U}$, τ and t, if $\tau \in \mathcal{I}(\mathcal{U}, t)$, then $t : \tau$.*

Proof. The cases are the same as before (p.142), with one new case for extensions.

case: $t = l \to t_1 \Cap v$.

(1)	$\exists e.\ \ \mathcal{I}(\mathcal{U}, t) = \{e\ \mathcal{I}_v(\mathcal{U}, t)\} = \{e\ \mathsf{isect}(\tau_1^s \cup \tau_2^s)\}$ where	$(\mathcal{I}.\text{val})/(\mathcal{I}.\text{ext})$
	$\tau_1^s = \{l \to T_1 \lhd \Gamma_1 \mid (T_1 \lhd \Gamma_1) \in \mathcal{I}(\mathcal{U}, t_1)\}$	
	$\tau_2^s = \{\sigma\ (T_2 \lhd \Gamma_2) \mid (T_2 \lhd \Gamma_2) \in \mathcal{I}(\mathcal{U}, v),$	
	$\sigma \in \mathcal{U}(\{e_1(\alpha[l] \to e_2\beta) \leq T_2\} \cup \{U \leq U \mid x : U \in \Gamma_2\}),$	
	$e_1, e_2, \alpha, \beta\ \mathsf{fresh}\}$	
(2)	For each $(l \to T_1 \lhd \Gamma_1) \in \tau_1^s$ where	(1)
	(2a) $(T_1 \lhd \Gamma_1) \in \mathcal{I}(\mathcal{U}, t_1)$	
(2b)	$t_1 : T_1 \lhd \Gamma_1$	ind.hyp. with (2a)
(2c)	$l \to t_1 \Cap v : l \to T_1 \lhd \Gamma_1$	(extl) with (2b)
(3)	$\tau' \in \tau_1^s \implies l \to t_1 \Cap v : \tau'$	(2)-(2c)
(4)	For each $\sigma\ (T_2 \lhd \Gamma_2) \in \tau_2^s$ where	from (1)
	(4a) $(T_2 \lhd \Gamma_2) \in \mathcal{I}(\mathcal{U}, v),$	”
	(4b) $\sigma \in \mathcal{U}(\{e_1(\alpha[l] \to e_2\beta) \leq T_2\} \cup \{U \leq U \mid x : U \in \Gamma_2\}),$	”
	(4c) e_1, e_2, α, β fresh	”
(4d)	$(e_1\ (\alpha[l] \to e_2\ \beta))::l$	def 5.6
(4e)	$(\sigma\ (e_1\ (\alpha[l] \to e_2\ \beta)))::l$	lem 5.8 with (4d)
(4f)	$\sigma\ (e_1\ (\alpha[l] \to e_2\ \beta)) \equiv \sigma\ T_2$	def 3.37 with (4b)
(4g)	$(\sigma T_2)::l$	lem 5.7 with (4e),(4f)
(4h)	$v : T_2 \lhd \Gamma_2$	ind.hyp. with (4a)

(4i)	$v : \sigma\ T_2 \lhd \sigma\ \Gamma_2$	lem 3.28 with (4h)
(4j)	$l \to t_1 \Cap v : \sigma\ T_2 \lhd \sigma\ \Gamma_2$	(extr) with (4i),(4g)
(5)	$\tau \in \tau_2^s \implies l \to t_1 \Cap v : \tau$	(4)-(4j)
(6)	$\forall \tau'.\ \tau' \in \tau_1^s \cup \tau_2^s \implies l \to t_1 \Cap v : \tau'$	(3),(5)
(7)	$l \to t_1 \Cap v : \mathsf{isect}(\tau_1^s \cup \tau_2^s)$	lem 3.55 with (6)
(8)	$l \to t_1 \Cap v : e\ \mathsf{isect}(\tau_1^s \cup \tau_2^s)$	(evar) with (7)
$\therefore$	if $\tau \in \mathcal{I}(\mathcal{U}, l \to t_1 \Cap v)$, then $l \to t_1 \Cap v : \tau$	(1),(8)

□

Restatement of Theorem 3.60 (Principality of $\mathcal{I}$ for E$^{\text{vcr}}$). *Given any $\mathcal{U}$, t and τ', if* $\mathsf{covering}(\mathcal{U})$ *and $t : \tau'$ and $\mathcal{I}(\mathcal{U}, t)$ terminates, then there exists a substitution σ and a typing $\tau \in \mathcal{I}(\mathcal{U}, t)$ such that $\tau' \equiv \sigma\tau$.*

Proof. Let D be the derivation of $t : \tau'$. The cases are the same as before (p.143), with two new cases for Rule (extl) and Rule (extr).

case: D ends with an application of Rule (extl) of the form

$$\frac{(2)\ t_1 : T_1' \lhd \Gamma'}{(1)\ l \to t_1 \Cap v : S \to T_1' \lhd \Gamma'}\ (3)\ S \equiv l$$

(5)	$\mathcal{I}(\mathcal{U},t)$ terminates	assumption
(6)	$\exists e.\ \ \mathcal{I}(\mathcal{U},t) = \{e\ \mathcal{I}_v(\mathcal{U},t)\} = \{e\ \mathsf{isect}(\tau_1^s \cup \tau_2^s)\}$ where	$(\mathcal{I}.\text{val})/(\mathcal{I}.\text{ext})$ with (5)
	$\tau_1^s = \{l \to T_1 \lhd \Gamma_1 \mid (T_1 \lhd \Gamma_1) \in \mathcal{I}(\mathcal{U},t_1)\}$	
	$\tau_2^s = \{\sigma_1\ (T_2 \lhd \Gamma_2) \mid (T_2 \lhd \Gamma_2) \in \mathcal{I}(\mathcal{U},v),$	
	$\sigma_1 \in \mathcal{U}(\{e_1\ (\alpha[l]{\to}e_2\beta){\underline{\le}}T_2\} \cup \{U{\underline{\le}}U \mid x:U{\in}\Gamma_2\}),$	
	$e_1, e_2, \alpha, \beta\ \mathsf{fresh}\}$	
(7)	$\mathcal{I}(\mathcal{U},t_1)$ terminates	(5),(6)
(8)	$\exists \sigma_1', T_1'', \Gamma''$ such that	ind.hyp. with (7),(2)
(9)	$(T_1'' \lhd \Gamma'') \in \mathcal{I}(\mathcal{U},t_1)$	"
(10)	$\sigma_1'(T_1'' \lhd \Gamma'') \equiv (T_1' \lhd \Gamma')$	"
(11)	$\sigma_1'(l \to T_1'' \lhd \Gamma'') \equiv (S \to T_1' \lhd \Gamma')$	(10),(3)
(12)	$(l \to T_1'' \lhd \Gamma'') \in \tau_1^s$	(9),(6)
(13)	$(l \to T_1'' \lhd \Gamma'') \in (\tau_1^s \cup \tau_2^s)$	(12)
(14)	$\exists E.(l \to T_1'' \lhd \Gamma'') \equiv E\ \mathsf{isect}(\tau_1^s \cup \tau_2^s)$	lem 3.56 with (13)
	$= (e := E)\ (e\ (\mathsf{isect}(\tau_1^s \cup \tau_2^s)))$	def 3.11

(15)	$S \to T_1' \lhd \Gamma'$	$\equiv$	$\sigma_1'(l \to T_1'' \lhd \Gamma'')$	(11)
		$\equiv$	$\sigma_1'((e := E)\ (e\ \mathsf{isect}(\tau_1^s \cup \tau_2^s)))$	lem 3.16 with (14)
		$=$	$(\sigma_1'(e := E))\ (e\ \mathsf{isect}(\tau_1^s \cup \tau_2^s))$	lem 3.14

$\therefore$ σ is $\sigma_1'(e := E)$ (15),(6)

case: D ends with an application of Rule (extr) of the form

$$\frac{(2)\ v : T_1 \lhd \Gamma_1}{(1)\ l \to t_1 \Cap v : T_1 \lhd \Gamma_1}\ (3)\ T_1{::}l$$

(4)	$\mathcal{I}(\mathcal{U},t)$ terminates	assumption
(5)	$\mathcal{I}(\mathcal{U},v)$ terminates	def 3.57 with (4)
(6)	$\exists T_1', \Gamma_1'.\quad \mathcal{I}(\mathcal{U},v) = \{T_1' \lhd \Gamma_1'\}$	($\mathcal{I}$.val) with (5)
(7)	$\exists e.\quad \mathcal{I}(\mathcal{U},t) = \{e\ \mathcal{I}_v(\mathcal{U},t)\} = \{e\ \mathsf{isect}(\tau_1^s \cup \tau_2^s)\}$ where	($\mathcal{I}$.val)/($\mathcal{I}$.ext) with (4)
	$\tau_1^s = \{l \to T_0 \lhd \Gamma_0 \mid (T_0 \lhd \Gamma_0) \in \mathcal{I}(\mathcal{U},t_1)\}$	
	$\tau_2^s = \{\sigma_0\ (T_1' \lhd \Gamma_1') \mid \sigma_0 \in \mathcal{U}(\Delta)\}$	(6)
	where $\Delta = \{e_1(\alpha[l] \to e_2\beta) \leq T_1'\} \cup \{U \leq U \mid x : U \in \Gamma_1'\}$, e_1, e_2, α, β fresh	
(8)	$\exists \sigma_1.\quad \sigma_1(T_1' \lhd \Gamma_1') \equiv (T_1 \lhd \Gamma_1)$	ind.hyp. with (2),(6)
(9)	e_1 does not occur in $T_1' \lhd \Gamma_1'$	e_1 is fresh
(10)	$\exists E.\quad E\ (\alpha[l] \to e_2\ \beta) \equiv T_1$	lem 5.14 with (3)
	let $\sigma_2 = (\sigma_1, e_1 := E)$	
(11)	σ_2 is a unifier for Δ	def 3.11, (8),(9),(10)
(12)	$\mathcal{U}(\Delta)$ terminates	(4),(7)
(13)	$\exists \sigma_3, \sigma_4.\quad \sigma_3 \in \mathcal{U}(\Delta)$ and $\sigma_2\Delta \equiv \sigma_4\sigma_3\Delta$	covering($\mathcal{U}$) with (12),(11)
(14)	$\sigma_3\ (T_1' \lhd \Gamma_1') \in \tau_2^s$	(13)
(15)	$\sigma_3\ (T_1' \lhd \Gamma_1') \in \tau_1^s \cup \tau_2^s$	(14)
(16)	$\exists E'.\quad \sigma_3\ (T_1' \lhd \Gamma_1') \equiv E'\ \mathsf{isect}(\tau_1^s \cup \tau_2^s)$	lem 3.56 with (15)
	$= (e := E')\ (e\ \mathsf{isect}(\tau_1^s \cup \tau_2^s))$	def 3.11
(17)	$\sigma_2(T_1' \lhd \Gamma_1') \equiv (T_1 \lhd \Gamma_1)$	(8), e_1 is fresh
(18)	$T_1 \lhd \Gamma_1 \equiv \sigma_2(T_1' \lhd \Gamma_1')$	(17)
	$\equiv (\sigma_4\sigma_3)\ (T_1' \lhd \Gamma_1')$	(13)
	$= \sigma_4\ (\sigma_3\ (T_1' \lhd \Gamma_1'))$	lem 3.14
	$\equiv \sigma_4\ ((e := E')\ (e\ \mathsf{isect}(\tau_1^s \cup \tau_2^s)))$	(16)
	$= (\sigma_4(e := E'))\ (e\ \mathsf{isect}(\tau_1^s \cup \tau_2^s)))$	lem 3.14
$\therefore$	$\sigma = \sigma_4(e := E')$	(18),(7) $\square$

5.4 Examples

This section demonstrates our implementation `evcr` on a set of extensible record-based examples and object-oriented examples encoded using extensible records. Section 5.4.1 first introduces examples of extensible records and examines the behaviour of our system in relation to field extension, field overriding, first-class labels and subtyping. Then, sections 5.4.2, 5.4.3 and 5.4.4 demonstrates `evcr` on the examples of object-orientation, first-class polymorphism and compositionality originally presented in Chapter 1 in sections 1.1.1, 1.1.2 and 1.1.3 respectively.

5.4.1 Extensible Records

Basic Syntax

The syntax for expressing extensible records in `evcr` is similar to the syntax used in the theory except that labels are now prefixed with a dot to distinguish them from ordinary variable identifiers. Also, braces { ... } can be used interchangeably with parentheses `( ... )` and by convention will be used instead of parentheses when enclosing extensible records. An extensible record containing two fields `name` and `employed` can be defined as follows:

```
$ {.name -> "John" ^ .employed -> true ^ {}};;
: a (b (.name -> c Str) ^ d (.employed -> e Bool))
```

`evcr` also supports the following record-like syntactic sugar, using a comma to separate fields and using `=` to separate field labels from field values:

```
$ {name = "John", employed = true, {}};;
: a (b (.name -> c Str) ^ d (.employed -> e Bool))
```

Since field labels are prefixed with a dot, the application of an extensible record to a field label resembles the traditional "dot" notation for field selection:

```
$ {name = "John", employed = true, age = 41, {}} .age;;
: a Int
= (.name -> "John" ^ .employed -> true ^ .age -> 41 ^ {}) .age
> (.employed -> true ^ .age -> 41 ^ {}) .age
> (.age -> 41 ^ {}) .age
> 41
```

Row Variables and Field Overriding

The following example defines a function that takes an unknown record as a parameter and returns an extension of it:

```
$ \r. {age = 41, r};;
: a (b c ([.age] -> d []) -> b (c ([.age] -> d []) ^ e (.age -> f Int)))
```

Here, `[.age]` represents a constrained simple type variable with the label constraint including only the label `.age`, while `[]` represents a constrained simple type variable with an empty label constraint. The encoded row variable `c ([.age] -> d [])` specifies that the given record type must not already expose an `age` field. The following example demonstrates the application of this function to a record:

```
$ (\r. {age = 41, r} ) {name = "John", employed = true, {}};;
: a (b (c (.name -> d Str) ^ e (.employed -> f Bool)) ^ g (.age -> h Int))
= (\r..age->41^r) (.name->"John"^.employed->true^{})
> .age->41^.name->"John"^.employed->true^{}
```

The next example shows the application of the same function to a record that *does* have an existing `age` field, thereby demonstrating field overriding:

```
$ (\r. {age = 41, r} ) {name = "John", employed = true, age = "nonsense", {}};;
: a (b (c (.name -> d Str) ^ e f (.employed -> g Bool)) ^ h (.age -> i Int))
= (\r..age->41^r) (.name->"John"^.employed->true^.age->"nonsense"^{})
> .age->41^.name->"John"^.employed->true^.age->"nonsense"^{}
```

This works because the constrained simple type variable `[.age]` requires only that the *type* of the given record does not expose an `age` field, even though the actual record may contain an `age` field. To analyse this term, `evcr` would have first analysed the argument as:

```
$ {name = "John", employed = true, age = "nonsense", {}};;
: j (k (.name -> l Str) ^ m (n (.age -> o Str) ^ p (.employed -> q Bool)))
```

with all fields present, and then eliminated the existing `.age` field by substituting the ω expansion for the E-variable `n` before introducing the new `.age` field.

First-Class Labels

Labels are first-class citizens and can be passed as arguments to extensions. They can also be bound to term variables, just as any other value can be bound to term variables. This means that it is possible to define abstractions over term variables that represent labels, and it is possible to infer types for these term variables compositionally.

In the following example, we have the application of a record to an unknown label represented by the term variable `x`:

```
$ {name = "John", employed = true, {}} x;;
: a Str <| x : .name
: a Bool <| x : .employed
```

Interestingly, the unknown label represented by `x` could actually be one (but not both) of two possibilities: either `x` is the label `.name` or `x` is the label `.employed`. These two possibilities cause `evcr` to infer two possible and independent typings, which are not intersected together due to the value restriction. This analysis is compositional and does not depend on any outside context.

When we abstract over `x`, we get a value which now permits us to intersect the two independent typings together into a single intersection type:

```
$ \x.{name = "John", employed = true, {}} x;;
: a (b (.employed -> c Bool) ^ d (.name -> e Str))
```

As desired, this "η-expansion" is inferred to have the same type as the original record itself:

```
$ {name = "John", employed = true, {}};;
: a (b (.employed -> c Bool) ^ d (.name -> e Str))
```

Another consequence of making labels first-class citizens is that extensible record types occuring in parameter type positions can sometimes be more general than is actually useful. For example, consider the following function:

```
$ let area = \rect. rect.width * rect.height;;
: a (((b .width -> Int) ^ (c .height -> Int)) -> d Int)
```

Although rect is intended to represent an extensible record, as far as the type inference algorithm is concerned, rect could potentially be a λ-abstraction that uses its argument zero times. Thus it is possible to substitute the ω expansion for E-variables a and c resulting in the following type:

```
$ let area = \rect. rect.width * rect.height;;
: a (((w -> Int) ^ (w -> Int)) -> d Int)
```

At this type, the area function would no longer accept an extensible record as an argument since by rules (extl) and (extr), no extensible record can be assigned field types w -> Int and w -> Int. The rect parameter is now permitted only to be an abstraction that uses its parameter zero times and returns an Int (e.g. \x.3).

Record Subtyping

The examples in this section demonstrate that intersection types are powerful enough to support code reuse that is usually supported via subtype polymorphism. Consider the following function that calculates the area of a rectangle:

```
$ let area = \rect. rect.width * rect.height;;
: a (((b .width -> Int) ^ (c .height -> Int)) -> d Int)
```

This function can of course be applied to a record with only width and height fields:

```
$ area {width=3, height=5, {}};;
: a Int
--- reduction steps omitted ---
> 15
```

However, in a way that resembles subtyping, this function can also be applied to a record that contains fields width and height plus additional fields x and y:

```
$ area {x=2, y=2, width=3, height=5, {}};;
: a Int
--- reduction steps omitted ---
> 15
```

The record argument's type adapts to match the parameter type, and so it is the record argument that is acting polymorphically, not the function. What happens during type inference is that first the argument's type is inferred in isolation:

```
$ {x=2, y=2, width=3, height=5, {}};;
: a (b (.x -> c Int) ^ d (e (.y -> f Int) ^ g (h (.height -> i Int) ^ j (.width -> k Int))))
```

Then, at the point of application, the type inference algorithm eliminates the fields labelled `x` and `y` from this type by substituting the ω expansion for E-variables `b` and `e`, and this transforms the record's type to the required parameter type.

This polymorphism can be exercised by writing a single program that uses the same record at both types. To demonstrate this, we define the following set of functions:

```
1  $ let area = \rect. rect.width * rect.height;;
2  : a (((b .width -> Int) ^ (c .height -> Int)) -> d Int)

3  $ let rect2str = \rect.
4       str(rect.x)++","++str(rect.y)++":"++str(rect.width)++"x"++str(rect.height);;
5  : a (((((b .x -> Int) ^ (c .y -> Int)) ^ (d .width -> Int)) ^ (e .height -> Int)) -> f Str)

6  $ let poly = \rect.
7              "rect=" ++ rect2str rect ++ ", area=" ++ str (area rect);;
8  : a ((((((b .x -> Int) ^ (c .y -> Int)) ^ (d .width -> Int)) ^ (e .height -> Int))
9      ^ (f .width -> Int) ^ (g .height -> Int))
10     -> h Str)
```

The `area` function uses the given record at a type exposing only the `width` and `height` fields. The `rect2str` function uses the given record at a type exposing all four fields. Finally, the `poly` function takes a rectangle and uses it with both of the other functions. Its parameter type is an intersection type showing that the parameter is used once for all 4 fields (Line 8), and then used a again just for its `width` and `height` fields (Line 9). The application of this function to our polymorphic rectangle record succeeds with the following result:

```
$ poly {x=2, y=2, width=3, height=5, {}};;
: a Str
--- reduction steps omitted ---
> "rect=2,2:3x5, area=15"
```

5.4.2 Object-Orientation

This section demonstrates our type inference algorithm on all of the object-oriented examples from Chapter 1, Section 1.1.1. These examples are supported by encoding the various object-oriented features, such as objects, classes, inheritance, method overriding and dynamic dispatch, into more primitive features, such as variables, functions and records.

There are two main approaches to encoding objects using record-like structures, and they differ primarily in their treatment of the special variable "`this`" by which objects can refer to themselves. Wand's approach [64] follows the *recursive record semantics* pioneered by Cardelli [11] and uses a fixed-point operator to recursively bind references to `this` within a record to the record itself. The other approach is the *self-application semantics* [32] where each method takes `this` as a parameter, and where invoking a method involves applying the method stored in a record to the record itself.

Operationally, the self-application semantics is simpler to implement since it does not require any special primitive to support recursion. However, from a typing perspective, the recursive record semantics is superior because it does not need a `this` parameter on each method, and this allows expected subtype relations to hold. For example, we shall see later that if a `this` parameter is exposed on all methods, then the type of `PositionedRectangle` objects can no longer be considered a subtype of the type of `Rectangle` objects.

Since we are in a calculus without a fixed-point operator, we adopt the self-application semantics. It is possible to use this semantics because `evcr` has no difficulty inferring typings for programs that involve self-application. This is in contrast to Hindley/Milner-based type inference which supports self-application only in limited situations where context information is available. Because the self-application semantics exposes the `this` parameter on all methods, only the non-subtyping examples can be typed. However, to handle the subtyping example, we use a variation on the self-application semantics in which *wrapper* objects are used to "hide" the `this` parameter. While not as elegant as the recursive record semantics, wrapper objects at least allow us to demonstrate that our system of expansion variables and extensible records produces expected results when the `this` parameter is hidden, and that it should therefore also produce expected results if the system were extended with a fixed-point operator to support the recursive record semantics.

First, the non-subtyping examples are demonstrated using the self-application semantics, and then the subtyping example is demonstrated using a variation of this semantics.

Self-Application Semantics

In the self-application semantics, an object is an extensible record containing both data fields and methods. A method is a function whose first parameter must be `this`. A class is simply a function that takes initial field values as parameters and returns an object based on those initial field values. For example, the `Rectangle` class is defined as follows:

```
$ let Rectangle = \w.\h. {
    width = w,
    height = h,

    area = \this. {
        this.width * this.height
    },

    toString = \this. {
        str(this.width) ++ "x" ++ str(this.height)
    },

    {}
};;
```

The inferred type shown below reflects that the `Rectangle` class is a function taking the two field values as parameters and returning a record containing fields `width` and `height` and methods `area` and `toString`.

```
: a (b c d e [] -> b (c f g h [] ->
       c (d (.width -> e [])
     ^ f (g (.height -> h [])
     ^ i (j (.area -> k (((l .width -> Int) ^ (m .height -> Int)) -> n Int))
     ^    o (.toString -> p (((q .width -> Int) ^ (r .height -> Int)) -> s Str)))))))
```

According to this type, the `area` method requires its `this` parameter to have type `(l .width -> Int) ^ (m .height -> Int)`. That is, it requires `this` to be a record containing fields `width` and `height`. The `toString` method requires exactly the same of its `this` parameter.

A `Rectangle` object `rect` is created by applying class `Rectangle` to initial field values, and its methods are invoked by passing `rect` itself to the `this` parameter:

```
$ let rect = Rectangle 10 20 in
let a = rect.area rect in
let s = rect.toString rect in
"area=" ++ str a ++ ", toString=" ++ s;;
: a Str
--- reduction steps omitted ---
> "area=200, toString=10x20"
```

The subclass `PositionedRectangle` is also defined as a function taking initial field values and returning an object. However, this object is constructed as an *extension* of `super` which is an instance of the superclass `Rectangle`:

```
$ let PositionedRectangle = \x.\y.\w.\h.
    let super = Rectangle w h;
{
    x = x,
    y = y,

    toString = \this.  {
        super.toString this ++ "(" ++ str(this.x) ++ "," ++ str(this.y) ++ ")"
    },

    super
};;
```

Note that the subclass `PositionedRectangle` inherits fields `width` and `height` and method `toString` from the superclass `Rectangle`. It then extends `Rectangle` by adding two fields `x` and `y` and *overriding* method `toString` with a new definition. Also note that the new implementation of `toString` is able to refer to and reuse the original inherited implementation of `toString` by invoking `super.toString`.

The inferred type for class `PositionedRectangle` shown below describes a function that takes 4 parameters, representing 4 field values, and returns a record containing fields `x`, `y`, `width` and `height`, and methods `area` and `toString`. Note that since the `toString` method of `PositionedRectangle` overrides the `toString` method of the superclass `Rectangle`, this method appears in the inferred type only once:

```
: a (b c d e f g [] -> b (c d e h i j [] -> c (d e h k l m n [] -> d (e h k l o p q [] ->
    e (f (.x -> g [])
    ˆ h (k (l (m (.width -> n [])
    ˆ o (p (.height -> q [])
    ˆ r s (.area
        -> t (((u .width -> Int) ˆ (v .height -> Int))
        -> w Int))))
    ^ x (.toString
       -> y (((((z .width -> Int) ^ (ba .height -> Int)) ^ (bb .x -> Int)) ^ (bc .y -> Int))
       -> bd Str)))
    ^ i (.y -> j [])))))))
```

As before, a `PositionedRectangle` object `r` can be created by applying the class to initial field values, and the new object `r` can be used by applying its methods to `r` itself:

```
let r = PositionedRectangle 1 2 3 4 in
"area=" ++ str(r.area r) ++ ", toString=" ++ r.toString r;;
: a Str
--- reduction steps omitted ---
> "area=12, toString=3x4(1,2)"
```

The self-application semantics is limited when it comes to subtyping due to the exposed `this` parameter. Ideally, a `PositionedRectangle` object should be coercible to have the same type as a `Rectangle`, but because of the exposed `this` parameter, it is not. To understand why, consider the following function which finds the larger of two rectangles:

```
$ let larger = \r1.\r2.  {
    if (r1.area r1 > r2.area r2) r1
    else r2
};
```

Since it is not known during static analysis whether this function will return `r1` or `r2`, the function will effectively coerce the given `r1` and `r2` to have the same type by unifying the two types:

```
let r1 = Rectangle 2 2;
let r2 = PositionedRectangle 1 2 10 20;
let bigRect = larger r1 r2;
bigRect;;

: a b c (d (e f (.width -> g Int)
  ^ h (i j k (.height -> l Int)
  ^ m n o p q (.area -> r (((s .width -> Int) ^ (t .height -> Int)) -> u Int))))
  ^ v x y z (.toString -> w))
```

The type of `bigRect`, which is the unification of the types of `r1` and `r2`, is satisfactory with respect to the members `width`, `height` and `area`, but the `toString` method is typed as ω making it unusable:

```
bigRect.toString bigRect;;

: No typings found
```

The reason that `toString` has type ω after unification is that the type of `r1`'s `toString` method and `r2`'s `toString` method are are incompatible and not unifiable: `r2.toString` requires its `this` parameter to have `x` and `y` fields, while `r1.toString` does not and cannot have this requirement due to the linear typing model.

This problem can be solved by hiding the `this` parameter, which will be done in the following section.

Hiding the "`this`" Parameter

In this section, we show how to create a wrapper around objects encoded in the self-application semantics that hides the `this` parameter, achieving similar typing benefits to the recursive record semantics.

The `Rectangle` class is now defined in three steps. First, we define a *raw class* called `Rectangle_raw` in the normal self-application semantics:

```
let Rectangle_raw = \w.\h. {
    width = w,
    height = h,

    area = \this. {
        this.width * this.height
    },

    toString = \this. {
        str(this.width) ++ "x" ++ str(this.height)
    },

    {}
};
```

Then, we define *wrapper class* called `Rectangle_wrap` which wraps around an instance of `Rectangle_raw` called `r`:

```
let Rectangle_wrap = \r.
{
    width = r.width,
    height = r.height,
    area = r.area r,
    toString = r.toString r,
    ()
};
```

This wrapper class defines the same fields and methods as the raw class but pre-applies the `this` parameter so that it is no longer exposed to users of the wrapper.

Finally, the actual `Rectangle` class is defined which creates a wrapped rectangle with a raw rectangle as its argument:

```
let Rectangle = \w.\h.Rectangle_wrap (Rectangle_raw w h);
```

A rectangle object created from this class now has a type that does not expose the `this` parameter:

```
Rectangle 2 3;;
: a (b c (.width -> d Int) ^ e (f (g (.area -> h Int) ^ i (.toString -> j Str))
 ^ k l m (.height -> n Int)))
```

The subclass `PositionedRectangle` is defined similarly in terms of a raw class, a wrapper class and the actual class. The raw class is defined as follows:

```
let PositionedRectangle_raw = \x.\y.\w.\h.
    let super = Rectangle_raw w h;
{
    x = x,
    y = y,

    toString = \this. {
        super.toString this ++ "(" ++ str(this.x) ++ "," ++ str(this.y) ++ ")"
    },

    super
};
```

The wrapper class needs its own `super` reference to be a *wrapper* rectangle rather than a *raw* rectangle, and is defined as:

```
let PositionedRectangle_wrap = \r.
    let super = Rectangle_wrap r;
```

```
{
    x = r.x,
    y = r.y,
    toString = r.toString r,
    super
};
```

Finally, the actual `PositionedRectangle` class is defined to create a wrapper positioned rectangle with a raw positioned rectangle as its argument:

```
let PositionedRectangle = \x.\y.\w.\h.
    PositionedRectangle_wrap (PositionedRectangle_raw x y w h);
```

By hiding the `this` parameter, a `PositionedRectangle` can now be coerced into a `Rectangle` and the subtyping example works as expected:

```
let larger = \r1.\r2.
{
    if (r1.area > r2.area) r1 else r2
};
let r1 = Rectangle 2 2;
let r2 = PositionedRectangle 1 2 10 20;
let bigRect = larger r1 r2;
bigRect.toString;;
: a Str
> "10x20(1,2)"
```

While a true fixed-point operator may still be a more elegant way to express the self-referential nature of objects, the above encoding at least demonstrates that our system of expansion variables and extensible records produces expected results when the `this` parameter is hidden, and should therefore also produce expected results if the recursive record semantics are used.

5.4.3 First-Class Polymorphism

This section demonstrates the substitution example from Chapter 1, Section 1.1.2. Because it requires first-class polymorphism, this example is *not* supported by object-oriented type inference systems in the Hindley/Milner style, such as those based on extensible records [62, 64, 28, 55, 9].

The example in the introduction was originally presented with 5 classes, however some of those classes, specifically `Substitution` and `Expression`, served only as types and contained no actual code.

In this section, those particular classes are excluded, and only the following 3 classes are needed: `Variable`, `IdentitySub` and `ExtendedSub`.

Class `Variable` is defined by the following function:

```
1  let Variable = \name.
2  {
3      name = name,

4      applySubToSelf = \this.\s.  {
5          let tail = s.tail;

6          if (this.name == s.x.name) s.v
7          else tail.apply tail this
8      },

9      toString = \this.  {
10         this.name
11     },

12     {}
13 };;
```

Method `applyToSelf`, as originally presented in Chapter 1, took a substitution `s` as its only parameter. When encoded using the self-application semantics, this method is augmented to take an additional `this` parameter. Furthermore, when invoking method `tail.apply` on Line 7, the first argument passed (i.e. to parameter "`this`") must be the object on which the method is being invoked, which in this case is the object `tail`, and then the normal method arguments follow after that.

Class `IdentitySub` is defined as follows:

```
1  let IdentitySub =
2  {
3      apply = \this.\item.  {
4          item
5      },

6      applySubToSelf = \this.\s.  {
7          s
8      },

9      toString = \this.  {
10         "{}"
11     },
    {}
};;
```

Even though `applySubToSelf` in this case does not use its `this` parameter, the parameter should still be present consistently on all methods.

Finally, class `ExtendedSub` is defined by the following function:

```
1  let ExtendedSub = \x.\v.\tail.
2  {
3      x = x,
4      v = v,
5      tail = tail,

6      apply = \this.\item.  {
7          item.applySubToSelf item this
8      },

9      applySubToSelf = \this.\s.  {
10         let newV = s.apply s (this.v);
11         let newTail = s.apply s (this.tail);

12         {v=newV, tail=newTail, this}
13     },

14     toString = \this.  {
15         let x = this.x;
16         let v = this.v;
17         let tail = this.tail;

18         x.toString x ++ ":=" ++ v.toString v ++ "," ++ tail.toString tail
19     },

20     {}
21 };;
```

Note that the `applySubToSelf` method of the `ExtendedSub` class uses `s.apply` polymorphically: first on Line 10 to take a variable to a variable, and then on Line 11 to take a substitution to a substitution. The inferred type of the `applySubToSelf` method reflects this polymorphism:

```
1 w (x ((y z ba ([.v,.tail] -> bb []) ^ (bg .tail -> bh [])) ^ (bi .v -> bj []))
2     -> x ((((bk .apply -> bl [] -> bh [] -> y z bc bd []) ^ bl [])
3           ^ (bm .apply -> bn [] -> bj [] -> y be bf []) ^ bn [])
4     -> y (be (.v -> bf []) ^ z (ba ([.v,.tail] -> bb []) ^ bc (.tail -> bd [])))))
```

Line 1 is the type of the `this` parameter, lines 2 and 3 are an intersection of the two different types at which the substitution `s` is used, and line 4 is the type of the substitution that is returned where the "row variable" `ba ([.v,.tail] -> bb [])` represents all of the unknown additional methods and fields of `this`.

Putting this polymorphism to the test, we create two substitutions, apply one to the other, and then produce a string representation of the result:

```
let a = Variable "a";
let b = Variable "b";
let c = Variable "c";
let d = Variable "d";
let id = IdentitySub;
let s1 = ExtendedSub b a id;
let s2 = ExtendedSub a b (ExtendedSub c d id);
let s1s2 = s1.apply s1 s2;

let s1Str = s1.toString s1;
let s2Str = s2.toString s2;
let s1s2Str = s1s2.toString s1s2;

"s1 is " ++ s1Str ++ " , s2 is " ++ s2Str ++ " , s1s2 is " ++ s1s2Str;;
: a Str
--- reduction steps omitted ---
> "s1 is b:=a,{} , s2 is a:=b,c:=d,{} , s1s2 is a:=a,c:=d,b:=a,{}"
```

Type inference succeeds, and evaluation produces the same answer as the original Java version of Chapter 1.

5.4.4 Compositionality

This section presents the examples of compositional analysis from Chapter 1, Section 1.1.3. Compositional analysis is *not* possible in type inference systems following Palsberg and Schwartzbach's approach [46], nor is it possible in Martin Plümicke's Java type inference algorithm [52, 51], since these systems are inherently based on a whole-program analysis.

Both of the examples presented in the introduction reference a class `A` which was left unspecified, but which we now define below:

```
let A = {
    m1 = \this.\arg.  {
        arg
    },

    {}
};;
: a (.m1 -> b (w -> c (d [] -> d [])))
```

Two examples were presented in the introduction.

Example 1

```
let C = {
    mono = \this.\b.  {
        let a = A;

        let _ = a.m1 a true;
        let _ = b.m2 b 42;

        ()
    },

    {}
};;
: a (.mono -> b (w -> c (((d .m2 -> e [] -> f Int -> w) ^ e []) -> g ())))
```

In Example 1, method `mono` makes use of two variables. Variable `a` is an object defined from a known class `A`. Variable `b`, on the other hand, is an object passed through the parameter of method `mono` and as such, the class of `b` is *not* known. By deconstructing the inferred type above, the inferred type of parameter `b` is found to be:

```
(d .m2 -> e [] -> f Int -> w) ^ e []
```

Rather than searching the whole program for a class name to be used as the type of `b`, `evcr` has instead inferred a *structural type*. The left component of the intersection type represents the type of an object with a single method called `m2` which takes a `this` parameter of type `e []`, and then takes an integer and returns a value that is not used. The right component of the intersection type says that b must also have the same type as the this parameter of method m2 so that the self-application semantics can work.

This example was used in the introduction to show that while a type for variable a could be easily inferred using known context information about class A, the same cannot be said for variable b since nothing is known of the origin of b. However, it is important to note that evcr ignores the fact that context information is actually available about a and infers types for *both* variables compositionally based only on how those variables are used within the method.

Example 2

```
1   let Foo = {
2       poly = \this.\b.  {
3           let a = A;
4           let _ = a.m1 a true;
5           let _ = a.m1 a 42;

6           let _ = b.m2 b true;
7           let _ = b.m2 b 42;

8           ()
9       },

10      {}
11  };;
12  : a (.poly -> b (w -> c ((
13                              ((d .m2 -> e [] -> f Int -> w) ^ e [])
14                             ^ (g .m2 -> h [] -> i Bool -> w) ^ h [])
15           -> j {})))
```

Example 2 is a slight variation on Example 1 in which both variables a and b are used polymorphically. evcr succeeds to infer a structural type for parameter b based on the way in which b is used within method poly. Lines 13 and 14 reveal the type of b as an intersection type with Line 13 revealing m2 as a method that takes integers, and Line 14 revealing m2 as a method that takes booleans.

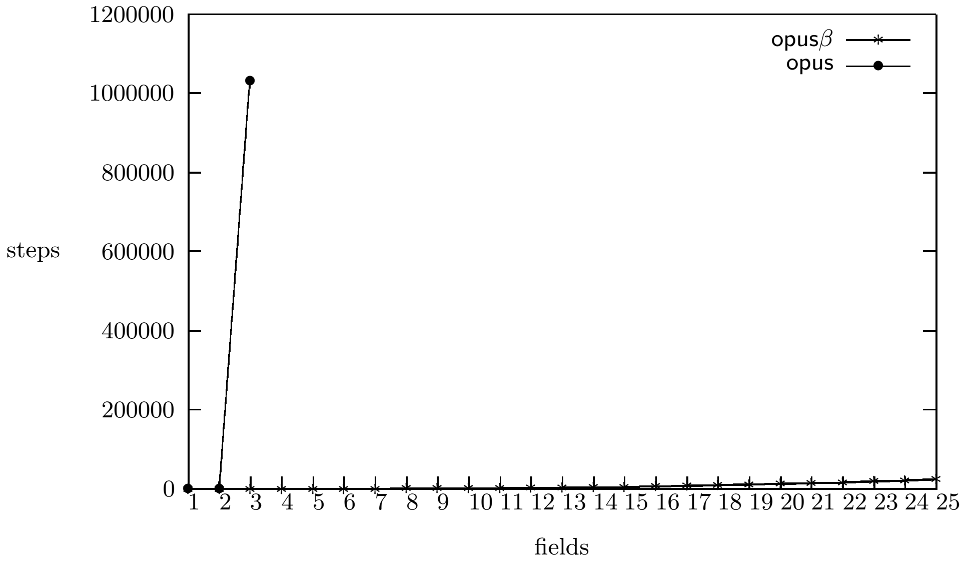


Figure 5.1: Performance of opusβ vs opus with extensible records

5.5 Efficiency

This section revisits the efficiency issues with opus that originally motivated the development of opusβ. In Section 3.6, we saw that the amount of branching performed by opus led the algorithm being prohibitively inefficient in many of the purely functional examples tested. When considering extensible records, the efficiency issues with opus become even more pronounced, to the extent that records with more than 2 fields cannot in practice be analysed without some optimisations such as those applied in opusβ.

To illustrate the efficiency issues, we run our implementation on a set of extensible records of the form $(l_1 \to 1, \ldots l_n \to 1, \{\})$ for $n = 1..25$, once using opusβ and once again using opus. Figure 3.2 plots on the X-axis the number of fields in each record, and on the Y-axis the number of $\overrightarrow{\text{opus}\beta}$ steps and $\overrightarrow{\text{opus}}$ steps needed to analyse the record using opusβ and opus respectively. The figure shows that type inference using opus quickly becomes impractical with extensible records that have a mere 3 fields. Based on these results, it is clear that opusβ is the more viable option for demonstrating Algorithm $\mathcal{I}$ on programs with records and therefore objects.

Figure 5.2 demonstrates opusβ on each of the object-oriented examples presented in this chapter,

Example	$\overrightarrow{\mathsf{opus}_\beta}$ steps
Section 5.4.2, Computing the largest rectangle (self-application semantics)	4259
Section 5.4.2, Computing the largest rectangle (wrapped semantics)	5240
Section 5.4.3, First-class polymorphism, substitution example	9892
Section 5.4.4, Compositionality, Example 1	124
Section 5.4.4, Compositionality, Example 2	214

Figure 5.2: Performance of $\mathsf{opus}\beta$ on object-oriented examples

showing in each case the number of $\overrightarrow{\mathsf{opus}_\beta}$ steps used. The cumulative 19,729 $\overrightarrow{\mathsf{opus}_\beta}$ steps took 18 seconds to complete on an Intel® Core™ i5-2400 CPU @ 3.10GHz with a single-threaded implementation.

Although $\mathsf{opus}\beta$ is a significant improvement over opus in terms of efficiency, it is clearly not the perfect unification algorithm. In addition to improving efficiency further, it would be desirable to have an algorithm that also offers the covering unifier sets property (as in opus) and offers some guarantee regarding termination (e.g. by limiting rank as in System I).

$\mathsf{opus}\beta$ might best be viewed as a first attempt to reduce opus's branching of the search path based on sound ideas from β-unification while retaining opus's ability to handle all constraint forms. Importantly, these efficiency improvements make it possible to demonstrate our type inference system successfully on the object-oriented examples of interest presented in the introduction.

5.6 Summary

This chapter presented System E^{vcr}, our final system including both extensible records and constants. We have shown that it is possible to adapt System E's technology of expansion variables to apply to extensible records. To do so, we encoded extensible records as functions from labels to values, and typed them using intersections of function types from labels to types, which has enabled us to reuse all of the function-oriented type machinery of System E. The challenging typing issue of row variables has been handled in System E^{vcr} through the novel use of constrained simple type variables which, in combination with function types, intersection types and E-variables, can be used to construct types that serve the same purpose as row variables. We showed how the object-oriented problem examples from Chapter 1 could be encoded into our calculus and, using our implementation `evcr`, demonstrated that Algorithm $\mathcal{I}$ was able to automatically infer typings for all of these examples. We then revisited our motivation to create an optimised derivative of `opus`, and we presented data to illustrate how `opus`β alleviates some of the significant performance deficiencies of `opus` in the context of extensible records.

Chapter 6

Conclusions

In this dissertation, we presented a new approach to type inference for object-orientation that addresses two objectives important to the object-oriented programming style:

1. *First-class polymorphism* should be used to support objects as first-class citizens.
2. *Compositional analysis* should be used to support the separate development of software modules.

The fact that our approach supports both of these objectives is significant because previous type inference approaches for objects have succeeded in satisfying only one or the other, but not both at the same time. For example, the approach of flow analysis by Palsberg and Schwartzbach [46] and the Java type inference algorithm by Martin Plümicke [52, 51] support first-class polymorphism, but at the cost of requiring a whole program analysis which prevents separate compilation. Conversely, the many approaches derived from the Hindley/Milner type inference system such as those based on extensible records [62, 64, 28, 55, 9] support a limited form of compositional analysis, but sacrifice the first-class polymorphism that is necessary to treat objects properly as first-class citizens.

Our solution combined System E with extensible records in ways that are novel in both disciplines. On the System E side, we showed that the expansion machinery of System E will work almost unmodified on extensible records if extensible records are interpreted as functions having intersections of function types, and record labels are treated as first-class constants that can be passed as arguments. Type safety in the call-by-value semantics is ensured by a value restriction on the expansion typing rules.

On the extensible records side, we invented a new alternative to row variables called *constrained simple type variables* which are useful when extensible records are treated as functions from first-class labels to values, and which can encode the usual notion of row variables when combined with the existing syntax of function types, intersection types and E-variables.

Our type inference algorithm departed from the previous algorithms for System E in order to support additional term forms beyond those of the pure λ-calculus. While previous type inference algorithms for System E are based on β-unification which assumes that all functions are λ-abstractions, our type inference algorithm was designed with covering unification in mind which makes no assumptions about the term language and thus supports extensible records posing as functions. However, covering unification is unfortunately a much slower process than β-unification because it needs to search more constraint reduction paths to produce a full covering set of unifiers. For this reason, we developed a hybrid unification algorithm that, like covering unification, handles all constraint forms, but like β-unification, uses asymmetric constraints to cut down the number of constraint reduction paths that need to be searched. Further study will be required to discover desirable properties for such a hybrid algorithm and to design an algorithm accordingly.

The main technical results contributed by this dissertation can be summarised as follows:

- A proof that System E^{v}, System E^{vc} and System $\mathrm{E}^{\mathrm{vcr}}$ are typesafe by subject reduction (Theorem 3.30) and progress (Theorem 3.31).
- A proof for each of System E^{v}, System E^{vc} and System $\mathrm{E}^{\mathrm{vcr}}$ that our type inference algorithm $\mathcal{I}$ terminates, is correct and finds principal typing sets whenever the chosen unification algorithm terminates, is correct and finds covering unifier sets respectively (Theorem 3.58, Theorem 3.59 and Theorem 3.60 respectively).
- A proof for each of System E^{v}, System E^{vc} and System $\mathrm{E}^{\mathrm{vcr}}$ that our hybrid type unification algorithm $\mathsf{opus}\beta$ is correct (Theorem 3.50).

In addition to these technical results, we also implemented and tested the type inference algorithm, verifying that it succeeds in inferring types for all of the object-oriented problem examples presented in the introduction. We compared our implementation to the System E type inference tool and found that the former matches the latter on all but one of the 61 λ-calculus examples from the System E

Inference Report in Appendix B (failing, where it does, in a way that is expected and consistent with the compositional approach of our algorithm). Using this implementation, we also highlighted the efficiency problems with opus and the improvements made in opusβ.

6.1 Future Work

Currently, our system supports only a core subset of object-oriented programming features. The following diagram shows what has been achieved to date, and in what direction future progress might be made:

$$\begin{tikzcd}
\textsf{Functions} & \textsf{Constants} & \textsf{Records} & \textsf{Recursion} & \textsf{Imperatives} \\
\mathrm{E}/\mathrm{E}^{v} \arrow[r, dotted] & \mathrm{E}^{vc} \arrow[r, dotted] & \mathrm{E}^{vcr} \arrow[r, dotted] & ? \arrow[r, dotted] & ?
\end{tikzcd}$$

While recursive programs are possible in System E by using a Y-combinator represented in pure λ-terms, type inference must resort to non-compositional analysis to succeed on such programs. Another issue is that current type inference algorithms for System E are restricted to the linear fragment of the type system, meaning that it is technically not feasible for them to infer a linear typing for a recursive function that may recurse for an unknown number of times (e.g. computing the factorial of an unknown number read from the user), or for an infinite number of times (e.g. a server operating system that (ideally) runs forever). These limitations might be addressed by adding a built-in Y-combinator with its own typing rule and making use of non-linear types to account for an unknown or infinite number of uses of a given term.

Imperative programming is also theoretically possible in System E by using monads [61] to capture the behaviour of state in a purely functional manner. However, there are efficiency benefits to building state-manipulating primitives directly into a language, and this will require changes to System E's type system.

Apart from adding new syntactic features to System E, another area for future research is to find further practical type system restrictions that offer decidable type inference. System I has decidable type inference for all finite-rank restrictions, although other as yet undiscovered restrictions may be possible. For example, papers on both of the previous type inference algorithms for System E [15, 6] state as future work the objective to develop type inference algorithms that perform any specified amount of evaluation (corresponding to partial evaluation of the program being analysed), followed by traditional monovariant analysis. The problems associated with constants also need further study. For example, the unifier of `not` : $\texttt{Bool} \to e\ \texttt{Bool}$ and $\lambda x.x : f[\,] \to f[\,]$ results in $\texttt{Bool} \to \texttt{Bool}$ which is unsatisfactory since the E-variable on the return type is lost (Section 4.5.1). Lastly, more work needs to be done on unification algorithms that support type unification in the general case, and not just in the specific case of β-unification which is hardwired to work for the pure λ-calculus. Such a unification algorithm will allow System E to be more easily extended to support the full range of term forms that can be found in real world object-oriented programming languages such as C++ and Java.

Bibliography

[1] TIOBE Programming Community Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[3] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 2–26, London, UK, UK, 1995. Springer-Verlag.

[4] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 15(4):575–631, 1993.

[5] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 233–246, New York, NY, USA, 1995. ACM.

[6] Adam Bakewell, Sbastien Carlier, A. J. Kfoury, and J. B. Wells. Inferring intersection typings that are equivalent to call-by-name and call-by-value evaluations. Technical report, 2005.

[7] Adam Bakewell and Assaf J. Kfoury. Unification with expansion variables: Preliminary results and problems. Technical report, 2004.

[8] Adam Bakewell and Assaf J. Kfoury. Properties of a rewrite system for unification with expansion variables. Technical report, 2005.

[9] Gérard Boudol. The recursive record semantics of objects revisited. *J. Funct. Program.*, 14:263–315, May 2004.

[10] Grard Boudol and Pascal Zimmer. On type inference in the intersection type discipline. *Electr. Notes Theor. Comput. Sci.*, 136:23–42, 2005.

[11] Luca Cardelli. A semantics of multiple inheritance. *Inf. Comput.*, 76:138–164, February 1988.

[12] Luca Cardelli and John C. Mitchell. Operations on records. In *Proceedings of the fifth international conference on Mathematical foundations of programming semantics*, pages 22–52, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[13] Sébastien Carlier. System E inference report. `http://www.macs.hw.ac.uk/DART/software/system-e/examples/terms.html`.

[14] Sébastien Carlier. System E type inference tool. `http://www.macs.hw.ac.uk/DART/software/system-e/`.

[15] Sébastien Carlier and J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β-reduction. In *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '04, pages 132–143, New York, NY, USA, 2004. ACM.

[16] Sébastien Carlier and J. B. Wells. Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation. *Electronic Notes in Theoretical Computer Science*, 136:173–202, 2005.

[17] Sbastien Carlier, Jeff Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *IN PROGRAMMING LANGUAGES & SYSTEMS, 13TH EUROPEAN SYMP. PROGRAMMING*, pages 294–309. Springer-Verlag, 2004.

[18] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies).* Princeton University Press, Princeton, NJ, USA, 1985.

[19] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. 1980.

[20] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. *Principal Type Schemes and Lambda-calculus Semantics*, pages 480–490. Accademic Press, London, 1980.

[21] William Damon. Multithreaded game programming and hyper-threading technology. `http://software.intel.com/en-us/articles/multithreaded-game-programming-and-hyper-threading-technology`, September 2011.

[22] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical report, 1996.

[23] Silvia Ghilezan. Strong normalization and typability with intersection types. *Notre Dame Journal of Formal Logic*, 37(1):44–52, 1996.

[24] Jean-Yves Girard. *Interprtation fonctionnelle et limination des coupures de l'arithmtique d'ordre suprieur.* Thèse d'état, Université Paris 7, June 1972.

[25] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 131–142, New York, NY, USA, 1991. ACM.

[26] Ryan Heise. `evcr`. `http://www.ryanheise.com/software/evcr/`.

[27] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23:396–450, May 2001.

[28] Lalita Jategaonkar and John Mitchell. ML with extended pattern matching and subtypes. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 198–211, New York, NY, USA, 1988. ACM.

[29] C. B. Jay. Methods as pattern-matching functions. *Foundations of Object-Oriented Languages*, page 16, 2004.

[30] Gregor Kiczales John, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. pages 220–242. Springer-Verlag, 1997.

[31] Mark P. Jones. First-class polymorphism with type inference.

[32] S. Kamin. Inheritance in smalltalk-80: a denotational definition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 80–87, New York, NY, USA, 1988. ACM.

[33] Sonya E. Keene. Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS. page 290, 1989.

[34] A. J. Kfoury. Beta-reduction as unification. In *Banach Center Publication*, pages 137–158. Springer-Verlag, 1996.

[35] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. *SIGPLAN Lisp Pointers*, VII(3):196–207, 1994.

[36] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–174, New York, NY, USA, 1999. ACM.

[37] Assaf J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3):1–70, 2004.

[38] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. Technical report, Ithaca, NY, USA, 2000.

[39] Didier Le Botlan and Didier Rémy. MLF: raising ML to the power of System F. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 27–38, New York, NY, USA, 2003. ACM.

[40] Daan Leijen. First-class labels for extensible rows. Technical Report UU-CS-2004-51, Department of Computer Science, Universiteit Utrecht, December 2004.

[41] Daan Leijen. HMF: Simple type inference for first-class polymorphism. *SIGPLAN Not*, 43(9):283–294, 2008.

[42] Daniel Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 88–98, New York, NY, USA, 1983. ACM.

[43] Brad Lushman and Gordon V. Cormack. A more direct algorithm for type inference in the rank-2 fragment of the second-order -calculus. Technical Report CS-2006-08, 2006.

[44] Henning Makholm and J. B. Wells. Type inference and principal typings for symmetric record concatenation and mixin modules. Technical report, 2005.

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

[46] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. pages 146–161. ACM Press, 1991.

[47] Benjamin C. Pierce. Bounded quantification is undecidable. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 305–315, New York, NY, USA, 1992. ACM.

[48] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[49] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, OOPSLA '94, pages 324–340, New York, NY, USA, 1994. ACM.

[50] G. Plotkin. Call-by-name, call-by-value and the -calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[51] Martin Plümicke. Typeless programming in Java 5.0 with wildcards. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ '07, pages 73–82, New York, NY, USA, 2007. ACM.

[52] Martin Plümicke and Jörg Bäuerle. Typeless programming in Java 5.0. In *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, PPPJ '06, pages 175–181, New York, NY, USA, 2006. ACM.

[53] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 567–577. Academic Press, Inc., New York, N.Y., 1980.

[54] Didier Rémy. Typing record concatenation for free. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 166–176, New York, NY, USA, 1992. ACM.

[55] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[56] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28(1-2):151 – 169, 1983.

[57] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theor. Comput. Sci.*, 59:181–209, July 1988.

[58] Martin Sulzmann. Designing Record Systems. Research Report YALEU/DCS/RR-1128, Yale University, Department of Computer Science, April 1997.

[59] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22:227–242, December 1987.

[60] Pawel Urzyczyn. Type reconstruction in F_ω. *Mathematical Structures in Computer Science*, 7:329–358, 1997.

[61] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

[62] Mitchell Wand. Complete type inference for simple objects. In *Proc. 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.

[63] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Inf. Comput.*, 93:1–15, July 1991.

[64] Mitchell Wand. Type inference for objects with instance variables and inheritance. In Carl Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 97–120. MIT Press, 1994. Originally appeared as Northeastern University College of Computer Science Technical Report NU-CCS-89-2, February, 1989.

[65] J. B. Wells. Typability is undecidable for F+eta. Tech. Rep. 96-022, Comp. Sci. Dept., Boston Univ., March 1996.

[66] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156, 1999.

[67] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.

[68] Andrew Wright. Simple imperative polymorphism. In *LISP and Symbolic Computation*, pages 343–356, 1995.

Appendix A

Additional Proofs

Lemma 3.14. *Given any E, F and K, $E\ (F\ K) = (E\ F)\ K$.*

Proof. The proof is by induction on the structure of E, F and K. This is similar to the proof in [17], except that since we do not impose equalities on types, induction is now directly possible on the structure of K, and there is no need to define a size function and prove an induction principle.

case: $E = \omega$.

$$\begin{array}{lll}
\text{LHS} & = \omega\ (F\ K) & \\
 & = \omega & \text{def 3.11} \\
 & = \omega\ K & \text{def 3.11} \\
 & = (\omega\ F)\ K & \text{def 3.11} \\
 & = \text{RHS} &
\end{array}$$

case: $E = E_1 \mathbin{\dot\cap} E_2$.

$$\begin{array}{lll}
\text{LHS} & = (E_1 \mathbin{\dot\cap} E_2)\ (F\ K) & \\
 & = E_1\ (F\ K) \mathbin{\dot\cap} E_2\ (F\ K) & \text{def 3.11} \\
 & = (E_1\ F)\ K \mathbin{\dot\cap} (E_2\ F)\ K & \text{ind.hyp. } (E_1 \text{ smaller}), \text{ ind.hyp. } (E_2 \text{ smaller}) \\
 & = (E_1\ F \mathbin{\dot\cap} E_2\ F)\ K & \text{def 3.11} \\
 & = ((E_1 \mathbin{\dot\cap} E_2)\ F)\ K & \text{def 3.11} \\
 & = \text{RHS} &
\end{array}$$

case: $E = e\ E_1$.

$$\begin{aligned}
\text{LHS} &= (e\ E_1)\ (F\ K) \\
&= e\ (E_1\ (F\ K)) && \text{def 3.11} \\
&= e\ ((E_1\ F)\ K) && \text{ind.hyp. } (E_1 \text{ smaller}) \\
&= (e\ (E_1\ F))\ K && \text{def 3.11} \\
&= ((e\ E_1)\ F)\ K && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $E = \sigma_1$.

case: $F = \omega$.

$$\begin{aligned}
\text{LHS} &= \sigma_1\ (\omega\ K) \\
&= \sigma_1\ \omega && \text{def 3.11} \\
&= \omega && \text{def 3.11} \\
&= \omega\ K && \text{def 3.11} \\
&= (\sigma_1\ \omega)\ K && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $F = F_1 \capdot F_2$.

$$\begin{aligned}
\text{LHS} &= \sigma_1\ ((F_1 \capdot F_2)\ K) \\
&= \sigma_1\ (F_1\ K \capdot F_2\ K) && \text{def 3.11} \\
&= \sigma_1\ (F_1\ K) \capdot \sigma_1\ (F_2\ K) && \text{def 3.11} \\
&= (\sigma_1\ F_1)\ K \capdot (\sigma_1\ F_2)\ K && \text{ind.hyp. } (F_1 \text{ smaller}), \text{ ind.hyp. } (F_2 \text{ smaller}) \\
&= (\sigma_1\ F_1 \capdot \sigma_1\ F_2)\ K && \text{def 3.11} \\
&= (\sigma_1\ (F_1 \capdot F_2))\ K && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $F = e\ F_1$.

case: $\sigma_1 = \boxdot$.

$$\begin{aligned}
\text{LHS} &= \boxdot\ ((e\ F_1)\ K) \\
&= (e\ F_1)\ K && \text{lem 3.13} \\
&= (\boxdot\ (e\ F_1))\ K && \text{lem 3.13} \\
&= \text{RHS}
\end{aligned}$$

case: $\sigma_1 = \sigma_1', e := E_1$.

$$\begin{aligned}
\text{LHS} &= (\sigma_1', e := E_1)\ ((e\ F_1)\ K) \\
&= (\sigma_1', e := E_1)\ (e\ (F_1\ K)) && \text{def 3.11} \\
&= E_1\ (F_1\ K) && \text{def 3.11} \\
&= (E_1\ F_1)\ K && \text{ind.hyp. } (E_1 \text{ and } F_1 \text{ smaller}) \\
&= ((\sigma_1', e := E_1)\ (e\ F_1))\ K && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $\sigma_1 = \sigma_1', X_1 := K_1$ where $X_1 \neq e$.

$$\begin{aligned}
\text{LHS} &= (\sigma_1', X_1 := K_1)\ ((e\ F_1)\ K) \\
&= (\sigma_1', X_1 := K_1)\ (e\ (F_1\ K)) && \text{def 3.11} \\
&= \sigma_1'\ (e\ (F_1\ K)) && \text{def 3.11} \\
&= \sigma_1'\ ((e\ F_1)\ K) && \text{def 3.11} \\
&= (\sigma_1'\ (e\ F_1))\ K && \text{ind.hyp. } (\sigma_1' \text{ smaller}) \\
&= ((\sigma_1', X_1 := K_1)\ (e\ F_1))\ K && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $F = \sigma_2$.

case: $K = \omega$.

$$\begin{aligned}
\text{LHS} &= \sigma_1\ (\sigma_2\ \omega) \\
&= \sigma_1\ \omega && \text{def 3.11} \\
&= \omega && \text{def 3.11} \\
&= (\sigma_1\ \sigma_2)\ \omega && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $K = K_1 \capdot K_2$.

$$\begin{aligned}
\text{LHS} &= \sigma_1\ (\sigma_2\ (K_1 \capdot K_2)) \\
&= \sigma_1\ (\sigma_2\ K_1 \capdot \sigma_2\ K_2) && \text{def 3.11} \\
&= \sigma_1\ (\sigma_2\ K_1) \capdot \sigma_1\ (\sigma_2\ K_2) && \text{def 3.11} \\
&= (\sigma_1\ \sigma_2)\ K_1 \capdot (\sigma_1\ \sigma_2)\ K_2 && \text{ind.hyp. } (K_1 \text{ smaller}), \text{ ind.hyp. } (K_2 \text{ smaller}) \\
&= (\sigma_1\ \sigma_2)\ (K_1 \capdot K_2) && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $K = e\ K'$.

case: $\sigma_2 = \boxdot$.

LHS $= \sigma_1\ (\boxdot\ (e\ K'))$

$= \sigma_1\ (e\ K')$ lem 3.13

$= (\sigma_1\ \boxdot)\ (e\ K')$ def 3.11

$=$ RHS

case: $\sigma_2 = \sigma_2', e := E_1$.

LHS $= \sigma_1\ ((\sigma_2', e := E_1)\ (e\ K'))$

$= \sigma_1\ (E_1\ K')$ def 3.11

$= (\sigma_1\ E_1)\ K'$ ind.hyp. (E_1 and K' smaller)

$= (\sigma_1\ \sigma_2', e := \sigma_1\ E_1)\ (e\ K')$ def 3.11

$= (\sigma_1\ (\sigma_2', e := E_1))\ (e\ K')$ def 3.11

$=$ RHS

case: $\sigma_2 = \sigma_2', X_2 := K_2$, where $X_2 \neq e$.

LHS $= \sigma_1\ ((\sigma_2', X_2 := K_2)\ (e\ K'))$

$= \sigma_1\ (\sigma_2'\ (e\ K'))$ def 3.11

$= (\sigma_1\ \sigma_2')\ (e\ K')$ ind.hyp. (σ_2' smaller)

$= (\sigma_1\ \sigma_2', X_2 := \sigma_1\ K_2)\ (e\ K')$ def 3.11

$= (\sigma_1\ (\sigma_2', X_2 := K_2))\ (e\ K')$ def 3.11

$=$ RHS

case: $K = \sigma_3$.

case: $\sigma_3 = \boxdot$.

LHS $= \sigma_1\ (\sigma_2\ \boxdot)$

$= \sigma_1\ \sigma_2$ def 3.11

$= (\sigma_1\ \sigma_2)\ \boxdot$ def 3.11

$=$ RHS

case: $\sigma_3 = \sigma_3', X_3 := K_3$.

$$
\begin{array}{lll}
\text{LHS} & = \sigma_1\ (\sigma_2\ (\sigma'_3, X_3 := K_3)) & \\
& = \sigma_1\ (\sigma_2\ \sigma'_3, X_3 := \sigma_2\ K_3) & \text{def 3.11} \\
& = \sigma_1\ (\sigma_2\ \sigma'_3), X_3 := \sigma_1\ (\sigma_2\ K_3) & \text{def 3.11} \\
& = (\sigma_1\ \sigma_2)\ \sigma'_3, X_3 := (\sigma_1\ \sigma_2)\ K_3 & \text{ind.hyp. } (\sigma'_3 \text{ smaller}), \text{ ind.hyp. } (K_3 \text{ smaller}) \\
& = (\sigma_1\ \sigma_2)\ (\sigma'_3, X_3 := K_3) & \text{def 3.11} \\
& = \text{RHS} &
\end{array}
$$

case: $K = \alpha[L]$.

case: $\sigma_2 = \boxdot$.

$$
\begin{array}{lll}
\text{LHS} & = \sigma_1\ (\boxdot\ \alpha[L]) & \\
& = \sigma_1\ \alpha[L] & \text{def 3.11} \\
& = (\sigma_1\ \boxdot)\ \alpha[L] & \text{def 3.11} \\
& = \text{RHS} &
\end{array}
$$

case: $\sigma_2 = \sigma'_2, \alpha[L] := \bar{T}$ where $\bar{T}\upharpoonright L$.

$$
\begin{array}{lll}
\text{LHS} & = \sigma_1\ ((\sigma'_2, \alpha[L] := \bar{T})\ \alpha[L]) & \\
& = \sigma_1\ \bar{T} & \text{def 3.11} \\
& = (\sigma_1 \sigma'_2, \alpha[L] := \sigma_1 \bar{T})\ \alpha[L] & \text{lem 5.5,conv 5.3,def 3.11} \\
& = (\sigma_1\ (\sigma'_2, \alpha[L] := \bar{T}))\ \alpha[L] & \text{def 3.11} \\
& = \text{RHS} &
\end{array}
$$

case: $\sigma_2 = \sigma'_2, X_2 := K_2$ where $X_2 \neq \alpha[L]$.

$$
\begin{array}{lll}
\text{LHS} & = \sigma_1\ ((\sigma'_2, X_2 := K_2)\ \alpha[L]) & \\
& = \sigma_1\ (\sigma'_2\ \alpha[L]) & \text{def 3.11} \\
& = (\sigma_1\ \sigma'_2)\ \alpha[L] & \text{ind.hyp. } (\sigma'_2 \text{ smaller}) \\
& = (\sigma_1\ \sigma'_2, X_2 := \sigma_1\ K_2)\ \alpha[L] & \text{def 3.11} \\
& = (\sigma_1\ (\sigma'_2, X_2 := K_2))\ \alpha[L] & \text{def 3.11} \\
& = \text{RHS} &
\end{array}
$$

case: $K = C$.

$$\begin{aligned}
\text{LHS} &= \sigma_1\ (\sigma_2\ C) & \\
&= \sigma_1\ C & \text{def 3.11} \\
&= C & \text{def 3.11} \\
&= (\sigma_1\ \sigma_2)\ C & \text{def 3.11} \\
&= \text{RHS} &
\end{aligned}$$

case: $K = T_1 \to T_2$.

$$\begin{aligned}
\text{LHS} &= \sigma_1\ (\sigma_2\ (T_1 \to T_2)) & \\
&= \sigma_1\ (\sigma_2\ T_1 \to \sigma_2\ T_2) & \text{def 3.11} \\
&= \sigma_1\ (\sigma_2\ T_1) \to \sigma_1\ (\sigma_2\ T_2) & \text{def 3.11} \\
&= (\sigma_1\ \sigma_2)\ T_1 \to (\sigma_1\ \sigma_2)\ T_2 & \text{ind.hyp. } (T_1 \text{ smaller}), \text{ ind.hyp. } (T_2 \text{ smaller}) \\
&= (\sigma_1\ \sigma_2)\ (T_1 \to T_2) & \text{def 3.11} \\
&= \text{RHS} &
\end{aligned}$$

□

Lemma 3.16. *Given any T, T' and E, if $T \equiv T'$, then $E\ T \equiv E\ T'$.*

Proof. The proof is by induction on the structure of the derivation of $T \equiv T'$, and then the structure of E. However, the cases are enumerated first by the cases of E and when $E = \sigma$, by the cases of $T \equiv T'$.

case: $E = \omega$.

$$\begin{aligned}
\text{LHS} &= \omega\ T & \\
&= \omega & \text{def 3.11} \\
&= \omega\ T' & \text{def 3.11} \\
&= \text{RHS} &
\end{aligned}$$

case: $E = E_1 \cap E_2$.

$$\begin{aligned}
\text{LHS} &= (E_1 \cap E_2)\ T & \\
&= E_1\ T \cap E_2\ T & \text{def 3.11} \\
&\equiv E_1\ T' \cap E_2\ T' & \text{ind.hyp. } (E_1 \text{ smaller}), \text{ ind.hyp. } (E_2 \text{ smaller}), \text{ def 3.7} \\
&= (E_1 \cap E_2)\ T' & \text{def 3.11} \\
&= \text{RHS} &
\end{aligned}$$

case: $E = e\ E_1$.

$$\begin{aligned}
\text{LHS} \quad &= (e\ E_1)\ T \\
&= e\ (E_1\ T) && \text{def 3.11} \\
&\equiv e\ (E_1\ T') && \text{ind.hyp. } (E_1 \text{ smaller}), \text{ def 3.7} \\
&= (e\ E_1)\ T' && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $E = \sigma$. We now consider the cases for the derivation of $T \equiv T'$.

case: $(T_1 \capdot T_2) \capdot T_3 \equiv T_1 \capdot (T_2 \capdot T_3)$.

$$\begin{aligned}
\text{LHS} \quad &= \sigma\ ((T_1 \capdot T_2) \capdot T_3) \\
&= (\sigma T_1 \capdot \sigma T_2) \capdot \sigma T_3 && \text{def 3.11} \\
&\equiv \sigma T_1 \capdot (\sigma T_2 \capdot \sigma T_3) && \text{def 3.7} \\
&= \sigma\ (T_1 \capdot (T_2 \capdot T_3)) && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $T_1 \capdot T_2 \equiv T_2 \capdot T_1$.

$$\begin{aligned}
\text{LHS} \quad &= \sigma\ (T_1 \capdot T_2) \\
&= \sigma T_1 \capdot \sigma T_2 && \text{def 3.11} \\
&\equiv \sigma T_2 \capdot \sigma T_1 && \text{def 3.7} \\
&= \sigma\ (T_2 \capdot T_1) && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $T \capdot \omega \equiv T$.

$$\begin{aligned}
\text{LHS} \quad &= \sigma\ (T \capdot \omega) \\
&= \sigma T \capdot \sigma\omega && \text{def 3.11} \\
&= \sigma T \capdot \omega && \text{def 3.11} \\
&\equiv \sigma T && \text{def 3.7} \\
&= \text{RHS}
\end{aligned}$$

case: $e\ \omega \equiv \omega$.

$$\begin{aligned}
\text{LHS} \quad &= \sigma\ (e\ \omega) \\
&= (\sigma\ e)\ \omega && \text{def 3.11} \\
&= \omega && \text{def 3.11} \\
&= \sigma\ \omega && \text{def 3.11} \\
&= \text{RHS}
\end{aligned}$$

case: $e\ (T_1 \capdot T_2) \equiv e\ T_1 \capdot e\ T_2$.

LHS $= \sigma\ (e\ (T_1 \Cap T_2))$

$= (\sigma\ e)\ (T_1 \Cap T_2)$ def 3.11

$\equiv (\sigma\ e)\ T_1 \Cap (\sigma\ e)\ T_2$ lem 3.15

$= \sigma\ (e\ T_1) \Cap \sigma\ (e\ T_2)$ def 3.11

$= \sigma\ (e\ T_1 \Cap e\ T_2)$ def 3.11

$=$ RHS

case: If $T_1 \equiv T_1'$ and $T_2 \equiv T_2'$, then $T_1 \to T_2 \equiv T_1' \to T_2'$.

LHS $= \sigma\ (T_1 \to T_2)$

$= \sigma T_1 \to \sigma T_2$ def 3.11

$\equiv \sigma T_1' \to \sigma T_2'$ ind.hyp. ($T_1 \equiv T_1'$ smaller), ind.hyp. ($T_2 \equiv T_2'$ smaller), def 3.7

$= \sigma\ (T_1' \to T_2')$ def 3.11

$=$ RHS

case: If $T_1 \equiv T_1'$ and $T_2 \equiv T_2'$, then $T_1 \Cap T_2 \equiv T_1' \Cap T_2'$.

LHS $= \sigma\ (T_1 \Cap T_2)$

$= \sigma T_1 \Cap \sigma T_2$ def 3.11

$\equiv \sigma T_1' \Cap \sigma T_2'$ ind.hyp. ($T_1 \equiv T_1'$ smaller), ind.hyp. ($T_2 \equiv T_2'$ smaller), def 3.7

$= \sigma\ (T_1' \Cap T_2')$ def 3.11

$=$ RHS

case: If $T_1 \equiv T_1'$, then $e\ T_1 \equiv e\ T_1'$.

LHS $= \sigma\ (e\ T_1)$

$= (\sigma\ e)\ T_1$ def 3.11

$\equiv (\sigma\ e)\ T_1'$ ind.hyp. ($T_1 \equiv T_1'$ smaller), def 3.7

$= \sigma\ (e\ T_1')$ def 3.11

$=$ RHS

case: $T \equiv T$.

LHS $= \sigma T$

$\equiv \sigma T$ def 3.11

$=$ RHS

case: If $T' \equiv T$ then $T \equiv T'$. This can be seen by reading the proofs for the other cases in reverse.

case: If $T \equiv T'$ and $T' \equiv T''$, then $T \equiv T''$.

$$
\begin{aligned}
\text{LHS} &= \sigma T \\
&\equiv \sigma T' && \text{ind.hyp. } (T \equiv T' \text{ smaller}) \\
&\equiv \sigma T'' && \text{ind.hyp. } (T' \equiv T'' \text{ smaller}) \\
&= \text{RHS}
\end{aligned}
$$

□

Lemma 3.27. *Given any t, T, T' and Γ, if $t : T \lhd \Gamma$ and $T \equiv T'$, then $t : T' \lhd \Gamma$.*

Proof. Let D be the derivation of $t : T \lhd \Gamma$. The proof is by induction on the structure of D. The cases proceed first by term form and then by the possible rules that may have been used to conclude $T \equiv T'$.

case: t is v. Now we consider the cases for the rule used to conclude $T \equiv T'$.

case: Axiom $(T_1 \Cap T_2) \Cap T_3 \equiv T_1 \Cap (T_2 \Cap T_3)$.

Then D ends with

$$\dfrac{\dfrac{(2)\ v : T_1 \lhd \Gamma_1 \quad (3)\ v : T_2 \lhd \Gamma_2}{v : T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2} \qquad (1)\ v : T_3 \lhd \Gamma_3}{v : (T_1 \Cap T_2) \Cap T_3 \lhd (\Gamma_1 \Cap \Gamma_2) \Cap \Gamma_3}$$

Then,

$$
\begin{array}{lll}
(4) & v : T_2 \Cap T_3 \lhd \Gamma_2 \Cap \Gamma_3 & \text{(int) with (3) and (1)} \\
\therefore & v : T_1 \Cap (T_2 \Cap T_3) \lhd \Gamma_1 \Cap (\Gamma_2 \Cap \Gamma_3) & \text{(int) with (2) and (4)}
\end{array}
$$

case: Axiom $T_1 \Cap T_2 \equiv T_2 \Cap T_1$.

Then D ends with

$$\dfrac{(1)\ v : T_1 \lhd \Gamma_1 \quad (2)\ v : T_2 \lhd \Gamma_2}{v : T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2}$$

Then,

$$v : T_2 \Cap T_1 \lhd \Gamma_2 \Cap \Gamma_1 \quad \text{(int) with (2) and (1)}$$

case: Axiom $T' \Cap \omega \equiv T'$.

Then D ends with

$$\dfrac{(1)\ v : T' \lhd \Gamma \quad v : \omega \lhd \Gamma_\omega}{v : T' \Cap \omega \lhd \Gamma}$$

Then (1) is the result.

case: Axiom $e\ \omega \equiv \omega$.

Then D ends with

$$\frac{(1)\ v : \omega \lhd \Gamma_\omega}{v : e\ \omega \lhd \Gamma_\omega}$$

Then, (1) is the result.

case: Axiom $e\ (T_1 \Cap T_2) \equiv e\ T_1 \Cap e\ T_2$.

Then D ends with

$$\cfrac{\cfrac{(1)\ v : T_1 \lhd \Gamma_1 \quad (2)\ v : T_2 \lhd \Gamma_2}{v : T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2}}{v : e\ (T_1 \Cap T_2) \lhd e\ (\Gamma_1 \Cap \Gamma_2)}$$

Then,

(3)	$v : e\ T_1 \lhd e\ \Gamma_1$	(evar) with (1)
(4)	$v : e\ T_2 \lhd e\ \Gamma_2$	(evar) with (2)
$\therefore$	$v : e\ T_1 \Cap e\ T_2 \lhd e\ (\Gamma_1 \Cap \Gamma_2)$	(int) with (3) and (4)

case: Structural congruence rule $\dfrac{(1)\ T_1 \equiv T_1' \quad (2)\ T_2 \equiv T_2'}{T_1 \to T_2 \equiv T_1' \to T_2'}$

By Lemma 3.25, there are five possible cases for D.

case: D is the following application of Rule (var):

$$\frac{}{x : T_1 \to T_2 \lhd x : \langle T_1 \to T_2 \rangle}$$

Then,

(3)	$x : T_1' \to T_2' \lhd x : \langle T_1' \to T_2' \rangle$	(var)
$\therefore$	$x : T_1' \to T_2' \lhd x : \langle T_1 \to T_2 \rangle$	def. 3.7 with (3),(1),(2)

case: D is the following application of Rule (con):

$$\frac{}{(3)\ c : T_1 \to T_2 \lhd \Gamma_\omega}\ (4)\ T_1 \to T_2 \equiv \sigma\ \mathsf{typeof}(c)$$

Then,

(5)	$T_1 \to T_2 \equiv T_1' \to T_2'$	Assumption
(6)	$T_1' \to T_2' \equiv \sigma\ \mathsf{typeof}(c)$	def. 3.7 with (4) and (5)
$\therefore$	$c : T_1' \to T_2' \lhd \Gamma_\omega$	(con) with (6)

case: D ends with the following application of Rule (abs):

$$\frac{(3)\ t_1 : T_2 \lhd \Gamma, x : \langle T_1 \rangle}{\lambda x.t_1 : T_1 \to T_2 \lhd \Gamma}$$

Then,

(4)	$t_1 : T_2' \lhd \Gamma, x : \langle T_1 \rangle$	ind.hyp with (3) and (2)
(5)	$t_1 : T_2' \lhd \Gamma, x : \langle T_1' \rangle$	def. 3.7 with (4)
	$\lambda x.t_1 : T_1' \to T_2' \lhd \Gamma$	(abs) with (5)

case: D ends with the following application of Rule (extl)

$$\frac{(4)\ t_1 : T_2 \lhd \Gamma}{(3)\ l \to t_1 \capdot v : T_1 \to T_2 \lhd \Gamma}\ (5)\ T_1 \equiv l$$

Then,

(6)	$t_1 : T_2' \lhd \Gamma$	ind.hyp. with (4) and (2)
(7)	$T_1' \equiv l$	def 3.7 with (5) and (1)
$\therefore$	$l \to t_1 \capdot v : T_1' \to T_2' \lhd \Gamma$	(extl) with (6) and (7)

case: D ends with the following application of Rule (extr)

$$\frac{(4)\ v : T_1 \to T_2 \lhd \Gamma}{(3)\ l \to t_1 \capdot v : T_1 \to T_2 \lhd \Gamma}\ (5)\ (T_1 \to T_2){::}l$$

where

(6)	$T_1 \to T_2 \equiv T_1' \to T_2'$	Assumption
(7)	$v : T_1' \to T_2' \lhd \Gamma$	ind.hyp. with (4) and (6)
(8)	$\exists \bar{T}_1.\quad T_1 \equiv \bar{T}_1$	def 5.6 with (5)
(9)	$\bar{T}_1 \upharpoonright l$	"
(10)	$T_1' \equiv \bar{T}_1$	(1),(8)
(11)	$(T_1' \to T_2')::l$	def. 5.6 with (10),(9)
$\therefore$	$l \to t_1 \Cap v : T_1' \to T_2' \lhd \Gamma$	(extr) with (7) and (11)

case: Structural congruence rule $\dfrac{(1)\ T_1 \equiv T_1' \quad (2)\ T_2 \equiv T_2'}{T_1 \Cap T_2 \equiv T_1' \Cap T_2'}$

Then D ends with the following application of Rule (int)

$$\frac{(3)\ v : T_1 \lhd \Gamma_1 \quad (4)\ v : T_2 \lhd \Gamma_2}{v : T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2}$$

Then,

(5)	$v : T_1' \lhd \Gamma_1$	ind.hyp. with (3) and (1)
(6)	$v : T_2' \lhd \Gamma_2$	ind.hyp. with (4) and (2)
$\therefore$	$v : T_1' \Cap T_2' \lhd \Gamma_1 \Cap \Gamma_2$	(int) with (5) and (6)

case: Structural congruence rule $\dfrac{(1)\ T_1 \equiv T_1'}{e\ T_1 \equiv e\ T_1'}$

Then D ends with the following application of Rule (evar)

$$\frac{(2)\ v : T_1 \lhd \Gamma_1}{v : e\ T_1 \lhd e\ \Gamma_1}$$

Then,

(3)	$v : T_1' \lhd \Gamma_1$	ind.hyp. with (2) and (1)
$\therefore$	$v : e\ T_1' \lhd e\ \Gamma_1$	(evar) with (3)

case: Equivalence rule $T \equiv T$.

Done.

case: Equivalence rule $\dfrac{T' \equiv T}{T \equiv T'}$.

For the symmetric rules that have the same meaning when reflected from left to right, these cases have already been proved. For the non-symmetric rules, we have the following cases:

case: $T_1 \Cap (T_2 \Cap T_3) \equiv (T_1 \Cap T_2) \Cap T_3$.

Similar to the reverse case.

case: $T \equiv T \Cap \omega$.

$$\begin{array}{lll} (1) & v : T \lhd \Gamma & \text{Assumption} \\ (2) & v : \omega \lhd \Gamma_\omega & \text{(omega)} \\ & v : T \Cap \omega \lhd \Gamma & \text{(int) with (1) and (2)} \end{array}$$

case: $\omega \equiv e\ \omega$.

Then D ends with the following application of Rule (omega)

$$\frac{}{(1)\ v : \omega \lhd \Gamma_\omega}$$

Then,

$v : e\ \omega \lhd \Gamma_\omega$ (evar) with (1)

case: $e\ T_1 \Cap e\ T_2 \equiv e\ (T_1 \Cap T_2)$.

Then D ends with

$$\frac{\dfrac{(1)\ v : T_1 \lhd \Gamma_1}{v : e\ T_1 \lhd e\ \Gamma_1} \quad \dfrac{(2)\ v : T_2 \lhd \Gamma_2}{v : e\ T_2 \lhd e\ \Gamma_2}}{v : e\ T_1 \Cap e\ T_2 \lhd e\ \Gamma_1 \Cap e\ \Gamma_2}$$

Then,

$$\begin{array}{lll} (3) & v : T_1 \Cap T_2 \lhd \Gamma_1 \Cap \Gamma_2 & \text{(int) with (1) and (2)} \\ \therefore & v : e\ (T_1 \Cap T_2) \lhd e\ \Gamma_1 \Cap e\ \Gamma_2 & \text{(evar) with (3)} \end{array}$$

case: $\dfrac{T \equiv T'' \quad T'' \equiv T'}{T \equiv T'}$.

This case follows automatically from the transitivity of $\Longrightarrow$ in the statement of the lemma.

case: t is $t_1\ t_2$.

Then D ends with

$$\frac{(1)\ t_1 : T_1 \to T \lhd \Gamma_1 \quad (2)\ t_2 : T_1 \lhd \Gamma_2}{t_1\ t_2 : T \lhd \Gamma_1 \Cap \Gamma_2}\ \text{(app)}$$

Then,

(3)	$T \equiv T'$	Assumption
(4)	$T_1 \to T \equiv T_1 \to T'$	def. 3.7 with (3)
(5)	$t_1 : T_1 \to T' \lhd \Gamma_1$	ind.hyp. with (1) and (4)
$\therefore$	$t_1\ t_2 : T' \lhd \Gamma_1 \Cap \Gamma_2$	(app) with (5) and (2)

□

Appendix B

Examples from the System E Inference Report

This appendix show the output of `evcr` on all of the 61 example terms from the System E Inference Report.

```
Term 01
x
: a [] <| x : a []

Term 02
\x.x
: a (b [] -> b [])

Term 03
x y
: a [] <| y : b [],  x : b [] -> a []

Term 04
x x
: a [] <| x : (b [] -> a []) ^ b []

Term 05
f x y
: a [] <| f : b [] -> c [] -> a [],  y : c [],  x : b []

Term 06
(\x.x) y
: a [] <| y : a []
```

```
Term 07
(\x.x x) y
: a [] <| y : (b [] -> a []) ^ b []

Term 08
(\h->(\x->h (x x)) (\x->h (x x))) (\f->\n->n (\v->\x->\y->y) (\x->\y->x) (\f->\x->f x)
    (\g->\x->n (f (n (\p->\s->s (p (\x->\y->y)) (\f->\x->f (p (\x->\y->y) f x)))
    (\s->s (\f->\x->x) (\f->\x->x)) (\x->\y->x)) g) x)) (\f->\x->f (f x))
: Fails to terminate

Term 09
(\z.\x.x x) z (\y.y)
: a (b [] -> b [])

Term 10
(\z.\x.x x) y (\z.z)
: a (b [] -> b [])

Term 11
(\two.\k.two two two k) (\f.\x.f (f x)) (\v.\w.v)
: a (b c d e f g h i j k l m n o p q r [] -> b (w -> c (w -> d (w -> e (w -> f (w ->
    g (w -> h (w -> i (w -> j (w -> k (w -> l (w -> m (w -> n (w -> o (w -> p (w ->
    q (w -> r []))))))))))))))))))

Term 12
(\two.\k.two two k) (\f.\x.f (f x)) (\v.\w.v)
: a (b c d e f [] -> b (w -> c (w -> d (w -> e (w -> f [])))))

Term 13
(\f.\x.f (f x)) (\f.\x.f (f x)) (\v.\w.v)
: a (b c d e f [] -> b (w -> c (w -> d (w -> e (w -> f [])))))

Term 14
(\z.\x.x x) (\y.y)
: a (((b [] -> c []) ^ b []) -> c [])

Term 15
(\z.\x.x) (\z.z)
: a (b [] -> b [])

Term 16
(\z.\x.x) z
: a (b [] -> b [])

Term 17
(\y.(\x.w (x y) z) a) b
: a [] <| w : c [] -> d [] -> a [],  b : b [],  a : b [] -> c [],  z : d []

Term 18
(\y.\x.x) (\y.y)
: a (b [] -> b [])
```

```
Term 19
(\y.\x.x) z
: a (b [] -> b [])

Term 20
(\x.x x x x x x x x x x x) (\y.y)
: a (b [] -> b [])

Term 21
(\x.x x x x x x x x) (\y.y)
: a (b [] -> b [])

Term 22
(\x.x x x x x x x) (\y.y)
: a (b [] -> b [])

Term 23
(\x.x x x x x x) (\y.y)
: a (b [] -> b [])

Term 24
(\x.x x x x xx) (\y.y)
: a [] <| xx : a []

Term 25
(\x.x x x x x) (\y.y)
: a (b [] -> b [])

Term 26
(\x.z (x (\f.\u.f u)) (x (\v.\g.g v))) (\y.y y y)
: a [] <| z : b (c (d [] -> e []) -> c (d [] -> e [])) -> f ((g (h i [] ->
    h ((i [] -> j []) -> j [])) -> k []) -> k []) -> a []

Term 27
(\x.x x) (\y.y z y)
: a [] <| z : ((b [] -> c [] -> d (((e [] -> f [] -> g []) ^ f []) -> g [])
    -> a []) ^ c []) ^ b [] ^ d e []

Term 28
(\x->x x) (\x->x x)
: Fails to terminate

Term 29
(\x.x w) (\y.y z y)
: a [] <| w : (b [] -> c [] -> a []) ^ c [],  z : b []

Term 30
(\x.x I) (\y.A)
: a [] <| A : a []

Term 31
(\x.w x) (\y.y z y)
: a [] <| w : b (((c [] -> d [] -> e []) ^ d []) -> e []) -> a [],  z : b c []
```

```
Term 32
(\x.\y.y) z
: a (b [] -> b [])

Term 33
(\x.\y.x) (\y.y y)
: a (w -> b (((c [] -> d []) ^ c []) -> d []))

Term 34
(\x.\w.x) y
: a (w -> b []) <| y : a b []

Term 35
(\x.x) (f y)
: a [] <| f : b [] -> a [],  y : b []

Term 36
(\x.x) (\y.y y)
: a (((b [] -> c []) ^ b []) -> c [])

Term 37
(\x.x) (\y.y)
: a (b [] -> b [])

Term 38
(\w.\x.(\z.x) w) y
: a (b [] -> b [])

Term 39
(\w.\x.x) y
: a (b [] -> b [])

Term 40
(\x.x) (\x.x) (\x.x) (\x.x) (\x.x) (\y.y y)
: a (((b [] -> c []) ^ b []) -> c [])

Term 41
(\y.\x.x (f (x x)) x y) (\y.y y) (\x.x)
: a [] <| f : b (c [] -> c []) -> d (e [] -> e []) -> f (((g [] -> h [])
    ^ g []) -> h []) -> a []

Term 42
(\y->\x->x (f (x x)) x y) (\x->x) (\y->y y)
: Fails to terminate

Term 43
(\y.\x.x (f (x x)) x) (\y.y y) (\x.x)
: a [] <| f : b (c [] -> c []) -> d (e [] -> e []) -> a []

Term 44
(\x.x) y z
: a [] <| z : b [],  y : b [] -> a []
```

```
Term 45
(\x.x x x x) (\y.y)
: a (b [] -> b [])

Term 46
(\x.x x x) (\y.y)
: a (b [] -> b [])

Term 47
(\x.x x a) (\y.y)
: a [] <| a : a []

Term 48
(\x.a x x) ((\y.b (c y)) (d e))
: a [] <| d : f [] -> e [],  e : f [],  b : d [] -> (b [] ^ c []),
    c : e [] -> d [],  a : b [] -> c [] -> a []

Term 49
(\x.x (x y)) ((\z.z) (\x.a x))
: a [] <| a : (b [] -> a []) ^ (c [] -> b []),  y : c []

Term 50
(\x.x y) z
: a [] <| z : b [] -> a [],  y : b []

Term 51
(\x.x x) z
: a [] <| z : (b [] -> a []) ^ b []

Term 52
(\x.f x) y
: a [] <| f : b [] -> a [],  y : b []

Term 53
(\x.x) y
: a [] <| y : a []

Term 54
(\f.\x.f (f x)) (\x.x)
: a (b [] -> b [])

Term 55
(\f.\x.f (f x)) (\g.\y.g (g y))
: a (b (((c [] -> d []) ^ (e [] -> c [])) ^ (f [] -> e []) ^ (g [] -> f []))
    -> b (g [] -> d []))

Term 56
(\f.\x.f (f x)) (\g.\x.g (g x))
: a (b (((c [] -> d []) ^ (e [] -> c [])) ^ (f [] -> e []) ^ (g [] -> f []))
    -> b (g [] -> d []))
```

```
Term 57
f (g (h x))
: a [] <| f : b [] -> a [],  g : c [] -> b [],  h : d [] -> c [],  x : d []

Term 58
(\x.x x) _if _false (\x.x) (\f.\x.f (f x))
: a [] <| _if : (c [] -> b [] -> d (e [] -> e []) -> f (g ((h [] -> i [])
    ^ (j [] -> h [])) -> g (j [] -> i [])) -> a []) ^ c [],  _false : b []

Term 59
_if _true (\x.x) (\w.y)
: a [] <| _if : b [] -> c (d [] -> d []) -> e (w -> f []) -> a [],  _true : b [],  y : e f []

Term 60
(\x.x x) (_if _false (\x.x) (\f.\x.f (f x)))
: a [] <| _if : b [] -> c (d [] -> d []) -> e (f ((g [] -> h []) ^ (i [] -> g []))
    -> f (i [] -> h [])) -> ((j [] -> a []) ^ j []),  _false : b []

Term 61
(\x.x x) (\y.y)
: a (b [] -> b [])
```

Appendix C

The opus Unification Algorithm

This appendix provides a formal definition of the opus algorithm [7] which succeeds in finding a covering unifier set for any constraint that has a solution. This presentation of opus will use the notation below:

- $X_1 := K_1, \ldots X_n := K_n$ abbreviates $\boxdot, X_1 := K_1, \ldots X_n := K_n$.
- e/σ abbreviates $e := e\ \sigma$ and applies substitution σ under namespace e.
- $\vec{e}$ abbreviates a sequence of E-variables $e_1\ e_2\ \ldots\ e_n$.
- $\vec{e}/\sigma$ abbreviates $e_1/\ldots/e_n/\sigma$ and applies substitution σ under namespace $\vec{e}$.
- Metavariable Σ ranges over sets of substitutions.
- Metavariable τ ranges over typings.

Figure C.1 presents the complete opus algorithm. Note that opus is defined in such a way that no equalities are assumed to have been imposed on types. So, for example, T is not equal to $T \Cap \omega$. When it is beneficial for this particular equality to hold, an explicit equivalence relation $\overset{\text{unit}}{\equiv}$ is used instead, although $\overset{\text{unit}}{\equiv}$ only takes the place of the $\Cap$-unit equality rule (and its reflection) and not any of the other equality rules.

A covering unifier set is computed by taking all possible reduction paths under $\underset{\text{c}*}{\Longrightarrow}$ from a given Δ to solved constraint sets. That is, a covering unifier set for Δ is the set $\{\sigma \mid \Delta \overset{\sigma}{\underset{\text{c}*}{\Longrightarrow}} \Delta', \mathsf{solved}(\Delta')\}$. A constraint reduction $\Delta \overset{\sigma}{\underset{\text{c}*}{\Longrightarrow}} \Delta'$ happens through a series of reduction steps of the form $\Delta_1 \overset{\sigma_1}{\underset{\text{c}}{\Longrightarrow}} \Delta_2$. In

Preliminary definitions

- A *constraint* δ is an *unordered* pair of types written $S \doteq T$, where $\mathsf{solved}(S \doteq T)$ iff $S = T$.
- A *constraint set* Δ is a set of constraints, where $\mathsf{solved}(\Delta)$ iff $\forall \delta \in \Delta.\ \mathsf{solved}(\delta)$.
- An *expansion join* $E_1 \sqcup E_2$ is defined by:

$$\begin{aligned} \omega \sqcup \omega &= \omega \\ (E_1 \mathbin{\dot\cap} F_1) \sqcup (E_2 \mathbin{\dot\cap} F_2) &= (E_1 \sqcup E_2) \mathbin{\dot\cap} (F_1 \sqcup F_2) \\ e\ E \sqcup e\ F &= e\ (E \sqcup F) \\ \sigma_1 \sqcup \sigma_2 &= \sigma \text{ if for all } X,\ \sigma X \text{ is defined by one of the following cases:} \end{aligned}$$

$$\sigma\ X = \begin{cases} \sigma_1(X) & \text{if } \neg\mathsf{supported}(\sigma_2, X) \text{ or } \sigma_1(X) = \sigma_2(X) \\ \sigma_2(X) & \text{if } \neg\mathsf{supported}(\sigma_1, X) \\ \sigma_1(e) \sqcup \sigma_2(e) & \text{if } \mathsf{supported}(\sigma_1, e), \mathsf{supported}(\sigma_2, e) \text{ and } X = e \end{cases}$$

where: $\mathsf{supported}(\sigma, e)$ iff $\sigma\ e \neq e\ \boxdot$ and $\mathsf{supported}(\sigma, \alpha)$ iff $\sigma\ \alpha \neq \alpha$

- The *expansion restriction* $E|_T$ results in E' iff $E'\ T{=}E\ T$, and $\forall F.\ F\ T{=}E\ T \implies \exists G.\ F{=}E' \sqcup G$.
- $\overset{\text{unit}}{\equiv}$ is the equivalence relation formed by taking the compatible closure of $T \mathbin{\dot\cap} \omega \overset{\text{unit}}{\equiv} T$ and $\omega \mathbin{\dot\cap} T \overset{\text{unit}}{\equiv} T$.
- $S \underline{\dot\cap} T$ indicates the intersection type $S \mathbin{\dot\cap} T$ when $S \overset{\text{unit}}{\not\equiv} \omega$ and $T \overset{\text{unit}}{\not\equiv} \omega$.

Constraint factorisation

$$\begin{array}{lll} \Delta & \xrightarrow{\mathsf{rfactor}} & \bigcup\{\Delta' \mid \delta \in \Delta, \delta \xrightarrow{\mathsf{rfactor}} \Delta'\} \\ \delta & \xrightarrow{\mathsf{rfactor}} & \Delta \text{ if } \delta \overset{\text{unit}}{\equiv} \vec{e}\ (S_1 \underline{\dot\cap} T_1 \doteq S_2 \mathbin{\dot\cap} T_2) \text{ and } \{\vec{e}\ (S_1 \doteq S_2), \vec{e}\ (T_1 \doteq T_2)\} \xrightarrow{\mathsf{rfactor}} \Delta \\ \delta & \xrightarrow{\mathsf{rfactor}} & \Delta \text{ if } \delta \overset{\text{unit}}{\equiv} \vec{e}\ (S_1 {\to} T_1 \doteq S_2 {\to} T_2) \text{ and } \{\vec{e}\ (S_1 \doteq S_2), \vec{e}\ (T_1 \doteq T_2)\} \xrightarrow{\mathsf{rfactor}} \Delta \\ \delta & \xrightarrow{\mathsf{rfactor}} & \{\delta\} \text{ if } \delta \neq S_1 {\to} T_1 \doteq S_2 {\to} T_2 \text{ and } \delta \neq S_1 \underline{\dot\cap} T_1 \doteq S_2 \underline{\dot\cap} T_2 \end{array}$$

Constraint part unification (where σ_Δ freshly renames the variables in Δ)

$$\begin{array}{rlll} \alpha \doteq T & \xrightarrow{\mathsf{opus}}_\Delta & \{\sigma \mid \sigma = (\alpha := T), \sigma\ \alpha = \sigma\ T\} & \text{(C T-unify)} \\ e\ T \doteq T_1 {\to} T_2 & \xrightarrow{\mathsf{opus}}_\Delta & \{(e := \sigma_\Delta)\} & \text{(C EA-unify)} \\ e\ T \doteq \omega & \xrightarrow{\mathsf{opus}}_\Delta & \{(e := \sigma_\Delta), (e := \omega)\} & \text{(C EO-unify)} \\ e\ T \doteq T_1 \mathbin{\dot\cap} T_2 & \xrightarrow{\mathsf{opus}}_\Delta & \{(e := \sigma_\Delta), (e := e_1 \boxdot \mathbin{\dot\cap} e_2 \boxdot) \mid e_1, e_2\ \mathsf{fresh}\} & \text{(C EI-unify)} \\ e\ T_1 \doteq e\ T_2 & \xrightarrow{\mathsf{opus}}_\Delta & \{(e := e(e_1\sigma_1 \mathbin{\dot\cap} \ldots \mathbin{\dot\cap} e_n\sigma_n)) \mid e_1 \ldots e_n\ \mathsf{fresh}, (T_1 {\doteq} T_2) \xrightarrow{\mathsf{opus}}_\Delta \Sigma \cup \{\sigma_i\}_{i=1}^n\} & \text{(C E-unify)} \\ e_1\ T_1 \doteq e_2\ T_2 & \xrightarrow{\mathsf{opus}}_\Delta & \{(e_1 := e_5\ (E_L\ e_3 \boxdot \mathbin{\dot\cap} E_R|_{T_1}), e_2 := e_5\ (E_L|_{T_2} \mathbin{\dot\cap} E_R\ e_4 \boxdot)) & \text{(C EE-unify)} \\ & & \mid e_3, e_4, e_5, e_1^L \ldots e_p^L, e_1^R \ldots e_q^R\ \mathsf{fresh}, e_1 \neq e_2, & \\ & & \quad e_3\ T_1 \doteq T_2 \xrightarrow{\mathsf{opus}}_\Delta \Sigma_1 \cup \{\sigma_i^L\}_{i=1}^p, \qquad E_L = e_1^L\ \sigma_1^L \mathbin{\dot\cap} \ldots e_p^L\ \sigma_p^L, & \\ & & \quad T_1 \doteq e_4\ T_2 \xrightarrow{\mathsf{opus}}_\Delta \Sigma_2 \cup \{\sigma_i^R\}_{i=1}^q, \qquad E_R = e_1^R\ \sigma_1^R \mathbin{\dot\cap} \ldots e_q^R\ \sigma_q^R\} & \end{array}$$

Constraint reduction

$$\Delta \overset{\sigma}{\underset{\text{c}}{\Longrightarrow}} \Delta_2 \text{ if } \Delta \xrightarrow{\mathsf{rfactor}_\beta} \Delta_1 \text{ and } \delta \in \Delta_1 \text{ and not } \mathsf{solved}(\delta) \text{ and } \delta \xrightarrow{\mathsf{opus}}_\Delta \sigma \text{ and } \sigma\Delta_1 \xrightarrow{\mathsf{rfactor}_\beta} \Delta_2$$

$$\Delta \overset{\boxdot}{\underset{\text{c}*}{\Longrightarrow}} \Delta$$

$$\Delta_1 \overset{\sigma_2\sigma_1}{\underset{\text{c}*}{\Longrightarrow}} \Delta_3 \text{ if } \Delta_1 \overset{\sigma_1}{\underset{\text{c}}{\Longrightarrow}} \Delta_2 \text{ and } \Delta_2 \overset{\sigma_2}{\underset{\text{c}*}{\Longrightarrow}} \Delta_3$$

Figure C.1: The opus algorithm

each step, the constraint set is factored with $\xrightarrow{\mathsf{rfactor}}$, one step of unification is performed with $\xrightarrow{\mathsf{opus}}$, and the constraint set is once again factored.

The relation $\xrightarrow{\mathsf{rfactor}}$ is non-deterministic. This non-determinism is introduced through the use of the $\overset{\text{unit}}{\equiv}$ relation and its purpose is to handle the unification of an intersection type with a type that cannot become an intersection type via substitution (i.e. a simple type or ω). For example, given the constraint $S \sqcap T \doteq U_1 \to U_2$, there are potentially two alternative ways for it to be solved. One way is to interpret the constraint as $S \sqcap T \doteq (U_1 \to U_2) \sqcap \omega$ and the other way is to interpret the constraint as $S \sqcap T \doteq \omega \sqcap (U_1 \to U_2)$.

The relation $\xrightarrow{\mathsf{opus}}$ is then used to perform a single step of unification. Each rule operates only on the outer variables of the types being unified, and in many cases, parallel reduction paths are generated:

- (C T-unify) unifies a type variable with a type and performs an occurs check. This rule also applies when the type variable is on the right since opus interprets constraints as *unordered* pairs.
- (C EA-unify) eliminates E-variable e, using a renaming to avoid E-variable capture.
- (C EO-unify) introduces two possible paths for unification. One is to eliminate e using a renaming to avoid E-variable capture, and the other is to simply substitute the ω expansion for e.
- (C EI-unify) introduces two possible paths. One is again the elimination of e, and the other is to substitute an intersection expansion for e to make the left side the same shape as the right side.
- (C E-unify) substitutes for e an intersection of unifiers for the inner constraint $T_1 \doteq T_2$, with each component wrapped in a distinct E-variable so that they can be individually selected.
- (C EE-unify) covers all possibilities for two expansion applications with different E-variables by inductively applying opus′ to the constraints $e_3\ T_1 \doteq T_2$ and $T_1 \doteq e_4\ T_2$. All of the results are wrapped in distinct E-variables so that they can be individually selected.

C.1 Adapting opus to System $\mathrm{E}^{\mathrm{vcr}}$

The opus algorithm is easily adapted to System $\mathrm{E}^{\mathrm{vcr}}$. First, the new syntactic categories must be included in the type language:

$$\bar{S}, \bar{T}, \bar{U} ::= \ldots \mid C \mid \alpha[] \qquad\qquad \textsf{simple types}$$

With the introduction of constrained simple type variables, the regular type variables can be removed and substitutions should follow Convention 5.3. The new types can be handled by updating just two rules of opus:

$$\begin{aligned} \alpha[L] \doteq \bar{T} &\quad \overrightarrow{\textsf{opus}}_{\Delta} \quad \{\textsf{hunify}(\alpha[L] \doteq \bar{T})\} && \text{(C T-unify)} \\ e\ T \doteq \bar{T} &\quad \overrightarrow{\textsf{opus}}_{\Delta} \quad \{(e := \sigma_{\Delta})\} && \text{(C EA-unify)} \end{aligned}$$

The first rule invokes hunify which was established by Lemma 5.13 to find principal unifiers. As such, it is expected that opus should continue to find covering unifier sets with the amendment to this rule.

The second rule generalises the original (C EA-unify) rule to support not only function types, but all simple types (including type constants and constrained simple type variables). The proof that Rule (C EA-unify) leads to a covering unifier set is given in case 3 of Lemma 2 in the opus paper [7]. The logic of this proof continues to hold if "simple types" are mentioned in place of "function types".