LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

An Incremental Type Inference System for the Programming Language Id

MIT / LCS / TR-488

November 1990

Shail Aditya Gupta

Originally published as Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1990.

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

An Incremental Type Inference System for the Programming Language Id

Shail Aditya Gupta

MIT / LCS / TR-488
November 1990

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

An Incremental Type Inference System for the Programming Language Id

Shail Aditya Gupta

Technical Report MIT / LCS / TR-488
November 1990

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

Modern computing environments strive to be robust and reliable, and at the same time, aim at providing enough flexibility to an interactive user to edit, debug, and test programs easily and efficiently. Strongly typed languages satisfactorily meet the former goal by guaranteeing that "type-consistent" programs will not incur run-time type-errors. But most programming environments for such languages have to sacrifice the flexibility of interactive and incremental program development in order to achieve this consistency over the whole program. The problem is further complicated by the presence of polymorphism in many of these languages, where the definition of "type-consistency" is one of inclusion, rather than equality.

In this thesis, we address the latter goal of providing an interactive and incremental program development environment in context of Id, which is a polymorphic, strongly typed, incrementally compiled, parallel programming language, developed at the Laboratory for Computer Science, MIT. Id's typing mechanism is based on the Hindley/Milner type inference system. We describe modifications and extensions to its inference algorithm to allow the building and testing of programs in an incremental, definition by definition basis, and in arbitrary order. We also prove the correctness (soundness and completeness) of our incremental typing scheme with respect to the original type system.

We also examine the problem of systematically identifying overloaded identifiers in a program using type information, and translating them into efficient actual code. Finally, we describe the problems encountered while typing non-functional, polymorphic objects in our type system and discuss some possible solutions.

Key Words and Phrases: Hindley/Milner Type System, Id, Incremental Compilation, Incremental Type Inference, I-Structures, Overloading, Polymorphism, Polymorphic References, Static Semantics, Type Checking, Type Inference.

Acknowledgements

I am most grateful to my thesis advisor, Rishiyur S. Nikhil, for his clear guidance and encouragement. He was always ready to listen to my ideas and make sense out of them and help me understand them better. He pulled me through three attempts at the proof of the correctness theorem, the first two ending in dismal failures, with patience, a clear sense of direction, and an unshaken confidence in my abilities; I can't thank him enough for that. His sharp wit and sense of humor provided an exhilarating atmosphere to work in.

I am also grateful to Arvind, the leader of the Computation Structures Group, for his amazing grasp of the high-level ideas and showing his interest in me when I first came to MIT in 1987 as a fresh graduate. I am priviledged to be a part of his group, which is truely one of the most talented group of individuals I have ever met.

I thank the members of my OQE Committee consisting of Albert Meyer, Fernando Corbato, and Robert Kennedy, for their constructive criticism of my thesis presentation. Their concerns and suggestions have helped me organize this thesis in a better way.

I heartily thank all the members of the Computation Structures Group, both past and present, for their continuing help and support, making it all feel like a big family. I must especially mention James Hicks, who helped me with the Id Compiler and patiently resolved my innumerable queries; Andy Boughton, who kept all the machines alive and without whom this thesis would currently be in the bit-bucket; and Paul Johnson, Suresh Jagannathan, Madhu Sharma, and Zena Ariola, for their delightful friendship and support.

I also thank all my friends at MIT, and from IIT, Delhi, for their enjoyable company and memorable experiences, creating welcome diversions from work and making these past few years one of the most pleasurable moments of my life.

Finally, I am grateful beyond words for the love and affection bestowed upon me from my family. The encouragement and moral support I received from my parents, brothers, and sisters is beyond measure. They instilled hope, confidence, and perseverance in me that helped me endure through difficult times.

[0]Funding for this work has been provided in part by the Advanced Research Projects Agency of the Department of Defence under the Office of Naval Research contract N00014-84-K-0099.

To my parents,

Vidyaratna and Kusum.

Contents

List of Figures

Chapter 1

Introduction

In recent years, a desire for robust, reliable, and efficient software systems, that are also easy to develop and maintain, has led to some rethinking in the programming language research community. Efforts have been underway to design and implement programming languages that guarantee reliability and robustness of program code, and at the same time, do not compromise efficiency of execution. Strongly typed[1] languages offer a nice high-level programming paradigm which allows procedure and data abstraction, modularization, and means to guarantee code reliability: "type-consistent" programs written in such languages are guaranteed not to incur any run-time type-errors. Additional features, such as type polymorphism[2] and static type inferencing[3], further add to the expressibility of these languages and their ease of programming.

On the other hand, other research efforts have concentrated on techniques to provide interactive, user-friendly programming environments that support incremental program development and maintenance. Incremental and interactive compilation, version control, and support for editing, testing, and debugging programs fall under this category. But there is an apparent conflict between the incrementality offered by these environments and the need to maintain consistent compile-time information over the entire program by strongly typed languages. Therefore, most programming environments for such languages are biased towards one or the other goal. Highly flexible and interactive environments (*e.g.*, LISP) either do not enforce type-consistency over complete programs, or enforce it dynamically and pay a run-time price, while environments for statically typed languages (*e.g.*, PASCAL) generally impose restrictions on user interaction and do not support incremental development and maintenance of their programs.

[1]Languages in which all expressions are guaranteed to be type consistent.

[2]Permitting the use of the same program code under multiple type contexts.

[3]Inferring the type of every fragment of a program at compile time with very little or no help from the programmer.

Some recent efforts in the context of strongly typed functional languages such as ML [24, 25, 41, 42] and Miranda[4] [60, 61] have tried to come up with a workable solution for the interactive user, but their approach is not entirely satisfactory. ML, in most of its implementations [2, 3, 51, 24, 10], has an interactive session to which definitions can be incrementally added, but it still requires a complete program to be specified bottom-up before any of its parts can be tested. Miranda, on the other hand, only permits complete files (also called *scripts*) to be interactively manipulated, though definitions within the files are allowed to be in arbitrary order. It is clear that there is still a need for integrated, interactive computing environments that combine the flexibillity and incrementality required for smooth program development and maintenance with the reliability and robustness offered by strongly typed languages.

Id is a high-level language proposed by the Computation Structures Group at the Laboratory for Computer Science, MIT, for fine-grain dataflow parallel computation [48, 49]. It has a large purely functional subset with a few non-functional and non-deterministic extensions catering to areas where functional languages are either inadequate or are inefficient. It is strongly typed and employs the Hindley/Milner polymorphic type system [26, 40] to perform type inferencing. It supports higher-order functions, algebraic types, abstract data types, and overloading of identifiers, and in some of these respects it is similar to one or more of its contemporaries in the realm of functional langauges, namely, ML, Hope [14], Miranda, Haskell [27] etc. But the programming environment for Id is different from that of these other languages in the following important ways.

1. The Id compiler [59] is incremental in nature. Each top-level phrase, which may be a constant binding, a function, a type definition, or a type annotation, is treated as a single individual compilation unit.

2. The compiler accepts input interactively from an editor or in batch-mode from a file. In either case, the top-level definitions do not have to be processed in any particular order. The user may edit and compile definitions interactively in arbitrary order.

3. The compiler is smoothly integrated with the editor and an emulator for the Tagged-token Dataflow Architecture [5, 7] into "Id-World" [50], a friendly, interactive, parallel programming environment. Each top-level compilation unit may be compiled, loaded, and linked with other similar units and then executed *via* commands given to the interactive

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

emulator.

4. The compiler incrementally generates enough persistent global information in order to maintain consistency across compilations of several pieces of the same program.

5. It is even possible to test and debug incomplete Id programs some of whose parts may not yet be defined. Unknown definitions, if invoked, simply cause a run-time trap.

The purpose of this research is to design and develop a typing mechanism for Id based on the Hindley/Milner type system that fits its incremental programming environment. In particular, we have the following design goals in mind.

1. The type system should enforce a strongly typed, polymorphic, and high-level programming paradigm, without requiring the users to provide explicit type declarations in their programs.

2. The typing mechanism must maintain type-consistency among all the top-level compilation units and be able to generate and maintain this information incrementally, in view of the overall incremental design philosophy.

3. The type system is required to provide support for resolving and compiling overloaded identifier definitions and blend it with its existing system of polymorphism (a la Wadler and Blott [63]).

4. The type system must also provide support for sound type inference of polymorphic, non-functional, referential data-structures (a la Tofte [57]).

1.1 Thesis Outline

The outline of this thesis is as follows.

In chapter 2 we carefully examine the problem of polymorphic type inference for a functional language in the light of the Hindley/Milner type system [16, 20] and establish facts and notation used in the rest of the thesis. We describe the basic Hindley/Milner type inference system using an abstract syntax. This forms the basis for extension and reasoning for the subsequent chapters. We also describe the relationship between the abstract syntax and the actual syntax of Id as a means of providing readable examples throughout the rest of the thesis.

Chapter 3 describes our approach to incremental type inferencing and the algorithm that maintains type consistency over the whole program. First, we describe a general book-keeping mechanism to record, maintain, and verify interesting compilation properties in an incremental fashion. Our scheme is oriented towards definition level incrementality–individual, top-level definitions can be independently edited, compiled and finally loaded into the run-time system, while interprocedural consistency is maintained using the properties recorded by the compiler.

Then, we describe an application of the general mechanism, specifically to maintain type information incrementally. We show the necessary modifications and extensions to the basic Hindley/Milner type inferencing algorithm described in chapter 2 for this purpose. We also provide a correctness proof of our algorithm for well-typed programs, which establishes an exact correspondence between the types inferred by our algorithm and those inferred by the basic system. This is the major result of this thesis and to the author's knowledge, this is the first time such a proof has appeared in the literature.

In chapter 4, we discuss the issues involved in overloading of identifiers. The original proposal as adopted in the language Haskell [27] defines type classes and translates overloading into parameterized function calls. Our system[5] is a slight simplification of that proposal. Every operator has its own class which makes it easier to allow either parameterization or specialization on demand in order to make the program more efficient. We also discuss the extensions necessary to incorporate overloading into our basic type inferencing system.

In chapter 5, we extend the basic type inferencing mechanism for functional programs to incorporate non-functional language constructs. Specifically, the problem is to correctly type-check a polymorphic reference to a shared assignable location. The problem has been examined before in context of ML in [24] and semantically analysed recently by Tofte in [57]. Our proposal, developed independently, is a slight extension of the latter. We not only type-check polymorphic references, but also allow the programmer to delineate functional and non-functional boundaries in the program, so that the compiler is able to perform functional optimizations in most parts of the program.

Finally, in chapter 6, we present a summary of the results and conclude with future directions for research.

[5]A preliminary version of the system described in this thesis was presented by Nikhil in [47] for internal circulation.

Chapter 2

Basic Type Inferencing

The classical notion of **Type** is that of a set containing semantically similar values. A **Type error** is, to a first approximation, a situation where a given object is not found to be in its expected set of semantic values, or in other words, the object fails to have the expected type. The goal of type checking is to ensure that such unexpected surprises do not crop up while executing a program. For example, consider the following LISP program to find the sum of all the elements of a given list:

```
(defun reduce (l)
   (if (null l)
       0
       (+ (first l) (reduce (rest l)))))
```

The `reduce` function expects a list of numbers as input and adds them up. What will happen if it is not supplied with a list of numbers?

(reduce '(1 2 3)) $\Longrightarrow$ 6
(reduce '("1" "2" "3")) $\Longrightarrow$ *error!*

The second case above fails to execute and produces a runtime type error because the "+" operator expected a number but was supplied a string instead. The error could have been caught during compilation of the second form because it was already known that `reduce` expects a list of numbers and not a list of strings[1].

The design of a programming language can go a long way towards achieving this goal of detecting runtime type errors. Languages in which the type of every expression can be determined by static program analysis are called **Statically Typed** languages. This may be

[1]It may be argued that LISP, in fact, provides type-checking, although at runtime. But we will assume that this is only a mechanism for trapping unnoticed errors. The goal of type-checking should be to forewarn the user of any possibility of running into such errors at runtime.

too strong a condition to impose, so many languages are **Strongly Typed**, which means that every expression in that language is type consistent even though its actual type cannot be found out by static analysis.

Id is a strongly typed programming language. Furthermore, it supports type inferencing and polymorphism. We motivate these features below and present a basic type inferencing system for Id. An excellent introduction and survey of these concepts is found in [17].

We assume no previous knowledge on the part of the reader regarding the basic concepts of type inferencing, polymorphism, Hindley/Milner type system, its associated theoretical basis or the Id language. The informed reader is free to skip the introductory sections 2.1 and 2.2, refer back to section 2.3 for notations, definitions and results that have appeared in literature before, and skim through sections 2.4 and 2.5 for a discussion of Id syntax and the Hindley/Milner type system in general.

2.1 Type Inferencing

The process of **Type Inferencing** involves automatically inferring the type of every subexpression in a given program without explicit type declarations from the programmer. **Type checking**, on the other hand, refers to the task of simply checking that the type of an expression in its declaration is consistent with its use. Since Id supports type inferencing, we do not make a distinction between these terms later in the thesis and will always mean "type inferencing" when referring to either term. But for now, we highlight their differences and the advantage in doing type inferencing rather than type checking by means of an example.

As an aside, we will not attempt to give a complete description of Id language syntax and semantics, though we do present the abstract syntax of the Id Kernel language at the end of this chapter. For a concrete treatment of the Id syntax, the reader is referred to [46, 48]. A preliminary discussion of Id dynamic semantics is found in [8], while a more recent and revised version appears in [4]. We will present examples from Id, explaining their semantics informally. For the most part, the meaning should be fairly clear and we will rely on the reader's intuition in this matter.

Consider the two programs in figure 2.1. In (a) it shows a PASCAL function to compute the square of an integer and in (b) a similar function written in Id. The figure also shows how these definitions may be used subsequently in the same program. A function declaration in Id takes

<pre>program example1(input,output);
var
  x,xsq:integer;
function square(x:integer):integer;
begin
  square = x * x;
end;
begin
  ...
  xsq := square(x);
  ...
end.</pre>
	<pre>def square x = x * x;
	...
	xsq = square x;</pre>

(a): PASCAL code.	(b): Id code.

Figure 2.1: A code comparison of a squaring function written in PASCAL and in Id.

the form of an equation preceded by a `def` keyword. The left hand side of the equation specifies the function identifier and formal parameters, while the right hand side specifies the body of the function written as an Id expression, computing the value of the function. A simple variable binding is expressed as an equation as well. Function application in Id is by juxtaposition (as in `square x`) and it associates to the left.

The interesting difference between the two programs is the necessary presence of type declarations in one and their absence in the other. While the programmer has to supply full type declarations for every identifier appearing in the PASCAL program, the same information can be inferred from context in the Id program. The compiler knows the type of the binary operator `*` and it can infer that the type of its operand `x` as well as its result must be integer, hence the function `square` must take an integer and produce an integer[2]. This is precisely the information explicitly declared in the PASCAL program.

The inference mechanism is governed by the unification algorithm of Robinson [54]. The known type of an identifier or an operator is unified with the type expected in the context where it is used. This may either result in a failure if the types do not match, which means we have detected a type violation, or it may produce a unifying substitution, which refines the type of the context. Thus, we can infer the unknown types in the context.

The obvious advantage of type inferencing is freeing the programmer from writing redundant type declarations at the cost of increasing the task of the compiler slightly. We may also envision

[2]We assume for the moment that arithmetic operators operate on integers only. Later, in chapter 4, we will address the issue of overloading these operators for integers as well as reals. In PASCAL, the issue may be resolved by inferring the right operator from context using the explicit type declarations.

an intermediate situation where the programmer supplies only a few type annotations in the program that help the compiler in type inferencing, provide self documentation and increase understanding. The issue of how much type inferencing a compiler can do becomes more subtle when we have polymorphism, as discussed below, but the design of the language can still guarantee that the compiler will be able to succesfully infer the types of the expressions within reasonable resource limits. The design of the type inference system for Id was motivated with this view in mind.

2.2 Polymorphism

Consider the PASCAL function `length_listint` in figure 2.2 (a) to count the number of elements in a list of records containing integers. Since this function explicitly declares the types, it may be used only on lists of records containing integers. If the complete program uses lists of various kinds, then one may think of writing one such function for each type of list used. Obviously, this seems to be quite wasteful and unnecessary because the function itself does not care about the type of the elements of the list that it is provided with (namely, integers). It simply counts the number of elements in it. It should be possible to write simply one such function and use it for all types of lists.

Now consider the LISP and the Id version[3] of the same function as shown in (b) and (c) respectively. LISP is a dynamically typed language, which means that expressions are checked for type compatibility only at run time and different invocations of the same program fragment can contain different types as long as it is type consistent for each invocation. So the same function `lisp_length` suffices for all types of lists. Of course, there is no guarantee that the program will not generate any runtime type errors. If `lisp_length` was given an array as an argument in an invocation, the error will not be detected until run time when `lisp_length` tries to take the `cdr` of its argument.

While strongly typed languages like PASCAL provide more reliability in their programs avoiding runtime type errors, they appear to overly constrain their type so that it is impossible to share the code even in places where apparently there is no type conflict. It is possible to relax this constraint and still maintain strong typing by recognizing the safe sharing of a function

[3]The Id definition uses pattern matching to deal with various cases of the input argument, executing the clause corresponding to the matched case on a given input. A "colon" (:) represents the infix "cons" operator for lists. An underscore (_) matches any input.

```
program example2(input,output);
type
  LISTINT = record
              x:integer;
              next:↑LISTINT;
            end;
...
function length_listint(l:↑LISTINT):integer;
begin
  if (l = nil)
     then length_listint := 0
     else length_listint := 1 + length_listint(l↑.next);
end;
```

(a): PASCAL code.

```
(defun lisp_length (l)
   (if (null l)
       0
       (+ 1 (lisp_length (cdr l)))))
```

(b): LISP code.

```
def id_length nil = 0
 |  id_length (_:l) = 1 + id_length l;
```

(c): Id code.

Figure 2.2: A Code Comparison of a length function for Lists.

over objects of several different types. Such functions are called **Polymorphic** in nature as against **Monomorphic** functions that allow objects of only one type. Languages that allow polymorphic functions are said to exhibit *Polymorphism*. Note that LISP is automatically polymorphic since it is dynamically typed.

Coming back to the Id example in figure 2.2 (c), we can see that a smart compiler can realize that the `id_length` function imposes no restriction on the type of elements in the incoming list and hence will record its type as a function from "list of anything to integers". In Id syntax, this may be written as a type annotation as follows:

```
typeof id_length = (list *0) -> I;
```

Here `*0` stands for a "type variable" which represents any type and `I` stands for integers. We will look at more examples later.

Now the same function can be used in different contexts to determine the length of different types of lists. Note that, if the function had used the elements of the list in some way as to restrict their type, (for example, summing them up to restrict them to integers) it will be noticeable in the type inferencing and the compiler will then restrict the type of the function appropriately. We will soon see mechanisms that allow us to do such inferencing. Thus, we see that polymorphism and static type inferencing are useful features that help to combine the reliability obtained from strong typing with ease and efficiency of coding seen in untyped languages.

2.2.1 Kinds of Polymorphism

Cardelli and Wegner describe various kinds of polymorphism in [17]. We have already seen *Parametric Polymorphism* exhibited by `id_length`, which essentially means that the polymorphic functions have an explicit or implicit type parameter which determines the type of the argument for each application of that function. In Id, we take this approach using static type inferencing to provide the implicit type parameters.

We also deal with overloading or *ad hoc Polymorphism.* An identifier is said to be *overloaded* when it is used to represent different functions in different type contexts. Here only the name of the identifier is shared. For example, we may wish to overload usual arithmetic operators $+, -, *, /$ etc. with both integer and floating point meaning. Usually, integers and floating point numbers have different machine representations and arithmetic operators. But the programmer need not be burdened with using different symbols, say, for integer addition and floating point addition. Thus, a smart compiler will be able to translate the same code in different ways according to the type context[4]:

```
typeof a = I;
typeof b = I;

c = a + b ⟹ c = intplus a b

typeof a = F;
typeof b = F;

c = a + b ⟹ c = floatplus a b
```

We will investigate mechanisms to do this in Id in chapter 4.

Object oriented languages allow *Inclusion Polymorphism*, *i.e.*, objects in those languages may be viewed as belonging to many different classes that may include one another, thereby possessing a hierarchy of types by inheritance. As an example, we could declare integers to be a subclass of rationals (pairs of integers in their lowest factor form) which in turn is a subclass of reals. Each class specifies its own set of permitted operations[5] which get superseded by

[4]Here `I` and `F` represent the types integers and floats, and `intplus` and `floatplus` are functions to perform integer and floating point addition respectively.

[5]Also called *Methods* in the literature.

operations of the subclass. If an operation is not found in a particular class, it is looked up in its superclass. Thus, addition, subtraction and multiplication among integers may be specially treated and compiled into (presumably more efficient) integer operations, but division may be handled by the class of rationals. Similarly, the square root operation may be handled only at the level of reals. Once we have this machinery going, it makes sense to divide integers and expect to get back a rational. Similarly we can ask for, say, the square root of 2, and the appropriate operation in the superclass of reals will be invoked to compute it.

In Id, we do not have inheritance and type classes[6]; parametric polymorphism is the default while it is also possible to explicitly declare some identifiers as overloaded.

2.2.2 Higher-Order Functions

Another concept that allows further code sharing and imparts a good structure to the program is that of higher-order functions. We allow functions that manipulate functions as their arguments and compute them as result values. Giving this "first class" status to functions, we can write generalized functions that are useful to a large class of applications and specialize them later as necessary during use. A higher-order style of programming imparts a nice and clean structure to programs and hides details and clutter to bring out just the basic mathematical properties that the functions possess.

The `mapcar` function in LISP is a good example of a higher-order function. Consider its definition below in Id along with its inferred type.

```
def mapcar f nil = nil
 |  mapcar f (x:xs) = (f x):(mapcar f xs);

typeof mapcar = (*0 -> *1) -> (list *0) -> (list *1);
```

`mapcar` expects a function that operates on the elements of a given list and produces a list of those transformed elements. The only restriction on the input list is that it should contain elements consistent with the expected argument type of the function. The "`->`" in the type signature is the function type constructor and associates to the right. The scope of the type variables `*0` and `*1` is the complete type signature. Thus the signature of `mapcar` says that it expects first argument to be a mapping function from some type `*0` to another type `*1` and the second argument to be a list of elements of the first type, and it returns a list of elements

[6] At least not at the moment. But Id is an experimental language and it may be possible later to add inclusion polymorphism to its already rich repertoire of types.

of the second type as the result. The type variables with the same number constrain those objects to possess the same type. Even though the type of the actual function and that of the list supplied to mapcar may not be known during compilation (which, in fact, accounts for its polymorphism), we can still check for type consistency between these two wherever mapcar is used. Thus, we can maintain strong typing while allowing polymorphism.

The mapcar function can be used with different functions in different contexts as shown below:

```
def cube x = x↑3;
typeof cube = I -> I;

mapcar cube (1:2:3:nil) ⟹ 1:8:27:nil

def zero? x = if x==0 then true else false;
typeof zero? = I -> B;

mapcar zero? (0:3:1:0:nil) ⟹ true:false:false:true:nil
```

With cube, it produces a list of integers and with zero? a list of boolean values. Note that the infix list cons operator ":" associates to the right. The lists to the right of the arrow ($\Longrightarrow$) are the results of the computation shown at the left.

While polymorphic, higher-order functions are useful and compact, they pose some theoretical problems for type inferencing. Such functions, in their full generality, are usually modelled using second-order typed Lambda Calculus [13, 23, 53]. The question of decidability of type inference in this model is still open. Some recent work in this field attempts to identify subsets of the full polymorphic model where this question can still be answered [30].

There are two ways out of this problem. We can restrict the polymorphism of arguments and results of functions in certain ways so that we are still able to decidably infer their type using known algorithms. The most popular approach, taken by many modern languages such as ML [24, 25, 41, 42], Miranda [60, 61], Hope [14], Haskell [27] and Id, is to use a special polymorphic type inference system now commonly referred to as the Hindley/Milner type system [20, 26, 40]. Other, more recent efforts to strengthen this inference system have met with only partial success [30, 32, 43].

The other way out is to help the type inference system by giving explicit type declarations in the program while still retaining polymorphic functions as first class objects. It is a non-trivial problem to figure out just where such type declarations need to be inserted. Moreover, it is not clear that the additional expressive power gained by this approach would outweigh the loss in ease of programming and the increase in complexity of the compiler. There are experimental systems based on this approach [22] and only future research may shed further light on the advantages of one over the other.

In Id, we take the former position adopting the Hindley/Milner type inference mechanism, largely due to the fact that these type systems have been tested in several languages with success and have proved to be a good mix of polymorphism with type inferencing, keeping the clarity and ease of programming intact even in large programs. We will now describe this type system in detail.

2.3 Hindley/Milner Type Inference System

For the purpose of our discussion, we first define an abstract, functional mini-language, which is a slight extension of Lambda Calculus and forms the core of the languages mentioned above. We only extend it with a `let` construct, and arithmetic and logical constants. The latter are included only to show meaningful examples without obscuring the clarity. This language will form the basis of all our discussions on types later on and we will make extensions to it where necessary.

We also present some basic notation that will be used throughout the rest of this thesis. We try to adopt a notation similar to one that has appeared in literature before, extending it where necessary. Then we present the Hindley/Milner type inference system as a set of inference rules and show its semantic soundness and other properties. Finally, we discuss an algorithm which uses these rules to derive a type for every expression in the given program and show its semantic properties.

The basic inferencing mechanism was first devised by Hindley [26], which was later independently rediscovered and extended by Milner [40] to allow polymorphic functions. The theoretical basis was firmly established in Denotational Semantics by Damas [20, 19] and in Operational Semantics by Tofte [57]. Since then, several expositions have appeared in the literature, either in context of the languages that use this system or those that extend the richness or power of the system in some way (typing polymorphic references, type-containment, type-coercion etc.). Our notation is the same as used by Tofte in [57] and the description is primarily derived from the work of Damas [20], Tofte [57], and Clément *et al.*[18].

2.3.1 A Mini-Language

Expressions

We start with a set of basic constants and identifiers in the language as shown below. The meta-variables on the left range over the sets on the right.

$$c \in \text{Constants} \quad = \quad \{\texttt{true}, \texttt{false}, 1, 2, \ldots\} \tag{2.1}$$

$$x \in \text{Identifiers} \tag{2.2}$$

Using these we define the set of expressions admissible in the abstract language:

$$\begin{aligned} e \in \text{Expressions} \quad ::= \quad & c \\ \mid \quad & x \\ \mid \quad & \lambda x.\, e_1 \\ \mid \quad & e_1\; e_2 \\ \mid \quad & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \end{aligned} \tag{2.3}$$

The only difference in this definition from the usual extended Lambda Calculus syntax is the introduction of a `let` expression, and that really makes all the difference for polymorphism in this system.

Type Expressions

We start as usual with a set of type constants or nullary type constructors and an infinite set of type variables as shown below.

$$\pi \in \text{Type-Constructors} \quad = \quad \{\mathit{int}, \mathit{bool}, \ldots\} \tag{2.4}$$

$$\alpha, \beta \in \text{Type-Variables} \quad = \quad \{t, t', t_1, t_2, \ldots\} \tag{2.5}$$

Using these, we define the type expressions of the language:

$$\begin{aligned} \tau \in \text{Types} \quad ::= \quad & \pi \\ \mid \quad & \alpha \\ \mid \quad & \tau_1 \rightarrow \tau_2 \end{aligned} \tag{2.6}$$

$$\begin{aligned} \sigma \in \text{Type-Schemes} \quad ::= \quad & \tau \\ \mid \quad & \forall \alpha.\, \sigma_1 \end{aligned} \tag{2.7}$$

The function type constructor $(\rightarrow)$, associates to the right. Also note that *Types* are not quantified, while the *Type-Schemes* are universally quantified at the outermost. A type scheme $\forall\alpha_1.\forall\alpha_2.\cdots\forall\alpha_n.\tau$ is also written as $\forall\alpha_1\alpha_2\cdots\alpha_n.\,\tau$.

2.3.2 Some Notations and Definitions

First, we define some general set notation to help us later.

Notation 2.1 *If A and B are sets, then,*

1. *powerset(A) is defined to be the set of all subsets of A.*
2. *$A + B$ is the disjoint union of A and B.*
3. *$A \setminus B$ is the set difference $A - B$.*
4. *$A \times B$ is the cartesian product of A and B.*
5. *$A \rightarrow B$ is a set of functions from A to B. The precise interpretation of $(\rightarrow)$ depends on the characteristics of the sets A and B. For example, if A and B are simple syntactic sets, then $A \rightarrow B$ merely denotes the set of all total functions from A to B; if A and B are scott-domains, then $A \rightarrow B$ denotes the set of continuous functions from A to B, and so on.*
6. *$A \xrightarrow{fin} B$ is the set of finite maps[7] from A to B.*

Notation 2.2 *If f and g are functions (maps) then,*

1. *$Dom(f)$ is the Domain of f.*
2. *$Ran(f)$ is the Range of f.*
3. *$f \in A \rightarrow B$ means that f is a function from A to B as determined by the interpretation of the set $A \rightarrow B$.*
4. *$f \in A \xrightarrow{fin} B$ means that f can be written as $\{a_1 \mapsto b_1, \ldots, a_i \mapsto b_i, \ldots, a_n \mapsto b_n\}$ where $a_i \in A$ and $b_i \in B$.*
5. *$\{\}$ represents the empty map.*
6. *$f \circ g$ (read f* **composed with** *g) is the composite map defined by $(f \circ g)(a) = f(g(a))$.*
7. *$f + g$ (read f* **modified by** *g) is the map with domain $Dom(f) \cup Dom(g)$ defined by $(f + g)(a) =$ if $a \in Dom(g)$ then $g(a)$ else $f(a)$.*
8. *If f and g have disjoint domains then $f \mid g$ (read f* **simultaneously composed with** *g) is the map with domain $Dom(f) \cup Dom(g)$ formed by the concatenation of the two maps.*

[7] A finite map is a function with a finite domain.

This serves in place of either f or g and emphasizes the disjointness of their respective domains.

9. *$f\backslash_A$ is the map f with its domain restricted to $Dom(f) \setminus A$.*
10. *$f \downarrow_A$ is the map f with its domain restricted to A.*

We can relate the identifiers of the language to the types assigned to them in a type environment, which is a finite map from identifiers to type-schemes.

$$TE \in \text{Type-Environments} = \text{Identifiers} \xrightarrow{fin} \text{Type-Schemes} \tag{2.8}$$

Next, we define some notation and operations on type expressions which would be useful later.

Definition 2.3 *Given a type scheme $\sigma = \forall \alpha_1 \cdots \alpha_n . \tau$,*

1. *$\alpha_1, \ldots, \alpha_n$ are* **bound** *in σ.*
2. *A type variable α is* **free** *in σ if it is not bound and it occurs in τ.*
3. *A type variable α is* **free** *in a type environment TE, if it is free in some type scheme from its range $Ran(TE)$.*

In short, the quantified type variables are bound and the others are free in a type scheme, just as one might expect. Using the above definition, we can define the following functions for computing free and bound variables of types, type schemes and type environments:

Definition 2.4 *The function $FV \in$ Types $\rightarrow$ powerset(Type-Variables) computes the free type variables of a type expression as follows:*

1. *$FV(\tau)$ is the set of all the type variables in type τ, all of which are free.*
2. *By extension, $FV(\sigma)$ and $FV(TE)$ denote the set of free type variables in σ and TE respectively as defined above in definition 2.3.*

Definition 2.5 *The function $BV \in$ Type-Schemes $\rightarrow$ powerset(Type-Variables) gives the set of bound variables of the type scheme σ.*

If $FV(\tau) = \phi$ then τ is called a **ground type** or **monotype**. Similarly , if $FV(\sigma) = \phi$ and $FV(TE) = \phi$, then σ and TE are said to be **closed**, respectively. When it is clear by context that we are talking about only free type variables of an object, we may also use the symbol $tyvars(\sigma)$ instead of $FV(\sigma)$.

Now we define the important concept of substituting free and bound type variables in type and type schemes with other types.

Definition 2.6 *A* **Substitution** $S \in$ Type-Variables $\rightarrow$ Types *is a map from type variables to types. Moreover,*

1. *By extension, a substitution S is applied to a type τ yielding τ', written as $\tau' = S\tau$, by applying it to all the (free) type variables $FV(\tau)$. The type τ' is called a* **substitution instance** *or sometimes simply an* **instance** *of type τ.*
2. *Similarly, for a type scheme $\sigma = \forall \alpha_1 \cdots \alpha_n . \tau$, a substitution $\sigma' = S\sigma$ is applied only to its free type variables $FV(\sigma)$ renaming all its bound type variables. More formally, we define $\sigma' = \forall \beta_1 \cdots \beta_n . \tau'$ where $\beta_1 \cdots \beta_n$ are new type variables, $\tau' = (S' \mid \{\alpha_i \mapsto \beta_i\})\tau$ and $S' = S \downarrow_{tyvars(\sigma)}$. As before, σ' is called a* **substitution instance** *of σ.*
3. *A substitution to a type environment TE is applied to the free variables of each type scheme σ in its range.*

Using notation 2.2, a finite substitution is also written as $S = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$. We may also abbreviate it as $S = \{\alpha_i \mapsto \tau_i\}$. Composition of Substitutions $(S_2 \circ S_1)\alpha$ means $S_2(S_1\alpha)$ where the parentheses can be dropped by assuming that substitutions associate to the right.

Note that substitutions are defined for free type variables only. Substitutions for the bound variables in a type scheme are done differently and are called Instantiations[8].

Definition 2.7 *Given a type scheme $\sigma = \forall \alpha_1 \cdots \alpha_n . \tau$ with $BV(\sigma) = \{\alpha_1, \ldots, \alpha_n\}$,*

1. *A type τ' is said to be a* **generic instance** *(also called an Instantiation) of σ, written as $\sigma \succ \tau'$, if there exists a finite substitution $S = \{\alpha_i \mapsto \tau_i\}$ with $Dom(S) \subseteq BV(\sigma)$ such that $\tau' = S\tau$.*
2. *A type scheme $\sigma' = \forall \beta_1 \cdots \beta_m . \tau'$ is said to be a* **generic instance** *of σ, written as $\sigma \succeq \sigma'$, when any of the following equivalent conditions hold,*
   - *For all types τ_1, $\sigma' \succ \tau_1$ implies $\sigma \succ \tau_1$.*
   - *$\sigma \succ \tau'$ and no β_j is free in σ.*
   - *$\sigma \succ \tau'$ and $FV(\sigma) \subseteq FV(\sigma')$.*

[8] We differ slightly from the terminology used in [20] where these are called Instantiations and Generic Instantiations.

3. Similarly, by extension, instantiation of a type environment TE yields a **generic instance** *TE', written as $TE \succeq TE'$, obtained by pointwise instantiation of the type-schemes in its range.*

Note here that Instantiations act only on bound variables of type schemes since the domain of the substitution S used in showing $\sigma \succ \tau$ is limited to $BV(\sigma)$. Thus substitution instances are created *via* substitutions and generic instances are created *via* instantiations.

We may sometimes write $\sigma \succeq \tau$ to mean $\sigma \succ \tau$ where σ could also be just a type without any bound variables. In fact, it can be easily seen from the above definitions that for types τ and τ', $\tau \succeq \tau'$ implies $\tau = \tau'$.

Finally, we admit quantification of the free type variables in a type and converting it to a type scheme as an inverse operation of instantiation.

Definition 2.8 Generalization *or* **Closing** *a type τ under a type environment TE is defined as follows:*

$$close(TE, \tau) = \begin{cases} \forall \alpha_1 \cdots \alpha_n . \, \tau & (FV(\tau) \setminus FV(TE)) = \{\alpha_1, \ldots, \alpha_n\} \\ \tau & (FV(\tau) \setminus FV(TE)) = \phi \end{cases}$$

When $FV(TE) = \phi$, we may also write simply $close(\tau)$ instead of $close(TE, \tau)$.

This is the operation that introduces polymorphism in the system. Note that we are not allowed to quantify over free variables of a type that may already exist freely in the type environment. This is an important point that makes this system sound.

2.3.3 Operational Semantics of the Mini-Language

Now we are ready to present the operational semantics of the mini-language. Both dynamic and static semantics are presented as a set of inference rules [52] operating on the expressions of the mini-language. The static semantic rules constitute the basic Hindley/Milner type inference rules, from which valid assertions about the type of the expressions can be inferred.

Dynamic Semantics

Dynamic operational semantics tells us how to evaluate the expressions of the language. For this, we need to define the objects created and manipulated during the course of evaluation.

IDENTIFIER:	$$\frac{x \in Dom(E)}{E \vdash x \longrightarrow E(x)}$$
ABSTRACTION:	$$\frac{}{E \vdash \lambda x.\, e_1 \longrightarrow [x, e_1, E]}$$
APPLICATION:	$$\frac{\begin{array}{c} E \vdash e_1 \longrightarrow [x_0, e_0, E_0] \\ E \vdash e_2 \longrightarrow v_2 \\ E_0 + \{x_0 \mapsto v_2\} \vdash e_0 \longrightarrow v \end{array}}{E \vdash e_1\; e_2 \longrightarrow v}$$
LET:	$$\frac{E \vdash e_1 \longrightarrow v_1 \qquad E + \{x \mapsto v_1\} \vdash e_2 \longrightarrow v}{E \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \longrightarrow v}$$

Figure 2.3: Dynamic Inference Rules.

These are defined below:

$$c \in \text{Constants} = \{true, false, 1, 2, \ldots\} \tag{2.9}$$

$$v \in \text{Values} = \text{Constants} + \text{Closures} \tag{2.10}$$

$$[x, e, E] \in \text{Closures} = \text{Identifiers} \times \text{Expressions} \times \text{Environments} \tag{2.11}$$

$$E \in \text{Environments} = \text{Identifiers} \xrightarrow{fin} \text{Values} \tag{2.12}$$

We assume an environmental model of evaluation. The inference rules appear in figure 2.3. The rules consist of two parts. In each rule, establishing all the premises above the line allows us to conclude all the conclusions below the line. Each premise or conclusion is an *evaluation* of the following form[9]:

$$E \vdash e \longrightarrow v$$

which is read as "starting with environment E, the expression e *evaluates* to value v".

The inference rules are easy to understand. Constants represent themselves and hence are excluded; identifiers are looked up in the environment for their value; lambda abstractions form closures with formal parameter x and body e over the lexical environment E which provides the values to the free variables of the body; applications (by juxtaposition) are the usual function applications; and finally, the `let` construct behaves exactly like an immediate application of an abstraction.

[9] We may extend this later with the notion of a store to incorporate imperative constructs in the mini-language.

$$\text{TAUT:}\quad \frac{(x \mapsto \sigma) \in TE \qquad \sigma \succeq \tau}{TE \vdash x : \tau}$$

$$\text{ABS:}\quad \frac{TE + \{x \mapsto \tau'\} \vdash e_1 : \tau}{TE \vdash \lambda x.\, e_1 : \tau' \rightarrow \tau}$$

$$\text{APP:}\quad \frac{TE \vdash e_1 : \tau' \rightarrow \tau \qquad TE \vdash e_2 : \tau'}{TE \vdash e_1\ e_2 : \tau}$$

$$\text{LET:}\quad \frac{TE \vdash e_1 : \tau' \qquad TE + \{x \mapsto close(TE, \tau')\} \vdash e_2 : \tau}{TE \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

Figure 2.4: Static Inference Rules DM'.

Note that there are no special constructs to deal with conditional expressions or recursive and mutually recursive functions. This is because it is easy to model them using predefined primitive functions. Conditionals expressions of the type "`if` e_1 `then` e_2 `else` e_3" can be written as "`cond` e_1 e_2 e_3" using a predefined primitive ternary operator *cond* which returns the second or the third argument accordingly as the first argument evaluates to *true* or *false*. Similarly, recursive functions are represented as functionals applied to a predefined fixpoint operator $fix = \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$. Of course, in a real implementation, for efficiency reasons, we may recognize and treat conditionals and recursion specially and the *cond* and *fix* operators may not figure in explicitly at all. Similarly, we will also assume a repertoire of the usual arithmetic functions $+, -, *, \div, <, \leq, >, \geq, =, \neq$ etc. predefined in our initial environment.

Static Semantics

Static operational semantics provides a mapping from the expressions of the language to the type expressions under a set of assumptions regarding the types of the free identifiers of the program. This set of assumptions is the type environment, which is a finite map from identifiers to type schemes (see equation 2.8).

The inference rules of the Hindley/Milner type system appear in figure 2.4. Again, in each rule, establishing all the premises above the line allows us to conclude all the conclusions below the line. Each premise or conclusion is an *elaboration* or *typing* of the following form:

$$TE \vdash e : \tau$$

$$
\begin{array}{ll}
\text{TAUT:} & \dfrac{(x \mapsto \sigma) \in TE}{TE \vdash x : \sigma} \\[2ex]
\text{INST:} & \dfrac{TE \vdash e : \sigma \qquad \sigma \succeq \sigma'}{TE \vdash e : \sigma'} \\[2ex]
\text{GEN:} & \dfrac{TE \vdash e : \sigma \qquad \alpha \notin FV(TE)}{TE \vdash e : \forall \alpha.\, \sigma} \\[2ex]
\text{ABS:} & \dfrac{TE + \{x \mapsto \tau'\} \vdash e_1 : \tau}{TE \vdash \lambda x.\, e_1 : \tau' \rightarrow \tau} \\[2ex]
\text{APP:} & \dfrac{TE \vdash e_1 : \tau' \rightarrow \tau \qquad TE \vdash e_2 : \tau'}{TE \vdash e_1\ e_2 : \tau} \\[2ex]
\text{LET:} & \dfrac{TE \vdash e_1 : \sigma' \qquad TE + \{x \mapsto \sigma'\} \vdash e_2 : \tau}{TE \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}
\end{array}
$$

Figure 2.5: Original Damas-Milner Inference Rules DM.

which is read as "assuming the type environment TE, the expression e *elaborates* to the type τ".

Readers familiar with the original Damas-Milner (DM) inference system as described in [20] (refer to figure 2.5) will recognize the following differences in the system presented here (DM'):

1. There are no separate rules for generalization of types or instantiation of type schemes. The former is done in by the *close* operation in the LET rule and the latter is done at the time of typing identifiers from the environment (TAUT rule).

2. There is only one rule applying to any syntactic construct. So, this system is in fact deterministic, while the Damas-Milner system was not.

3. The instantiation to types rather than type schemes in the TAUT rule makes the result of a typing to be a type instead of a type scheme. But it can be shown that these two systems admit exactly the same expressions and are thus equivalent in this sense.

The equivalence of DM and DM' is captured in the following theorem proved in [18].

Theorem 2.1 *The system DM' is equivalent to system DM in the following sense:*

$$\begin{array}{rcl} TE \overset{DM'}{\vdash} e : \tau & \Longrightarrow & TE \overset{DM}{\vdash} e : \tau \\ \forall TE, e\, \exists \tau \;\; TE \overset{DM}{\vdash} e : \sigma & \Longrightarrow & TE \overset{DM'}{\vdash} e : \tau \,\wedge\, close(TE, \tau) \succeq \sigma \end{array}$$

This means that every typing derivable in this DM' system is also derivable in the original DM system and conversely, every typing derivable in the DM system is an instantiation of the closed version of some typing derivable in this system.

Finally, we note the following lemmas which are proved in [57] for the DM' system. Their corresponding versions for the DM system are proved in [20].

Lemma 2.2 *If $\sigma \succ \tau'$ then for all substitutions S, $S\sigma \succ S\tau'$.*

Lemma 2.3 *If S is a substitution then,*

$$TE \vdash e : \tau \qquad \Longrightarrow \qquad S(TE) \vdash e : S\tau$$

Another lemma from [20] relates typings with instantiated environments.

Lemma 2.4 *If $\sigma \succeq \sigma'$ then,*

$$TE + \{x \mapsto \sigma'\} \vdash e : \sigma_0 \qquad \Longrightarrow \qquad TE + \{x \mapsto \sigma\} \vdash e : \sigma_0$$

This last lemma can easily be extended inductively (pointwise) for the case when $TE \succeq TE'$.

2.3.4 Denotational Model of the Mini-Language

Domains, Functions, and Types

A word about a denotational model of the above mini-language is appropriate here. The model that we present here is that of untyped Lambda Calculus. This is the position taken in the original paper by Milner [40]. For the most part, the dynamic semantics is fairly intuitive; constants represent themselves as elements of a Scott Domain [55, 56] and lambda abstractions create domains containing continuous functions over these domains. The syntactic categories we introduced in section 2.3.3 can thus be easily understood in terms of these domains. Environments would then become partial functions from identifiers to elements of these domains.

Using the same terminology to refer to the above mentioned semantic constructs, we write:

$$E \models e \longrightarrow v$$

read as, "given environment E, expression e *denotes* the domain value v under this model". Note that the "`let` $x = e_0$ `in` e_1" expression has exactly the same semantics as $(\lambda x.\, e_1)e_0$.

We will not spend any time in the details of writing down the denotational evaluation function for the dynamic semantics since our main goal is type checking or static semantics. Later in this section, we will discuss a little more about the semantic model of the language, in order to relate it to the operational inference rules seen above. The literature is filled with the discussion of semantics of Lambda Calculus, both operational and denotational. A complete theoretical treatment can be found in [12]. A very good discussion on Scott domains and denotational semantics can be found in [56].

The semantics of the type expressions is also fairly simple. The nullary type constructors (π) represent the type of basic Scott domains as described above that contain those constants. $\tau_0 \rightarrow \tau_1$ is the type of the domain containing continuous functions from the domain with type τ_0 to the domain with type τ_1. Again, type environments become partial functions from identifiers to elements of this type structure. So we write,

$$TE \models e : \tau$$

read as, "given a type environment TE, the value denoted by the expression e has the type τ under this model".

Polymorphism and Quantification

The above strategy is sufficient to ascribe meaning to the type of monomorphic objects (objects with ground types), for example, all the predefined binary arithmetic operators would have the type $int \rightarrow int \rightarrow int$[10].

Polymorphic objects are typed by type schemes. A type scheme can be viewed as a concise statement of the different ground types a polymorphic object is capable of possessing. More theoretically, their type is a semantic quantification over the types of these Scott domains. For example, consider the identity function defined below:

`let` $identity \;=\; \lambda x.\, x$ `in` e

We treat this function as polymorphic, accepting an object of any kind and returning the same object. Thus, to type this identity function, we can view it as possessing a family of types, one for each type of monomorphic object that it can accept. In other words, a typing of this

[10] As mentioned earlier, we will deal with the overloading of these operators for real arguments in chapter 4.

function exists for every type domain $\tau \to \tau$. This can be written in a type scheme as shown below:

$$identity : \forall \alpha.\, \alpha \to \alpha$$

Similarly, the primitive conditional operator *cond* will be typed as:

$$cond : \forall \alpha.\, bool \to \alpha \to \alpha \to \alpha$$

In general we write,

$$TE \models v : \sigma \qquad \text{iff} \qquad \forall \tau \preceq \sigma.\, (TE \models v : \tau)$$

Note that the $\preceq$ relation between τ and σ is syntactic and not semantic.

We have seen that the quantification in a type scheme is only at the outermost level. This means that we do not allow arbitrary polymorphic arguments and results of functions, restricting ourselves to a proper subset of the general class of semantically meaningful expressions. This is done specifically to obtain a subset of expressions for which the question of typability is known to be decidable. In spite of this restriction, we are able to express most of the polymorphism inherent in functions like *identity* within this subset.

One apparently important function that can not be type checked in this system is the universal fixpoint operator, $fix = \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$. Our system is not powerful enough to type it as a polymorphic function. This is because self application requires more than just outermost quantification. But we can get around this problem by pre-defining a polymorphic operator in the initial environment with the following type:

$$fix : \forall \alpha.\, (\alpha \to \alpha) \to \alpha$$

This effectively asserts that there exists a typing for a fixpoint operator in every domain of interest. This type specification is sufficient to guarantee correct typings for recursive and mutually recursive functions.

Soundness of Inference

Soundness of a type inference system implies that expressions that are inferred to be type-correct, do not lead to a run-time type error on evaluation, or denotationally speaking, they do not denote a type error. Further, the values they denote or evaluate to, have the corresponding inferred types. This is a fundamental property necessary for any inference mechanism to establish its validity with respect to the associated operational or denotational computation model.

In his original paper [40], Milner showed the semantic soundness of well-typed expressions. In the framework of Damas-Milner system *DM* as described in [20, 19], Damas proved the soundness of his system using Denotational Semantics. Due to theorem 2.1, that system and the system presented here are equivalent. Therefore, intuitively it should be clear that the soundness theorem extends to this system as well. Indeed, Tofte in [57] showed a similar consistency result for our system in an Operational Semantics setting. So, henceforth we shall assume that the semantic properties proved for one system naturally extend to the other and we may only state the results for one of them. Since *DM* infers type schemes in its typings while DM' infers types, it should be clear by context which one is being referred to, if at all we need to differentiate between the two.

Theorem 2.5 (Soundness of Inference) *For an expression e,*

$$TE \vdash e : \sigma \qquad \Longrightarrow \qquad TE \models e : \sigma$$

Typed Lambda Calculus Model

As an aside, we would like to point out that the model presented above is not the only one possible. In fact, the second-order typed Lambda Calculus provides a much cleaner and easier to understand framework for a more general and richer class of functions [13, 23, 53]. Historically, though, only recently has there been an attempt to understand the various levels of type inference possible in that model, so as to keep it decidable [30]. It is possible to fit this system in that model by restricting the polymorphism of its typed terms to outermost quantification only. The model easily extends to addition of primitive value and type constants as well as more *kinds* of type constructors rather than just having the function type constructor and the universal quantification.

In that model, the notion of *type* becomes more complicated and general than simply a set of domain values. Every identifier is assumed to possess a type, which is explicitly quantifiable by lambda abstractions over types to achieve polymorphism. The meaning of untyped syntactic terms is taken to be an *erasure* of type information from the corresponding typed terms and the purpose of type inferencing is to *reconstruct* that type information[11].

[11] It is for this reason that the term *Type Reconstruction* is also used in the literature instead of *Type Inferencing* .

To give an example, the identity function for a type τ would look like:

$$\begin{aligned} identity_\tau &= \lambda x : \tau.\, x \\ identity_\tau &: \tau \rightarrow \tau \end{aligned}$$

and its polymorphic version looks like:

$$\begin{aligned} identity &= \lambda t.\, \lambda x : t.\, x \\ identity &: \forall \alpha.\, \alpha \rightarrow \alpha \\ identity_\tau &= identity\ \tau \end{aligned}$$

Note that the type obtained for the polymorphic identity is the same as before. We will not present this model here since our purpose is served by working with operational static semantics (symbol pushing) only and it does not matter what model is used as long as we are confident that the syntactic inference system has sound theoretical basis. The interested reader is refered to [13] for further discussion in this regard.

2.3.5 The Type Inference Algorithm

Given a set of inference rules, it is possible to implement them in various direct or indirect ways, addressing issues such as efficiency and correctness. We were concerned about the determinacy of our inference system primarily to obtain a direct algorithm for it. Even though all the semantic features of Hindley/Milner inference system are captured in its inference rules, for pragmatic reasons and for the sake of completeness, we still present the inference algorithm most widely used in the literature to implement these rules.

Our inference algorithm uses the Unification algorithm of Robinson [54] which we state below as a theorem.

Theorem 2.6 (Robinson) *There is an algorithm U which, given a pair of types, either returns a substitution S or fails. Further,*

1. *If $U(\tau, \tau')$ succeeds and returns a substitution S, then S unifies τ and τ',* i.e., *$S\tau = S\tau'$.*
2. *If R unifies τ and τ', then $U(\tau, \tau')$ succeeds and returns S such that there exists another substitution T satisfying $R = T \circ S$. Moreover, S involves only variables in τ and τ'.*

The inference algorithm W appears in figure 2.6. It is essentially a case statement based directly on the deterministic inference rules of figure 2.4. It accepts a type environment and

```
Def W(TE,e) =
  case e of
   x           ⇒ if x ∉ Dom(TE) then  fail.
                  else let
                         ∀α1 ··· αn. τ = TE(x).
                         β1, ..., βn be new type variables.
                       in
                         (ID,{αi ↦ βi}τ).
                  endif.
   λx. e1      ⇒ let
                   β be a new type variable.
                   (S1, τ1)  =  W(TE + {x ↦ β}, e1).
                 in
                   (S1, S1β → τ1).
   e1 e2       ⇒ let
                   (S1, τ1)  =  W(TE, e1).
                   (S2, τ2)  =  W(S1(TE), e2).
                   β be a new type variable.
                   S3  =  U(S2τ1, τ2 → β).          (may  fail)
                 in
                   (S3S2S1, S3β).
   let x  =  e1 in e2 ⇒
                 let
                   (S1, τ1)  =  W(TE, e1).
                   σ  =  close(S1(TE), τ1).
                   (S2, τ2)  =  W(S1(TE)  +  {x ↦ σ}, e2).
                 in
                   (S2S1, τ2).
  endcase.
```

Figure 2.6: Pseudo-code for the inference algorithm W.

an expression, and returns a substitution S and a type τ, if succesful. The algorithm is sound, *i.e.*, it never computes a wrong derivation. This fact is captured in the following theorem.

Theorem 2.7 (Soundness of W) *If the call $W(TE, e)$ succeeds returning (S, τ), then there is a derivation of the typing $S(TE) \vdash e : \tau$.*

In fact, W computes the most general type or the *principal type* as defined below.

Definition 2.9 *Given a type environment TE and expression e, τ is called a* **principal type** *of e under TE iff,*

1. *$TE \vdash e : \tau$.*
2. *For any other τ' such that $TE \vdash e : \tau'$, we have $close(TE, \tau) \succ \tau'$.*

$close(TE, \tau)$ is also called the **principal type scheme** *of e under TE.*

The fact that W computes the principal types, is captured in the following theorem.

Theorem 2.8 (Completeness of W) *Given a type environment TE and expression e, let $TE' = S'(TE)$ be a substitution instance of TE such that $TE' \vdash e : \tau'$. Then,*

1. *$W(TE, e)$ succeeds.*
2. *If $W(TE, e) = (S, \tau)$ then there exists a substitution R such that $TE' = (R \circ S)(TE)$ and $R(close(S(TE), \tau)) \succ \tau'$.*

Basically, this says that all typings of an expression can be obtained as instantiations of the closed version of the type returned by W in the final type environment.

The proof of these results appears in [57]. The corresponding results for the *DM* system are proved in [19]. This completes our discussion of the Hindley/Milner type inference system.

2.4 Id Kernel Language

In this section, we present the kernel language for Id. We will discuss how it fits with our mini-language described above and show how to perform basic type inferencing for programs written in it.

Id is an evolving language [45, 46, 48]. But most of its important syntactic features can be expressed in a small subset of the full language. [8] discusses standard techniques for such "kernelization"; converting *for*-loops to *while*-loops and *while*-loops to tail-recursive procedure calls, infix operators to function applications, n-tuple manipulations (making and selection) into family of primitive functions, removing n-tuples from function arguments and block bindings etc. Literature has standard techniques to translate pattern matching into nested conditionals [10, 11, 15] as well. Thus, our task of static type analysis is greatly simplified after these transformations[12].

The kernel language description has been revised extensively by Ariola and Arvind in [4]. There the reader will find precise operational dynamic semantics and proofs of confluence etc. We will not go into those details since we are concerned with static semantics only, but we will draw from their description of the Id kernel syntax. We also draw heavily from the description of I-structures by Arvind *et al.* in [9] which are assignable data structures used to extend the functional programming paradigm to provide efficient synchronisation and storage without losing useful properties such as confluence. We will explore the issue of polymorphic type inferencing in presence of such assignable structures in chapter 5.

2.4.1 Mini-Language *vs* Id Kernel Language

The kernel language P-TAC as shown in [4] and [9] is reproduced here in figure 2.7. The start symbol is *Program*. The superscripts on constants and identifiers denote their syntactic arity. We will restrict ourselves to the functional subset of P-TAC which simply amounts to ignoring the I-structure manipulation functions, namely, `allocate`, `I_select`, and `I_set`. But it still has far too many details for our purpose. That is why the mini-language was introduced. Since we already know how to do type inference for expressions in the mini-language, we will only describe a translation from the functional subset of the Id kernel language to the mini-language described above for the purpose of type inference.

Before we show the translation, a word about the mini-language itself is appropriate. It is easy to simulate tuples and lists in the mini-language. Tuples are essentially elements of a product type domain $(A \times B)$ and lists and other such recursive type structures are usually modelled using disjoint union domains $(A+B)$. We state without proof that these domains can be modelled using the function type constructor $(\rightarrow)$ only. These domain constructions identify simple syntactic transformations necessary to represent everything in terms of the function type constructor. Thus, we will freely use these constructs as part of our mini-language where

[12] Some of these transformations are not performed in the actual implementation for reasons of efficiency, but that is not our concern here. Our goal here is to simplify the syntax to the level of the mini-language presented above, so that we can reason about it.

necessary.

2.4.2 Translation of Id Kernel Language into Mini-Language

Instead of specifying a formal translation from the Id kernel to our mini-language, we will only provide guidelines for such a translation, relying largely on the reader's intuition. The following features of the Id kernel motivate the translation guidelines.

1. A program is a list of user-defined procedures and a main expression. The definitions may be recursive and mutually recursive and in arbitrary order.

2. The definitions may have an arbitrary arity.

3. There are nullary as well as higher arity constants.

4. A block is similar to a `let` construct, only that it allows several bindings to be established at once that may be recursive or mutually recursive.

Note that the mini-language requires that there be only one non-recursive binding in a `let` construct and that all the identifiers used in an expression be either pre-defined in the environment or must have been defined in a binding enclosing that expression[13]. The mini-language also requires that the recursive and mutually recursive functions be expressed using the predefined fixpoint operator *fix* shown in section 2.3. Our translation must take care of these syntactic constraints without losing generality.

The various nullary constants can be directly incorporated into the class of constants in the mini-language. The other higher arity constants can be pre-defined in the environment.

The translation of multi-arity functions into nested λ-expressions is straightforward using "currying" as shown below. This transformation does not affect the static semantics of the program.

$$\texttt{def f x1 x2} \cdots \texttt{xn =}\ e \implies f = \lambda x_1.\, \lambda x_2.\, \cdots\, \lambda x_n.\, e$$

We translate the recursive functions into an application of *fix* to their corresponding non-recursive functionals where the function identifier is abstracted out as a parameter. Mutually recursive functions are treated the same way to obtain a joint fixpoint in a tuple as shown

[13] The `let` and the λ-abstraction constructs are the only means of binding (or defining) identifiers to values. The binding of x in "$\lambda x.\, e$" is visible inside e and that in "$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$" is visible only inside e_2.

$$
\begin{array}{lll}
\textit{Program} & ::= & \textit{Definition}\ ;\ \cdots\ ;\ \textit{Definition}\ ;\ \textit{Main} \\[1ex]
\textit{Definition} & ::= & \texttt{Def}\ \textit{ProcId}\ \underbrace{\textit{Identifier} \cdots \textit{Identifier}}_{n}\ \texttt{=}\ \textit{Expression} \qquad (n > 0) \\[1ex]
\textit{ProcId} & ::= & \textit{Identifier} \\
\textit{Main} & ::= & \textit{Block} \\[1ex]
\textit{Expression} & ::= & \textit{SimpleExpr} \\
 & | & \textit{Constant}^n\ \underbrace{\textit{SimpleExpr} \cdots \textit{SimpleExpr}}_{n} \qquad (n > 0) \\
 & | & \textit{Block} \\
\textit{SimpleExpr} & ::= & \textit{Constant}^0 \mid \textit{Identifier} \mid \textit{ProcId} \\[1ex]
\textit{Constant}^0 & ::= & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid \cdots \\
\textit{Constant}^1 & ::= & \texttt{mk\_closure} \mid \texttt{hd} \mid \texttt{tl} \mid \texttt{nil?} \mid \cdots \\
\textit{Constant}^2 & ::= & \texttt{+} \mid \texttt{-} \mid \cdots \mid \texttt{<} \mid \texttt{=} \mid \cdots \mid \texttt{allocate} \mid \texttt{I\_select} \mid \\
 & & \texttt{cons} \mid \texttt{apply} \mid \cdots \\
\textit{Constant}^3 & ::= & \texttt{cond} \mid \cdots \\[1ex]
\textit{Block} & ::= & \{\quad \textit{Statement}\ ; \\
 & & \qquad \cdots \\
 & & \qquad \textit{Statement} \\
 & & \ \texttt{In} \\
 & & \qquad \textit{SimpleExpr}\ \} \\
\textit{Statement} & ::= & \textit{Binding} \mid \textit{Command} \\
\textit{Binding} & ::= & \textit{Identifier}\ \texttt{=}\ \textit{Expression} \\
\textit{Command} & ::= & \texttt{null} \mid \texttt{I\_set}\ \textit{SimpleExpr}\ \textit{SimpleExpr}\ \textit{SimpleExpr}
\end{array}
$$

Figure 2.7: Syntax of Id Kernel Language.

below:

$$f_1 = e_1; f_2 = e_2; \cdots; f_n = e_n \implies f_1, f_2, \ldots, f_n = \mathit{fix}\ \ \lambda f_1, f_2, \ldots, f_n.\, e_1, e_2, \ldots, e_n$$

The only task left now is to show how to partition all the bindings into groups of mutually recursive ones and then to generate nested `let` bindings acceptable to our mini-language syntax. This can be accomplished using the standard graph algorithm for finding the strongly connected components in a directed graph [1]. Each binding is assumed to be a node in the graph. There is an edge from binding $f_i = e_i$ to binding $f_j = e_j$ in the graph whenever there is a call to f_j inside e_i, or in other words, the definition of f_i *uses* f_j. This information is easily generated by the compiler at the scope-analysis phase. This graph is called the **Static Call Graph**.

The algorithm for finding the strongly connected components partitions the set of nodes into groups so that every pair of nodes from a group is inside some cycle; in other words, the function bindings corresponding to the nodes of a group are all mutually recursive. Considering each group as one supernode (using the fixpoint tuple binding transformation as described above), the graph becomes a directed acyclic graph (DAG) which can then be topologically sorted [1] bottom up. This numbers the set of nodes so that the nodes being *used* by other nodes appear earlier in the numbering than those using them. This ordered list of bindings (corresponding to each node in the above numbering) is then directly used to generate a series of nested `let` bindings in the mini-language syntax.

It is a simple matter to see that the same steps can also be applied to the internal blocks as well as to the main expression of the Id kernel program (which is simply a block). Thus, at each block level all the bindings are collected and transformed into nested `let` statements using the above procedure. Finally, embedding the translated main expression inside the innermost top-level `let` binding yields a giant mini-language expression corresponding to the original Id kernel program.

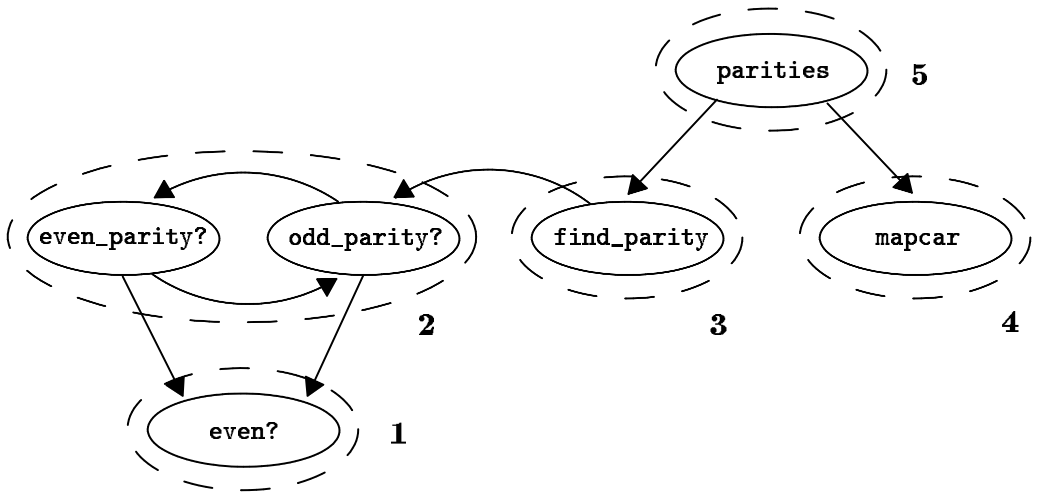
2.4.3 A Translation Example

The above procedure is illustrated in figure 2.8 with a somewhat contrived example. Here, we wish to compute parities (*odd* or *even*) of a list of numbers (refer to figure 2.8 (a)). The function `parities` uses `mapcar` given earlier to map `find_parity` over the given list. The parity is computed by a pair of mutually recursive functions `odd_parity?` and `even_parity?` using a test for evenness (`even?`). The above set of functions could be used as shown in the example

```
def parities l = mapcar find_parity l;
def find_parity n = if odd_parity? n then "ODD" else "EVEN";
def odd_parity? 0 = false
 |..odd_parity? n =
    if even? n then (odd_parity? (n/2)) else (even_parity? ((n-1)/2));
def even_parity? 0 = true
 |..even_parity? n =
    if even? n then (even_parity? (n/2)) else (odd_parity? ((n-1)/2));
def even? n = (floor (n/2)) * 2 == n;

parities (2:5:nil) ⟹ "ODD":"EVEN":nil
```

(a): Id code.



(b): Static Call Graph.

```
let even? = λn.(== (* (floor (/ n 2)) 2) n)
in let odd_parity?,even_parity? =
    fix λo,e.
        λn. (cond (== n 0) false (cond (even? n) (o (/ n 2)) (e (/ (- n 1) 2)),
        λn. (cond (== n 0) true  (cond (even? n) (e (/ n 2)) (o (/ (- n 1) 2))
   in let find_parity = λn. (cond (odd_parity? n "ODD" "EVEN")
      in let mapcar =
          fix λm. λf. λl.
               (cond (== l nil) nil
                     (let x,xs = hd l,tl l in (cons (f x) (m f xs))))
         in let parities = λl. (mapcar find_parity l)
            in parities (cons 2 (cons 5 nil))
```

(c): Mini-Language Code.

Figure 2.8: Translating a program in Id Kernel to that in Mini-Language.

(a): Id code.	(b): Mini-Language code.

`def id_length nil = 0`
`  |  id_length (_:l) = 1 + id_length l;`
 	
`id_length (true:nil), id_length (2:nil)`
	`let id_length =`
	`  fix λf.`
	`    (λx.`
	`      (cond (== x nil)`
	`        0`
	`        (let l = tl x`
	`         in`
	`           (+ 1 (f l)))))`
	`in`
	`  (make_2_tuple`
	`    (id_length (cons true nil))`
	`    (id_length (cons 2 nil)))`

Figure 2.9: List length function and its translation into the Mini-language.

query. The static call graph appears in (b). The graph contains a node for each top-level definition and edges to reflect the calling dependency. The graph has been partitioned into its strongly connected components (enclosed within dashed lines) and the components have been assigned a topologically sorted order. Finally, in (c) we use this sorted ordering to generate the translation into the mini-language.

We will come back to this process of finding the strong components and then topological sorting bindings at the top-level in the next chapter when we consider an incremental compilation environment. There we will show how to generate the call graph incrementally, since we will have only one top-level definition binding to process at a time.

2.4.4 An Inference Example

Now we will present a simpler example and show the mechanism of inference in detail. Our program will be in standard Id syntax, while its inference will be in the form of a derivation tree on the mini-language syntax. With the above discussion on translation in mind, we can do so freely.

Consider the `id_length` function in figure 2.2 (c), which is reproduced in figure 2.9 for convenience along with its translation into the mini-language. Primitive functions have been used freely to enhance clarity.

The inference tree appears in figure 2.10. The syntactic terms, the type environment, and the type expressions have been abbreviated appropriately for clarity and their expansions appear in figure 2.11.

$$TE_4 \vdash \mathbf{cons} : int \to (list\ int) \to (list\ int) \quad TE_4 \vdash 2 : int$$

$$TE_4 \vdash \mathbf{cons}\ 2 : (list\ int) \to (list\ int) \quad TE_4 \vdash \mathtt{nil} : (list\ int)$$

$$TE_4 \vdash \mathtt{id\_length} : (list\ int) \to int \qquad TE_4 \vdash e_{20}\ \mathtt{nil} : (list\ int)$$

$$TE_4 \vdash \mathbf{cons} : bool \to (list\ bool) \to (list\ bool) \quad TE_4 \vdash \mathtt{true} : bool$$

$$TE_4 \vdash \mathbf{cons}\ \mathtt{true} : (list\ bool) \to (list\ bool) \quad TE_4 \vdash \mathtt{nil} : (list\ bool)$$

$$TE_4 \vdash \mathtt{id\_length} : (list\ bool) \to int \quad TE_4 \vdash e_{19}\ \mathtt{nil} : (list\ bool)$$

$$TE_4 \vdash \mathtt{make\_2\_tuple} : int \to int \to (int, int) \quad TE_4 \vdash \mathtt{id\_length}\ e_{18} : int$$

$$TE_4 \vdash \mathtt{make\_2\_tuple}\ e_{16} : int \to (int, int) \qquad TE_4 \vdash \mathtt{id\_length}\ e_{17} : int$$

$$TE_3 \vdash + : int \to int \to int \quad TE_3 \vdash 1 : int \qquad TE_3 \vdash f : (list\ t) \to int \quad TE_3 \vdash \mathtt{l} : (list\ t)$$

$$TE_3 \vdash +\ 1 : int \to int \qquad TE_3 \vdash f\ \mathtt{l} : int$$

$$TE_2 \vdash \mathtt{tl} : (list\ t) \to (list\ t) \qquad TE_2 \vdash x : (list\ t)$$

$$TE_2 \vdash \mathtt{tl}\ x : (list\ t) \qquad TE_2 + \{\mathtt{l} \mapsto (list\ t)\} \vdash e_{12}\ e_{13} : int$$

$$TE_2 \vdash \mathbf{eq} : (list\ t) \to (list\ t) \to bool \quad TE_2 \vdash x : (list\ t)$$

$$TE_2 \vdash \mathbf{eq}\ x : (list\ t) \to bool \quad TE_2 \vdash \mathtt{nil} : (list\ t)$$

$$TE_2 \vdash \mathbf{cond} : bool \to int \to int \to int \quad TE_2 \vdash e_{11}\ \mathtt{nil} : bool$$

$$TE_2 \vdash \mathbf{cond}\ e_{10} : int \to int \to int \qquad TE_2 \vdash 0 : int$$

$$TE_2 \vdash e_7\ 0 : int \to int \qquad TE_2 \vdash \mathtt{let\ l\ =}\ e_8\ \mathtt{in}\ e_9 : int$$

$$TE_1 + \{x \mapsto (list\ t)\} \vdash e_5\ e_6 : int$$

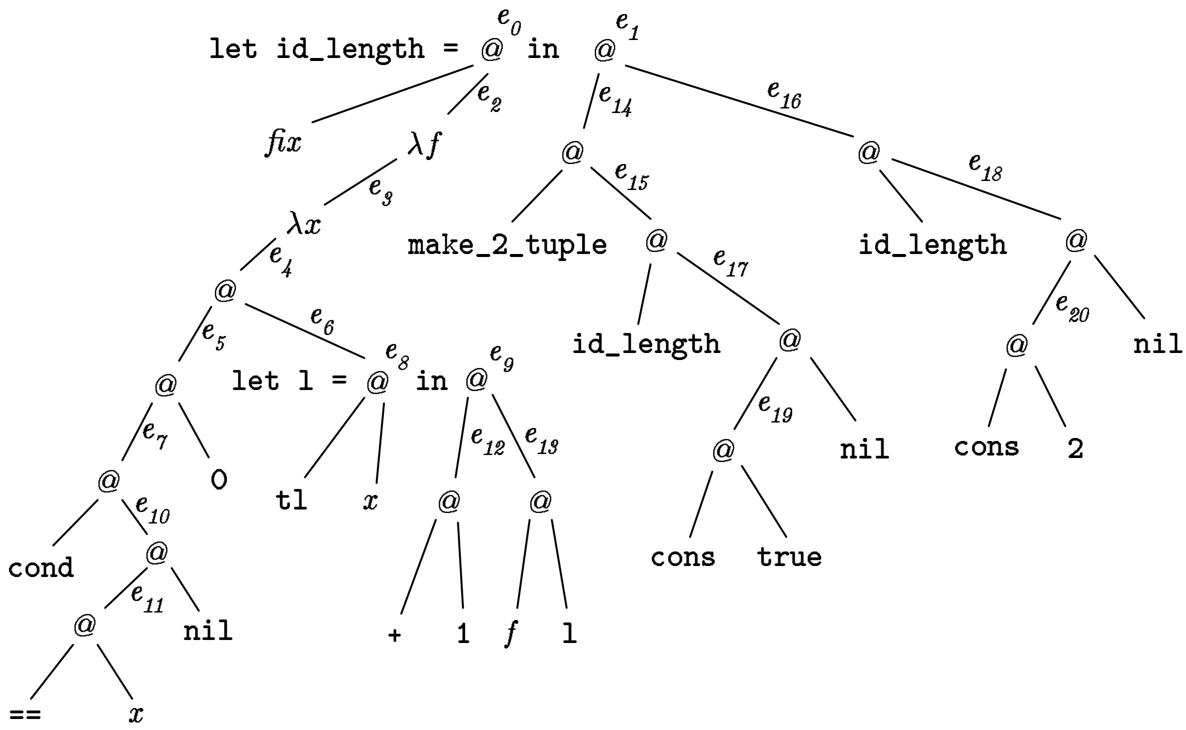
$$TE + \{f \mapsto \tau_1\} \vdash \lambda x.\, e_4 : \tau_1$$

$$TE \vdash \mathit{fix} : (\tau_1 \to \tau_1) \to \tau_1 \qquad TE \vdash \lambda f.\, e_3 : \tau_1 \to \tau_1$$

$$TE \vdash \mathit{fix}\ e_2 : \tau_1 \qquad TE{+}\{\mathtt{id\_length} \mapsto \forall t.(list\ t) \to int\} \vdash e_{14}\ e_{15} : int, int$$

$$TE \vdash \mathtt{let\ id\_length\ =}\ e_0\ \mathtt{in}\ e_1 : int, int$$

Figure 2.10: Inference tree for the list length example.



The meaning of subexpressions (bottom-up)

$$
\begin{aligned}
e_0 &= \mathit{fix}\ e_2 \\
e_2 &= \lambda f.\, e_3 \\
e_3 &= \lambda x.\, e_4 \\
\tau_1 &= (\mathit{list}\ t) \rightarrow \mathit{int} \\
TE_1 &= TE + \{f \mapsto \tau_1\} \\
e_4 &= e_5\, e_6 \\
TE_2 &= TE_1 + \{x \mapsto (\mathit{list}\ t)\} \\
e_5 &= e_7\ 0 \\
e_6 &= \texttt{let l = } e_8 \texttt{ in } e_9 \\
e_7 &= \textbf{cond}\ e_{10} \\
e_{10} &= e_{11}\ \textbf{nil} \\
e_{11} &= \textbf{eq}\ x \\
e_8 &= \textbf{tl}\ x
\end{aligned}
$$

$$
\begin{aligned}
e_9 &= e_{12}\, e_{13} \\
TE_3 &= TE_2 + \{\texttt{l} \mapsto (\mathit{list}\ t)\} \\
e_{12} &= \texttt{+}\ 1 \\
e_{13} &= f\ \texttt{l} \\
e_1 &= e_{14}\, e_{15} \\
TE_4 &= TE + \{\textbf{id\_length} \mapsto \forall t.\, (\mathit{list}\ t) \rightarrow \mathit{int}\} \\
e_{14} &= \textbf{make\_2\_tuple}\ e_{16} \\
e_{15} &= \textbf{id\_length}\ e_{17} \\
e_{16} &= \textbf{id\_length}\ e_{18} \\
e_{17} &= e_{19}\ \textbf{nil} \\
e_{19} &= \textbf{cons true} \\
e_{18} &= e_{20}\ \textbf{nil} \\
e_{20} &= \textbf{cons}\ 2
\end{aligned}
$$

Figure 2.11: Syntactic abbreviations for the inference tree.

The tree should be read bottom-up which actually corresponds to a top-down inference strategy. The type of the whole program has been inferred as (int, int) at the bottom-most inference for the outermost `let` construct. The initial type environment TE contains type bindings for all the primitive functions. Note the instantiations of polymorphic primitive functions like `cons`, `nil`, `cond`, and the *fix* operator. Also note that the LET rule of inference does not allow closure on the free type variable t inside the definition of `id_length`, while it permits it for the outermost `let` construct. There, it is no longer free and can be safely closed, giving rise to the polymorphism exhibited by `id_length` in its two applications.

2.5 Discussion

Restricting quantification in a type scheme to only the outermost level is indeed a restriction on the class of well-typed programs accepted by this type system. Type systems for Second-order Lambda Calculus accept a strict superset of this class. The quantifications, in that case, may be anywhere in the type expression, not just at the outermost. To give an example, consider the following program fragment which fails to type-check in this system[14]:

```
(λf. if (f 1) = 1 then (f true) else (f false))(λx. x)
```

Here, the argument of the first lambda-abstraction is expected to be a polymorphic function, so that its type would have an embedded quantifier and this would not be allowed in this system. But, if the application argument is statically known to be a polymorphic function, we can indeed take advantage of this system by declaring it in a `let` construct as follows:

```
let id = λx. x
in
  if (id 1) = 1
       then (id true)
       else (id false)
```

As we have seen earlier, the polymorphism is initiated by a `let` declaration for an identifier. Then, wherever that identifier is used in the body of the `let` declaration, we can treat it as polymorphic in the sense described above. So, the above program fragment type-checks, because the use of *id* in the predicate instantiates it to the type $int \rightarrow int$, while its use in the arms of the conditional instantiate it to the type $bool \rightarrow bool$.

Indeed, this may seem to be a spurious kind of polymorphism. The astute reader can see that this is exactly equivalent to the situation where we interpret the `let` construct as a macro

[14] "`if` e_1 `then` e_2 `else` e_3" is used instead of "`cond` e_1 e_2 e_3" to enhance readability.

and perform textual substitution of the declared identifier with its binding everywhere within the body and then proceed to type check with only ground types. So, the above example would macro-expand to:

```
if ((λx. x) 1) = 1
   then ((λx. x) true)
   else ((λx. x) false)
```

Here, each identity function can be individually monomorphically typed to the desired type as before.

So there is no active or dynamic polymorphism in this system in the sense of Girard-Reynolds second-order polymorphic Lambda Calculus [13, 23, 53] and indeed the class of programs accepted by this system is exactly the same as that of simply typed first-order Lambda Calculus. But this system offers it in a package that gives the appearance of (static) polymorphism.

The down side of this succinct representation of polymorphic programs is that the worst-case running complexity of this type inferencing mechanism is provably exponential. There are very recent results in the literature to show that the Hindley/Milner type inference system is DEXPTIME-*complete* [31, 38]. This gives a tight bound over the previous lower bound result of PSPACE-*hard* by Mitchell [29]. These results are contrary to the common folklore that this system is fairly efficient in practice, and came as a surprise to the programming language community.

The point is that this system has much more to offer to a compiler writer than simply an exponential time type inferencing mechanism for a restricted subset of polymorphic typed Lambda Calculus. The system allows the programmer to dissociate the declaration from its usage in a polymorphic setting in a general, safe, and systematic manner, leading the way to resolving important pragmatic issues of incremental compilation and type-checking, separate compilation, and modularization. It is for these pragmatic reasons that this system is now one of the most widely used type inferencing mechanisms for modern programming languages. Once the proper mechanisms get established, other advantages crop up too, as they fit well in the same philosophy. This includes the treatment of overloaded operators and polymorphic references. We will see these extensions in later chapters.

Chapter 3

Incremental Type Inference

A reasonably rich type system equipped with features like polymorphism and type inference as we saw in the last chapter offers much more programming convenience over PASCAL-like languages. Still, this leaves a lot to be said for our goal of a flexible program environment that integrates these features seamlessly with incremental program development, testing, and editing, requiring minimal effort on part of the user or the system while maintaining overall consistency and correctness of the program being developed.

System designers have approached this problem in various ways. Some systems support separate compilation of physical partitions (such as files) of a large program that allows them to maintain consistency among the various components automatically, given some inter-partition dependency information; the UNIX[1] *make* facility [21] is an example. But then, it is the user's responsibility to setup the inter-dependence explicitly and the system has no automatic knowledge of the actual dependency among the partitions. The partitioning itself is artificial and has no default logical relationship with the actual code dependencies in the program. It is the user's responsibility again to group the logically related procedures into the same physical file partition in order to use this facility meaningfully. Furthermore, the violation of the dependencies is based, pessimistically, on the date/time of their creation, instead of some syntactic or semantic consistency check involving the partitions. Thus, even a simple operation such as reading a file and writing it back may cause a violation of its dependencies and trigger a recompilation of the entire application.

Some programming environments such as that for Miranda[2] [60, 61] ensure that each physical file partition (called a Miranda-*script*) also represents an independent logical partition of the

[1] "UNIX" is a trademark of Bell Laboratories.
[2] "Miranda" is the trademark of Research Software Ltd.

complete system by requiring the user to explicitly declare its logical interface: the names and types of objects and procedures to be exported or imported etc. Other structured programming languages like ML or CLU provide independent language constructs for logical partitioning of large programs into *modules* [37, 25] or *clusters* [36] respectively. But such partitioning is not geared towards fast, interactive and incremental program development. Such partitioning helps in overall structuring of a large system and individual maintainence of each of its logical components but is still too coarse (usually consisting of several definitions per module) for the purpose of interactive and incremental manipulation of individual definitions contained within. Also, these systems cannot handle debugging and testing of incomplete programs; all imported objects in an interface specification are expected to be present before anything in that logical partition can be executed, again imposing unnecessary ordering requirements on the interactive user during the program development phase.

As far as we are concerned, an **incremental compilation** system should allow the user to partition his/her program into several, small and independent **compilation units** that may be provided in arbitrary order, while it automatically generates inter-unit dependency information and maintains consistency among them. The information should be inferred from the program text and may be continually updated as the user develops or edits the program. The size of the compilation unit is a key factor in the flexibility offered by such a system.

LISP programming environments have turned out to be good role models for the desired programming ease. The unit of compilation in these systems is a single top-level definition; a higher-order, function oriented[3] style of programming keeps their size small and easily manipulable. These systems allow the user to furnish a sequence of definitions constituting the program, one by one and in arbitrary order resolving global references to other definitions automatically by dynamic linking. They allow editing and testing parts of an incomplete program or debug those parts that are incorrect, without worrying about the status of the rest of the program. Such situations are common during program development and debugging/testing. Higher level partitioning into modules and clusters or physical partitioning into files is still possible and is orthogonal to the issue of providing this definition level incrementality. Contrast this with a PASCAL-like programming environment where a complete, correctly compiled program has to be provided before any of its components can be exercised and even a small amount of editing in some part calls for a recompilation of the whole program.

[3]This may or may not be *functional* which generally implies being side-effect free.

In a dynamically typed system such as LISP, providing this flexibility is not too difficult since most of the inter-procedural consistency checking is postponed until the run time, which is also the earliest possible time when we detect a type-error. But for a compiler of a strongly typed language aiming at static type analysis, as we have in Id, it is clear that this incrementality means extra work in maintaining interprocedural consistency which gets further complicated due to type inference, polymorphism and higher-order functions.

It is our intention in this thesis to incorporate LISP-like programming incrementality within the framework of Hindley/Milner static type inference mechanism. Nikhil [44] raised this problem in the context of ML programming environment and outlined a scheme for incremental polymorphic type inference that we have concretized and implemented here for the programming environment of Id. In particular, we will show how to modify the Hindley/Milner type system to work in an incremental fashion, while still retaining its nice properties of soundness and completeness. Our unit of compilation is a single top-level definition just as in LISP[4]. A complete Id program is a sequence of user defined functions followed by a main expression to evaluate (refer to section 2.4). The sequence of definitions, input through the editor, are allowed to be incrementally compiled and loaded into the system. The run time environment then simply becomes an interactive extension of the editor-compiler giving the flexibility of an interpretive environment with the execution speed of compiled code to form an edit-compile-execute loop.

The rest of the chapter is organized as follows. First in section 3.1, we describe a general framework for incremental compiler analysis. In section 3.2, we describe the basic mechanism of our incremental Hindley/Milner type inference system. A proof of the correctness of this system is developed in sections 3.3 through 3.6. Extensions to the basic system and a discussion of its complexity appear in sections 3.7 and 3.8 respectively. Finally in section 3.9, we discuss some optimizations and general features our system, describe related work in this field, and report the current status of our implemention.

3.1 Incremental Analysis

Before we describe the incremental type inference algorithm, it is useful to introduce a general notion of incremental book-keeping and analysis of compile-time properties that we will use

[4]There are other systems such as GLIDE by Toyn *et al.*[58] that carry the same incremental philosophy to an even finer grain: for each subexpression. We will describe such related work at the end of this chapter.

throughout the rest of this chapter.

Equation 2.3 expresses a complete program as a giant expression with top-level definitions appearing in nested `let` expressions. In section 2.4 we went into some detail of translating the Id-kernel language into this mini-language. However, in an incremental system with the grain of incrementality being a single definition, this translation is not always possible at the top-level. The user may supply one or few definitions at a time in some arbitrary order and refer to definitions that are as yet undefined. We need to get a good handle on maintaining partial information for the compilation properties of the top-level definitions. So, we extend the grammar of our mini-language in order to model this scenario more closely.

3.1.1 Incremental Mini-Language

We introduce the notion of top-level, independent bindings as a unit of compilation. A complete program now is a set of such bindings. As before, we will freely (and informally) use tuple expressions and tuple bindings in this extended mini-language as well[5]. The extended mini-language appears below.

$$\begin{array}{rcll} e \in \text{Expressions} & ::= & c & (3.1) \\ & | & x & \\ & | & \lambda x.\, e_1 & \\ & | & e_1\ e_2 & \\ & | & \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 & \\ B \in \text{Bindings} & ::= & x = e & (3.2) \\ P \in \text{Programs} & ::= & B_1;\ B_2; \cdots;\ B_n;\ \mathtt{it} = e & (3.3) \end{array}$$

The expression syntax remains the same, but a program, instead of being a giant nested expression rearranged according to our static call graph transformation (section 2.4.2), now consists of an explicit sequence of top-level bindings followed by an expression query. This means that top-level mutually recursive definitions are no longer grouped together in an application of the fixpoint operator to a functional as was done in section 2.4.2. Our incremental algorithm will take care of that. This setup resembles the organisation of a program in the Id kernel language (figure 2.7). The difference is that the final query is expressed here as a special

[5] As stated in section 2.4, this does not affect the Soundness and Completeness properties of our type inference system (theorems 2.5, 2.7 and 2.8) because tuples can always be simulated by the function type constructor ($\rightarrow$) alone.

binding to a predefined variable "`it`"[6] which is also evaluated after being compiled within the current environment. The RHS of each top-level binding is, as before, completely translated into an expression in the mini-language, which means that mutually recursive internal bindings are still converted into an application of the fixpoint operator to a functional constructed out of those bindings.

3.1.2 Incremental Book-Keeping

The basic idea in incremental compilation is to be able to compute some desired compile-time properties for an aggregate of identifiers in an incremental fashion. This aggregate forms the **identifier namespace** that we operate in. For our purposes, this is the set of all top-level identifiers.

The union of all the property mappings of namespace identifiers to their property-values constitutes a **compilation environment**. The goal is to incrementally accumulate information into the compilation environment about all the desired properties of all the relevent identifiers by successive refinements without having to look at everything at the same time.

We distinguish between two kinds of properties. Some *local* properties may be computed independently and locally within a single unit of incremental compilation, as in the case of finding the free identifiers used in the body of a top-level definition. Others are *global* properties that might require information on the same or other properties of other identifiers in order to be computed. The type of top-level identifiers is an example of such a property. Obviously, it is the latter that make incremental book-keeping non-trivial because of the non-local interdependence among the identifiers. We introduce a general book-keeping mechanism to keep track of all such interdependences[7].

First, we formally define what constitutes a property.

Definition 3.1 *A* **property** $\mathcal{P} = (\mathcal{D}, \sqsubseteq)$ *is characterized by a domain of values* $\mathcal{D}$ *and a binary predicate test* $\sqsubseteq$*. Given two values,* $v_1, v_2 \in \mathcal{D}$ *for a property* $\mathcal{P} = (\mathcal{D}, \sqsubseteq)$*,* $v_1 \sqsubseteq v_2$ *holds if and only if* v_1 *is consistent with* v_2 *according to the property we want to characterize.*

The domain of values is simply a syntactic set of values with some structural relationship defined among its elements through the predicate relation (see figure 3.1 (a)). The predicate

[6] This is adapted from the ML interactive runtime environment where the last expression evaluated is referred to as "`it`".

[7] A variation of the mechanism described here was first implemented (without formal characterization) as part of the Id Compiler by Traub [59].

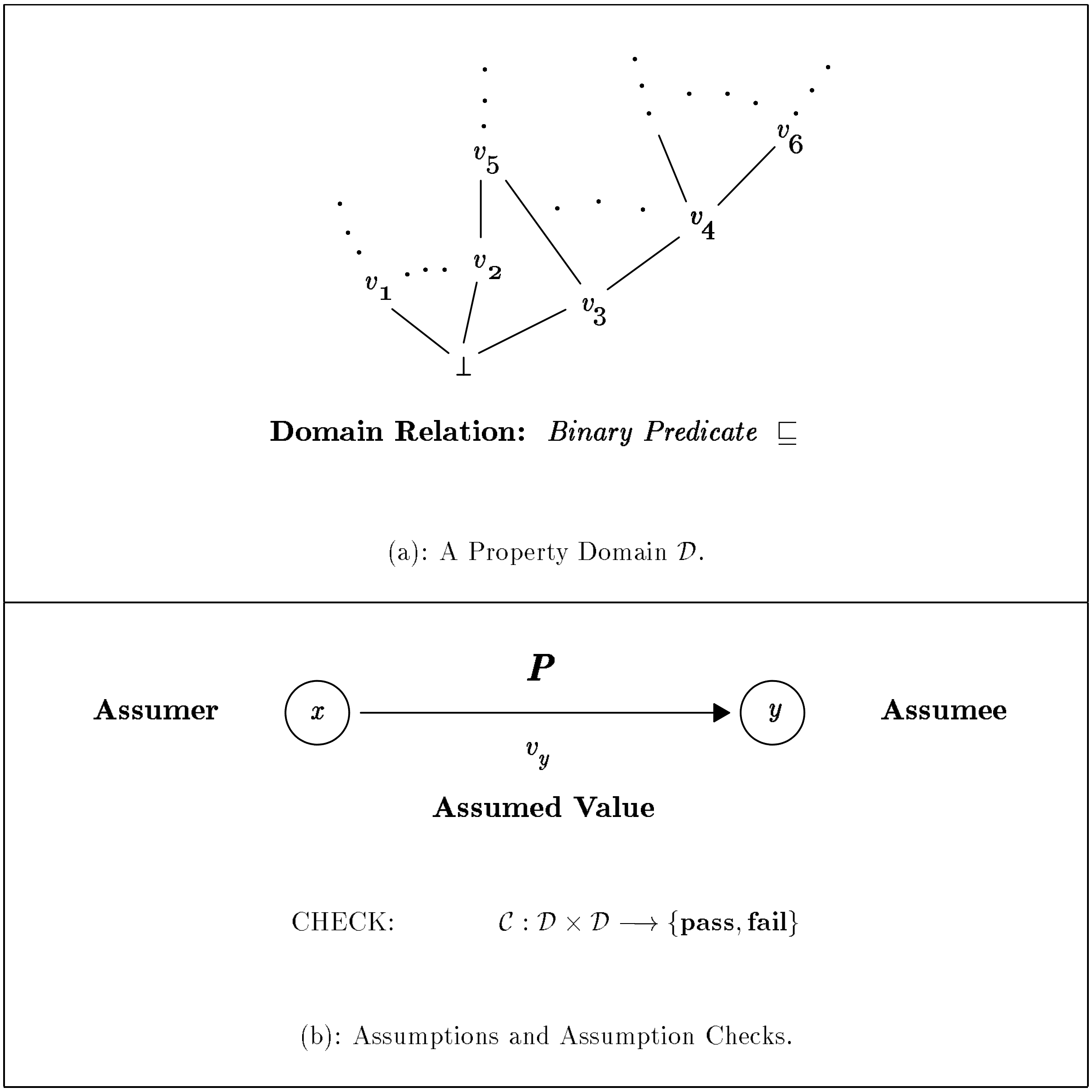


Figure 3.1: A general framework for Properties and Assumptions.

actually defines the structure of the domain according to whatever notion of consistency we want to impart to the desired property. For example, the compilation property denoting the syntactic arity of function definitions is defined to be consistent if each function identifier is used with the same arity throughout the program. Thus, the property will be characterized by the domain of natural numbers along with the predicate test for equality. The domain also contains a special element "$\perp$" (read "bottom") that corresponds to the default property value assigned to as yet undefined or unknown identifiers in the namespace.

The test of consistency is useful, for instance, in comparing the values of some global property of a particular object before and after a certain event. For example, we can compare the type of a function before and after its free identifiers get defined; the test in this case would be a subsumption (instantiation) test between the involved type-schemes. Also note that the test is irrelevent for local properties because by definition, no external event can affect a local property of an identifier. Once computed, the value of a local property remains the same unless it is re-computed. In such cases we will ostensibly record the test to be that of equality[8].

The interdependences among the global properties of identifiers at various times during incremental compilation is maintained *via* sets of assumptions (see figure 3.1 (b)).

Definition 3.2 *An* **assumption** $(x, y, v_y) \in A_{\mathcal{P}}$ *made by an* **assumer** *identifier* x *is a record of an acceptable value* $v_y \in \mathcal{D}$ *of some property* $\mathcal{P} = (\mathcal{D}, \sqsubseteq)$ *of another* **assumee** *identifier* y*. Associated with an assumption set, is a binary predicate* **assumption check** C*, that verifies a subsequent actual value* v'_y *of the assumee identifier against its previously assumed value* v_y *for consistency/compatibility. This check may use the property predicate* $\sqsubseteq$ *for this purpose.*

A set of assumptions for a given property $\mathcal{P} = (\mathcal{D}, \sqsubseteq)$ can be represented in any one of the following three isomorphic forms each of which identifies the relationship among the assumer identifier, the assumee identifier, and the assumed property value.

- As a set of triples of assumer, assumee, and the assumed value:

$$A_{\mathcal{P}} \in \mathit{powerset}(\text{Identifier-Namespace} \times \text{Identifier-Namespace} \times \mathcal{D})$$

- As a map from assumer to a set of assumee and assumed value pairs:

$$A_{\mathcal{P}} \in \text{Identifier-Namespace} \xrightarrow{fin} \mathit{powerset}(\text{Identifier-Namespace} \times \mathcal{D})$$

[8]Usually the meaning of a test for equality is clear from the property domain; in case of structured objects we usually opt for structural equality rather than pointer equality.

- As a map from assumer to another map from assumee to its assumed value:

$$A_{\mathcal{P}} \in \text{Identifier-Namespace} \xrightarrow{fin} (\text{Identifier-Namespace} \xrightarrow{fin} \mathcal{D})$$

The assumption check C is always a binary predicate denoted by $C \in (\mathcal{D} \times \mathcal{D}) \rightarrow \{\textbf{pass}, \textbf{fail}\}$. We separate the assumption check C from the property predicate $\sqsubseteq$ because several assumption domains may be associated with the same property using different assumption compatibility checks that are all derivatives of the same property predicate test. We will see examples of this later on.

By definition, each property generates a property environment as a map from the identifier namespace to the property domain, which becomes a part of the overall compilation environment. The sets of assumptions associated with each property make up the book-keeping overhead of the compilation environment.

3.1.3 Overall Plan for Incremental Analysis

The overall scheme for incremental property computation and maintenance appears in figure 3.2. We undertake the following steps.

1. First, we initialize the compilation environment for the incremental computation and accumulation of each of the desired compilation properties.

2. Next, for each new or edited binding, we compute its properties and collect the appropriate property assumptions. While computing properties for a top-level binding, the identifier on the LHS is taken to be the assumer and it can make assumptions about the various properties of the free identifiers in the RHS.

3. After editing and compiling some definitions we may wish to evaluate a given expression query. Before we do that, all the assumptions for each of the top-level identifiers are checked for consistency against their latest property values using the associated assumption checks. Success in the test indicates an agreement between the expected and the actual property value and the compilation is considered successful. A failure indicates a non-trivial discrepency between the compilation environment existing at the time when the assumer was compiled and the latest environment. This calls for an error message to be flagged or a recompilation[9] of the assumer within the latest environment.

[9] We use the term *recompilation* somewhat loosely here. Usually it refers to a re-analysis of the involved definition. We will characterize what constitutes a recompilation more concretely in section 3.9.

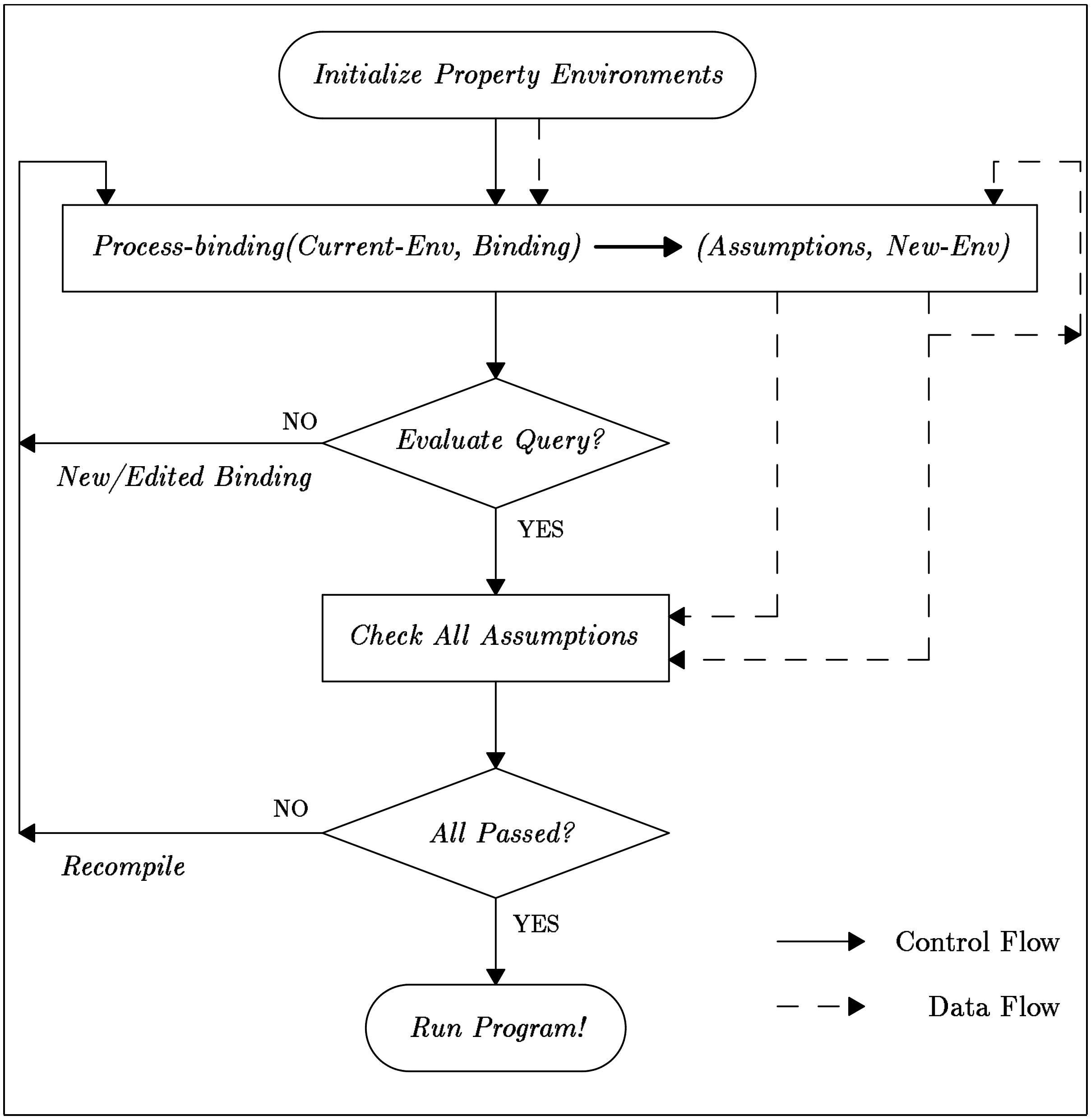


Figure 3.2: Overall Plan for Incremental Property Maintenance.

4. Finally, when all the collected assumptions have passed their checks successfully, then we allow the evaluation of the expression query to proceed. The incremental property maintenance system guarentees that all the desired compilation properties have been computed correctly, *i.e.*, the property values for each of the top-level identifiers are exactly same as if the complete program had been compiled all together.

The above operational description of the incremental book-keeping mechanism is necessarily quite high-level. The success or failure of this mechanism in computing the desired properties correctly and efficiently depends upon several factors such as the semantic characteristics of each property domain and how it relates to other domains, the internal structure of the property domain governed by the binary relation, the computation algorithm for the property, the kinds of assumptions collected and the assumption checks performed for each domain, and the incremental compilation environment history and its maintainence. We show the correctness of this incremental system in computing the types of identifiers within the well defined framework of Hindley/Milner type inference system.

3.2 The Incremental Type Inference System

Our goal is to design an incremental type inference system that guarantees that the types of all top-level identifiers in a given program are computed correctly. Here correctness implies that the incremental system infers exactly the same type-schemes for each of the top-level identifiers in a given Id program as the simplistic complete program approach described in section 2.4, using the basic type inference system of the last chapter.

We will present our system in several layers. First, we present a basic incremental type inference system and subsequently show its correctness. We will make several simplifying assumptions and ignore complexity considerations in the basic system in order to show its correctness easily. In later sections, we will relax these restrictions and describe optimizations that maintain correctness.

3.2.1 Components of the Basic System

We make the following assumptions in our basic system.

Assumption 1 *The unit of compilation is a single top-level definition (single top-level binding in the mini-language). We also assume that each binding is processed in its own separate* *invocation of the compiler*[10].

To achieve correctness in the sense mentioned above, we must be able to reject the programs that generate a type error when typed under the complete program system and also compute the same types for each top-level identifier when given a complete and type-correct program. Note that we can only make a correspondence claim for complete and type-correct programs: complete because that is the only way to verify the computed types against the simplistic complete program approach, and type-correct because we are not guaranteed to find the same type-errors since the two systems may have different ordering of top-level bindings[11]. Thus in showing this correspondence, we make the following additional assumptions.

Assumption 2 *We assume that all nodes in the complete static call graph are type-checked at least once in the incremental system.*

This means that eventually the incremental system gets to look at all the information necessary (the complete program) even though it looks at it incrementally. This constraint is easily met by explicitly detecting the end of a complete program. We will incorporate such a test later in section 3.7. For the basic system, it is the user's responsibility to supply the complete program.

Assumption 3 *We assume that the top-level definitions are not edited between the individual invocations of the compiler on each one of them.*

This only means that the top-level definitions already supplied are not allowed to change interactively. This restriction will be relaxed later on as well.

Compilation Properties

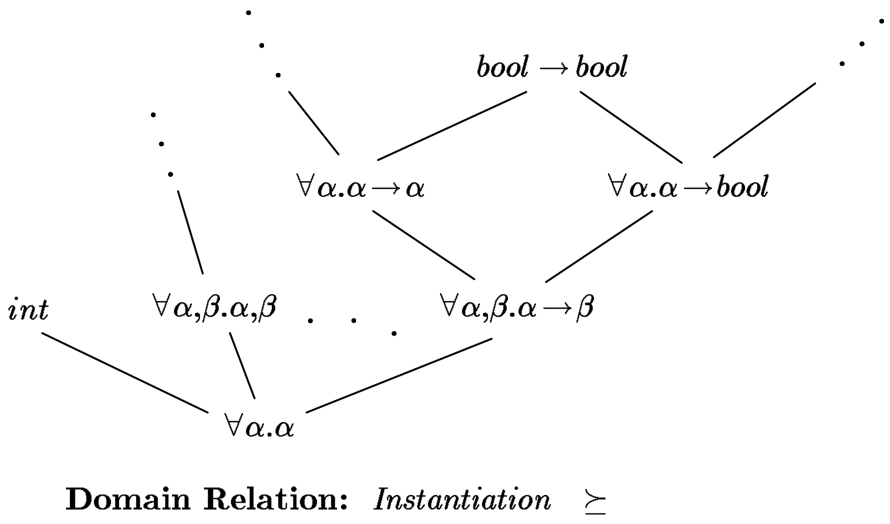
We maintain the following compilation properties in our system.

1. Global property *Type* = (Type-Schemes, $\succeq$) (see figure 3.3 (a)). This property records the type[12] of all the identifiers. The domain is structured using the instantiation relation

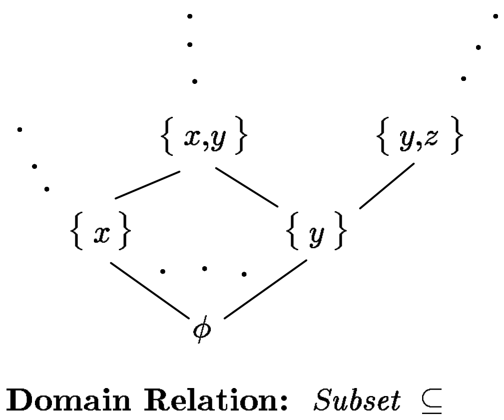
[10] A group of bindings can be handled easily by invoking the compiler on each binding one by one.

[11] In fact, it is possible to run into different type-errors with the same program if the order of definitions is altered. This is because that the place where a type-error is detected depends on the exact sequence of unifications performed during type-checking.

[12] Note that the word *type* is used here as the name of a compiler property which, in fact, corresponds to the property domain of Type-Schemes.



(a): Type Property Domain.



(b): SCC Property Domain.

Figure 3.3: Compilation Properties used in Algorithm W_1.

$\succeq$ (definition 2.7). The "bottom" element of the domain is the most general type-scheme $\forall\alpha.\alpha$. The main goal of our type inference system is to compute this property incrementally for all the top-level identifiers. The mechanism of computation of this property for the complete program was described in sufficient detail in the last chapter. As defined by equation 2.8, we denote the mapping of identifiers to their types in a Type-Environment (TE).

2. Global property *Strongly-Connected-Components* = (*powerset*(Identifiers), $\subseteq$) (see figure 3.3 (b)). This is the second global property that we need to compute. The *strongly connected component* (SCC) of an identifier is the set of all identifiers in the static call graph that are mutually recursive with the given identifier. The domain of this property is the powerset (set of all subsets) of the set of all top-level identifiers. Therefore, it is appropriately structured using the subset relation for sets of symbols (identifiers). We denote the mapping of the identifiers to their recursive sets in a SCC-Environment (SE).

$$SE \in \text{SCC-Environments} = \text{Identifiers} \xrightarrow{fin} \mathit{powerset}(\text{Identifiers}) \tag{3.4}$$

There are other properties that are useful in computing the above properties and we may record them in the compilation environment as well. Such properties will be computed and recorded in compiler phases preceding the type inferencing stage. In particular, we need to compute the set of free identifiers (FId) occuring in each top-level binding in the scope-analysis phase of the compiler. This is a local property and is necessary in order to reconstruct the static call graph and subsequently its strongly connected components. We will not worry about such outside properties and focus our attention only to those computed during the type inferencing phase as enumerated above.

Property Assumptions

We maintain a set of type assumptions (M) to record the actual type instance of each use of a free generic identifier in a top-level definition (see figure 3.4). The final type-scheme of the assumee identifier is expected to be at least as polymorphic as to be able to generate all these type instances. In some sense, these type instances describe an upper boundary within the type-scheme domain that the final type-scheme of the assumee identifier must be able to instantiate. So, we record this boundary[13] to verify it later against the actual type scheme of

[13] It is possible to record a single type-scheme instead, which acts as the greatest lower bound (glb) of this

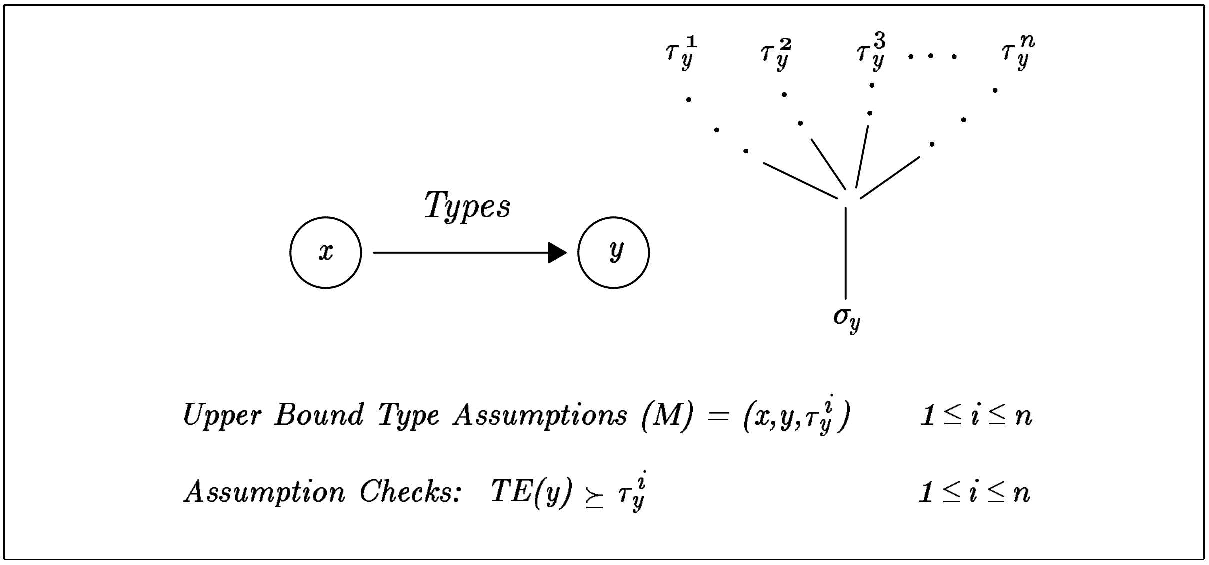


Figure 3.4: Upper Bound Type Assumptions (M) used in Algorithm W_1.

the assumee identifier using the subsumption test on types. If the final assumee type scheme fails to instantiate any of these upper bounds, we can conclude that the upper bounds for the assumer were collected under overly relaxed (polymorphic) or substantially different type environment and the assumer identifier should be recompiled.

$$
\begin{aligned}
& M \in \text{Upper-bd-type-assumptions} = \text{Identifiers} \xrightarrow{fin} \text{Types} \\
& \text{CHECK: } \forall (y \mapsto \tau_y) \in M,\ TE(y) \succeq \tau_y
\end{aligned}
\tag{3.5}
$$

3.2.2 Mechanism of the Basic System

Following the outline of the last section (refer figure 3.2), now we describe how the properties defined above are initialized and computed.

Initialization

Our identifier namespace consists of the set $X = \{x_1, x_2, \ldots, x_n\} \cup \{\texttt{it}\}$ of identifiers bound in the LHS of the top-level bindings in a complete program as defined by equation 3.3. The initial compilation environment (TE_0, SE_0) assigns the bottom elements of the respective property

upper boundary, but we chose not to do so for simplicity.

```
Def W_1(TE_{i-1},SE_{i-1},B_x) =
    Let B_x be the binding 'x = e_x'.
    Construct the current call graph G_i using the free identifiers (FId) property.
    SCC^i_x = {z_1, ..., x, ..., z_n} be the current strongly connected component of x in G_i.
```

SCC Environment: $SE_i = SE_{i-1} \setminus_{SCC^i_x} + \{z_k \mapsto SCC^i_x\} \qquad \forall z_k \in SCC^i_x.$

for each *top-level identifier* $z_k \in SCC^i_x$ **do**

Let e_k *be the RHS expressions associated with* z_k.

Let $\beta_{k1}, \ldots, \beta_{kn}$ *be new type variables.*

New-$TE^i_k = TE_{i-1} \setminus_{SCC^i_x} + \{z_1 \mapsto \beta_{k1}, \ldots, z_n \mapsto \beta_{kn}\}$.

$W'(\textit{New-}TE^i_k, e_k) = (S^i_k, \tau^i_{z_k}, A^i_k)$.

endfor.

$$U_i = \textit{Unify-Columns}\begin{pmatrix} S^i_1(\beta_{11}) & \cdots & S^i_1(\beta_{1k}) & \cdots & S^i_1(\beta_{1n}) \\ \vdots & & \vdots & & \vdots \\ S^i_n(\beta_{n1}) & \cdots & S^i_n(\beta_{nk}) & \cdots & S^i_n(\beta_{nn}) \\ \tau^i_{z_1} & \cdots & \tau^i_{z_k} & \cdots & \tau^i_{z_n} \end{pmatrix}$$

Type Environment: $TE_i = TE_{i-1} \setminus_{SCC^i_x} + \{z_k \mapsto \textit{close}(U_i \tau^i_{z_k})\} \qquad \forall z_k \in SCC^i_x.$

Upper-bd Type Assumptions: $M_{z_k} = U_i A^i_k \setminus_{SCC^i_x} \qquad \forall z_k \in SCC^i_x.$

Return (TE_i, SE_i).

Figure 3.5: The Incremental Algorithm W_1.

domains to all the identifiers in the namespace.

$$TE_0 = TE_{lib} \cup \{x_i \mapsto \forall\alpha.\alpha\} \qquad \forall x_i \in X. \tag{3.6}$$

$$SE_0 = \{x_i \mapsto \phi\} \qquad \forall x_i \in X. \tag{3.7}$$

TE_0 also includes a standard library type environment TE_{lib} that maps all the standard library function identifiers, predefined operators etc. to their appropriate type schemes. There are no free type variables in this environment by assumption.

The Incremental Type Inference Algorithm W_1

The algorithm W_1 that processes each top-level binding by accumulating its incremental properties and generating the assumptions is shown in figure 3.5.

There are a few points to note.

1. For each binding B_x, its current static call graph G_i is constructed out of the currently available nodes. Such a graph is, in fact, a subset of the complete static call graph

```
Def W′(TE,e) =
  case e of
    x           ⇒ let
                     ∀α1 ··· αn. τ = TE(x).
                     β1, ..., βn be new type variables.
                     τ′ = {αi ↦ βi}τ.
                  in
                     (ID,τ′,{(x ↦ τ′)}).
    λx. e1      ⇒ let
                     β be a new type variable.
                     TE′ = TE + {x ↦ β}.
                     (S1, τ1, A1) = W′(TE′, e1).
                  in
                     (S1, S1β → τ1, A1\{x}).
    e1 e2       ⇒ let
                     (S1, τ1, A1) = W′(TE, e1).
                     (S2, τ2, A2) = W′(S1(TE), e2).
                     β be a new type variable.
                     S3 = U(S2τ1, τ2 → β).          (may fail)
                  in
                     (S3S2S1, S3β, S3(S2A1 ∪ A2)).
    let x = e1 in e2 ⇒
                  let
                     (S1, τ1, A1) = W′(TE, e1).
                     TE′ = S1(TE) + {x ↦ close(S1(TE), τ1)}.
                     (S2, τ2, A2) = W′(TE′, e2).
                  in
                     (S2S1, τ2, S2A1 ∪ A2\{x}).
  endcase.
```

Figure 3.6: Pseudo-code for the Inference Algorithm W'.

G_t described in section 2.4.2 and includes all and only those nodes that are currently reachable from the node corresponding to B_x.

2. For the current binding "$x = e_x$", SCC_x^i is the set of all (already defined) top-level identifiers that are mutually recursive with x. In our basic system, we treat all the bindings in a SCC as a single *supernode* of the static call graph and type all of them together. This is similar in principle to their treatment in section 2.4.2 where the bindings of a SCC are physically grouped together in a single expression. Since our system is incremental, this may require keeping track of the actual code of each definition as it is compiled. This information is not too difficult to maintain in an integrated editor-compiler environment like we have for Id.

3. We type-check the set of bindings for each $z_k \in SCC_x^i$ in two phases. First, we compute the type of the RHS expression e_k for each binding $z_k = e_k$ independent of the types of other top-level identifiers of SCC_x^i. We also collect type assumptions from the RHS expression e_k with z_k being the assumer. Then in the second phase, we unify the defined type $\tau_{z_k}^i$ so obtained for each of the bindings with the type of z_k as used while typing the other bindings. We do this operation as one giant unification of all the terms corresponding to the same top-level identifier. This operation effectively simulates the task of the fixpoint operator used in translation of Id kernel in section 2.4.2. The upper bound type assumptions are also appropriately adjusted to account for this unification.

4. The algorithm W' used internally to actually compute the type of each top-level RHS expression is shown in figure 3.6. It recursively collects all type instances generated for each use of a freely occuring identifier into a set of type assumptions for the current assumer. We will show some assertions about these collected assumptions in the next section. The free identifiers of the RHS of a top-level binding are either other user-defined top-level identifiers or pre-defined library identifiers. In either case, type assumptions corresponding to identifiers outside SCC_x^i (generic identifiers) are recorded as upper bound type assumptions. Apart from this extra book-keeping, the algorithm is exactly the same as the standard type inference algorithm W of figure 2.6. Consequently, the theorems of soundness and completeness of W (theorems 2.7 and 2.8) are also valid for W'.

3.2.3 An Example

Environment Snapshot 1:		
Type Environment	:	$TE_0 = TE_{lib} + \{\mathtt{f} \mapsto \forall\alpha.\alpha, \mathtt{g} \mapsto \forall\alpha.\alpha\}$
SCC Environment	:	$SE_0 = \{\mathtt{f} \mapsto \phi, \mathtt{g} \mapsto \phi\}$

```
def f x = (x+1):(g x);
```

Environment Snapshot 2:		
Type Environment	:	$TE_1 = TE_{lib} + \{\mathtt{f} \mapsto$ `N -> (list N)`$, \mathtt{g} \mapsto \forall\alpha.\alpha\}$
SCC Environment	:	$SE_0 = \{\mathtt{f} \mapsto \{\mathtt{f}\}, \mathtt{g} \mapsto \phi\}$
Assumptions	:	$M_f = \{$`(:)` $\mapsto$ `N -> (list N) -> (list N)`,
	:	`(+)` $\mapsto$ `N -> N -> N`, `g` $\mapsto$ `N -> (list N)`$\}$

```
def g x = x:nil;
```

Environment Snapshot 3:		
Type Environment	:	$TE_2 = TE_{lib} + \{\mathtt{f} \mapsto$ `N -> (list N)`, `g` $\mapsto$ `*0 -> (list *0)`$\}$
SCC Environment	:	$SE_0 = \{\mathtt{f} \mapsto \{\mathtt{f}\}, \mathtt{g} \mapsto \{\mathtt{g}\}\}$
Assumptions	:	$M_f = \{$`(:)` $\mapsto$ `N -> (list N) -> (list N)`,
	:	`(+)` $\mapsto$ `N -> N -> N`, `g` $\mapsto$ `N -> (list N)`$\}$
	:	$M_g = \{$`nil` $\mapsto$ `(list *0)`,
	:	`(:)` $\mapsto$ `*0 -> (list *0) -> (list *0)`$\}$

Figure 3.7: An Example to show Incremental Type Inference.

We now show an example of our incremental analysis.

Figure 3.7 shows a small Id program consisting of two function definitions `f` and `g`. We take snapshots of the compilation environment at each definition boundary. The initial environment is empty. When function `f` is compiled, its type and the SCC properties are computed and get recorded in the environment. All the assumptions made by `f` about its free identifiers are also recorded. When function `g` is compiled, similar book-keeping takes place. Note that the assumed type `(N -> (list N))` for `g` recorded in `f`'s assumption set M_f, passes the instantiation check because the defined type of `g` turns out to be `(*0 -> (list *0))`, which can instantiate the assumed type. So, the system ends up with correct types for both the functions, even though `g` was used inside `f` before it was ever defined.

3.3 Proving Correctness of the Incremental System

3.3.1 Overall Plan of Proof

Figure 3.8 gives the overall plan that we will follow in showing the correspondence between our incremental strategy and the complete program approach of section 2.4. We start with an Id kernel program as a set of top-level identifier definitions followed by a final expression query. In principle, we first need to transform this program to our mini-language syntax because our inference rules and the algorithm are geared towards that. For our purposes, we will assume that this transformation is already performed and we are supplied a sequence of mini-language bindings[14].

The difference between the incremental system and the complete program system lies in the way we organize type inferencing on each of these bindings. The incremental system compiles each individual top-level binding in succession in separate calls to W_1 as described in the last section, whereas the complete program system reorders them into one giant expression as described in section 2.4.2 and calls the algorithm W on it. Thus, the incremental system corresponds to an interactive user session, while the complete program system more resembles batch-mode compilation.

In figure 3.8, we have marked the state of the type environment at various intermediate stages of the two systems. This gives us an idea about the actual type environments under which the various bindings get type-checked in the two systems[15]. We are interested in the

[14]The actual Id Compiler does its type inferencing on the Id syntax tree directly.

[15]The actual type-checking environments could be very different from each other even for the same bindings.

Type-checking an Id Kernel Program: *Definition ; $\cdots$; Definition ; Main.* (Translate Id Kernel Program to Mini-Language)	
Incremental System	**Complete Program System**
$TE_0 = TE_{lib} + \{x_i \mapsto \forall\alpha.\alpha\}$: B_1 TE_1 : B_2 $\vdots$ TE_{n-1} : B_n TE_n : $\texttt{it} = e_{main}$ $\vdots$ TE_i : B_x $\vdots$ TE_f.	$TE^t_{x_1} = TE_{lib}$: $\texttt{let}\ x_1 = e_1\ \texttt{in}$ $TE^t_{x_2} = TE^t_{x_3}$: $\texttt{let}\ x_2, x_3 = e_{23}\ \texttt{in}$ $\vdots$ $TE^t_{x_n}$: $\texttt{let}\ x_n = e_n\ \texttt{in}$ $TE^t_{e_{\text{main}}}$: e_{main}. $TE_t = TE^t_{e_{\text{main}}} + \{\texttt{it} \mapsto \sigma_{e_{\text{main}}}\}$.

A Session in the Incremental System and its Corresponding Complete Program.

Proof Strategy			
SCC Environment		**Type Environment**	
Completeness: $\forall i.\ SE_i \subseteq SE_t$	Lemma 3.6.	Completeness: $\forall i.\ TE_i \succeq TE_t$	Lemma 3.8.
Soundness: $SE_t \subseteq SE_f$	Lemma 3.9.	Soundness: $TE_t \succeq TE_f$	Lemma 3.11.

Figure 3.8: The Overall Plan of Correctness Proof of Incremental System.

final type environments derived in the two cases. For the complete program case, the final type environment TE_t is obtained by extending the type environment $TE^t_{e_{\text{main}}}$ (under which the final expression query is typed) with the type-scheme $\sigma_{e_{\text{main}}} = close(\tau_{e_{\text{main}}})$ of the final expression query itself. We compare this with the final type environment TE_f computed by the incremental compilation scheme when all the assumption checks (including those of the special binding "$\texttt{it} = e_{main}$") have been passed. Our goal is to show that $TE_t = TE_f$.

We show this correspondence in two steps. First in section 3.4, we show that starting with a most general type environment (equation 3.6), the type environment in the incremental compilation scheme gets increasingly more refined as each new top-level definition is type-checked (lemma 3.7). We also show that at each step the existing type environment is more general (in the sense of the $\succeq$ predicate on type-schemes) than the final type environment TE_t of the complete program case (lemma 3.8). In a way, this is a statement of completeness of the algorithm W_1, *i.e.*, any typing derivable when the complete program is available, is an instance of the typing derivable at any point during the above compilation scheme and hence is also derivable. Since the type property directly uses the strongly connected component property, we have to make similar completeness arguments about the SCC environments first (lemmas 3.5 and 3.6).

As the second step, in section 3.5, we show that the final type environment TE_f obtained after passing all the assumption checks is less general than (or at least as refined as) the final environment TE_t in the complete program case (lemma 3.11). Thus, any typing derivable from TE_f, is also derivable using TE_t in the complete program case. This verifies the soundness of algorithm W_1 that its final type environment cannot derive any extraneous typings. As in the first case, we have to support this with a similar argument for the SCC environments as well (lemma 3.9).

Combining these two steps in section 3.6, we conclude that final type environments in the two cases are exactly the same and therefore derive exactly the same typings (theorem 3.12).

3.3.2 Properties of the Assumptions

Before we go into the details of proving algorithm W_1 correct, let us first examine the characteristics and significance of the upper bound type assumptions (M) we collect for each of the

This is because the bindings actually get rearranged in topological order in the complete program system, thereby fixing a particular type environment for each definition, while in the incremental system, the bindings are processed in texual order interspersed with random requests for recompilation of some of the earlier bindings.

generic top-level identifiers *via* algorithm W'. By definition, these assumptions record all the generic type instances created from the top-level type-schemes within the call to W'. They serve as an exact type specification of the interface between the current top-level definition and other definitions.

Below, we define these assumption sets in a general setting and then examine their properties in the context of algorithm W', which will help us later in the correctness proofs.

Notation 3.3 *Given a typing $TE \vdash e : \tau$ and its associated derivation tree, we write $A^{TE} \vdash e : \tau$, or simply $A \vdash e : \tau$ (when TE is clear by context), as another typing representing the same derivation tree that explicitly records all type instances of its free (external) identifiers. A^{TE} is the set of assumptions constructed inductively as below*[16]*.*

Case *1:* — $e \Rightarrow x$ — *Define* $A^{TE} = \{(x \mapsto \tau)\}$.

Case *2:* — $e \Rightarrow \lambda x.e_1$ — *If inductively,* $A^{TE'} \vdash e_1 : \tau_1$ *represents the typing for the body* e_1 *where* $TE' = TE + \{x \mapsto \tau_2\}$*, then define* $A^{TE} = A^{TE'} \backslash_{\{x\}}$.

Case *3:* — $e \Rightarrow e_1\ e_2$ — *If inductively,* $A_1^{TE} \vdash e_1 : \tau_2 \rightarrow \tau_1$ *and* $A_2^{TE} \vdash e_2 : \tau_2$*, then define* $A^{TE} = A_1^{TE} \cup A_2^{TE}$.

Case *4:* — $e \Rightarrow \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ — *If inductively,* $A_1^{TE} \vdash e_1 : \tau_1$ *and* $A_2^{TE'} \vdash e_2 : \tau_2$ *where* $TE' = TE + \{x \mapsto close(TE, \tau_1)\}$*, then define* $A^{TE} = A_1^{TE} \cup A_2^{TE'} \backslash_{\{x\}}$.

Informally, the typing $A^{TE} \vdash e : \tau$ simply collects the actual type instances assigned to each occurrence of an external identifier inside the derivation tree of the typing $TE \vdash e : \tau$ into the assumption set A^{TE}. A^{TE} is a special multi-set mapping in the sense that several type instance may be recorded for the same identifier x corresponding to its various occurrences within e, and also because it always refers to a particular derivation tree expansion of the typing $TE \vdash e : \tau$, recording a specific set of type instances used within it.

Note that A^{TE} is not a type environment in itself, but it serves as a cumulative record of the use of the free identifiers (FId) of the expression e during the derivation of the typing $TE \vdash e : \tau$. Note that the notation respects the scoping rules for free identifiers of an expression; the identifiers bound locally *via* a `let` or a λ-construct are free internally but bound in the whole expression, therefore, the assumptions collected for these identifiers are discarded as soon as they become bound.

[16]Unless it is necessary to specify the assumer explicitly, we treat a set of assumptions as a mapping from the assumee to the assumed value. The identifier on LHS of the current top-level binding is implicitly taken to be the assumer.

Now we define operations on these assumption sets and show some of their properties.

Definition 3.4 *Let A be a set of assumptions collected for the typing $TE \vdash e : \tau$ as defined above. Then, for a specific occurrence of a free identifier x within e, define $A(x) = \tau_x$ if and only if $(x \mapsto \tau_x) \in A$ and this is the assumption corresponding to the specified occurrence of x as recorded in $A \vdash e : \tau$.*

It is usually clear by context which occurrence of x is being referred to in $A(x)$[17] because it makes sense to use an assumption set A only as prescribed by the derivation tree of $A \vdash e : \tau$.

Definition 3.5 *Given an assumption set A collected for a typing $TE \vdash e : \tau$ and another type environment TE', we define $TE' \succeq A$ if and only if $Dom(A) \subseteq Dom(TE')$ and $\forall (x \mapsto \tau_x) \in A$ with $TE'(x) = \sigma_x$, we have $\sigma_x \succeq \tau_x$. Note that $TE \succeq A$ by construction.*

The above definition provides a meaningful way to compare an assumption set with type environments other than that for which it was originally collected. The following two lemmas follow directly from this definition.

Lemma 3.1 *Given an assumption set A collected for the typing $TE \vdash e : \tau$ and another type environment TE' satisfying $TE' \succeq A$ according to definition 3.5, then $TE' \vdash e : \tau$ is also a valid typing.*

Proof: By definition, A records the actual type instances of all the free identifiers used within the derivation tree of the typing $TE \vdash e : \tau$. So, any type environment capable of generating those external type instances is an acceptable candidate for preserving the derivation tree. In other words, any type environment TE' satisfying $TE' \succeq A$ will be able to regenerate the same derivation tree and hence produce a valid typing. □

Note that in the above lemma, the original type environment TE may be incomparable to the given environment TE', but as long as they can instantiate the same assumption set, both are able to generate the same typing. This further corroborates the notion that the assumption sets can be used as a minimal description of a typing with respect to the types of the free identifiers used in it.

Lemma 3.2 *Given an assumption set A collected for a typing $TE \vdash e : \tau$ and another type environment TE', if $TE' \succeq A$ then for all substitutions S, $S(TE') \succeq S(A)$.*

[17] Remember, A is a multi-set and can have multiple assumptions for x corresponding to its various occurrences within e.

Proof: It follows trivially from definition 3.5 above and by pointwise extension of lemma 2.2.

□

We may note that the range of an assumption set is a set of types, and in general, may contain several type variables all of which are free. In order to reason about substitutions as applied to assumption sets, we partition these type variables into the following categories.

Notation 3.6 *The set of all type variables in $Ran(A^{TE})$ denoted by $tyvars(A^{TE})$ is partitioned into three categories.*

1. *First, there may be some type variables that occur free in TE as well. These are represented by,*

$$freevars(A^{TE}) = tyvars(A^{TE}) \cap tyvars(TE)$$

2. *There may be some type variables that are closed over by the generalization operation (definition 2.8) of the LET rule while typing some* `let`*-construct within the expression e. By definition of generalization, these have to be disjoint from the free type variables of TE and become bound in that branch of the derivation tree. These type variables are represented by $genvars(A^{TE})$.*

3. *All other type variables that are neither free in TE nor appear bound in any branch of the derivation tree are simply fresh type variables used in the derivation tree that have not yet been closed over and are denoted by $othervars(A^{TE})$.*

By definition, we have,

$$tyvars(A^{TE}) = freevars(A^{TE}) \cup genvars(A^{TE}) \cup othervars(A^{TE})$$

In order to make the partitioning clear, consider the following simple example in mini-language. The expression e in equation 3.8 has a typing 3.9 while we collect the assumptions as shown in equation 3.10. We have superscripted all identifiers with their types for clarity.

$$e \quad ::= \quad \texttt{let}\ f^{(list\ \alpha),\beta} = \texttt{nil}^{(list\ \alpha)}, y^{\beta}\ \texttt{in}\ \texttt{nil}^{(list\ \delta)}; \tag{3.8}$$

$$TE + \{y \mapsto \beta\} \quad \vdash \quad e : (list\ \delta) \tag{3.9}$$

$$A \quad = \quad \{\texttt{nil} \mapsto (list\ \alpha), y \mapsto \beta, \texttt{nil} \mapsto (list\ \delta)\} \tag{3.10}$$

Among the type variables of the assumption set A, β is a free variable since it occurs free in the supplied type environment as well, α is a generic variable since it is closed over in the let-binding of f (assuming that α is not free in TE as well), and δ belongs to neither of the above two categories. Also note that there are two assumptions for nil corresponding to its two occurrences within e as expected.

The following lemma forms the backbone of the assumption based reasoning in the subsequent proofs for correctness of our incremental system.

Lemma 3.3 *Let $A^{TE} \vdash e : \tau$ be a typing corresponding to $TE \vdash e : \tau$ defined* via *notation 3.3. Then for any substitution S that does not involve genvars(A) in its domain or range, $S(A^{TE}) \vdash e : S\tau$ is also a valid typing corresponding to $S(TE) \vdash e : S\tau$.*

Proof: Using lemma 2.3 we obtain,

$$\begin{array}{lrcl} & TE & \vdash & e : \tau \\ \Longrightarrow & S(TE) & \vdash & e : S\tau \end{array} \tag{3.11}$$

We only have to show that the assumption set $A^{S(TE)}$ corresponding to typing 3.11 is the same as $S(A^{TE})$. We will prove this inductively by traversing the derivation tree of typing 3.11 and verifying that the assumption set $A^{S(TE)}$ generated at each step (*via* notation 3.3) satisfies the condition $A^{S(TE)} = S(A^{TE})$.

Case 1: — $e \Rightarrow x$ — Typing 3.11 directly generates an assumption set $A^{S(TE)} = \{x \mapsto S\tau\}$ which is the same as $S(A^{TE})$.

Case 2: — $e \Rightarrow \lambda x.e_1$ — If inductively, $A^{S(TE')} = S(A^{TE'})$ for the antecedent $S(TE') \vdash e_1 : S\tau_1$ where $TE' = TE + \{x \mapsto \tau_2\}$, then $A^{S(TE)} = A^{S(TE')}\backslash_{\{x\}} = S(A^{TE'})\backslash_{\{x\}} = S(A^{TE})$ using notation 3.3.

Case 3: — $e \Rightarrow e_1\ e_2$ — If inductively, $A_1^{S(TE)} = S(A_1^{TE})$ and $A_2^{S(TE)} = S(A_2^{TE})$ for the antecedents $S(TE) \vdash e_1 : S(\tau_2 \rightarrow \tau_1)$ and $S(TE) \vdash e_2 : S\tau_2$ respectively, then $A^{S(TE)} = A_1^{S(TE)} \cup A_2^{S(TE)} = S(A_1^{TE}) \cup S(A_2^{TE}) = S(A^{TE})$ using notation 3.3.

Case 4: — $e \Rightarrow$ let $x = e_1$ in e_2 — We expand the immediate antecedents of typing 3.11 for this case[18].

$$\frac{TE' \vdash e_1 : S'\tau_1 \qquad TE' + \{x \mapsto close(TE', S'\tau_1)\} \vdash e_2 : S\tau_2}{TE' \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : S\tau_2} \tag{3.12}$$

[18] For a proof of how these antecedents were arrived at, see [57], pages 18–20.

where,

$$\begin{aligned}
TE' &= S(TE) = S'(TE) \\
\text{and} \qquad S' &= S \downarrow_{tyvars(TE)} \\
\Longrightarrow \qquad S(close(TE, \tau_1)) &= close(TE', S'\tau_1)
\end{aligned}$$

We are given that the substitution S does not involve any generic type variables that are closed over in any branch of the derivation tree. In other words, S may involve only those type variables that are not allowed to be closed over as far as this typing is concerned. So we must have,

$$S(close(TE, \tau_1)) = close(S(TE), S\tau_1)$$

Thus, we can replace S' with S in typing 3.12 and use the inductive assumption for the two antecedents, yielding the desired result $A^{S(TE)} = S(A^{TE})$ *via* notation 3.3.

$\square$

The above lemma provides a way to use the assumption sets to obtain new correct typings from old typings for the same expression. We can compare this with the lemma 2.3 which establishes the substitution transformation using the type environments.

Now we come back to the algorithm W' (refer figure 3.6) and show that the assumptions collected there actually correspond to the typing generated by the algorithm. The proof of this lemma is very similar to that of the soundness theorem 2.7.

Lemma 3.4 *If assumption set A is collected in the call $W'(TE, e) = (S, \tau, A)$ then $A \vdash e : \tau$ is a valid typing that corresponds to the typing $S(TE) \vdash e : \tau$ generated through the soundness theorem 2.7.*

Proof: We prove this by structural induction on the expression e.

Case 1: — $e \Rightarrow x$ — From figure 3.6, $W'(TE, x) = (ID, \tau', \{(x \mapsto \tau')\})$. Clearly, $\{(x \mapsto \tau')\} \vdash x : \tau'$ is a valid typing corresponding to $ID(TE) \vdash x : \tau'$ directly from its definition in notation 3.3.

Case 2: — $e \Rightarrow \lambda x.e_1$ — From figure 3.6, the internal recursive call to W' for the body e_1

yields (S_1, τ_1, A_1). So, inductively we obtain the following valid typing,

$$\begin{array}{lrcl} & A_1 & \vdash & e_1 : \tau_1 \\ \text{where} & S_1(TE) + \{x \mapsto S_1\beta\} & \vdash & e_1 : \tau_1 \end{array}$$

Using notation 3.3 for the above typing, we arrive at the following,

$$\begin{array}{lrcl} & A_1\backslash_{\{x\}} & \vdash & \lambda x.e_1 : S_1\beta \to \tau_1 \\ \text{where} & S_1(TE) & \vdash & \lambda x.e_1 : S_1\beta \to \tau_1 \end{array} \tag{3.13}$$

As we can see from figure 3.6, $A_1\backslash_{\{x\}}$ is exactly the assumption set returned from W', so that typing 3.13 is what we want.

Case 3: — $e \Rightarrow e_1\ e_2$ — Again, the internal recursive calls to W' inductively yield the following,

$$A_1 \vdash e_1 : \tau_1 \tag{3.14}$$

$$\text{where} \quad S_1(TE) \vdash e_1 : \tau_1$$

$$\text{and} \quad A_2 \vdash e_2 : \tau_2 \tag{3.15}$$

$$\text{where} \quad S_2S_1(TE) \vdash e_2 : \tau_2$$

The substitution S_2 does not involve any type variable from *genvars*(A_1) (see notation 3.6) because its type variables can only come from either the free type variables of TE, or absolutely fresh type variables generated while typing e_2. Similarly, the substitution $S_3 = U(S_2\tau_1, \tau_2 \to \beta)$ does not involve any type variables from either *genvars*(A_1) or *genvars*(A_2). Therefore, lemma 3.3 can be applied to the typings 3.14 and 3.15 giving,

$$S_3S_2(A_1) \vdash e_1 : S_3S_2\tau_1$$

$$\Longrightarrow \quad S_3S_2(A_1) \vdash e_1 : S_3\tau_2 \to S_3\beta \tag{3.16}$$

$$\text{and} \quad S_3(A_2) \vdash e_2 : S_3\tau_2 \tag{3.17}$$

Using notation 3.3 and the typings 3.16 and 3.17 we obtain the following valid typing,

$$\begin{array}{lrcl} & S_3S_2(A_1) \cup S_3(A_2) & \vdash & e_1\, e_2 : S_3\beta \\ \text{where} & S_3S_2S_1(TE) & \vdash & e_1\, e_2 : S_3\beta \end{array}$$

which exactly corrresponds to the assumption set returned from the outer call to W'.

Case 4: — $e \Rightarrow$ `let` $x = e_1$ `in` e_2 — Just like in the last case, the internal recursive calls to

W' inductively yield the following,

$$\begin{array}{llcll} & A_1 & \vdash & e_1 : \tau_1 & (3.18) \\ \text{where} & S_1(TE) & \vdash & e_1 : \tau_1 & \\ \text{and} & A_2 & \vdash & e_2 : \tau_2 & (3.19) \\ \text{where} & S_2S_1(TE) + \{x \mapsto \sigma\} & \vdash & e_2 : \tau_2 & \\ \text{and} & \sigma & = & close(S_2S_1(TE), S_2\tau_1) & \text{As in lemma 3.3.} \end{array}$$

We always allocate fresh type variables in generating instances of previously closed type-schemes, including the type-scheme generated from the current `let`-binding. Therefore, the type variables of S_2 are either from these fresh type variables or from the free type variables of TE, neither of which can appear in $genvars(A_1)$. So just like the previous case, we can apply S_2 to typing 3.18 using lemma 3.3 and then combine it with typing 3.19 using notation 3.3 to obtain the following valid typing,

$$\begin{array}{llcl} & S_2(A_1) \cup A_2\backslash_{\{x\}} & \vdash & \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2 \\ \text{where} & S_2S_1(TE) & \vdash & \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2 \end{array}$$

which exactly corresponds to the assumption set returned from the outer call to W'.

□

The above lemma validates the assumptions collected by algorithm W' in the sense that the collected assumptions denote a valid typing. The fact that these assumptions describe a minimum specification of the interface between the binding currently being processed and the external environment (lemma 3.1), and that we can manipulate them as *pseudo*-type environments (lemma 3.3), permits us to use them as the basis of our incremental book-keeping.

3.4 Completeness of Algorithm W_1

Following the plan for the proof of correctness of our incremental system (section 3.3.1), we first show some simple results about the strongly connected components of top-level identifiers.

3.4.1 Completeness of SCC Computation

Lemma 3.5 *Starting from SE_0 (equation 3.7), if SE_i and SE_j ($j > i$) are the SCC environments obtained after the i-th and the j-th call to W_1 then $\forall x \in X$, $SE_i(x) \subseteq SE_j(x)$ (as an abbreviation, we write $SE_i \subseteq SE_j$ for this condition).*

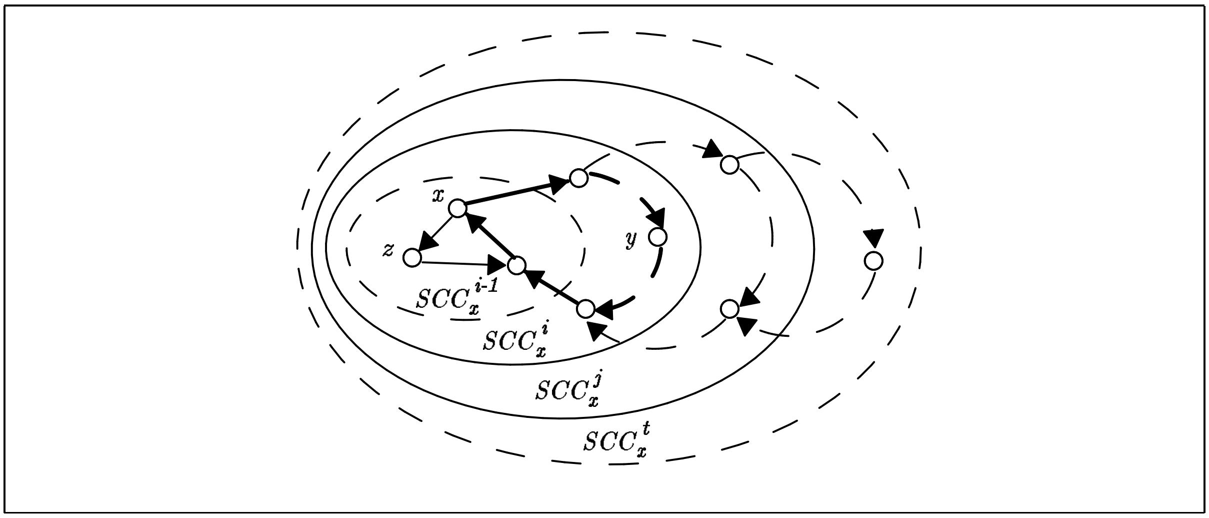


Figure 3.9: The Strongly Connected Components of x at various incremental stages.

Proof: We prove this by induction on i (refer figure 3.9).

Base Step: It is clear from equation 3.7 that $SE_0 \subseteq SE_j$ for every $j > 0$ since ϕ is a subset of any set.

Inductive Step: Assume that $SE_{i-1} \subseteq SE_j$ for every $j > (i-1)$.

Looking at the algorithm W_1 (figure 3.5), we see that SE_i is the same as SE_{i-1} except for the identifiers in SCC_x^i, where it is updated to the freshly computed component SCC_x^i. From assumption 3, no edge or node gets removed from the call graph once it is added. If the current binding B_x has already appeared earlier, then there are no new additions to the call graph either, and $SCC_x^i = SCC_x^{i-1}$. Therefore, $SE_i = SE_{i-1} \subseteq SE_j$ for every $j > i$ using the inductive assumption.

If the binding B_x is being processed for the first time, then all its calling dependency edges and some of its free identifiers may have to be added to the call graph. In this case, SCC_x^i may differ from SCC_x^{i-1}. Now, for every *new* element $y \in SCC_x^i$ (if any) which is not present in SCC_x^{i-1}, there is an edge path from x to y and vice-versa[19] since y is now in the SCC of x. This path never gets destroyed in subsequent calls to W_1 by the above observation, so y will remain in the SCC of x forever. Therefore we conclude that $SCC_x^i \subseteq SCC_x^j$, and consequently $SE_i \subseteq SE_j$ for every $j > i$.

□

[19]This path appears in bold in figure 3.9.

Lemma 3.6 *Define SE_t to be the SCC environment for the complete program case so that $\forall x \in X$, $SE_t(x)$ is the set of all top-level identifier that are mutually recursive with x in the whole program. Then for every $i \geq 0$, $SE_i \subseteq SE_t$.*

Proof: The proof is exactly similar to the one above; again hinging on the fact that once added, no edge gets removed from the call graph and that the call graph existing at each incremental stage is always a subgraph of the call graph of the whole program. Therefore, the strongly connected components of top-level identifiers when the whole program is available are always a superset of the corresponding components at any intermediate stages. □

In short, the above two lemmas state that in our incremental compilation scheme, the strongly connected components of the top-level identifiers start from an empty set and steadily increase in size while never exceeding their maximum possible value obtained in presence of the full program. We will show in section 3.5 that we stop further computation only when we have reached this maximum value.

3.4.2 Completeness of Type Computation

The above pair of lemmas embody a simple pattern that is at the heart of the incremental property computation. Starting with no information, we can refine our knowledge about interesting global properties as we are given more details about the program. Now, we apply the same principle to the type property.

Lemma 3.7 *Starting from TE_0 (equation 3.6), if TE_i and TE_j $(j > i)$ are the type environments obtained after the i-th and the j-th call to W_1 then $TE_i \succeq TE_j$.*

Proof: We prove this by induction on i.

Base Step: It is clear from equation 3.6 that $TE_0 \succeq TE_j$ for every $j > 0$ since $\forall\alpha.\alpha \succeq \sigma$ for any σ.

Inductive Step: Assume that $TE_{i-1} \succeq TE_j$ for every $j > (i-1)$. Let x be the identifier being typed at the i-th call.

Looking at algorithm W_1 (figure 3.5), we see that TE_{i-1} is the same as TE_i except for the identifiers $z_k \in SCC_x^i$, where it has been updated to the new type schemes $\sigma_{z_k}^i = close(U_i\tau_{z_k}^i)$. So the inductive assumption carries over to TE_i for all other identifiers except these. For any identifier $z_k \in SCC_x^i$, let TE_j $(j > i)$ be some later type environment in which the type-scheme for z_k is updated again to $\sigma^j_{z_k} = close(U_j\tau^j_{z_k})$[20]. We need to show that $\sigma^i_{z_k} \succeq \sigma^j_{z_k}$ for each $z_k \in SCC^i_x$, so that we can combine this information with the inductive assumption and arrive at $TE_i \succeq TE_j$ for every such $j > i$. For convenience, we denote the number of elements in sets SCC^i_x and SCC^j_x by n and m respectively.

In order to prove that $\sigma^i_{z_k} \succeq \sigma^j_{z_k}$ for each $z_k \in SCC^i_x$, we need to construct an alternate typing for each z_k in terms of the types obtained at the j-th call, and then use the completeness theorems 2.6 and 2.8 over the types computed at the i-th call to give the above instantiation relationship.

The following typings are obtained through the internal calls to W' and the U_j unification at the j-th call to W_1. For every $z_k \in SCC^j_x$ we have,

$$\begin{array}{lrcll} & \textit{New-TE}^j_k & = & TE_{j-1} \setminus_{SCC^j_x} +\{z_1 \mapsto \beta_{k1}, \ldots, z_m \mapsto \beta_{km}\} & \\ & W'(\textit{New-TE}^j_k, e_k) & = & (S^j_k, \tau^j_{z_k}, A^j_k) & \\ \Longrightarrow & S^j_k(\textit{New-TE}^j_k) & \vdash & e_k : \tau^j_{z_k} & \text{Using theorem 2.7.} \\ \Longrightarrow & U_j S^j_k(\textit{New-TE}^j_k) & \vdash & e_k : U_j\tau^j_{z_k} & \text{Using lemma 2.3.} \end{array}$$

which when expanded yields the following typing for each RHS expression e_k corresponding to each top-level identifier $z_k \in SCC^j_x$.

$$TE_{j-1} \setminus_{SCC^j_x} +\{z_1 \mapsto U_j\tau^j_{z_1}, \ldots, z_m \mapsto U_j\tau^j_{z_m}\} \vdash e_k : U_j\tau^j_{z_k} \tag{3.20}$$

Note that by construction, there are no free type variables in the environment TE_{j-1}, and the U_j substitution ultimately reduces each term $S^j_k(\beta_{k1}), \ldots, S^j_k(\beta_{km})$ for each $1 \leq k \leq m$ to the same term $U_j\tau^j_{z_k}$ by unification.

We compare the type environment in the typing 3.20 above with the environments $\textit{New-TE}^i_k$ setup during the i-th call for each k $(1 \leq k \leq n)$.

$$\begin{array}{lrcll} & \textit{New-TE}^i_k & = & TE_{i-1} \setminus_{SCC^i_x} +\{z_1 \mapsto \beta_{k1}, \ldots, z_n \mapsto \beta_{kn}\} & \\ \text{Let} & S^{j-i}_k & = & \{\beta_{k1} \mapsto U_j\tau^j_{z_1}, \ldots, \beta_{kn} \mapsto U_j\tau^j_{z_n}\} & \\ \Longrightarrow & S^{j-i}_k(\textit{New-TE}^i_k) & = & TE_{i-1} \setminus_{SCC^i_x} +\{z_1 \mapsto U_j\tau^j_{z_1}, \ldots, z_n \mapsto U_j\tau^j_{z_n}\} & \\ \Longrightarrow & S^{j-i}_k(\textit{New-TE}^i_k) & \vdash & e_k : U_j\tau^j_{z_k} & (3.21) \end{array}$$

The last typing 3.21 is obtained by comparing its environment with that in typing 3.20. We use the inductive assumption to get $TE_{i-1} \succeq TE_{j-1}$ and lemma 3.5

[20]That would be the case when some identifier x' (possibly x again) gets type-checked at the j-th call and $z_k \in SCC^j_{x'}$.

to ensure that $SCC_x^i \subseteq SCC_x^j$ so that lemma 2.4 can be used for the identifiers in $SCC_x^j - SCC_x^i$. This equation specifies an alternate typing for each $z_k \in SCC_x^i$ which we now show to be an instance of the type-scheme computed at the i-th call.

For each of the internal calls to W' during the i-th call to W_1, we can use the completeness theorem 2.8 over its corresponding alternate typing 3.21 to give the following,

$$\exists R_k.\ (R_k \circ S_k^i)(\mathit{New\text{-}TE}_k^i) = S_k^{j-i}(\mathit{New\text{-}TE}_k^i) \qquad (1 \le k \le n). \tag{3.22}$$

$$R_k \mathit{close}(S_k^i(\mathit{New\text{-}TE}_k^i), \tau_{z_k}^i) \succeq U_j \tau_{z_k}^j \qquad \forall z_k \in SCC_x^i. \tag{3.23}$$

In equation 3.22 above, the domains of each of the R_k substitutions $(1 \le k \le n)$ are completely disjoint because none of the internal calls to W' share any non-generic type variables. Furthermore for each k, $\mathit{Dom}(R_k) \subseteq \mathit{tyvars}(S_k^i(\mathit{New\text{-}TE}_k^i))$, so that it involves only the free type variables of $\tau_{z_k}^i$. Therefore, for each type $\tau_{z_k}^i$ $(z_k \in SCC_x^i)$, the instance relationship of 3.23 can be explicitly written as a substitution equation $(R_k \mid I_k)\tau_{z_k}^i = U_j \tau_{z_k}^j$ where I_k is the instantiation substitution operating only on the generic type variables of $\tau_{z_k}^i$.

Finally, we combine all these disjoint substitutions into one giant substitution U_{j-i} that unifies the column terms of the unification matrix in the second phase of the i-th call into the types obtained in the j-th call.

$$\begin{array}{llcll} & U_{j-i} & = & R_1 \mid I_1 \mid \cdots \mid R_n \mid I_n & \\ \Longrightarrow & U_{j-i}\tau_{z_k}^i & = & U_j \tau_{z_k}^j & \forall z_k \in SCC_x^i. \end{array}$$

Now by the completeness theorem of unification (theorem 2.6) U_i is the most general unifier for its nodes[21], and all other unifiers are substitution instances of U_i. *I.e.*,

$$\begin{array}{llcll} & \exists R.\ R \circ U_i & = & U_{j-i} & \\ \Longrightarrow & (R \circ U_i)\tau_{z_k}^i & = & U_{j-i}\tau_{z_k}^i & \forall z_k \in SCC_x^i. \\ \Longrightarrow & (R \circ U_i)\tau_{z_k}^i & = & U_j \tau_{z_k}^j & \forall z_k \in SCC_x^i. \\ \Longrightarrow & \mathit{close}(U_i \tau_{z_k}^i) = \sigma_{z_k}^i & \succeq & \mathit{close}(U_j \tau_{z_k}^j) = \sigma_{z_k}^j & \forall z_k \in SCC_x^i. \end{array}$$

The last equation is what we wanted to prove.

$\square$

[21]Theorem 2.6 deals with the unification of a pair of type terms whereas here we have several groups of terms to be unified under the same unifying substitution. But it is easy to construct a pair of giant tuple-type terms from the given set of terms whose unification yields the same substitution U_i. So the completeness result is valid for this case also.

Lemma 3.8 *If TE_t is the final type environment in the complete program case with the type-scheme obtained for the final program query being assigned to the special variable "*`it`*", then for every $i \geq 0$, $TE_i \succeq TE_t$.*

Proof: The basic strategy of the proof is same as in proving the last lemma. We construct an alternate typing for each $z_k \in SCC_x^i$ at the i-th call in terms of the types obtained in the complete program system and then use the completeness theorems to show that $\sigma_{z_k}^i \succeq \sigma_{z_k}^t$ for each $z_k \in SCC_x^i$. Since only the identifiers from SCC_x^i get updated at the i-th call, we combine this with the inductive assumption to obtain the desired result just like in the previous lemma.

Base Step: From equation 3.6, $TE_0 \succeq TE_t$ since $\forall\alpha.\alpha \succeq \sigma$ for any σ.

Inductive Step: Assume that $TE_{i-1} \succeq TE_t$. As before, let x be the identifier being typed at the i-th call. Also, let the number of elements in sets SCC_x^i and SCC_x^t be denoted by n and m respectively, where SCC_x^t stands for the strongly connected component of x in the complete call graph.

In the complete program system, the algorithm W is called over the complete program expression as described in section 2.4.2 with the initial library environment TE_{lib}. The type-scheme σ_x^t for x appearing in the final type environment TE_t is actually computed during an internal recursive call to W when its corresponding `let`-binding is being typed. In the general case, x appears together with the identifiers in its strongly connected component SCC_x^t in a `let`-binding,

$$\texttt{let}\ z_1, \ldots, x, \ldots, z_m = \mathit{fix}\ \lambda z_1 \cdots x \cdots z_m.(e_{z_1}, \ldots, e_x, \ldots, e_{z_m})\ \texttt{in}\ e \tag{3.24}$$

We traverse the derivation tree generated by the overall program typing down to the branch where the inner tuple of the RHS expressions $(e_{z_1}, \ldots, e_x, \ldots, e_{z_m})$ is being typed to obtain the following typing for each RHS expression e_k corresponding to each top-level identifier $z_k \in SCC_x^t$.

$$TE_{prev} + \{z_1 \mapsto \tau_{z_1}^t, \ldots, z_m \mapsto \tau_{z_m}^t\} \vdash e_k : \tau_{z_k}^t \tag{3.25}$$

The type environment TE_{prev} comes from previously typed `let`-bindings in addition to the initial library environment TE_{lib}. The final type-schemes $\sigma_{z_k}^t$ for each $z_k \in SCC_x^t$ are obtained by completely closing their corresponding types obtained in typing 3.25 above.

$$TE_t(z_k) \;=\; \sigma^t_{z_k} \;=\; close(\tau^t_{z_k}) \qquad\qquad \forall z_k \in SCC^t_x.$$

Typing 3.25 is similar to typing 3.20 of the last lemma. As before, we can setup conversion substitutions S^{t-i}_k for the environments $New\text{-}TE^i_k$ created during the i-th call that produce an alternate typing for each $z_k \in SCC^i_x$ in the same spirit as typing 3.21. We only note that TE_{prev} is a subset map of TE_t so the inductive assumption on TE_{i-1} yields $TE_{i-1} \succeq TE_{prev}$[22] and lemma 3.6 ensures that $SCC^i_x \subseteq SCC^t_x$ as required in the proof. We omit the details.

So just as in the previous lemma, we obtain,

$$close(U_i\tau^i_{z_k}) \;=\; \sigma^i_{z_k} \;\succeq\; close(\tau^t_{z_k}) \;=\; \sigma^t_{z_k} \qquad\qquad \forall z_k \in SCC^i_x.$$

□

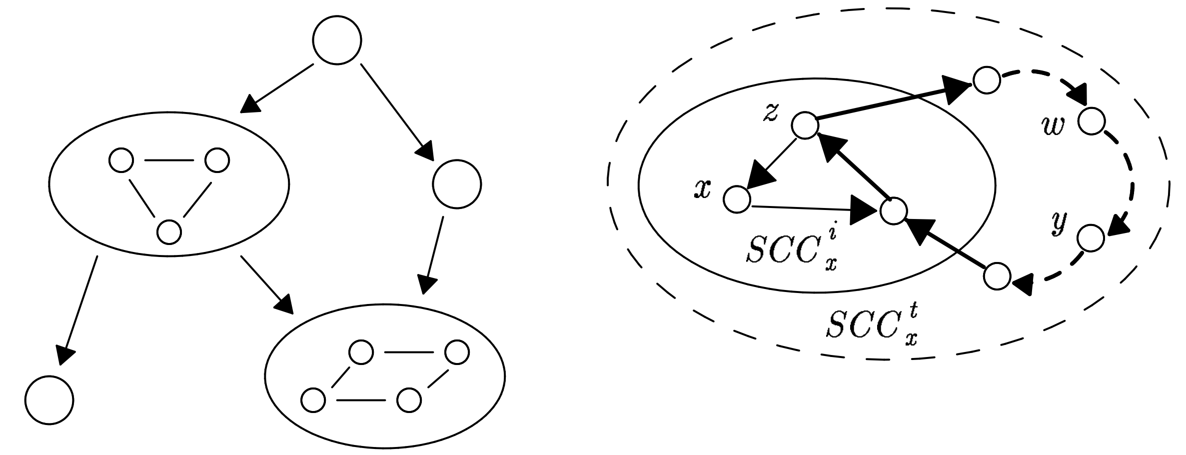
The above two lemmas together show that starting with the most general type-scheme $\forall\alpha.\alpha$, the type-schemes of the top-level identifiers get increasingly more refined, and that at each stage they are complete with respect to the actual type-schemes.

3.5 Soundness of Algorithm W_1

As a second step in the proof of correctness of algorithm W_1, we now show that the process of incrementally refining the compilation environment eventually terminates and when it does, it exactly corresponds to the compilation environment obtained when the complete program is available.

As noted in section 3.1.3, we stop further compilation only when all the assumption checks collected during various calls to W_1 are passed successfully by the existing property-values in the current compilation environment. The assumption checks play an important role here in guaranteeing that only a sound compilation environment is acceptable, otherwise the process of incremental refinement continues. Termination is shown by proving that the assumptions checks are eventually passed by some environment.

[22] Technically, the domain of TE_{i-1} will have to be restricted to that of TE_{prev} in order to compare the two, but we will ignore this for clarity.



(a): Form of Final Call Graph. (b): Set of Identifiers in an SCC.

Figure 3.10: Proving the Soundness of SCC Computation.

3.5.1 Soundness of SCC Computation

First, we prove that the SCC environment is computed correctly. Since no property assumptions are collected for the SCC property, no assumption checks are necessary and no requests for recompilations are possible. So we only have to show that each top-level identifier x is assigned its final strongly connected component SCC_x^t automatically after some number of calls to W_1 without any assumption checks or recompilations.

Lemma 3.9 *In the sequence of SCC environments $SE_0, \ldots, SE_i, \ldots$ generated from the various calls to W_1, there exists an environment SE_f (and every environment after that) such that $SE_f(x) = SE_t(x)$ for each $x \in X$.*

Proof: We prove this by structural induction on the call graph of the whole program G_t (refer figure 3.10 (a)).

Case 1: Consider a leaf of the call graph G_t which is a single identifier x.

The complete SCC for x, $SCC_x^t = \{x\}$, is computed the very first time x is type-checked, say at the i-th call. Due to assumption 2, this happens at least once. Once computed, it never changes from that value since it can only monotonically increase in size (lemma 3.5) but has already reached its maximum (lemma 3.6). So we can designate SE_f to be any SCC environment with $f \geq i$ since $SE_i(x) = \{x\} = SE_t(x)$ trivially.

Case 2: Consider a leaf of the call graph G_t which is a group of mutually recursive identifiers. Suppose an identifier x from this group is type-checked at some i-th call, updating the SCC of all $z_k \in SCC_x^i$ to SCC_x^i in the environment SE_i.

From lemma 3.6, $SCC_x^i \subseteq SCC_x^t$. If $SCC_x^i = SCC_x^t$ then our job is already done and we can designate SE_f to be any SCC environment with $f \geq i$.

If $SCC_x^i \neq SCC_x^t$ then let y represent one of the identifiers in SCC_x^t that are not present in SCC_x^i (refer figure 3.10 (b)). There must be at least one such $y \in (SCC_x^t - SCC_x^i)$ that has not been compiled even once when SCC_x^i is computed, so that the edges coming out of the node y are not yet visible in the intermediate call graph G_i. Otherwise, each such y will have to be present in SCC_x^i contradicting our assumption that $SCC_x^i \neq SCC_x^t$.

Now using assumption 2, every node in G_t must be type-checked at least once, so let $w \in (SCC_x^t - SCC_x^i)$ be the last of all such nodes appearing at some j-th call ($j > i$). At that time, all nodes in SCC_x^t as well as all their edges are visible in the call graph G_j. Hence SCC_w^j, the SCC of w at the j-th call, must contain all these identifiers, *i.e.*, $SCC_w^j = SCC_x^t = SCC_w^t$. Moreover, the SCC property for all $z_k \in SCC_w^j$ gets updated to $SCC_w^j = SCC_x^t$ in the latest environment SE_j. Thus, the final SCC are computed at the j-th call and we can designate SE_f to be any SCC environment with $f \geq j$.

Case 3: Consider an internal node of the call graph G_t which is a single identifier x. Let $Poly_x^t$ denote the set of all identifiers that x directly calls in G_t, and m be the number of elements in $Poly_x^t$.

Just like in case 1, the complete SCC for x, $SCC_x^t = \{x\}$, is computed the very first time it is type-checked, say at the i-th call. By inductive assumption, for each $z_k \in Poly_x^t$ there exists an SCC environment SE_{f_k} such that $SE_{f_k}(z_k) = SE_t(z_k)$. So we can designate SE_f to be any SCC environment with $f \geq max(i, f_1, \ldots, f_m)$.

Case 4: We can easily compose the cases 2 and 3 above and designate SE_f to be any SCC environment that is computed after all the nodes in the group of mutually recursive identifiers have become visible and all the nodes that any of them calls have also been assigned their final SCC values.

□

The above proof relies heavily on assumption 2 and the fact that we update the SCC value of all the identifiers $z_k \in SCC_x^i$ together during the i-th call to W_1. We will later examine another strategy where the assumption 2 is explicitly enforced by means of an end-of-program test and is no longer the user's responsibility. Alternate strategies are also possible that update the properties of only the current binding being type-checked but ensure that the latest information is always propagated to other the nodes in the same SCC. Detailed analysis of these strategies is beyond the scope of this thesis.

Note that operationally there is no way to distinguish among the various cases of the above lemma in the incremental system. Many identifiers, out of a set of mutually recursive ones, may start out being singleton, until some other identifier becomes visible that links them all together. The incremental system does not examine or update the SCC properties of the complete call graph at once[23]. Nevertheless, the lemma guarantees that by simply ensuring that all the identifiers in the call graph G_t have been compiled at least once, all of them will eventually compute their complete SCC correctly. In fact, the final SCC will be computed as soon as all the identifiers have been compiled at least once, as shown in the next lemma.

Lemma 3.10 *If SE_n is the SCC environment obtained after the n-th call to W_1 when all the identifiers $x \in X$ have been compiled at least once, then $SE_n = SE_t$.*

Proof: It can be easily seen that in each case of lemma 3.9, we can choose SE_f to be SE_n as defined above. $\square$

The above lemma ensures that if we have type-checked each top-level definition in our program once then we already have the desired final SCC environment without the need to recompile any of the definitions.

3.5.2 Soundness of Type Computation

Now, we show the correctness of the type environment computation. Here we will use the upper bound assumption checks to show that the type environment that passes all these checks is correct with respect to the complete program. We will use the lemmas established in section 3.3.2 in the proof.

[23]Indeed, if we were given the complete call graph together, there would be no need to have an incremental system.

Lemma 3.11 *In the sequence of type environments $TE_0, \ldots, TE_i, \ldots$ generated from the various calls to W_1, there exists an environment TE_f (and every environment after that) that passes all the upper bound type assumptions (M) for all the top-level identifiers,* i.e.,

$$\forall x \in X,\ \forall (z \mapsto \tau_z^i) \in M_x,\ \ TE_f(z) \succeq \tau_z^i$$

and moreover, $TE_t \succeq TE_f$.

Proof: Our proof strategy is the reverse of that in lemma 3.8. We will construct an alternate typing for the top-level identifiers in each SCC of the complete call graph in terms of the types obtained in the incremental algorithm. We will show that this can be done whenever all the collected upper bound assumptions pass their checks.

As mentioned in the proof of lemma 3.8, the type-schemes of the top-level identifiers in the complete program case are computed in a recursive call to W while typing a giant `let`-binding (equation 3.24) associated with all $z_k \in SCC_x^t$. We assume that the number of identifiers in SCC_x^t is n.

$$W(TE_{prev}, (\mathit{fix}\ \ \lambda z_1, \ldots, z_n.e_1, \ldots, e_n)) = (S^t_{z_1 \cdots z_n}, (\tau^t_{z_1}, \ldots, \tau^t_{z_n})) \tag{3.26}$$

where the type environment TE_{prev} is a subset of environment TE_t and it comes from previously typed `let`-bindings in addition to the initial library environment TE_{lib}.

Now we construct an alternate typing for the same expression using the type assumptions collected in the incremental system. For simplicity, we will only consider those iterations of W_1 where $SE_i = SE_t$[24]. In general at the i-th call to W_1, the following typing can be generated for each $z_k \in SCC_x^i = SCC_x^t$ $(1 \leq k \leq n)$ (refer to algorithm W_1 in figure 3.5).

$$\begin{array}{llcll} & W'(\mathit{New\text{-}TE}_k^i, e_k) & = & (S_k^i, \tau_{z_k}^i, A_k^i) & \\ \Longrightarrow & A_k^i & \vdash & e_k : \tau_{z_k}^i & \text{Using lemma 3.4.} \\ \Longrightarrow & U_i(A_k^i) & \vdash & e_k : U_i \tau_{z_k}^i & \text{Using lemma 3.3.} \end{array} \tag{3.27}$$

In equation 3.27, U_i only involves type variables present in the unification matrix. All the type variables from terms of the form $S_k^i(\beta_{k1}), \ldots, S_k^i(\beta_{kn})$ occur free in the environment $S_k^i(\mathit{New\text{-}TE}_k^i)$ and hence in the corresponding assumption set A_k^i for each $1 \leq k \leq n$. Other

[24]From the last lemma, this holds true for any iteration i after all the top-level bindings have been type-checked at least once. Moreover for each top-level identifier, this holds at least for the iteration in which its final SCC is computed.

type variables of U_i come from the terms $\tau^i_{z_k}$ $(1 \le k \le n)$ and are fresh variables that have not yet been closed over. Thus, no type variable involved in U_i can be a generic variable of any A^i_k and lemma 3.3 is applicable.

We can combine all the typings in equation 3.27 using the assumption notation 3.3 into a single alternate typing for the giant expression of equation 3.26.

$$\bigcup_{k=1}^{n} U_i A^i_k \backslash_{SCC^t_x} \vdash \mathit{fix} \quad \lambda z_1, \ldots, z_n.e_1, \ldots, e_n : U_i\tau^i_{z_1}, \ldots, U_i\tau^i_{z_n} \tag{3.28}$$

We discard all the assumptions corresponding to identifiers in SCC^t_x in accordance with notation 3.3 since these identifiers are now bound in the above expression. Note that the set of assumptions in the LHS of the above typing 3.28 is exactly the set of upper bound type assumptions that we record in algorithm W_1.

Now we are ready to show a structural induction on the call graph of the whole program G_t.

Case 1: Consider a leaf of the call graph G_t which in the general case, is a group of mutually recursive identifiers with their strongly connected component denoted by SCC^t_x. Let the identifiers of this group be type-checked at the i-th call.

Since this group is a leaf of the call graph, there are no upper bound type assumptions to check. Moreover, TE_{prev} of equation 3.26 will trivially satisfy the typing 3.28 because no external identifier is used for this leaf case. Thus, using the completeness theorem 2.8 for algorithm W with alternate typing 3.28 we get,

$$\begin{array}{rrcll} & \exists R,\ \ Rclose(\tau^t_{z_k}) & \succeq & close(U_i\tau^i_{z_k}) & \forall z_k \in SCC^t_x. \\ \Longrightarrow & TE_t(z_k) = close(\tau^t_{z_k}) & \succeq & TE_i(z_k) = close(\tau^i_{z_k}) & \forall z_k \in SCC^t_x. \end{array}$$

So we can designate TE_f to be any type environment with $f \ge i$ in order to satisfy the statement of the lemma.

Case 2: Consider an internal node of the call graph G_t which in the general case is a group of mutually recursive identifiers with their strongly connected component denoted by SCC^t_x. Let $Poly^t_x$ denote the set of all identifiers that any $z_k \in SCC^t_x$ directly calls in G_t, and m be the number of elements in $Poly^t_x$ (see figure 3.11).

Now, suppose the identifiers of SCC^t_x are last type-checked at the i-th call. By inductive assumption, for each $z_k \in Poly^t_x$ there exists a type environment TE_{f_k} that passes the assumption checks for all identifiers occuring in the subgraph below z_k. More-

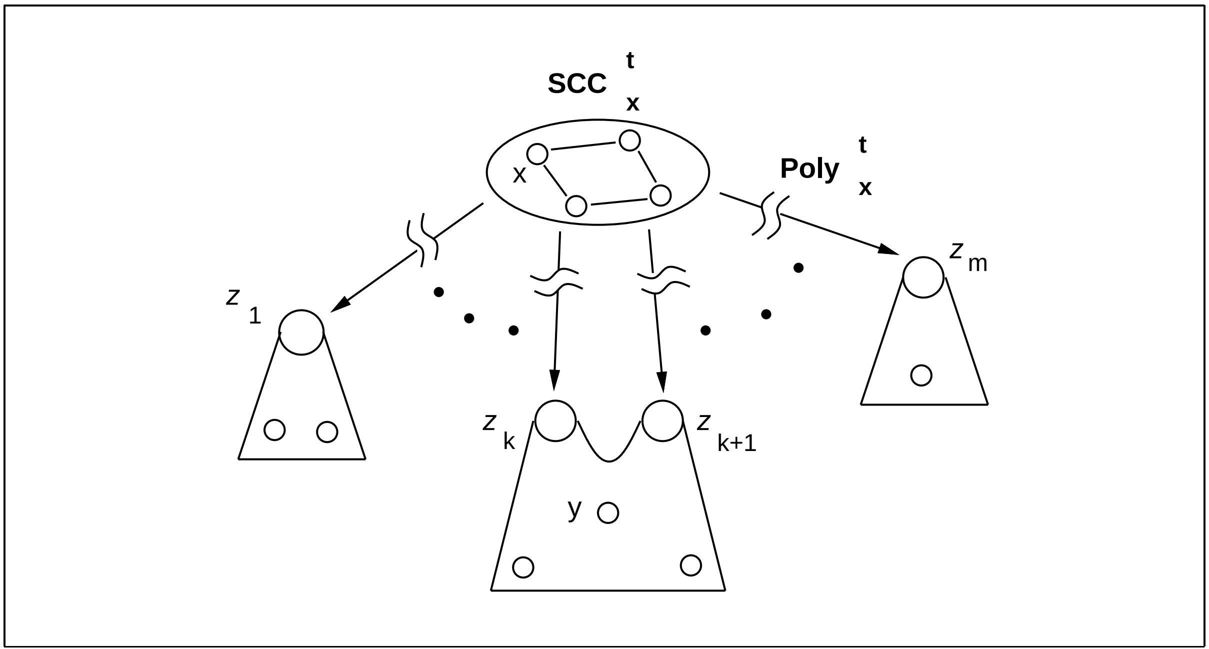


Figure 3.11: The Inductive Step in the Soundness Proof of Type Computation.

over, $TE_{prev}(y) \succeq TE_{f_k}(y)$ for each such identifier y in the subgraph. Let us denote $f' = max(f_1, \ldots, f_k, \ldots, f_m)$. Then by completeness lemma 3.7, $TE_{f_k} \succeq TE_{f'}$ for each $(1 \leq k \leq m)$, and $TE_{f'}$ satisfies the inductive assumption for the complete subgraph below the current node SCC_x^t by transitivity. Now there are two cases to consider.

SubCase 2.1: If $i > f'$ then it means that the current node SCC_x^t is being type-checked in bottom-up order after all its children have been type-checked. Using lemma 3.7 we have $TE_{f'} \succeq TE_i$, and by construction, TE_i satisfies all the assumptions of typing 3.28. Therefore,

$$TE_{prev} \succeq TE_{f'} \succeq TE_i \succeq \bigcup_{k=1}^{n} U_i A_k^i \backslash_{SCC_x^t}$$

so that TE_{prev} will again satisfy the typing 3.28 and the completeness theorem can be used just as in the case 1 above. This implies that we can designate TE_f to be any environment with $f \geq i$.

SubCase 2.2: If $i < f'$ then it means that the current node SCC_x^t was type-checked some time earlier than one of its children was type-checked. Now the assumptions play an important part. One possibility is that the environment $TE_{f'}$ still satisfies all the assumptions collected for the earlier i-th call and we have,

$$TE_{prev} \succeq TE_{f'} \succeq \bigcup_{k=1}^{n} U_i A_k^i \backslash_{SCC_x^t}$$

The first instantiation relation comes from the inductive assumption as above and the second one derives from $TE_{f'}$ passing all its assumption checks. This will yield $TE_t \downarrow_{SCC_x^t} \succeq TE_i \downarrow_{SCC_x^t}$ using the completeness theorem for equation 3.26 with the alternate typing 3.28 as before. Since we assumed that the identifiers from SCC_x^t were last type-checked in the i-th call, their types will be unchanged in the f'-th call and we have $TE_t \succeq TE_{f'}$ for all the identifiers in the subgraph starting at the current node. Thus, we can designate TE_f to be any environment with $f \geq f'$. Note that in this case we save recompilation of the current node even though some of its children were type-checked later than the node itself.

The other possibility is that the environment $TE_{f'}$ fails some of the assumption checks associated with the current node. In that case, we flag a recompilation of the current node at some later j-th call with $j > f'$ which reduces to the first subcase.

Thus, in each case we are able to designate an environment TE_f that passes all assumption checks and satisfies $TE_t \succeq TE_f$. □

We have shown above that the criterion for ensuring that correct type-schemes are computed for each of the top-level identifiers is to pass the upper bound type assumption checks in addition to the condition that each definition of the program is compiled at least once. We combine this criterion with the completeness lemmas of the last section to obtain the main result of this thesis in the next section.

3.6 The Correctness Theorem

With the soundness and completeness lemmas in hand, we are now ready to show the correctness theorem for our incremental system.

Note that lemmas 3.6 and 3.8 ensure that the stongly connected components and the type-schemes derived at each stage are complete, while lemmas 3.10 and 3.11 provide a sufficient criterion for them to be sound. So we express this criterion as a termination strategy for our incremental system.

Termination Strategy 1 *Assuming that all the top-level definitions in the given program are compiled at least once, the incremental compilation system terminates when all the upper bound type assumptions (M) for each of the top-level identifiers pass their check in the latest compilation environment.*

The correctness theorem follows.

Theorem 3.12 *Given a complete and type-correct program, when the incremental system terminates, it has computed exactly the same type and SCC environments as computed in a system that type-checks the complete program as a whole.*

Proof: The incremental system starts in an empty compilation environment (TE_0 and SE_0) as given by equations 3.6 and 3.7. So the completeness lemmas 3.6 and 3.8 are applicable. The termination strategy ensures that the system terminates in an environment (TE_f and SE_f) that satisfies the soundness lemmas 3.9 and 3.11. Combining the result of these lemmas we straightaway obtain that $SE_f = SE_t$ and $TE_f = TE_t$. □

3.7 Extending the System

In order to compare our system with the one that compiles a whole program at a time, we assumed in section 3.2 that we are given a complete and type-correct program that does not undergo any editing. In an actual incremental system with an integrated editor-compiler neither of these assumptions is true. New functions and procedures may be added to the program in arbitrary order and old ones may be edited at the same time. Now we will examine how the incremental system maintains consistency in the compilation environment under these circumstances. In particular, we will see how to extend our system by adding more property assumptions and checks to handle these changes while maintaining the soundness and completeness of type inference as shown in the previous sections.

3.7.1 Handling Incomplete Programs

Assumption 2 guarantees that we always handle a complete program. Relaxing this constraint, we now allow incremental extensions to an otherwise type-correct program. Note that this does not violate assumption 3, since changes to previously compiled definitions are not yet allowed.

Complete Programs

Lemma 3.10 indicates that in an incomplete program with new definitions being added to it, we only need to check for the event that all the required definitions have been seen, in order to guarantee correct SCC computation for each of the identifiers. In order to specify such a test, we need to consider what constitutes a complete program more closely.

Definition 3.7 *Given an query binding "*`it` $= e_{main}$*" to be evaluated, a* **Complete Program** *is a minimal set of definitions that are self contained with respect to this query. It is a minimal subgraph rooted at the above query, from the call graph of the universe of definitions, that does not have any dependency edges going out of it.*

It may at first seem awkward to define a complete program in terms of the requirements of a query instead of a system of file or a group of libraries etc., but it is precisely that kind of artificial definition that we were trying to avoid in the beginning of this chapter in our incremental system. The above definition is much more flexible and captures the intrinsic requirement of the final result that we wish to compute.

Strictly speaking, the above definition specifies a link-time completeness of the program which may or may not be essential to fulfill at compile-time. What we mean is that, for instance, this definition requires that we provide actual module definitions along with the current program for each module that it uses. But we can "prune" these module dependency edges from the current program by abstracting just the module interface information (and not its complete definition) and explicitly declaring it in the current program. This module interface specification will ensure elegent modular decomposition of a complete program within the framework of our definition and is, in fact, a customary practice in all languages that offer modular design.

New Strategy for Incomplete Programs

Coming back to the question of how to handle incomplete programs, we can use the definition of a complete program as given above to clearly specify the set of identifiers we are interested in. We have to make sure that all these identifiers have appeared for compilation earlier. Observe that a required identifier that has not yet been compiled, must appear in the call graph of the complete program as a singleton leaf with no dependency edges going out of it. The converse may not be true; not all leaves with no dependency edges going out of them may still be undefined, so we only need to check whether a given leaf identifier has been compiled before or not. This can be done simply by keeping a boolean flag with each identifier that it has been *seen*. Or within the current framework of compilation properties, we can test if the SCC of the given identifier in the current SCC environment is non-null[25].

[25]From equation 3.7, every identifier initially has a null SCC, but the first time it appears for compilation, it will at least have itself in its SCC.

Thus, we modify the termination strategy described in section 3.6 to work in this extended system as follows.

Termination Strategy 2 *Each time we wish to test for termination of the incremental system, we execute the following steps.*

1. *Build a static call graph with the query binding "*$\mathtt{it} = e_{main}$*" as the root.*
2. *Check if any leaf identifier from the above call graph has not yet appeared for compilation.*
3. *Check if the upper bound type assumptions (M) for all identifiers in the call graph are satisfied in the latest type environment.*
4. *If both the above mentioned tests have been cleared, then the incremental system terminates.*

Note that if any of the upper bound type asumption checks fail then the system flags the appropriate definitions for recompilation and continues. Similarly, if we find an undefined identifier in step 2, we flag a message that the program is incomplete and take appropriate action. The most obvious action would be to abandon computation of the expression query and ask the user to supply the required definition. But a particular implementation may choose to continue while warning the user about possible abnormal termination if that definition is really required in the dynamic sequence of execution. This is useful, for example, in developing large systems where the user may want to exercise localized parts of the system using appropriate test data without bothering about the whole system. See [44] for description of such a strategy.

The completeness lemmas of section 3.4 are not affected by this extension of handling an incomplete program because none of the definitions is ever removed from the program–we are only allowed to add more, and secondly no definition can be edited so that the assumption 3 is still valid. The soundness lemmas of section 3.5 are also all satisfied using an updated sufficiency criterion that includes the test for the end of the program in addition to the usual upper bound type assumption checks. Thus, the correctness theorem 3.12 will still hold under this extension using the modified termination strategy and the definition of a complete program as described above.

3.7.2 Handling Incremental Editing of Programs

Now we relax the assumption 3 and allow incremental editing of the program while we make calls to W_1 to type-check each definition. The definition of a complete program as given above still holds but we need to reexamine the correctness lemmas in this light.

We will not attempt to formally prove the correctness of type inference in presence of editing. Instead, we will informally describe how editing affects the compilation environment by means of examples and what additional book-keeping will be necessary to take care of those effects.

Some Editing Examples

It is easy to see that lemmas 3.5 and 3.7 will not hold when editing and recompilation are allowed. This is because editing may actually reduce the SCC and relax the type constraints previously enforced on a definition identifier. Consider the example below,

```
def f x = (x+1):nil;                     % N -> (list N)
def g y = hd (f y);                      % N -> N
```

The system records upper bound type assumptions of `g` as $M_g = \{f \mapsto$ `(N -> (list N))`$\}$ for later verification. Now if we edit the function `f` so as to relax its type,

```
def f x = x:nil;                         % *0 -> (list *0)
```

then, the type of `g` that was initially constrained by the type of `f` to `(N -> N)`, can now be relaxed to `(*0 -> *0)`. Observe that the upper bound assumptions M_g will still pass their checks because the new type of `f` can still instantiate its previous use. If those were the only checks performed then no retyping will occur and `g` will end up with an incomplete (overly constrained) type.

A similar example is shown below for the case where editing causes a change in the strongly connected components. Consider the following definitions,

```
def fst (x,y) = x;                       % (*0,*1) -> *0

def h x = if true then
             x
          else fst (t x x);              % *0 -> *0

def t x y = h x,h y;                     % *0 -> *0 -> (*0,*0)
```

Since `h` and `t` are mutually recursive, the type of the two arguments of `t` are constrained to be the same. Now, if we edit the definition for `h` to be the simple identity function,

```
def h x = x;                             % *0 -> *0
```

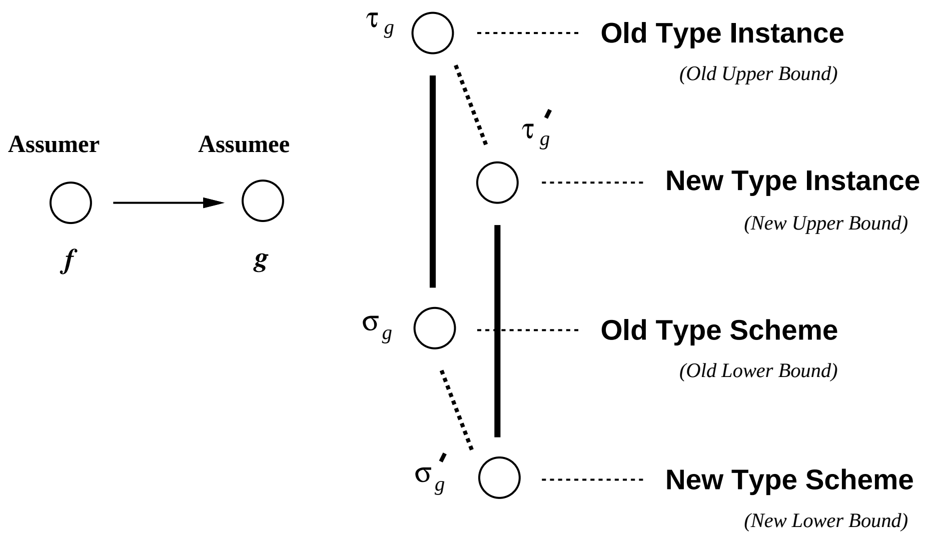
then the constraint on the arguments of `t` is no longer present since `h` can now be instantiated differently. The system has no way to detect this and flag recompilation of `t`.

It is the user's responsibility to recompile a definition after editing it but it is the compiler's responsibility to propagate those changes throughout the rest of the program and warn the user accordingly while maintaining soundness and completeness of the derived types. The problem in the first example above is that the upper bound type assumption checks help in maintaining soundness but not completeness. An upper bound type assumption $(f \mapsto \tau_f) \in M_g$ fails its check only if the latest type-scheme for `f` is more constrained than it was assumed to be during the compilation of `g`, so that we are able to catch that unsound typing and take appropriate action. But if the latest type-scheme for `f` happens to be less constrained than before then it goes unnoticed. In the second example, the type of `h` does not change even after editing, but its SCC has become smaller. That affects the type of `t` because its SCC also reduces implicitly. Without recording such implicit dependency for previous members of a SCC that has subsequently changed, it is impossible to compute complete types for them.

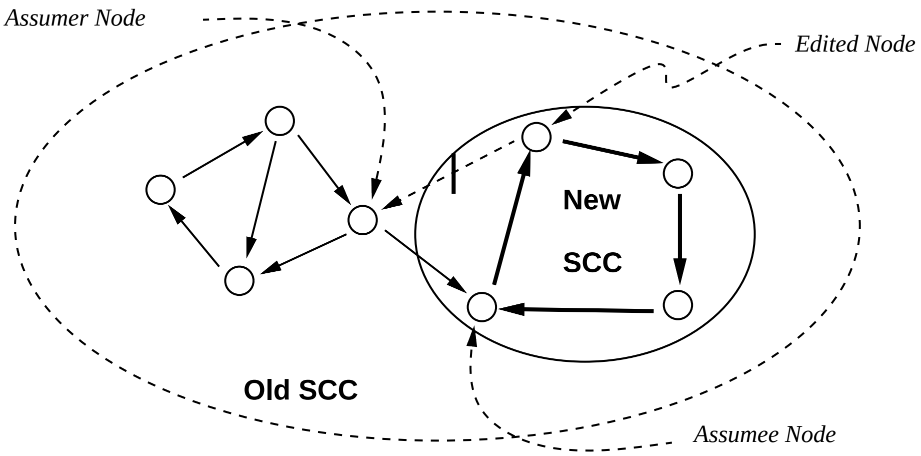
Additional Book-keeping and New Strategy for Editing

We can detect these constraint relaxations by using some additional property assumptions and checks as defined below. We can collect these extra assumptions along with the upper bound type assumptions collected already in algorithm W_1.

1. We record the actual type-schemes of free generic identifiers used within a top-level definition in an assumption set (N). The situation is pictorially depicted in figure 3.12 (a) in terms of the subsumption relationship of the type domain. If the final type-scheme (σ'_g) of the assumee identifier turns out to be more polymorphic than the type (σ_g) assumed for it earlier (depicted as being lower in the type domain hierarchy), then we can deduce that the assumee type constraint for that compilation has been relaxed and recompilation is necessary in order to propagate its effect to the assumer. The actual type instance used by the assumer, τ_g, may get relaxed to τ'_g as well. This means that the assumer will acquire a more general type than before, as in the first example above. These assumptions record a lower boundary limit for the assumee type-schemes, just like the upper bound type asumptions (M) record the upper boundary limit. Again, the verification check is



(a): Using the Lower Bound Assumptions (N) to detect Type Constraint Relaxation.



(b): Using the SCC Assumptions (Q) to detect Reduced SCCs.

Figure 3.12: Additional Assumptions to detect Constraint Relaxation during Editing.

the same subsumption test, but now in the reverse direction.

$$\begin{aligned}
&N \in \text{Lower-bd-type-assumptions} = \text{Identifiers} \xrightarrow{fin} \text{Type-Schemes} \\
&\text{COLLECT: } N_x = \{z \mapsto TE(z)\}, \qquad \forall z \in (FId(e_x) - SCC_x) \\
&\text{CHECK: } \forall (z \mapsto \sigma_z) \in N,\ \sigma_z \succeq TE(z)
\end{aligned} \tag{3.29}$$

2. We also record the SCC property of each identifier used directly by another identifier from within its own SCC in an assumption set (Q) (see figure 3.12 (b)). The verification check in this case is a test of (set) equality. This ensures that if the SCC of one of the identifiers changes, then all identifiers that were excluded from its old SCC during this change will fail this check and will be flagged for recompilation. So in our second example, the assumed SCC of `h`, {`h,t`}, will not match its latest SCC, {`h`}, and the assumer `t` will be recompiled in order to revise its SCC properties to its correct value, {`t`}.

$$\begin{aligned}
&Q \in \text{SCC-assumptions} = \text{Identifiers} \xrightarrow{fin} \mathit{powerset}(\text{Identifiers}) \\
&\text{COLLECT: } Q_x = \{z \mapsto SCC_x\}, \qquad \forall z \in (FId(e_x) \cap SCC_x) \\
&\text{CHECK: } \forall (z \mapsto SCC_z) \in Q,\ SCC_z = SE(z)
\end{aligned} \tag{3.30}$$

Our new termination strategy is now only slightly more complex.

Termination Strategy 3 *When we wish to test for termination of the extended incremental system, we execute the following steps.*

1. *Build a static call graph with the query binding "*`it` $= e_{main}$*" as the root.*
2. *Check if any leaf identifier from the above call graph has not yet appeared for compilation.*
3. *Check if the upper bound type assumptions (M), the lower bound type assumptions (N), and the SCC assumptions (Q) for each identifiers in the call graph are all satisfied in the latest compilation environment.*
4. *If all the above mentioned tests have been cleared, then the incremental system terminates.*

Modifications to Correctness Proof

We hypothesize that the correctness theorem 3.12 will still be valid under this editing extension with the modified termination strategy above. Here, we need to establish a correspondence between the final strongly connected components and the type-schemes of identifiers obtained with the above termination strategy and those obtained had the final edited program been given together as a complete program in the first place. Note that the final edited program must be type-correct in order to make this correspondence meaningful. A detailed analysis of the necessary modifications to the various lemmas of section 3.4 and 3.5 is beyond the scope of this thesis. We simply present an informal argument below.

The correspondence can be shown by considering the tail-end of the sequence of compilation environments under which each individual definition from the final edited program will get type-checked. The only difference is that the initial compilation environment for this tail-end sequence may be an arbitrary environment and not as given by equations 3.6 and 3.7. So, the type-schemes and SCCs of some definitions may have to be refined, while those for others may have to be relaxed. Technically speaking, both completness and soundness lemmas will have to be modified to deal with the state of individual identifiers placed in an arbitrary compilation environment, instead of treating all the given identifiers in a similar fashion.

Informally, we can see that the end of complete program test and the upper bound type assumption checks help to maintain soundness–they force the user to supply and compile all the required definitions at least once and flag for recompilation when there is a possibility of unsound inference due to refinement of previously assumed properties. While the SCC assumptions and the lower bound type assumption checks help to maintain completeness–they flag for recompilation when there is a possibility of incomplete inference due to constraint relaxation of previously assumed properties. Thus, starting in an arbitrary compilation environment, each of these checks will cause the corresponding properties of each of the identifiers to be pushed towards the direction of soundness and completeness, which in the terminal state will exactly match those obtained while typing the whole program together.

Also note that the user is free to edit the definitions in entirety, so that an edited definition may have a completely different type from its earlier version. But the trouble is that the old type will still be present among the cached assumptions of other dependent definitions that were compiled in the earlier environment. In such cases, both lower and upper bound type assumptions involving the edited definition will fail for the dependent definitions, forcing the user to reexamine all of them. This implies that it may be a good strategy to edit the definitions in bottom-up topological order in order to minimize the recompilation effort.

3.7.3 Comparison of Extensions

Finally, we present an overall comparison of the original incremental system with its two extensions in figure 3.13.

The Incremental System and its Extensions		
The Basic System	Incomplete Program Extn.	Editing Extn.
Assumptions		
1. A complete program is given. 2. No editing allowed.	1. No editing allowed.	None.
Definition of a Complete Program		
The given program.	Nodes in the call graph of the expression query.	Nodes in the call graph of the expression query.
Property Assumptions Collected		
1. Upper bound type assumptions (M).	1. Upper bound type assumptions (M).	1. Upper bound type assumptions (M). 2. Lower bound type assumptions (N). 3. SCC assumptions (Q).
Termination Strategy		
1. Each definition from the given program has been compiled at least once. 2. Assumptions M pass their checks.	1. Each singleton leaf identifier in the query's call graph has been compiled before. 2. Assumptions M pass their checks.	1. Each singleton leaf identifier in the query's call graph has been compiled before. 2. Assumptions M, N, and Q pass their checks.
The Correctness Theorem (W.r.t the Type and SCC Properties derived in the Non-Incremental System)		
On termination, the properties for each identifier in the given program are exactly the same.	On termination, the properties for each node in the query's call graph are exactly the same.	On termination, the properties for each node in the query's call graph from the final edited program are exactly the same. (Not Shown)

Figure 3.13: A Comparison of the Incremental System with its Extensions.

3.8 Complexity of the Incremental System

In this section we will compare the complexity of our incremental system with that of the complete program system. We make this comparison for each of the properties computed in our incremental system. As described in the last section, the book-keeping is more complex when editing is permitted on the program, so we analyze these two situations separately.

Analysis of the incremental system without editing will characterize the reduced cost of type-checking an incremental addition to the program as against retyping the whole program. We will also delineate the book-keeping overheads involved in this task. Analysis of the incremental system with editing will tell us about the additional overhead for permitting editing. Here, we will attempt to bound the cost per editing change.

3.8.1 Complexity of the SCC Computation

In the complete program case, it may be the user's responsibility to properly group all definitions into mutually recursive groups and to reorder them bottom up. Or else, the compiler can do that once in the beginning using the SCC algorithm described in [1], which also identifies the correct topological ordering of the definitions. This can be done in time and space which is linear in the size of the complete static call graph ($O(n+e)$, where n is the total number of nodes in the complete static call graph and e is the number of edges).

In the case of the incremental system without editing, there are no book-keeping overheads for SCC computation since there are no SCC assumptions to keep track of and no recompilations flagged thereof. The space requirement is the same $O(n+e)$, since we only need the "free-identifiers" (FId) local property for each definition and always compute the latest static call graph afresh. As for the time complexity, the call graph is computed each time a definition is compiled or recompiled. So the SCC cost of an incremental addition to the program is the cost of computing the static call graph of the program. Note that it is sufficient to compute only the call graph rooted at the given node at each stage which may be appreciably smaller. In that case, the amount of work involved depends heavily on the topology of the call graph as well as on the actual order in which the nodes are presented.

The recompilations performed due to the failure of upper bound type assumptions also recompute the SCC property of the nodes involved. This is again proportional to the size of the static call graph. We will attribute this cost to the type computation.

When editing is allowed, there is a topology dependent space overhead for keeping the SCC assumptions. Note that the assumptions for an assumer identifier consist of the strongly connected components of its free identifiers that are in its own SCC. This could be as large as $O(n^3)$, as in the case of a completely connected graph. It could also be zero, as in the case of an acyclic graph. In most real situations, the size of the SCCs is bounded by a small constant, so that this overhead is only $O(n)$ and the overall space complexity is still $O(n+e)$. As for the time complexity, failed SCC assumptions for a node leads to the recompilation of all nodes that were previously in its SCC but now have been excluded from it. Thus, the SCC cost of editing a node is bounded by the cost of recompiling all nodes in its old SCC.

An interesting observation is that recompilation of nodes that failed their SCC assumptions resets the SCC property of all the identifiers in their own SCC and hence might alleviate other SCC assumption failures. But this might trigger other lower bound type assumption failures, due to constraint relaxation as described in the last section. We will attribute that cost to the type computation.

It is clear from the above discussion that the space and time overhead in our system for the SCC computation is topology dependent. It is also dependent on the order of compilation of definitions and closely interacts with cost of type computation. A more detailed analysis is beyond the scope of this thesis.

3.8.2 Complexity of the Type Computation

The complexity of the type computation is the most crucial aspect of our incremental system. We identify four distinct levels of computational complexity.

1. We mentioned in section 2.5 that the Milner's type inference algorithm has an intrinsic exponential complexity. This is not only because of the potentially exponential size of the final type of a given program (see examples in [29]) but also due to the intrinsic complexity of simply determining whether the program is type correct or not. This complexity is completely independent of whether we compute the type incrementally or not, so we will disregard this aspect in further discussion.

2. The basic computation vehicle is the unification algorithm. There are many algorithms for unification in the literature (see [35] for a survey), with computational complexity varying from linear to exponential in the size of the expressions being unified. Our implementation

leans more towards pragmatic simplicity than asymptotic efficiency, and is a variation of Huet's algorithm [28] based on the almost linear Union-Find algorithm [1]. Again, the choice of an implementation is independent of our incremental book-keeping mechanism so we will not discuss this aspect any further.

3. The essential difference between the type computation done in the complete program system from that in our incremental system is the number of times unification has to be performed which amounts to estimating the number of recompilations that may be necessary in our system. We will focus our analysis on this aspect.

4. Finally, the cost of computing, maintaining, and using the book-keeping information is an added overhead in the computational complexity of our system, which we will also characterize.

During our analysis, we will measure the space overhead by the number of extra type expressions we have to carry as compilation properties and assumptions. We will measure the time overhead by the extra work necessary in each call to algorithm W_1 in order to construct and use these type expressions, the overhead of checking the assumptions time and again, and most importantly, the number of recompilations performed due to the failures of these assumptions.

Space Overhead in Type Computation

It is clear that the space overhead is proportional to the number of occurrences of top-level identifiers in the given program. The assumptions collected from algorithm W' simply record the type expressions associated with each such occurrence in every top-level definition. The upper bound type assumptions record one type expression for each occurrence of a generic free identifier, which in the case of an acyclic call graph amounts to recording all occurrences. If m is the total number of occurrences of top-level identifiers in the program and n is the total number of top-level identifier definitions, then the space overhead in computing the assumptions and storing the upper bound type assumptions is bounded by $O(m+n)$ for the incremental system without editing.

For the case when editing is allowed, we also collect lower bound type assumptions, exactly one type-scheme per free, generic identifier in every top-level definition, so the total overhead is bounded by the number of edges e in the static call graph. Note that this number may be substantially smaller than m because an identifier occurring several times within the same definition will be counted only once in the static call graph. So the overhead is still bounded by $O(m+n)$[26].

Time Overhead in Type Computation

In the complete program system, the algorithm W recursively descends through the expression structure of the given program in topological order, typing all bindings of an SCC together. Each top-level binding is type-checked exactly once.

In the incremental system a similar recursive descent is made internally for each top-level definition in the call to W'. Collecting assumptions at each stage does not involve any extra computation. Since we treat each SCC as a supernode, each call to W_1 type-checks all nodes in the current SCC together and matches the cost of type-checking a complete SCC in the complete program system. But allowing the definitions to be supplied incrementally and in arbitrary order opens up the possibility of recompilations due to failed assumption checks, editing, and newly supplied bindings. The overhead of each such request for recompilation of a definition is the cost of retyping its complete SCC. Below, we analyze the total cost of making an incremental change to the program including the cost of recompilations caused by it.

When either incremental additions are being made to the program or some previously supplied definition is being edited, the SCCs of the involved definition nodes may change substantially. There, we may justifiably (re)type the complete SCC for each user initiated (re)compilation. But such changes potentially affect all the topologically preceeding definitions in the call graph. Recompilations of those definitions is the extra price paid by the user for incremental flexibility, because it could be avoided if the definitions were given in the right toplogical order. Our mechanism of maintaining upper and lower bound assumptions attempts to reduce this set of affected definitions. When the definition of an identifier appearing later in the incremental sequence of compilations falls within the assumed bounds, no recompilation is flagged. Even when these assumptions fail, we can localize the retyping to just the assumer identifier and its SCC instead of the whole program. But in the worst case each such recompilation can give rise to more failures in the topologically preceeding SCCs and we may end up recompiling the whole program.

As an example, consider the set of definitions in figure 3.14 (a). The call graph in this case

[26]Note that for most cases (connected graphs that are not trees), we have $n \leq e \leq m$, so that the total space complexity can be bounded by $O(m)$.

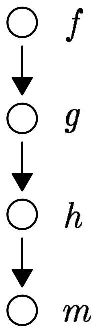
```
def f x = g x;

def g x = h x;

def h x = m x;

def m x = x:nil;
```

(a)



(b): Static Call Graph.

```
def f x = g x;

def g x = (h x):nil;

def h x = (m x):nil;

def m x = x:nil;
```

(c)

Figure 3.14: Example program to illustrate the high number of Recompilations.

is a simple chain as shown in (b). The upper bound type assumptions of the identifier `h` fails its checks as soon as `m` is defined. A subsequent recompilation of `h` propagates the type information of `m` onto `h` and causes the failure of assumption checks of `g` and so on. Thus each one of `f`, `g`, and `h` needs to be recompiled once (but in the reverse order) in order for its type to reflect the latest type information due to `m`. This example shows that the number of recompilations could be linear in the size of the program. Note that a simple strategy such as compiling the whole set of definitions twice does not help since the order of recompilation is important. In fact, that strategy will perform a quadratic number of recompilations in this example since on the second round of compilations the definitions of `f` and `g` will still not see the latest updated type of `h`, and `g` will fail its assumption checks nevertheless.

It is further possible to amortize the cost of retyping an SCC over several assumption failures by adopting a lazy recompilation policy (as against an eager one). The idea is as follows: a recompilation is flagged when some significant change is detected in the program, therefore it may help to postpone the recompilations as late as possible so as to let all the changes accumulate. To see an example of this, consider the program in figure 3.14 (c) which has the same call graph as in (b). Now the upper bound type assumptions of `f` fail as soon as `g` is introduced. If we choose to eagerly recompile all definitions as soon as their recompilation is flagged, we will end up compiling `f` d times where d is the DAG-height[27] of the call graph, because its type gets affected by each of the subsequent definitions. Similarly `g` will be compiled $d-1$ times and so on. In call graphs that have large depth, so that $d = O(n)$, we will end

[27] Collapsing each SCC into a single node, the static call graph becomes a directed acyclic graph (DAG). DAG-height refers to the longest path from the root of this DAG to some leaf node.

up doing $O(n^2)$ recompilations in total with this eager strategy. Whereas if we postponed all the recompilations till the end, a single sweep of recompilations in the reverse order will settle all the types to their correct values. When editing is allowed, the number of recompilations is again dependent on the DAG-height and the strategy used. The only difference is that some definitions may need to relax their types while others may need to refine them.

Thus, it is clear that the total time and space overhead for the type computation depends on the topology as well as the strategy of recompilation. For most real situations, where the size of the SCCs can be bounded by a small constant, the worst case overhead for type-checking a given program in the incremental system is still proportional to the size of the program, but on an average may turn out to be much more localized if we delay the recompilations as much as possible and do them in the right order. A more detailed analysis is beyond the scope of this thesis.

3.8.3 Comments on Complexity

Note that the overhead for computing the SCCs is proportional to the size of the static call graph, while that for the type computation is proportional to the size of the actual program. As mentioned earlier, the size of the static call graph may be much smaller than the size of the program. Moreover for the type computation, the actual overhead is weighted by the size of the type expressions which could themselves be exponentially large, so we usually ignore the overhead due to the SCC computation even though it may be quadratic in the size of the call graph.

Even though the worst case complexity of our system is no better than recompiling the whole program again, judging by our experience with its implementation, it is substantially smaller than that on an average. The overhead of incremental book-keeping is usually more than offset by avoiding the recompilation of unrelated or unaffected nodes. Further reserach is necessary in this direction to quantify the average complexity of this system over a variety of programs.

Polymorphism is the other aspect from which our system derives benefit. Note that the assumptions about the upper and lower bounds of the types of top-level identifiers provide a spring like resilience to the type system. As long as the actual polymorphic type of a top-level assumee identifier used by an assumer identifier conforms to these limits recorded in the assumptions of the assumer, the assumer need not be recompiled. If there was no polymorphism in the type system then there would not be separate upper and lower type limits. There would only be a single monomorphic assumed type tested for equality against the current type of the assumee and the slightest change in its type would force a recompilation of the assumer.

3.9 Discussion

3.9.1 Recompilation *vs* Retyping

We have been vague in our terminology for the compiler action when some property assumptions fail their checks, using the terms recompilation and retyping somewhat loosely and interchangeably. Now we will make a distinction between the two.

Arbitrary editing may involve not only correcting erring definitions but possibly redefining some definitions altogether. We need to distinguish these cases of true recompilation of definitions where no previous property of the redefined top-level identifier can be allowed to be retained by the compiler, from the cases of recompilation issued by the incremental type system itself when some assumer identifier fails the assumption checks. The latter is a request to simply recompute the global properties of an unchanged assumer definition in order to incorporate the effects of the changes in the properties of its neighboring nodes. None of the local compilation properties change for that definition and can be retained from the previous compilation. We will call this recomputation as **retyping** as against the former situation which we will continue to call **recompilation**.

Note that retyping is properly subsumed by recompilation so an implementation may choose to process all requests for retyping as requests for recompilation. But the distinction is necessary for the benefit of an implementation that significantly improves performance by retaining all the unchanged properties of the top-level identifiers and does retyping on the fly when some assumptions fail.

The other significant difference between retyping and recompilation is that recompilation is usually user initiated while retyping is always requested by the inference system itself. Doing retyping is safe when it is guaranteed that the given program has not changed textually. This is true for the case when no editing is allowed as well as in the general case after the compiler has been invoked on a given set of definitions[28]. Then, all internal requests for retyping, initiated due to the failure of some assumptions, can be processed automatically within the

[28]Unless we have a compiler that *corrects* programs!

compiler followed by a retry for the failed checks. If during the process of retyping a type-error is encountered, the compiler can then abort the process and request a recompilation because some user editing is necessary.

3.9.2 Optimizations

It is possible to further reduce the cost of an incremental change to the program, if it does not affect the properties of other nodes. In our current scheme, whenever a node is flagged for recompilation, we recompile all the definitions in its latest SCC. This is necessary when the new SCC of the node is different from its old one. But this may be wasteful if the editing changes do not affect the SCC or the type properties of the node. Indeed, the only computation really necessary in such cases is the compilation of the edited node itself.

Even when the type of the edited node is different but its SCC is the same, it may not be necessary to recompile all the other definitions in its SCC. This is because type-checking a given definition does not use any information about other definitions in its SCC (apart from the fact that they are all in the same SCC) until the latter part of algorithm W_1 when we unify all the assumed types of the definitions with their computed types in a unification matrix (see figure 3.5). Therefore, if we save the types of SCC identifiers assumed while typing a given definition, we can use them directly in the latter part of the W_1 algorithm when some other identifier from the same SCC is being recompiled.

We show the updated algorithm W_2 in figure 3.15, where we have also incorporated the collection of the SCC and the lower bound type assumptions introduced in section 3.7. The algorithm now works in two phases.

The first phase computes the SCC of the given definition and compares it with its earlier value. If the SCC has changed then we proceed as before, compiling each of the definitions belonging to the new SCC afresh and accumulating their properties. We also record the locally inferred type of each identifier and its local type assumptions about other identifiers in its SCC for future use. If the SCC has not changed from its earlier value, then it implies that only the current binding needs to be compiled and we can use previously saved local properties of the other identifiers of that SCC in the second phase, instead of computing all of them afresh.

The second phase, as before, constructs a unification matrix using the freshly computed or previously saved local properties of all the identifiers of the SCC and updates their type properties. It also records the upper and the lower bound type assumptions for the identifiers

Def $W_2(TE_{i-1}, SE_{i-1}, B_x)$ =

PHASE I:

Let B_x be the binding '$x = e_x$'.
Construct the current call graph G_i using the free identifiers (FId) property.
$SCC_x^i = \{z_1, \ldots, x, \ldots, z_n\}$ be the current strongly connected component of x in G_i.

if $SCC_x^i \neq SE_{i-1}(x)$ **then**
SCC Environment: $SE_i = SE_{i-1} \setminus_{SCC_x^i} + \{z_k \mapsto SCC_x^i\}$ $\qquad \forall z_k \in SCC_x^i$.

for each top-level identifier $z_k \in SCC_x^i$ **do**
Let e_k be the RHS expressions associated with z_k.
Let $\beta_{k1}, \ldots, \beta_{kn}$ be new type variables.
$New\text{-}TE_k^i = TE_{i-1} \setminus_{SCC_x^i} + \{z_1 \mapsto \beta_{k1}, \ldots, z_n \mapsto \beta_{kn}\}$.
$W'(New\text{-}TE_k^i, e_k) = (S_k^i, \tau_{z_k}^i, A_k^i)$.
Local Assumptions: $A_k^{local} = \{z_1 \mapsto S_k^i \beta_{k1}, \ldots, z_n \mapsto S_k^i \beta_{kn}\}$.
endfor.

Local Assumption Environment: $AE_i = AE_{i-1} \setminus_{SCC_x^i} + \{z_k \mapsto A_k^{local}\}$ $\qquad \forall z_k \in SCC_x^i$.
Local Types Environment: $WE_i = WE_{i-1} \setminus_{SCC_x^i} + \{z_k \mapsto \tau_{z_k}^i\}$ $\qquad \forall z_k \in SCC_x^i$.
else
Let $\beta_1, \ldots, \beta_n$ be new type variables.
$New\text{-}TE_x^i = TE_{i-1} \setminus_{SCC_x^i} + \{z_1 \mapsto \beta_1, \ldots, z_n \mapsto \beta_n\}$.
$W'(New\text{-}TE_x^i, e_x) = (S_x^i, \tau_x^i, A_x^i)$.

Local Assumptions: $A_x^{local} = \{z_1 \mapsto S_x^i \beta_1, \ldots, z_n \mapsto S_x^i \beta_n\}$.
Local Assumption Environment: $AE_i = AE_{i-1} \setminus_{\{x\}} + \{x \mapsto A_x^{local}\}$.
Local Type Enironment: $WE_i = WE_{i-1} \setminus_{\{x\}} + \{x \mapsto \tau_x^i\}$.
endif.

PHASE II:

$$U_i = \textit{Unify-Columns} \begin{pmatrix} AE_i(z_1)(z_1) & \cdots & AE_i(z_1)(z_k) & \cdots & AE_i(z_1)(z_n) \\ \vdots & & \vdots & & \vdots \\ AE_i(z_n)(z_1) & \cdots & AE_i(z_n)(z_k) & \cdots & AE_i(z_n)(z_n) \\ WE_i(z_1) & \cdots & WE_i(z_k) & \cdots & WE_i(z_n) \end{pmatrix}$$

Type Environment: $TE_i = TE_{i-1} \setminus_{SCC_x^i} + \{z_k \mapsto close(U_i\, WE_i(z_k))\}$ $\qquad \forall z_k \in SCC_x^i$.

if only x was type-checked in PHASE I **then**
Upper Bound Type Assumptions: $M_x = U_i A_x^i \setminus_{SCC_x^i}$.
Lower Bound Type Assumptions: $N_x = \{z_p \mapsto TE_{i-1}(z_p)\}$ $\qquad \forall z_p \in (FId(e_x) - SCC_x^i)$.
else
Upper Bound Type Assumptions: $M_{z_k} = U_i A_k^i \setminus_{SCC_x^i}$ $\qquad \forall z_k \in SCC_x^i$.
Lower Bound Type Assumptions: $N_{z_k} = \{z_p \mapsto TE_{i-1}(z_p)\}$ $\qquad \forall z_p \in (FId(e_k) - SCC_x^i)$.
$\forall z_k \in SCC_x^i$.

endif.

Return (TE_i, SE_i).

Figure 3.15: The Updated Incremental Algorithm W_2.

compiled in the first phase.

The local properties we need to save in the first phase essentially correspond to all the type information necessary to reproduce the unification matrix in the second phase. These properties are local because they are independently computed for each identifier in a SCC and do not use any information about other identifiers from the same SCC. We need to maintain the following two local properties.

1. Local property *Local Type* $=$ (Types, $=$). For each identifier x, we record its local type τ_x^i computed in the call to W'. We save this property in a map from identifiers to types called Local-Type-Environment WE.

$$WE \in \text{Local-Type-Environments} = \text{Identifiers} \xrightarrow{fin} \text{Types} \tag{3.31}$$

2. Local property *Local Assumptions* $=$ (Local-Type-Environments, $=$). For each identifier x, its local assumption set A_x^{local} is a map from the identifiers in its SCC to their types as inferred from the call to W'. This is later used in the second phase for unification with similar local assumptions and actual inferred type of other identifiers in the same SCC. We collect this property in Local-Assumption-Environment AE.

$$AE \in \text{Local-Assumption-Environments} = \text{Identifiers} \xrightarrow{fin} \text{Local-Type-Environments} \tag{3.32}$$

Our incremental algorithm is now complete. It type-checks a set of definitions incrementally, allowing intermediate editing. It also attempts to minimize the overhead of recompilations by saving as much local information as possible while making sure that the types inferred are both sound and complete.

3.9.3 General Comments

While developing algorithms W_1 and W_2, we took some basic design decisions. Now we will discuss them with a general perspective.

The following points emerge by examining the two incremental algorithms.

1. Types and strongly connected components are closely related properties. This is a manifestation of the Hindley/Milner polymorphic type inference system which we described in detail in the last chapter. Therefore, we have to deal with both properties simultaneously in any implementation of this type system.

2. Even though the physical unit of compilation is a single top-level definition, our algorithms use a single SCC as a logical unit for type-checking. Again, this is a characteristic of the Hindley/Milner type system. We have to deal with definitions that are within an SCC differently from those that are outside it. Moreover, the definitions within an SCC are a tightly knit unit because their types are closely dependent upon each other. So it is logical to treat them as a unit.

3. With the above logical division in mind, we maintain the consistency of the compilation environment within a SCC unit using the SCC assumptions and outside it using the type assumptions. This is to obtain maximum advantage out of the polymorphism of the system. We do not care about changes in the SCCs of two definitions as long as they satisfy the type asumptions and are in different SCCs.

4. The compilation property domains $\mathcal{D}$ are entirely syntactic entities. They have a syntactic structure generated according to the associated predicate $\sqsubseteq$. For incremental property computation, it is necessary that these domains have a syntactic "bottom" element with respect to the predicate relation. This element denotes "no information" and has to be consistent with all the other elements of the domain. In practice, this element becomes the initial value of that property which then subsequently gets refined to one of the elements of the domain. For example, $\forall\alpha.\alpha$ is the bottom element of the type property and the null set ϕ is the bottom element of the SCC property. Also, the incremental property computation needs to be a syntactically monotonic operation over the domain. Otherwise, our mechanism of computing repeated refinements of a property starting from the bottom element of the domain will not be correct.

5. The incremental book-keeping mechanism we described in section 3.1.2 is general enough to capture almost all interesting incremental compilation properties. Indeed, this system, first implemented by Traub [59], forms the backbone of all the incremental book-keeping done inside the Id Compiler developed at the Laboratory for Computer Science, MIT. But this system only specifies a basic mechanism for incremental property maintenance. It does not guarantee correctness for an arbitrary property. As we saw in earlier sections, soundness and completeness of incremental property inference is closely tied to the peculiarities of the property domain, the actual strategy used in computing and verifying the property, and the inter-relationship among the various properties being collected.

3.9.4 Related Work

Our work has grown out of the ideas presented by Nikhil in [44]. Subsequently, Toyn *et al.*[58] used some of those ideas in their GLIDE system. Even though their incremental type inference system is based on the same principles as ours, it is very different in detail.

The GLIDE system maintains much more fine-grain type information than our definition based book-keeping. They record type constraints for every identifier or literal in the form of its expected and inferred type. (Re)typing a definition modifies the set of type constraints maintained within the system and a (re)unification of the latest constraints determines the type of all identifiers. Keeping this fine-grain information allows them to reuse a lot of type information from within a definition during retyping instead of computing it afresh as in our system. But they do not take advantage of polymorphism of the type system in identifying the set of type constraints that need to be revised due to a change in the type of an identifier. Their consistency checks are based on syntactic sharing of type variables rather than upper and lower bounds in a structured type domain.

Another important feature of the GLIDE system is that they have an automatic loading mechanism that automatically compiles, loads, and links a definition only when it is actually needed in a computation. Thus all type-checking occurs on the fly at run-time, and it is important to resolve all type inconsistencies immediately so that the computation may proceed further. Therefore, they need to reuse as much type information as possible in order to cut down the retyping cost. This is completely in contrast with our approach of compile-time, static type-checking, where the user picks the order of (re)compilation of definitions and the system only informs what definitions need to be recompiled. In this way, our system can tolerate temporary type inconsistencies among definitions and reduce the number of recompilations by intelligent scheduling[29], while still guaranteeing the correctness of inferred types before run-time.

There are few other programming environments in the literature that have attempted to incorporate incrementality. In [39], Meertens describes an incremental polymorphic type inference scheme for the language B. Even though the language specific issues are quite different, his system incrementally maintains fine-grain type constraints like those in the GLIDE system.

The AIMLESS system developed at the Australian National University as part of the programming environment of ANU ML [51] takes a completely different approach. Here, the user is allowed to edit definitions in an editor buffer and then compile and use them in a ML script

[29]The reader is referred back to section 3.8.2 where we discuss eager *vs* lazy recompilation strategies.

that represents an interactive user session. The system records and maintains all dynamic information regarding the session. To edit a previously compiled definition, all phrases upto that definition, from most recent to the least recent, are moved from the script to the buffer and all their bindings and effects are *undone*. After the editing, they are moved back to the script, being recompiled in the process, and any discrepencies found are reported. It is possible to minimize the recompilation cost by saving some previous information.

The POPLOG [62] system is an umbrella for a variety of languages, including ML, Prolog, and LISP and it permits incremental compilation of functions. However, it requires type information to be presented in the right order and is therefore is not as flexible as our system.

3.9.5 Current Status

Finally, we want to state that the incremental type system presented in this chapter has been implemented as a module of the Id Compiler developed at Laboratory for Computer Science, MIT, and has been successfully running for the past two years. The implementation has helped in our understanding of pragmatic issues such as the incremental efficiency and the book-keeping overhead, as well as subtle theoretic issues in proving the correctness of the algorithm.

Technically, the current implementation is a slight variation of the algorithm W_1 presented in this chapter. It does not record the lower bound type assumptions (N), though it records the SCC assumptions (Q). Optimizations discussed in section 3.9.2 are also not implemented yet. The system does not perform any automatic retyping and has a lazy recompilation strategy–it checks the property assumptions for consistency only when an expression query is supplied and flags the appropriate definitions to be (re)compiled at that time. Our flexible implementation allows us to experiment with these ideas freely.

We plan to strengthen the system with the proposed extensions and optimizations. Our experience till now regarding the efficiency of the book-keeping mechanism and the flexibility offered by the definition level incrementality has been extremely favorable.

Chapter 4

Overloading

Overloading of identifiers, also known as *ad hoc* polymorphism, has appeared in conventional programming languages since their inception. In fact, in many situations it is so smoothly incorporated into the language syntax that it is hard to percieve the implicit sharing of names among various functions. As a working definition, we will say that an identifier is **overloaded** when it is used to represent different and possibly unrelated functions in different type contexts.

An example of such seamless interlinking between the overloading of identifiers and the language syntax can be seen in the way many conventional languages deal with arithmetic operators. Fortran and C use the same arithmetic operator symbols (`+,-,*,/` etc.) to do integer as well as floating point arithmetic. In Basic and some versions of Fortran, their interpretation extends even to vectors and matrices. The underlying functions (program code) that implement these operators are very different in each type context, but in order to reduce syntactic clutter and enhance readability, the language syntax names them all with the usual names of arithmetic operators. Such sharing of names is usually limited to a small set of pre-concieved identifiers and is, therefore, non-extensible. In most cases, the individual compilers recognize this sharing and generate the appropriate code in an *ad hoc* fashion. Only recently has there been some attempt to systematize the mechanism involved in this name sharing and to extend it to allow explicit user-defined overloading of identifiers in context of strongly typed, functional languages [27, 63].

In Id, we take a position close to that proposed by Wadler and Blott in [63]. In this chapter, we describe our proposal for explicit, user-defined overloading of identifiers and a systematic mechanism for the compiler to resolve and then translate such programs into appropriate code. A preliminary version of this proposal was described by Nikhil in [47] for internal circulation.

This chapeter is organized as follows. In section 4.1, we discuss the desirable features of an overloading scheme and its relationship to parametric polymorphism. In section 4.2, we characterize a way to explicitly declare overloaded identifiers and their instances and our strategy to resolve them during compilation. Subsequently in section 4.3, we show two mechanisms of translation, parameterization and specialization and debate the advantages of one over the other. Finally in section 4.4, we investigate the relationship between our proposal and that of Wadler and Blott in [63].

4.1 Issues in Overloading

There are several issues to consider in a possible scheme for overloading. The following discussion is geared towards the static, compile-time type analysis that we have chosen to follow. So even though we mention various issues and discuss their implications, we choose to emphasize those that either support or fit well into our approach.

4.1.1 Overloaded Identifiers and Resolution Choices

The most important aspect of overloading resolution is indicating what identifiers are overloaded and what are their possible resolution instances. Various schemes differ in various ways; whether there is a fixed, built-in set of overloaded identifiers with resolution choices specified by the language, or by a standard library; whether general, user-defined extensions are permitted; whether there is a finite number of choices or infinitely many; whether the resolution choices are specified/resolved at compile-time or run-time etc. The following features are desirable:

1. The set of overloaded identifiers should be user-declarable and extensible.
2. The set of resolution instances for an overloaded identifier should also be user-declarable and extensible.
3. The overloaded identifiers and their instances should be specified[1] and resolved at compile-time.
4. The process of resolving an overloaded identifier into one of the possible instances should be independent of the order of specification of instances and their position in the program. To this end, only non-overlapping instances may be permitted. Or, in other words, there

[1]Possibly in a library of standard declarations to be used as a prelude to the programs.

should not be any common contexts in which two or more instances are simultaneously applicable.

5. The process of resolution should always terminate. A possible measure to ensure this is to allow only a finite number of different instances for any given overloaded identifier.

As an example, we can declare that "`+`" is overloaded and has two resolution instances, `iplus` and `fplus`, the primitive addition operations for integers and floating point numbers respectively[2].

```
overload (+) :: I -> I -> I = iplus;
overload (+) :: F -> F -> F = fplus;
```

Then the following overloading resolutions and type inferences can be made:

```
... x + 2 ...  ⟹  ... (iplus x 2) ...
... y + 3.14 ...  ⟹  ... (fplus y 3.14) ...
... (a + 2) + 3.14 ... ⟹ TYPE ERROR: No resolution instance for + exists
                                      with type I -> F -> *0
```

Note that in the last example a type error was inferred because we have not declared any resolution instance for "`+`" to handle mixed types. Admitting only finite number of (compile-time specified) instances permits the compiler to enumerate and match the choices against the furnished context and then decide if it has found a valid resolution (*i.e.*meaning) for the overloaded identifier. Countably infinite number of choices are also acceptable as long as there is a tractable decision procedure to resolve among them and show validity of the resolution for the given type context.

Such a decision procedure also helps to distinguish decidedly improper uses of the overloaded identifier from the ones that lack sufficient contextual information. Otherwise, this decision has to be postponed until run-time when it can use the most recent list of instances for the actual objects involved. For example, if we declare equality (`==`) to be universally applicable to all types (either by making it fully polymorphic or by supplying a default resolution instance for every type), then applying it to functions or abstract data types will not generate a type error, as we may want it to. Indeed, in Miranda where equality has polymorphic status, an equality test over functions is caught only at run-time, and that over abstract objects is not caught at all and degenerates to an equality test over their representations[3].

[2]We will present the actual Id syntax for explicit declaration of overloaded identifiers and their instances in a later section. Till then, we request the reader to bear with us. The syntax,

`overload` *overloaded-identifier* `::` *type context* `=` *instance-identifier*`;`

declares the given instance identifier as an instance of the overloaded identifier for the specified type context.

[3]This may be regarded as a breach of the abstraction.

As a matter of clean semantics, we may wish to keep the various instances mutually non-overlapping. This in turn implies that the compiler is free to match the choices in any order and the user is free to declare them in any order without affecting the overall result of resolution. This property is particularly useful in an incremental compilation environment like ours.

Finally, different languages may permit the definition of resolution instances at different times. As in Fortran, C, or Pascal, it may occur at language design time only. In some cases, we may restrict it to only pre-defined operators and identifiers declared in a separate library or a standard prelude. Or, we may give the user full freedom to declare overloaded identifiers and their instances in their programs. In Id, we follow a mixed approach: while the usual arithmetic and relational operators etc. are declared in a standard basic library, the user has complete control over declaring his/her new overloaded identifiers, their instances or even extending the set of instances of the pre-defined ones. Thus, it is conceivable to have the following declaration as part of the math library, to be used for complex algebra,

```
def cmplxadd (x1,x2) (y1,y2) = x1+y1,x2+y2;
overload (+) :: (F,F) -> (F,F) -> (F,F) = cmplxadd;
```

In a language with module structure, we may, in fact, like to package up this definition in a module which can then be "used" by various applications. This is desirable, since we do not, in general, want to define addition on pairs of floating point numbers. This should be allowed only in context of complex numbers as declared here.

4.1.2 Propagation of Overloading Information

Consider the following function:

```
def double x = x + x;

...(double 2)...
...(double 2.718)...
```

What is the type of `double`? Is it `(I -> I)` as an integer function, or `(F -> F)` denoting a floating point function, or some generalization of both? Further, when does the type of `double` get completely known, at the time of its definition, or its use in a particular context, or at run time? In a polymorphic language like ours, we would like to declare only one such function and apply it to both integers and floating point numbers in different contexts as shown in the example above. In fact, keeping in mind that the identifier "`+`" is overloaded, we may wish to say that `double` is automatically overloaded with implicit instance declarations corresponding to every instance of "+". In other words, double is overloaded over any type that admits the addition operation. In ML, such an assertion is possible only for the equality operator (==) which is treated specially. Thus it is possible to define a generic list-membership function in ML, similar to the one shown below.

```
def member x nil = false
 |..member x (y:ys) = (x==y) or member x ys;

typeof member = *0(==) -> (list *0(==)) -> B;
```

The $*0_{(==)}$ denotes a type variable that is allowed to range only over types that admit equality and is called an *eq-type variable* in ML. We would, of course, like a solution that is universal.

Another problem with this implicit propagation of overloading is that the number of possible instances can grow very rapidly. For example we may define,

```
def doublepair x y = double x, double y;
```

Now, there are four possible instances for doublepair[4], taking arguments of type integer and integer, integer and float, float and integer, and float and float, respectively. We would like to handle this explosive situation appropriately in our overloading scheme.

4.1.3 Conditional Instances and Recursion

Combining the two concepts presented above, namely, declaring new resolution instances and propagating unresolved overloading information to other identifiers, poses a new challange to an overloading scheme. To see this, consider the following example of vector addition[5]:

```
def vadd X Y =
  { l,u = bounds X;
    in
     {array (l,u)
```

[4]Assuming that "+" (and hence double) has integer and floating point instances only.

[5]The syntax is called an "array comprehension":

```
a  =  {array bounds
          | [subscript] = value || generator
          ⋮
          | [subscript] = value || generator}
```

Essentially, this declares a one dimensional array with the specified bounds expressed as a tuple, and a sequence of filling clauses that fill the specified value at the specified subscript. The value and the subscript expression are evaluated at each of the values generated by a generator, which could be a nested combination of list producing expressions filterable through predicates. See [48] for details.

```
        | [i] = X[i]+Y[i] || i <- 1 to u}};
overload (+) :: (Vector *0) -> (Vector *0) -> (Vector *0) = vadd;
```

The challange here is to allow `vadd` as a resolution instance of "+". The idea is to be able to say that if `X` and `Y` are vectors, then `X+Y` is their vector addition defined by `vadd`. Now, note that the addition of the components `X[i]+Y[i]` within the definition is not fully resolved. It really depends on the type of the components of vectors `X` and `Y`. For example, if `X` and `Y` are integer arrays then `X[i]+Y[i]` should simply resolve to `(iplus X[i] Y[i])`, and if they are arrays of floating point numbers then the addition should resolve to `(fplus X[i] Y[i])` etc. In general, `vadd` is implicitly overloaded over vectors of any type whose components admit the addition operation. The declaration of `vadd` as an instance of "+" creates a recursive instance resolution problem: when components of `X` and `Y` are themselves vectors, we need to use the definition of `vadd` as a resolution instance of "+" in order to define `vadd` itself! Fortunately, a nice solution is possible by adding extra parameters to `vadd` and we will discuss this in the next section. For now, we only observe that the solution must be able to admit an infinity of overloadings for "+" through `vadd` – one each for vectors of integers, vectors of floats, vectors of vectors of integers, vectors of vectors of floats, and so on.

4.1.4 Ad hoc *vs* Parametric Polymorphism

The essential idea behind overloading is that it is a syntactic sharing of names. It does not interfere with the semantics of the language. So, for languages that do not emphasize the concepts of polymorphism and code sharing, we do not even think of overloading as a means of polymorphism. But a basic insight is that syntactic overloading can be traded off for semantically more polymorphic, higher-order, parametric polymorphism. Instead of attempting to replace syntactically overloaded programming constructs by their resolved instances in order to impart them a semantic meaning, we can choose to provide a parametric, semantic interpretation to the overloaded construct directly.

Consider the earlier example of `double`. Instead of treating it as implicitly overloaded, we may interpret it to possess an extra parameter supplying the appropriate addition operation. The intended interpretation is shown in figure 4.1, example (a), as a new transformed definition on the right. Now there is exactly one interpretation of `double`. It is now a parameterized function with an extra argument `(+)`. An actual resolved instance of "+" is passed to it as shown in any particular application.

Parameterization also takes care of the exponentially large number of possible instances by constructing only one function which is passed suitable arguments in various situations to behave like different instances. Translation of `doublepair` in example (b) illustrates this. Finally, it offers a clean solution to the recursive overloading problem as well (see example (c) for `vadd`). It converts the syntactic problem of choosing one of infinitely many different resolution instances into that of composing the right application from the same parameterized instance recursively.

It may appear from the above discussion that we have already found a plausible scheme for implementing overloading of identifiers. It is conceivable that a smart compiler will be able to generate the parameterized forms by looking at the type of the variables involved and the declaration of the various instances. Indeed, this is precisely the approach taken by Wadler and Blott in [63]. There the authors offer a general mechanism to systematically convert overloaded expressions into parameterized functions applying them to suitably resolved arguments.

It is possible to split this process into two parts. First, we may simply type-check the whole program and resolve all uses of overloaded identifiers to their appropriate instances as far as possible. Then, in the second step, we may translate the program using our preferred strategy that suitably deals with resolved and unresolved contexts. Thus, we can make the following observations at this point,

1. Overloading relieves the user from having to deal with some complex parameterizations in a particular implementation.

2. It is possible for the compiler to automatically infer the parameterizations using local type context information.

3. Overloading only specifies certain analysis and translation requirements for the compiler. A particular implementation may choose to satisfy them with more efficient compilation strategies than simple parameterization.

The last point needs some elaboration. Notice that the extra parameter `(+)` in `vadd'` is completely arbitrary. It has no knowledge that it is to be used for the instances of the overloaded addition operation only. Even though the compiler has this information, it simply discards it and any subsequent application of `vadd'` is no different from a general procedure call. An optimizing compiler can do better. Keeping in mind that `(+)` can only range over the useful instances of the addition operation (which is often a very small set in a given program), the

Source Code	Transformed Code
Example (a): `double`	

`def double x = x + x;`
 	
`...(double 2)...`
`...(double 2.718)...`
	`def double' (+) x = (+) x x;`
	
	`...(double' iplus 2)...`
	`...(double' fplus 2.718)...`

Example (b): `doublepair`	

`def doublepair x y = double x, double y;`
 	
`...(doublepair 2 2)...`
`...(doublepair 2.718 2)...`
	def doublepair' $(+)_1$ $(+)_2$ x y =
	  double' $(+)_1$ x, double' $(+)_2$ y;
	
	`...(doublepair' iplus iplus 2 2)...`
	`...(doublepair' fplus iplus 2.718 2)...`

Example (c): `vadd`	

`def vadd X Y = ...X[i]+Y[i]...;`
 	
`typeof X = (Vector I);`
`...X + Y...`
 	
`typeof XX = (Vector (Vector F));`
`...XX + YY...`
 	
`typeof XXX =`
`  (Vector (Vector (Vector I)));`
`...XXX + YYY...`
	`def vadd' (+) X Y =`
	`    ...((+) X[i] Y[i])...;`
	
	`typeof X = (Vector I);`
	`...(vadd' iplus X Y)...`
	
	`typeof X = (Vector (Vector F));`
	`...(vadd' (vadd' fplus) XX YY)...`
	
	`typeof XXX =`
	`  (Vector (Vector (Vector I)));`
	`...(vadd' (vadd' (vadd' iplus))`
	`          XXX YYY)...`

Figure 4.1: Translation of Overloading into Parameterization.

compiler may choose to generate a specialized version of `vadd` for each such used instance. The specialized versions will have inline instructions to do the relevent operation, consequently being far more efficient than parameter passing and general procedure calls. To take an example, in a matrix algebra library, it may suffice to generate a version of `vadd` for vectors and matrices of integers and floats each. The specialized versions for vectors may have inline instructions to do integer and floating point addition of the components and may be called `ivadd` and `fvadd` respectively. These versions will be much more efficient than their parameterized counterparts `(vadd' iplus)` and `(vadd' fplus)`.

It may be noted that all possible instances need not be specialized (of course, there may be infinitely many of them!). Obviously, it is non-trivial to decide which ones should be specialized. A compiler can perform price/performance analysis to decide this. There are opportunities for making several trade-offs here: compiling time *vs* execution time (efficiency of compiled code), added compiler complexity (for extra specialization analysis) *vs* simplistic translation mechanism, space (keeping several special versions) *vs* time (overhead in parameter passing and general procedure call).

4.1.5 Discussion

We may note in passing that different languages take different approaches when it comes to overloading. While conventional languages like Fortran and C allow only fixed, built-in interpretations of the overloaded symbols, Miranda does not allow overloading of arithmetic operators and treats equality to be fully polymorphic. Standard ML allows overloaded arithmetic operators in directly resolvable contexts such as `(2*2)` and `(2.72*2.72)` but cannot indirectly propagate it inside unresolved functions such as the function `double` above. As for the equality operator (`==`), ML treats it specially and overloads it with every type that admits equality. This gives it a limited, but extensible polymorphic flavor at the cost of some extra bookkeeping. Finally, Haskell directly supports the complete proposal of Wadler and Blott with type classes and parametric translation.

Our approach in Id is to investigate both parameterization and specialization schemes. The parameterization scheme is attractive since it is universal and systematic. The specialization scheme is attractive because of sheer run-time efficiency; the user need not pay any run-time price for the extra comfort of overloading. We describe our proposal below in two parts: first we establish a framework in which the compiler collects all the relevent information from the program and resolves the overloaded identifiers, then in the second step it uses this information to generate either parameterized or specialized code.

4.2 Overloading Resolution

First, we present the user's view of our overloading mechanism. We describe the type expression terms that can be constructed in Id and show the language constructs that the user may use to declare explicitly overloaded identifiers and their instances. Then we describe a mechanism of instance resolution using a set of operational reduction rules. This information is saved in the compiler for the later phase of actual code generation.

4.2.1 User's View of Overloading

We extend the notion of type of an identifier to include some information that it may be overloaded. We call this extra information a **type predicate**. In general, the complete type of an identifier has two parts, a standard type expression as derived by the Hindley/Milner type inference system (henceforth called the Hindley/Milner type or HMT for short), and a prefix type predicate, which is a boolean conjunction of zero or more predicate clauses. Each clause is a predicate function of some number of HMT agruments. A BNF grammar appears in figure 4.2.

A complete type expression is said to be **overloaded** if its type predicate prefix is non-null, *i.e.*, it contains at least one predicate clause, otherwise it is **non-overloaded**. A null type predicate (also called a trivial type predicate) will be represented by "ϵ" where explicit representation is desired. A non-trivial type predicate derived for a program expression specifies exactly what identifiers were found to be overloaded and under what type context. The predicate function in each clause identifies the overloaded identifier and the context information is manifest in its argument type expressions.

As discussed above in section 4.1, the identifiers may be **explicitly** or **implicitly** overloaded.

Explicit Overloading

It is often the case that the types of the intended instances of an overloaded identifier form a general pattern. For example, the Hindley/Milner type of `iplus` (`I -> I -> I`) and `fplus` (`F -> F -> F`) both fall into the general pattern of `(*0 -> *0 -> *0)`. Thus, it helps the type

$$
\begin{array}{lcll}
\textit{Type} & ::= & \textit{Type-predicate} \ . \ \textit{HMT} & \\[1em]
\textit{HMT} & ::= & \textit{Type-variable} & \\
 & | & \underbrace{\textit{HMT}, \cdots, \textit{HMT}}_{n} & (n > 1) \\
 & | & \textit{HMT} \texttt{ -> } \textit{HMT} & \\
 & | & \textit{Type-Constructor}^{n} \ \underbrace{\textit{HMT} \cdots \textit{HMT}}_{n} & (n \geq 0) \\
 & | & (\ \textit{HMT} \ ) & \\[1em]
\textit{Type-variable} & ::= & \texttt{*0} \mid \texttt{*1} \mid \texttt{*2} \mid \cdots & \\[1em]
\textit{Type-constructor}^{0} & ::= & \texttt{I} \mid \texttt{F} \mid \texttt{B} \mid \texttt{C} \mid \texttt{S} \mid \texttt{Sym} \mid \cdots & \\
\textit{Type-constructor}^{1} & ::= & \texttt{1d\_array} \mid \texttt{2d\_array} \mid \cdots \mid \texttt{list} \mid \cdots & \\[1em]
\textit{Type-predicate} & ::= & \underbrace{\textit{Predicate-clause} \wedge \cdots \wedge \textit{Predicate-clause}}_{n} & (n \geq 0) \\[1em]
\textit{Predicate-clause} & ::= & (\ \textit{Identifier}\texttt{?} \ \underbrace{\textit{HMT} \cdots \textit{HMT}}_{n} \ ) & (n > 0)
\end{array}
$$

Figure 4.2: Extended Syntax of Type Expressions in Id.

checker to restrict the type of all instances of an identifier to be a substitution instance of a general template[6]. In this way, arbitrary instance declarations can be caught as type errors. Of course, the template can be made as general as desired. Keeping this in view, we define the following Id syntax to declare overloaded identifiers and their instances:

`overload` *Overloaded-identifier* `=` *HMT-template* `;`

`instance` *Overloaded-identifier* `=` *Instance-identifier* `;`

For example, the declaration for "`+`" can be made as follows:

```
overload (+) = *0 -> *0 -> *0 ;
```

The type checker then infers the type of "`+`" as:

```
typeof (+) = (+? *0) . *0 -> *0 -> *0 ;
```

Now the type of the instances of `(+)` must be substitution instances of the template `(*0 -> *0 -> *0)`. The (non-trivial) predicate `(+? *0)` represents the fact that `(+)` was inferred to be overloaded and it requires one HMT-expression to specify the complete type

[6]Ideally, we would like to restrict the type of the overloaded identifier to precisely represent the set of declared instances, but there is no way to express this bounded quantification in our type system.

context for its resolution into an instance. This, of course, is the type of the objects that an instance operates upon[7]. So the following instance declarations are possible:

```
typeof iplus = ε . I -> I -> I;
instance (+) = iplus;                            % Substitution: {*0 ↦ I}

typeof fplus = ε . F -> F -> F;
instance (+) = fplus;                            % Substitution: {*0 ↦ F}

typeof cmplxadd = ε . (F,F) -> (F,F) -> (F,F);
instance (+) = cmplxadd;                         % Substitution: {*0 ↦ (F,F)}
```

Since these instances are not overloaded by themselves, their predicates are trivially null. Note that we only need to know the type of the instance at the time of its declaration. Its complete definition may appear somewhere else, and as long as its type is consistent with this declared signature, everything is fine. We also require that the instance types be mutually disjoint, which is indeed the case here. Both these characteristics are a direct consequence of the incremental nature of our type inference mechanism.

We will see in the next section how these substitutions are recorded and the later used for instance resolution.

Implicit Overloading

The implicit propagation of overloading information can also be easily represented using type predicates. Going back to our earlier examples,

```
def double x = x + x;
typeof double = (double? *0) . *0 -> *0;

def doublepair x y = double x, double y;
typeof doublepair = (doublepair? *0 *1) . *0 -> *1 -> (*0,*1);
```

Inferring the implicit overloading of `double`, we establish a type predicate `(double? *0)` by its name. Additional information is recorded internally to reflect that this overloading was caused by the presence of "`+`" inside its definition that went unresolved. Note that all the necessary type context information is collected in terms of the arguments to the type predicate function. Thus, `doublepair` requires two arguments to capture the entire context of both occurances of `double` which are independent of each other. Again, additional information is recorded to denote that `doublepair` is overloaded due to the two independent occurences of `double` inside its definition (remember, `double` is overloaded now, just like "`+`").

[7]Note that the scope of the type variable "`*0`" is the complete type signature. So, all the type information in a substitution instance of `(*0 -> *0 -> *0)` is caught by simply recording the substitution of "`*0`" inside the predicate `(+? *0)`.

Recursive overloading proceeds in a similar fashion,

```
def vadd X Y = ...X[i]+Y[i]...;
typeof vadd = (vadd? *1) . (Vector *1) -> (Vector *1) -> (Vector *1);
instance (+) = vadd;                      % Substitution: {*0 ↦ (Vector *1)}
```

The substitution shows the mapping from the HMT template of `(+)` to the Hindley/Milner type of `vadd`. In addition, the predicate `(vadd? *1)` denotes that `vadd` itself is overloaded and that this fact should be taken into account while resolving instances of the addition operation.

We have only shown how the overloading information is visible to the user. In the next section we describe the internal representations of the compiler and how it manipulates them to make the desired inferences.

4.2.2 Compiler's View of Overloading

We have introduced two new constructs, `overload` and `instance`. Along with that, we have to process implicit and recursive overloading too. We now give a two step, high level description of the extended type checking and overloading resolution algorithm.

The first step is to execute the type inference algorithm of chapter 3 to determine the Hindley/Milner types of all expressions in the given program. The only addition is that the type of an explicitly overloaded identifier is taken to be its HMT template.

The next step is to recursively traverse the given program bottom up assigning type predicates to all identifiers. This is the major overloading resolution step and is depicted algorithmically in figure 4.3. We process one definitional binding at a time, where a definitional binding is either a function definition or a top-level constant definition or one of `overload` or `instance` declarations. The syntax followed is a subset of the complete Id language syntax and is similar to that of Id Kernel language (figure 2.7) without performing some of its simplifying transformations[8].

There are certain characteristic properties of our resolution algorithm; we list them below.

1. Literals, *e.g.*, `1`, `2.718`, `"abracadabra"`, `'atom` etc. can not be overloaded[9].

[8]There are some differences from (figure 2.7); we have not lifted out the internal function definitions, neither do we need to flatten all subexpressions, and we have expanded the top-level *Main* block into a sequence of top-level constant definition bindings. Finally, new definitional bindings corresponding to `overload` and `instance` are also handled.

[9]These literals constitute the *base types* in our typed semantic domain. Without them, the type structure would be quite uninteresting and overloading them will make the job of the type inference system almost impossible. So we give them fixed interpretation and never overload them.

```
Def Overloading-Resolution(binding)
 case binding of

  'overload Identifier = HMT_o' ⇒
    Mark Identifier as explicitly overloaded.
    Generate its predicate function.

  'instance Overloaded-identifier = Instance-identifier' ⇒
    If Overloaded-identifier is not marked overloaded then error.
    else
        Let typeof Overloaded-identifier = Predicate_o . HMT_o.
        Let typeof Instance-identifier = Predicate_i . HMT_i.
        S = Unify(HMT_o,HMT_i).                                   (may fail)
        Instance-clause = S(Predicate_o).
        if Instance-clause overlaps with a previously generated instance clause then error.
        else
          Generate Explicit Overloading Rewrite Rule for Instance-identifier.
        endif.
    endif.

  'def Function-identifier parameters = body' ⇒
    Recursively process all definitional bindings inside the body in topological order.
    Collect all overloading predicates from the body instantiated with their type context.
    Resolve the predicates using the current set of Rewrite Rules.
    if there are no unresolved predicates left then
      Mark Function-identifier as non-overloaded.
    else if there is an unresolved predicate (Identifier? t_1 ··· t_n)
             such that some t_i is not a type variable then error.
        else
           Mark Function-identifier as implicitly overloaded.
           Generate Implicit Overloading Rewrite Rule for Function-identifier.
        endif.
    endif.

  'Constant-LHS = body' ⇒
    Recursively process all definitional bindings inside the body in topological order.
    Collect all overloading predicates from the body instantiated with their type context.
    Resolve the predicates using the current set of Rewrite Rules.
    if there is any unresolved predicate then error.

  endcase.
```

Figure 4.3: Overloading Resolution and Propagation Algorithm.

2. The meaning of an overloaded identifier is determined by its declared instances and the context of its use that selects one of those instances. The compiler machinery to do these resolutions consists of a set of rewrite rules, one for each implicitly overloaded identifier or a declared instance of an explicitly overloaded identifier. These rules are then applied to the type context where the overloaded identifier is used in order to generate its appropriate instance. The details of this rewrite mechanism will be presented shortly.

3. An identifier can be declared to be explicitly overloaded at the top-level or inside the scope of any block expression. The `overload` declaration *defines* that identifier for all syntactic purposes. The compiler notes this fact by generating its predicate function and its arity. The scope of overloading is exactly the same as the scope of visibility of that identifier[10].

4. An `instance` declaration is processed within the scope of the `overload` declaration of the corresponding overloaded identifier and the definitional binding of the instance identifier. The compiler action is first, to generate a type context for the overloaded identifier using the type signature of the instance identifier, and then to validate that instance (check for overlap with previous instances) and generate the appropriate rules for its resolution. At the moment, we flag an error if we find a previous overlapping instance. We will refine this behaviour later on.

5. Only function identifiers are allowed to be implicitly overloaded. To be precise, a function identifier is implicitly overloaded if there is some use of an overloaded identifier within its body that remains unresolved. So, we collect all unresolved uses of overloaded identifiers in a block expression and attribute it to to its nearest enclosing function definition identifier. When there is no such enclosing function definition, as in the case of top-level constants, we raise an error condition. The reason for not allowing top-level constants to be implicitly overloaded is essentially pragmatic. As stated earlier, overloading can be viewed as a possible parameterization of semantic meaning of the overloaded identifier with contextual information. Thus, an overloaded constant behaves like a function, *i.e.*, it has to be re-evaluated in every context in which the overloading gets resolved. This defeats the very purpose of having a constant, which is supposed to be evaluated only once[11]. We will

[10] The scope-analysis phase assigns unique names to all identifiers and effectively *flattens* the block structure, so overloading declarations inside a block are exactly equivalent to those at the top-level.

[11] Explicit overloading does not present this problem since each instance is actually a constant, so resolution

review this point at a later stage.

6. We allow a function identifier to be implicitly overloaded only if no contextual information is available to resolve its overloading predicates. We raise an error condition if some overloading predicate has partial contextual information but can not be resolved because there is no declared instance to handle it. This does not affect our resolution mechanism and is done essentially for pragmatic reasons.

4.2.3 Rewrite Rules for Overloading Resolution

Now we present the compiler machinery to actually keep track of the `overload` and `instance` declarations and to do instance resolution. We describe this in terms of a rewrite rule operational semantics. This, combined with the high level overloading processing algorithm described above, forms the complete overloading resolution mechanism of the compiler. Readers unfamiliar with the notation of Term Rewriting Systems should refer to [33, 34] for an excellent treatment.

Our term rewriting system consists of the type predicate terms as specified syntactically in figure 4.2 and rewrite rules over them. More formally, our rewrite system is a pair (Σ, R) where,

- Σ, also called the signature, is the set of function symbols in the system. It is exactly the union of the set of overloading predicate functions in the predicate clauses (`+?`, `double?`, ...) and the set of type constructors in our type system (`I`, `B`, `->`, `list`, `2_tuple`, ...). It also records the arity of each symbol.

- Σ also implicitly contains infinitely many variables which is precisely the set of all type variables in our case.

- R, the set of rewrite rules, consist of a left hand side (LHS) which is always a single predicate clause, and a right hand side (RHS) which is a collection of zero or more predicate clauses.

The predicate functions and their rewrite rules are accumulated by the compiler as it processes the explicitly and implicitly overloaded identifiers. Since this process is necessarily incremental, we will always view the rewrite system current at the time of a particular compilation to be the complete set of rewrite rules accumulated till that time. Any forward references can

is only a matter of choosing among constants.

$$
\text{OVERLOAD:} \quad \frac{\texttt{overload}\ x = \tau}{\vdash x : (x?\ \alpha_1 \cdots \alpha_n).\tau \qquad\qquad (\alpha_i \in FV(\tau))}
$$

$$
\text{INSTANCE:} \quad \frac{\begin{array}{c}\texttt{instance}\ x_o = x_i \\ TE \vdash x_o : \rho_o.\tau_o \\ TE \vdash x_i : \rho_i.\tau_i \\ S = U(\tau_o, \tau_i)\end{array}}{\text{Rule: } S\rho_o \xrightarrow{x_i'} S\rho_i}
$$

$$
\text{IMPL-FUNCTION:} \quad \frac{\begin{array}{c}\texttt{def}\ f\ x_1 \cdots x_n = \mathit{block} \\ \rho_b \text{ is the set of all overloading predicate clauses in } \mathit{block} \\ \rho_b \twoheadrightarrow \rho_u \qquad\qquad (\rho_u \text{ in normal form})\end{array}}{\text{Rule: } (f?\ \alpha_1 \cdots \alpha_n) \longrightarrow \rho_u \qquad\qquad (\alpha_i \in FV(\rho_u))}
$$

Figure 4.4: Meta-Inference rules to generate Rewrite Rules for Overloading Resolution.

be detected and flagged by the independent incremental compilation mechanism described in chapter 3.

The figure 4.4 shows the meta-inference rules used by the compiler to generate the rewrite rules of this system. We discuss them below. The meta-variable ρ varies over the set of type predicates. A type substitution when applied to a predicate performs the substitution over each type expression in each clause of the predicate.

An `overload` declaration specifies a HMT template that all its instances will share. We want to express restrictions on the template in order to help the type inference process as much as possible, because this template is the only visible type signature for that overloaded identifier. Note that a trivial template of the form `(*0)` has much less information than the template `(*0 -> *0 -> *0)`, which denotes a curried function of two arguments of the same type giving the result of the same type. The compiler also generates a new predicate function (the overloaded identifier suffixed with a "?") to be added to the current rewrite system. It assigns the arity of the predicate function to be the number of distinct type variables appearing in the HMT template. The distinct type variables represent the various independent ways in which that identifier can be overloaded. We capture these degrees of freedom by supplying an equal number of parameters to the predicate function, one for each type variable in the HMT template. For example, the following declarations,

```
overload (+) = *0 -> *0 -> *0;
overload (*) = *0 -> *1 -> *2;
```

generate the predicate clauses `(+? *0)` and `(*? *0 *1 *2)`. The latter form for "*" gives the flexibility of expressing multiplication of unlike types, *e.g.*, vectors with matrices to give vectors.

An `instance` declaration produces a rewrite rule for the overloaded identifier. The type context of the resolution is obtained by unifying a copy of the HMT template of the overloaded identifier with that of the instance identifier. The generated rule rewrites the instantiated predicate of the overloaded identifier into that of the instance identifier[12]. As an example, the following declarations,

```
typeof iplus = ϵ . I -> I -> I;
instance (+) = iplus;

typeof vadd = (vadd? *1) . (Vector *1) -> (Vector *1) -> (Vector *1);
instance (+) = vadd;
```

give rise to the rewrite rules,

$$\texttt{(+? I)} \xrightarrow{\texttt{iplus}} \epsilon$$

$$\texttt{(+? (Vector *1))} \xrightarrow{\texttt{vadd}} \texttt{(vadd? *1)}$$

respectively.

Implicit overloading of function identifiers is handled by the last rule in figure 4.4. During the initial type checking phase, the type of an overloaded identifier (its HMT template) gets suitably instantiated to reflect the type context in which it is being used. In the second pass, its type predicate is unified with this type context and resolved. This involves repeated rewriting of the instantiated type predicate using the existing set of rewrite rules until it can not be rewritten any more (it reaches a normal form). We will show in the next section that this rewriting always terminates (under certain restrictions) and produces the same answer irrespective of the order in which the rewrite rules are applied. In literature, these properties are called Strong Normalization and Confluence respectively. These are strong properties of the rewrite system and ensure correct compiler behaviour even in the face of random accumulation of rewrite rules as in our incremental compiler.

The normal form either consists of the trivial null predicate "ϵ", in which case the identifier in question has been completely resolved, or there are unresolved predicates left over. Predicates

[12]We also record an appropriate translation identifier over the rewrite rule to help us in translation. We will describe this in more detail in the next section.

may not get fully resolved either because the appropriate instances that cover the type context at hand are not available (as in attempting to resolve "`2+x`" without the `instance` declaration of `iplus`), or it could simply be that sufficient information is not present in the type context (as in attempting to resolve "`x+x`" in the declaration of `double`). We have chosen to interpret the former situation as an error in the resolution algorithm of figure 4.3 for essentially pragmatic reasons.

Finally, all unresolved predicates are collected up to the nearest enclosing function definition and we generate a rewrite rule to denote this implicit overloading as shown in figure 4.4. The predicate function goes by the name of the function identifier and its arity is the number of distinct type variables present in the unresolved predicates, just as in the case of explicit overloading declarations.

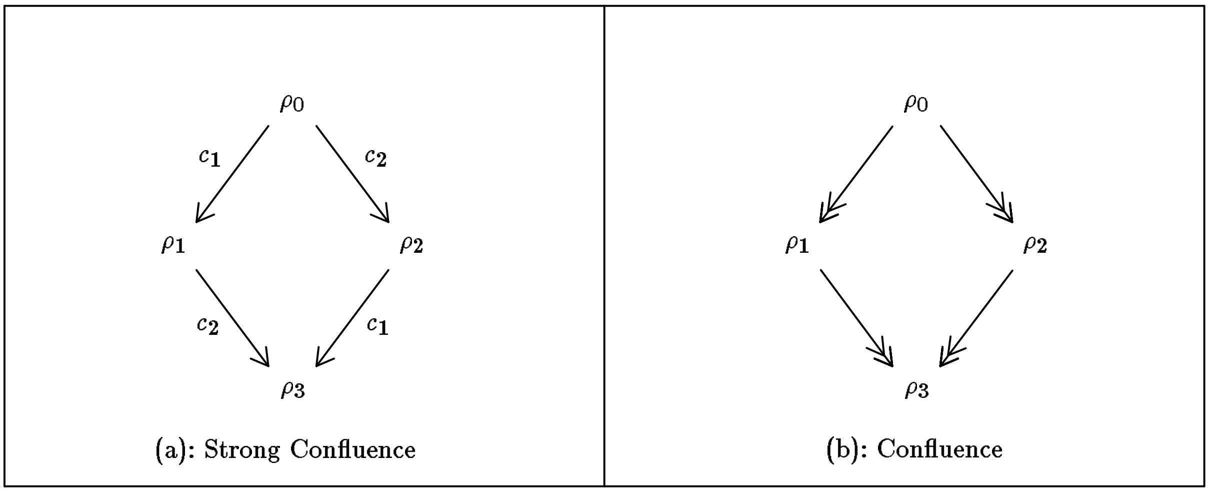
4.2.4 Properties of the Rewrite System

Confluence

A sub-expression which is a potential candidate for rewriting in a rewrite system is called a *redex*. An important observation about our rewrite system is that only predicate clauses are allowed to be a redex; arbitrary type expressions can only be manipulated as subterms of a redex clause. This stems directly from the fact that our rewrite rules deal only with predicate clauses as a whole, there are no rules to allow independent manipulation of a type expression embedded in a predicate clause.

The second important observation is that the predicate clauses are completely flat structures. We do not allow nesting of one clause inside the other, we can only have a group of clauses at the top level in a predicate. This is a direct consequence of the syntax of the predicate clauses. The arguments to a predicate function can only be pure Hindley/Milner type expressions, never a predicate clause. Note, that the type expressions themselves can be nested within one another, but due to our first observation above they are of no particular significance for rewriting.

The third important observation is that we carefully avoid overlapping instances every time we generate a rewrite rule. This is true of `instance` declarations where we explicitly check for an overlap, and implicit overloading never generates more than one rule anyway. Thus, atmost one rewrite rule can apply at a given predicate clause. This allows us to conclude that redexes in a predicate are completely independent and do not interefere with each other. Moreover,



(a): Strong Confluence

(b): Confluence

Figure 4.5: Confluence Properties of our Rewrite System.

from our second observation, one redex can not even manipulate another one (duplicate it or remove it) since it does not have a handle on any predicate clause except its own and possibly some brand new ones from its right hand side.

The following property of *strong confluence* falls out of the above discussion, which we state as a theorem.

Theorem 4.1 (Strong Confluence) *The system of rewrite rules as generated by the compiler for a given program is strongly confluent,* i.e., *if $\rho_0 \longrightarrow \rho_1$ and $\rho_0 \longrightarrow \rho_2$ are two single step reductions of the same predicate ρ_0 in this system, then there exists a predicate ρ_3 such that $\rho_1 \longrightarrow \rho_3$ and $\rho_2 \longrightarrow \rho_3$ in a single step each.*

Proof: Let c_1 and c_2 be the two redex clauses of ρ_0 that are reduced in the reductions $\rho_0 \longrightarrow \rho_1$ and $\rho_0 \longrightarrow \rho_2$ respectively (refer figure 4.5 (a)). By the above observations, these are completely disjoint and at the same level in ρ_0 (by the second observation, there is only one level). Thus, c_2 is present and can still be reduced in ρ_1; similarly c_1 is present and can still be reduced in ρ_2, both giving rise to the same predicate which we call ρ_3. $\square$

A more useful property is that of Confluence. It is the property of a rewrite system, where if $\rho_0 \twoheadrightarrow \rho_1$ and $\rho_0 \twoheadrightarrow \rho_2$ in zero or more steps then there exists another term ρ_3 such that $\rho_1 \twoheadrightarrow \rho_3$ and $\rho_2 \twoheadrightarrow \rho_3$ in zero or more steps (refer figure 4.5 (b)). We leave it as a simple exercise for the reader to verify that strong confluence implies confluence.

Strong Normalization

It would be nice to show that the rewrite system generated above always terminates. That way, the compiler is not in danger of running away with a non-terminating reduction sequence. Unfortunately, as the system stands, there is nothing to prevent the user from generating pathological instances that may give rise to cycles in rewriting. Consider, for example, the following program segment,

```
overload (+) = *0 -> *0 -> *0;
def vadd X Y = { typeof X = (Vector *0);
                 in X + Y};
instance (+) = vadd;
```

The type of `vadd` is inferred to be,

```
typeof vadd = (vadd? *0) . (Vector *0) -> (Vector *0) -> (Vector *0);
```

accordingly, the rewrite rules of the implicit overloading of `vadd` and the `instance` declaration become,

$$\begin{aligned} \texttt{(vadd? *0)} &\longrightarrow \texttt{(+? (Vector *0))} \\ \texttt{(+? (Vector *1))} &\longrightarrow \texttt{(vadd? *1)} \end{aligned}$$

which clearly indicate a non-terminating cycle of reductions.

Note that each individual declaration from among the above segment does not pose a problem. It is the combination that creates a cycle. The above declarations indicate a particularly problematic aspect of the rewrite system: since the compiler is incremental, the above declarations could appear separately anywhere in the program (even in separate files!), so it is a non-trivial task to detect such cycles. Even if we require that all instance declarations and definitions should appear together, it takes a fair amount of compiler analysis to figure out that a cycle of reductions is possible with the given set of declarations.

There are some trivial cases where it is straightforward to detect a cycle. For example, the declarations,

```
overload (+) = *0 -> *0 -> *0;
instance (+) = (+);
```

will generate the trivial cyclic rewrite rule,

$$\texttt{(+? *0)} \longrightarrow \texttt{(+? *0)}$$

Instead of trying to make the compiler ultra-smart, our current approach is to push the responsibility to the user. Since the user already has the responsibility of declaring explicit instances for the explicitly overloaded identifiers, we expect him/her to be extra cautious and knowledgeable about its consequences. In almost every case, a simple strategy can avoid the cycles; the declared instances, if themselves overloaded, should at least simplify the type context in which they apply. The notion of a simpler type context can be viewed in terms of substitution instances of chapter 2. Informally, if τ_1 and τ_2 are two type expressions, we say that τ_1 is simpler than τ_2 if we can embed τ_1 inside τ_2. In other words, we should be able to find a non-trivial type expression τ_3 and a substitution S over the domain $FV(\tau_3)$ and containing τ_1 in its range, such that $\tau_2 = S(\tau_3)$. For example, `(+? (Vector *0))` has a more complex type context than `(+? *0)` and so the usual pair of rewrite rules for `vadd` that map the former to the latter *via* `(vadd? *0)` will not lead to a cycle.

The above informal description of restricting conditions that guarantee termination is noticeably sketchy. Further research in this direction may shed more light on formal and algorithmic ways to detect and guarantee strong normalization properties for the benefit of the compiler.

4.3 Overloading Translation

In the last section, we saw a way of determining whether an identifier was overloaded or not and to what extent it could be resolved using the given type context. In this section we use that analysis to generate appropriate code for the resolved and unresolved overloaded identifiers. An overloaded identifier is a syntactically shared name, so this process can also be viewed as imparting semantic meaning to the syntactic occurrences of the overloaded identifier within the program.

The `overload` and `instance` declarations are merely directives to the compiler and do not give rise to any executable code. But we need to generate translated code for each applicative occurrence of an overloaded identifier in an expression and the implicitly overloaded function definitions.

4.3.1 Parameterization

In section 4.1.4 (figure 4.1), we gave examples of transforming occurrences of overloaded identifiers into suitably parameterized procedure calls. Now we will describe the mechanism to systematically generate that translation using our rewrite rules.

Parameterized Translation Mechanism

We note from the last section that none of the rewrite rules interfere with each other. Thus starting with a reducible type predicate, not only do we reach the same normalized form after an arbitrary sequence of reduction steps (the confluence property), but we always perform the same set of reductions, albeit in an undetermined order. Various reduction sequences can differ only in the order of reducing the available independent redexes at any point. We can record the actual reduction performed at each step and represent them in a resolution tree for the original type predicate. This resolution tree serves as a guideline for the parameterized translation.

We will demonstrate the translation mechanism *via* our earlier example of `doublepair`, which appears in figure 4.6 (a) along with the usual overloading declarations. The relevent rewrite rules generated are shown in (b) and the resolution tree generated for the application `(doublepair a b)` is shown in (c).

Intuitively, each reduction in the resolution tree that corresponds to an `instance` declaration contributes its associated instance identifier to the translation. This is the identifier recorded over the INSTANCE rewrite rule in figure 4.4. Thus, `iplus` and `fplus` are recorded over their respective rewrite rules in figure 4.6 (b). The instance declaration for `vadd` causes its corresponding translation instance `vadd'` to be recorded over the rewrite rule, since `vadd` is implicitly overloaded. All these recorded instance identifiers appear as annotations for their corresponding reduction steps in the resolution tree (see figure 4.6 (c)).

The overall translation is obtained by collecting all the annotations in an in-order traversal of the resolution tree. In this way, a subtree corresponding to an explicitly overloaded identifier translates into a single expression (a partial application) that is its implementation. A subtree corresponding to an implicitly overloaded identifier translates into a sequence of parametric arguments that need to be applied to its translation instance. This process of generating parametric arguments and partial applications continues until we reach the leaves of the resolution tree. A null ("ϵ") leaf indicates that the translation in that branch is complete. A non-null leaf represents an unresolved predicate and has to be translated into a new parameter for the overall expression.

```
overload (+) = *0 -> *0 -> *0;

instance (+) = iplus;

instance (+) = fplus;

def vadd X Y = ...X[i]+Y[i]...;

instance (+) = vadd;

def double x = x + x;

def doublepair x y = double x, double y;

typeof a = I;
typeof b = (Vector (Vector F));

...(doublepair a b)...
```

(a): Overloaded Source Program.

$\texttt{(+? I)} \xrightarrow{\texttt{iplus}} \epsilon$

$\texttt{(+? F)} \xrightarrow{\texttt{fplus}} \epsilon$

$\texttt{(vadd? *0)} \longrightarrow \texttt{(+? *0)}$

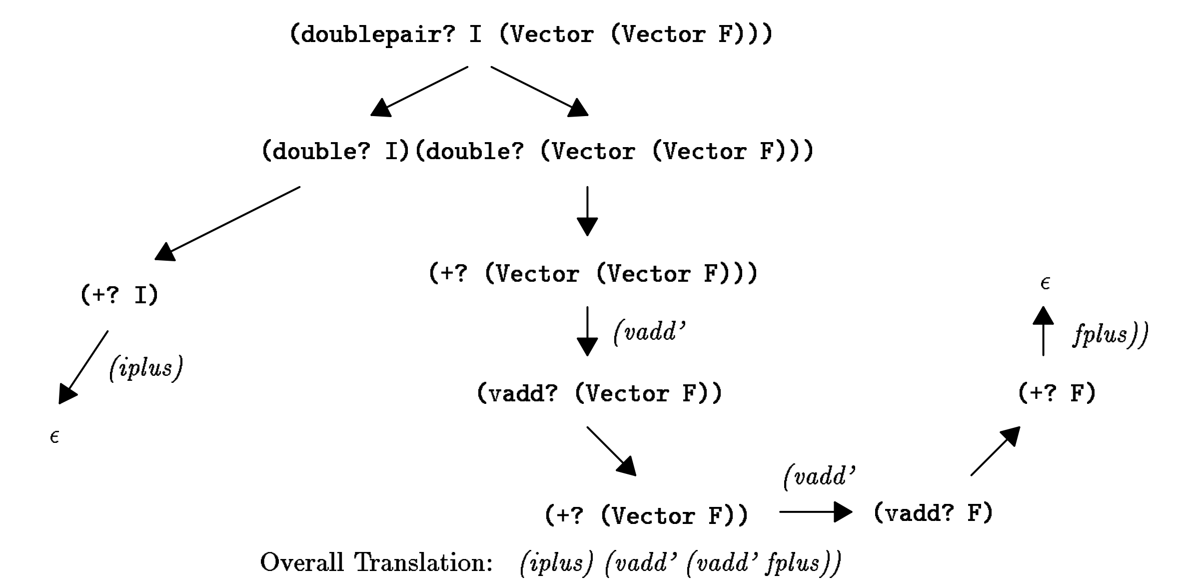
$\texttt{(+? (Vector *1))} \xrightarrow{\texttt{vadd}'} \texttt{(vadd? *1)}$

$\texttt{(double? *2)} \longrightarrow \texttt{(+? *2)}$

$\texttt{(doublepair? *3 *4)} \longrightarrow$
$\texttt{(double? *3)(double? *4)}$

...(doublepair? I (Vector (Vector F)))...

(b): Rewrite Rules.



(c): Predicate Resolution Tree.

Figure 4.6: An Example of Overloading Resolution and Parameterized Translation.

```
Def Translate-binding(TE,binding)
 case binding of

  'overload Identifier = HMT_o' ⇒ Do nothing.

  'instance Overloaded-identifier = Instance-identifier' ⇒
   if Instance-identifier is not overloaded then
     Record Instance-identifier over the instance rewrite rule.
   else if Instance-identifier is implicitly overloaded then
         Let Instance-identifier' be the translation instance of Instance-identifier.
         Record Instance-identifier' over the instance rewrite rule.
       endif.
   endif.

  'def Function-identifier parameters = body' ⇒
   if Function-identifier is implicitly overloaded then
     Let ρ = c_1 ∧ ··· ∧ c_n be the unresolved type predicate of Function-identifier.
     Let Function-identifier', v_1, ..., v_n be new identifiers.
     Record Function-identifier' to be the translation instance of Function-identifier.
     new-body = Translate-expression(TE + {c_i ↦ v_i},body);
     Return 'def Function-identifier' v_1 ··· v_n parameters = new-body'.
   else
     new-body = Translate-expression(TE,body);
     Return 'def Function-identifier parameters = new-body'.
   endif.

  'Constant-LHS = body' ⇒
   new-body = Translate-expression(TE,body);
   Return 'Constant-LHS = new-body'.

 endcase.
```

Figure 4.7: Parameterized Translation of Bindings.

Def *Translate-expression(TE,expression)*
case *expression* **of**

'Identifier' $\Rightarrow$ **if** Identifier *is implicitly overloaded* **then**
Let ρ_i = (Identifier? $\tau_1 \ \cdots \ \tau_n$) *be the type predicate of* Identifier.
Let Identifier$'$ *be the translation instance of* Identifier.
$TE \vdash \rho_i \Rightarrow e_1 \cdots e_k$.
Return 'Identifier$'$ $e_1 \ \cdots \ e_k$'.
else if Identifier *is explicitly overloaded* **then**
Let ρ_i = (Identifier? $\tau_1 \ \cdots \ \tau_n$) *be the type predicate of* Identifier.
$TE \vdash \rho_i \Rightarrow e_i$.
Return 'e_i'.
else *Return* 'Identifier'.
endif.
endif.

'$expression_1 \ \cdots \ expression_i \ \cdots \ expression_n$' $\Rightarrow$
e_i = *Translate-expression(TE,expression$_i$)*. $(1 \leq i \leq n)$
Return '$e_1 \ \cdots \ e_i \ \cdots \ e_n$'.

'{ $binding_1$; $\cdots$ $binding_i$; $\cdots$ $binding_n$ **in** *expression* }' $\Rightarrow$
b_i = *Translate-binding(TE,binding$_i$)*. $(1 \leq i \leq n)$
e_b = *Translate-expression(TE,expression)*.
Return '{ b_1; $\cdots$ b_n **in** e_b }'.

endcase.

Figure 4.8: Parameterized Translation of Expressions.

Parameterization Algorithm

The semi-formal algorithm for parameterized translation appears in figures 4.7 and 4.8. This is a pair of mutually recursive procedures, *Translate-binding* and *Translate-expression* that traverse the program in a top-down fashion. As before, the syntax followed is a subset of the complete Id language syntax with the usual extensions for overloading and partial kernelization[13]. Both these procedures accept a **translation environment** (TE)[14] as the first argument, which is a mapping from unresolved predicates to new parameterized arguments. The procedure *Translate-binding* processes definitional bindings, extending the translation environment with new parameter argument mappings for each implicitly overloaded functional binding that it encounters. The procedure *Translate-expression* processes an expression, replacing every overloaded identifier with a properly parenthesized partial application of instance identifiers generated from its resolution tree, the unresolved predicates being translated using the current translation environment.

The key points to observe are the following:

1. There is a distinct difference between the processing mechanisms of implicit and explicit overloading. This distinction arises due to a basic difference between the way we assign semantic meaning to implicitly and explicitly overloaded identifiers in a program. An explicitly overloaded identifier is a purely syntactic entity deriving its complete semantic interpretation from its instances, while an implicitly overloaded identifier already has a partially specified function template into which extra parametric arguments need to be inserted in order to complete its semantic meaning. Thus, during translation, an explicitly overloaded identifier gets completely replaced by a single partial application generated through the resolution tree, while an implicitly overloaded identifier translates into its parameterized version with only its parametric arguments being generated through the resolution tree.

2. As shown in figure 4.4, we need to record an appropriate translation (an instance identifier) over each rewrite rule generated *via* an explicit `instance` declaration. If the given instance identifier is non-overloaded, then it is recorded directly. Otherwise, if it is implicitly

[13]For convenience, we treat the predefined *Constant* classes of various arities and the class *ProcId* of (figure 2.7) to be included in the class *Identifier*.

[14]Not to be confused with *type environment* of our type inferencing algorithm, which is also abbreviated as TE.

UNRESOLVED:
$$\frac{(c_o \mapsto v) \in TE}{TE \vdash c_o \Rightarrow v}$$

EXPLICIT:
$$\frac{\begin{array}{c} c_o \text{ is a clause of an explicitly overloaded identifier} \\ c_o \xrightarrow{x'} c_1 \wedge \cdots \wedge c_i \wedge \cdots \wedge c_k \\ TE \vdash c_i \Rightarrow e_i \qquad\qquad (1 \leq i \leq k) \end{array}}{TE \vdash c_o \Rightarrow (x\ e_1 \cdots e_k)}$$

IMPLICIT:
$$\frac{\begin{array}{c} c_o \text{ is a clause of an implicitly overloaded identifier} \\ c_o \longrightarrow c_1 \wedge \cdots \wedge c_i \wedge \cdots \wedge c_k \\ TE \vdash c_i \Rightarrow e_i \qquad\qquad (1 \leq i \leq k) \end{array}}{TE \vdash c_o \Rightarrow e_1 \cdots e_k}$$

Figure 4.9: Inference rules to generate Parameterized Overloading Translation.

overloaded, then the identifier corresponding to its parameterized translation is recorded. In the pathological situation when the instance identifier is itself explicitly overloaded, we record nothing and simply let its own instance declarations record the translations.

3. The only places that appear to handle parameterization are those that handle implicitly overloaded function identifiers. This happens in *Translate-binding* while extending the translation environment with new argument variables corresponding to the unresolved predicates, and in *Translate-expression* while adding extra parameters to an application of an implicitly overloaded identifier. This is as expected. These are the only places that introduce extra overhead in the translated code and deserve our attention for optimization *via* specialization. We will come back to them in the next section.

4. The actual generation of the parameters and the partial applications from the type predicates is shown as an inference relation using the current translation environment. The inference rules for that are shown in figure 4.9. These follows the intuition of resolution tree based translation described above. The inference relation has the familiar form:

$$TE \vdash \rho \Rightarrow e$$

which is read as 'under the translation environment TE, the type predicate ρ generates the expression e'.

4.3.2 Specialization

We argued in section 4.1.4 that specialization is essential in order to achieve efficient translation of overloaded expressions. Now we show how to incorporate the generation of efficient specialized versions of implicitly overloaded function definitions into our translation scheme. Note that this is only an alternative translation step to the parameterization scheme described above. All our previous discussion of overloading resolution applies here equally well.

The basic idea behind specialization is that instead of translating an expression containing an implicitly overloaded function identifier into its parameterized instance supplied with suitable arguments, we choose to generate a completely new copy of the function using its definition template from the original program. The copy has the appropriate contextual information "built-in" so that no parameters are necessary. For example, we may specialize the definition of `vadd` for 1-dimensional integer and floating point arrays as shown below. The translated parameterized version is also shown for comparison.

```
def vadd′ (+) X Y = ...((+) X[i] Y[i])...;
def ivadd X Y = ...X[i] +ᵢ Y[i]...;
def fvadd X Y = ...X[i] +_f Y[i]...;
```

where $+_i$ and $+_f$ represent machine instructions for integer and floating point addition respectively. Using `(ivadd A B)` instead of `(vadd′ iplus A B)` will be much more efficient because first, there is no parameter passing overhead and second, the inline instruction $+_i$ adds its arguments directly while `(+)` is compiled as a general procedure call within the body of `vadd′`.

Specialization Translation Mechanism

To generate these specializations automatically, the compiler first needs to decide that a specialization is desired in a particular situation. This could either be indicated by the user through special annotations in the code or the compiler may decide that by itself. Such a decision will necessarily involve a non-trivial analysis of economic constraints, such as the cost of storing the definition template, cost of instantiating the template for a given situation, degree of efficiency achieved in the generated code, and the size of the compiled code etc. We will not go into these aspects and instead confine ourselves to the actual process of generating the specializations. The only important point to note is that if the compiler decides to specialize an implicitly overloaded function definition, it needs to store the complete definition in a template for future use.

The first step in overloading translation with specialization is to figure out what specialization is needed by looking at the type context. It makes sense to specialize only the fully resolved type contexts (where there are no unresolved type predicates) because the unresolved predicates will still require parameterization for their complete translation. The type context is available on the type predicate of the given overloaded function identifier. We resolve the predicate as before by repeated rewrite reductions using the existing set of rewrite rules and see if it resolves completely. If it does, then we have a possible candidate for specialization. If the economic analysis is favorable, we can generate a specialized version using the definitional template of that identifier, filling it with appropriate instances generated *via* the rewrite rules. It is clear that the same instances recorded over the rewrite rules that we used as parameter arguments will now be used to fill the template inline.

The second step is to record this specialization for future use. We do this by generating a specialization rewrite rule for the identifier in question with the given context (see figure 4.10 meta-rule SPL-INSTANCE1). Since we only specialize when the context is fully resolved, the generated rule always has a null RHS. In terms of the resolution tree, this rule represents pruning a reduction branch: we are allowed to short-circuit the resolution to null whenever we encounter this identifier in the same context. If this identifier also appears in the RHS of some other instance rewrite rule (it has been declared as an instance of some other overloaded identifier), then we need to generate another similar instance rewrite rule for its resolution in this special context (meta-rule SPL-INSTANCE2). These two rewrite rule generating steps can be added to our list of meta-inference rules of figure 4.4.

Note that these specialization rewrite rules will necessarily overlap with previously existing rewrite rules. This will affect the nice confluence property we maintained till now. Indeed, now it is possible to either take a one step specialization reduction generating the specialized translation or continue with the usual sequence of reductions aimed at parameterized translation. This is precisely the freedom of choice we wanted in the compiler. But it is worth noting that although we have lost the property of Strong Confluence (it is no longer possible to bring together two divergent one step reductions to the same term in a single step), we still retain Confluence, because this choice is only a short-cut to the final normalized predicate, "ϵ". We will leave it as an exercise for the reader to formally verify this. Of course, the translation in the two cases will be very different and is certainly not unique depending on the reduction path chosen. The compiler decides the appropriate reduction path and generates the translation ac-

$$\text{SPL-INSTANCE1:} \quad \frac{\begin{array}{c}\text{Predicate } \rho_x = (x?\ \tau_1 \cdots \tau_n) \text{ occurs in a type context} \\ \text{for an implicitly overloaded identifier } x \\ \rho_x \not\longrightarrow \epsilon\end{array}}{\text{Rule: } \rho_x \xrightarrow{x_{\tau_1 \cdots \tau_n}} \epsilon}$$

$$\text{SPL-INSTANCE2:} \quad \frac{\begin{array}{c}x \text{ is an implicitly overloaded identifier} \\ x \text{ already has a specialization for some type context} \\ \text{Rule: } (x?\ \tau_1 \cdots \tau_n) \xrightarrow{x_{\tau_1 \cdots \tau_n}} \epsilon \\ x \text{ appears in RHS of another instance rewrite rule} \\ \text{Rule: } \rho \xrightarrow{x'} (x?\ \alpha_1 \cdots \alpha_n) \\ S = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}\end{array}}{\text{Rule: } S\rho \xrightarrow{x_{\tau_1 \cdots \tau_n}} \epsilon}$$

Figure 4.10: Meta-Inference rules to generate Specialization Rewrite Rules.

cordingly. The added rules for specialization behave exactly like the rules introduced by explicit `instance` declarations, so the inference set in figure 4.9 is sufficient to derive a translation in this new setting.

Specialization Algorithm

In order to formalize the specialization process, we update our earlier parameterized translation algorithm. Procedure *Translate-binding* of figure 4.7 is virtually unchanged except that we also record a definitional template for the implicitly overloaded function identifiers. We still generate a parameterized version so that we may have a choice later on. The only significant change required to incorporate our specialized translation scheme is in the procedure *Translate-expression* of figure 4.8 and we show its updated version in figure 4.11. Whenever we have to translate an implicitly overloaded identifier, the algorithm first decides whether it wants to do parameterization or specialization and then proceeds accordingly. Note that it is even possible to mix the two; we may parameterize for most part and prune some branches of the resolution tree for some commonly occuring cases, short-circuiting them for specialization. Such a decision can be made while traversing and translating the resolution tree and is not shown in figure 4.11.

Def *Translate-expression*$_2$*(TE,expression)*
case *expression* **of**

'Identifier' $\Rightarrow$
if Identifier *is implicitly overloaded* **then**
Let $\rho_i =$ (Identifier? $\tau_1 \cdots \tau_n$) *be the type predicate of* Identifier.
$\rho_i \twoheadrightarrow \rho_u$ *where* ρ_u *is in normal form.*
if $\rho_u = \epsilon$ *and Specialization is desirable* **then**
Let Identifier$_{\tau_1 \cdots \tau_n}$ *be a new identifier.*
Instantiate the template of Identifier *using translations from the resolution tree of* ρ_i.
Generate the specialized instance Identifier$_{\tau_1 \cdots \tau_n}$ *with the above instantiated template.*
Generate and record the appropriate rewrite rules for Identifier$_{\tau_1 \cdots \tau_n}$.
Return 'Identifier$_{\tau_1 \cdots \tau_n}$'.
else
Let Identifier$'$ *be the parameterized translation instance of* Identifier.
$TE \vdash \rho_i \Rightarrow e_1 \cdots e_k$.
Return 'Identifier$'$ $e_1 \cdots e_k$'.
endif.
else if Identifier *is explicitly overloaded* **then**
Let $\rho_i =$ (Identifier? $\tau_1 \cdots \tau_n$) *be the type predicate of* Identifier.
$TE \vdash \rho_i \Rightarrow e_i$.
Return 'e_i'.
else *Return* 'Identifier'.
endif.
endif.

'*expression*$_1 \cdots$ *expression*$_i \cdots$ *expression*$_n$' $\Rightarrow$
e_i = *Translate-expression(TE,expression*$_i$*)*. $(1 \leq i \leq n)$
Return '$e_1 \cdots e_i \cdots e_n$'.

'{ *binding*$_1$; $\cdots$ *binding*$_i$; $\cdots$ *binding*$_n$ **in** *expression* }' $\Rightarrow$
b_i = *Translate-binding(TE,binding*$_i$*)*. $(1 \leq i \leq n)$
e_b = *Translate-expression(TE,expression)*.
Return '{ b_1; $\cdots$ b_n **in** e_b }'.

endcase.

Figure 4.11: Specialization Translation Algorithm.

```
def vadd X Y = ...X[i]+Y[i]...;

instance (+) = vadd;

typeof a = I;
typeof b = (Vector (Vector F));

...(doublepair a b)...
```

(a): Overloaded Source Program.

$\texttt{(vadd? *0)} \longrightarrow \texttt{(+? *0)}$

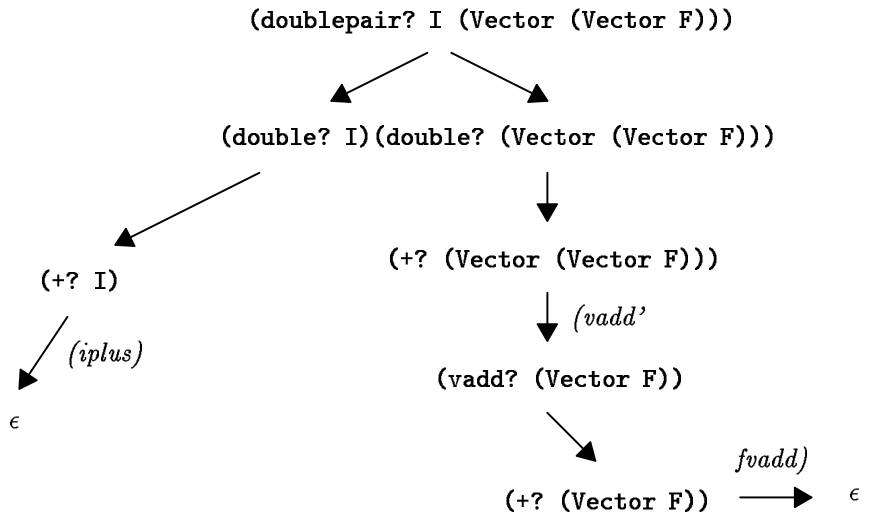
$\texttt{(vadd? F)} \xrightarrow{\texttt{fvadd}} \epsilon$

$\texttt{(+? (Vector *1))} \xrightarrow{\texttt{vadd'}} \texttt{(vadd? *1)}$

$\texttt{(+? (Vector F))} \xrightarrow{\texttt{fvadd}} \epsilon$

...(doublepair? I (Vector (Vector F)))...

(b): Rewrite Rules.



Overall Translation: *(iplus) (vadd' fvadd)*

(c): Predicate Resolution Tree.

Figure 4.12: An Example of Overloading Resolution and Specialized Translation.

Specialization Example

An example is appropriate at this point. We revisit the example of figure 4.6 in figure 4.12, this time deciding to specialize the resolution of the predicate `(+? (Vector F))` to generate `fvadd` as shown earlier. Note how the resolution tree gets pruned at that predicate and the additional rewrite rules generated involving `fvadd`. The overall translation has a simpler structure and will be more efficient.

The translation in figure 4.12 also illustrates how we can mix specialization and parameterization schemes together. The compiler may decide to specialize only some of the resolutions and parameterize the others depending upon the economics of the situation at hand.

Overloaded Constants Revisited

We argued in section 4.2.2 that it is undesirable to have implicitly overloaded top-level constants, and we flagged an error if we could not resolve all the type predicates in their body. With the above discussion in mind, we can relax that constraint slightly and say that implicit overloading of top-level constants is acceptable as long as they occur only in fully resolved type contexts so that a specialized version can be generated. We will have to store a definitional template for such constants as well. We will still flag an error if such constants occur in an unresolved or partially resolved context because we do not want to parameterize them for pragmatic reasons. The steps involved in their translation are exactly similar to those for function identifiers above and we do not show them separately.

4.3.3 Alternate Translation Strategies

So far, we have been quite ambitious in dealing with overloaded identifiers: we described a general parameterization mechanism and a specialization strategy that helps to improve the efficiency of the generated code. But these strategies are fairly costly and complex to implement. Parameterization pays a run-time penalty, while specialization requires a lot of compiler analysis and overhead in maintaining definition templates and multiple specialized versions.

It is possible to take a much more restricted view of overloading that eases the burden on the compiler and at the same time does not incur any run-time cost. In this scheme we do not allow implicit overloading at all. Only pre-defined operators are allowed to be overloaded with all their instances appearing in standard libraries. All overloading clauses within user-defined functions must be completely resolved, either *via* their inferred type contexts, or through explicit type annotations supplied by the user. In this way, each user-defined function has exactly one semantic interpretation that is as efficient as non-overloaded program code.

This strategy still does better than the *ad hoc* schemes adopted in the conventional languages because the compiler writer can control what pre-defined identifiers will be overloaded and include their instances in the standard libraries. The users may also be given some flexibility by allowing the scope of resolution of their definitions to be the whole program. In this way, an apparently overloaded definition can be resolved to its unique semantic interpretation by looking at the context of its use elsewhere.

A detailed description of this strategy is beyond the scope of this thesis, but we should mention that due to its efficiency, this scheme is used in the current version of Id [48] as a preliminary solution until the more general schemes described earlier are implemented.

4.4 Comparison with the Proposal of Wadler and Blott

Our overloading scheme is based on the proposal by Wadler and Blott described in [63] and used in the language Haskell [27]. In this section we will compare the mechanisms for overloading resolution and translation in the two systems, their expressive power, and the different objectives they achieve.

Both systems have the notion of a *predicated* type, which is the standard Hindley/Milner Type augmented with a set of overloading predicate clauses. The system of Wadler and Blott is geared towards a direct parameterized translation of overloaded expressions using their typing derived in the Hindley/Milner type system. They modify and extend the usual set of type inference rules (refer to figure 2.5 in section 2.3.3) with rules for `overload`, `instance`, and predicate resolution, in order to express typing of expressions and the resolution and parameterized translation of overloading predicates, all in a single step. They derive typings of the form,

$$TE \vdash e : \sigma \setminus \bar{e}$$

which is read as "assuming the type environment TE, the expression e has a well-typing σ with a translation $\bar{e}$". Here, the type environment TE also records the translation associated with each identifier along with its type-scheme.

On the other hand, we have chosen to completely separate the typing of expressions from the issue of overloading resolution and translation. First, the usual Hindley/Milner types are derived for each expression in the program using the declared type template for overloaded identifiers. Then, in another pass, we employ our rewrite rule mechanism to resolve overloaded expressions and translate them into appropriate code. This enables us to experiment with various resolution and translation policies without affecting the type system. Expressing our rewrite rules as a general Term Rewriting System also allows us to reason about their other useful properties such as confluence and strong normalization, independent of the typing mechanism.

Apart from differences in the mechanisms, our system is an overall simplification of the system proposed by Wadler and Blott. An important aspect of their system is the notion of a "*class*" with a set of operations (also called a *method dictionary* in [63]) defined for it. Each instance of that class defines its own method dictionary that can be passed around as an object and identifies the actual functions that implement the methods of that class. For example[15],

```
class Num a where
   (+), (*)  :: a -> a -> a
   negate    :: a -> a
   zero, one :: a

instance Num Int where
   (+)       = addInt
   (*)       = mulInt
   negate    = negInt
   zero      = 0
   one       = 1
```

declares a class `Num` with five operations and its instantiation for integers. This is interpreted by the compiler as,

```
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a) a a

add  (NumDict addop mulop negop zeroop oneop) = addop
mul  (NumDict addop mulop negop zeroop oneop) = mulop
neg  (NumDict addop mulop negop zeroop oneop) = negop
zero (NumDict addop mulop negop zeroop oneop) = zeroop
one  (NumDict addop mulop negop zeroop oneop) = oneop

numDInt :: NumD Int
numDInt =  NumDict addInt mulInt negInt 0 1
```

which declares a method dictionary type `NumD` for the class `Num` along with method names to choose the appropriate method when given a dictionary. The interpretation of the instance

[15] These examples are in Haskell. The syntax is mostly self explanatory. Early alphabets like `a`, `b`, etc. denote type variables, `data` is an algebraic type declaration, and function definitions appear as equations. For details, the reader is referred to [63, 27].

declaration is to define a dictionary object `numDInt` that carries the method implementing definitions for that instance.

Our scheme can be viewed as a special case of their scheme where each class is allowed to have only one operation and the class name is the same as the operation name. In this way, we do not have to introduce any new notion of *class*. Secondly, we offer a simple mechanism for specialization of implicitly overloaded user-defined functions that seamlessly integrates with the parameterized translation mechanism to generate more efficient code.

One advantage of their scheme is that in some cases, their type predicates may have fewer clauses and hence give rise to fewer parameters than in our scheme, if we choose to parameterize. The following example shows this.

```
def vip A B =
  { sum = zero;
    (l,u) = bounds A
   in
    {for i <- l to u do
      sum = sum + (A[i] * B[i])
     finally sum }};
```

Our type system will translate this as follows,

```
typeof vip = (+? *0)∧(*? *0)∧(zero? *0) . (array *0) -> (array *0) -> *0;

def vip' (+) (*) zero A B =
  { sum = zero;
    (l,u) = bounds A
   in
    {for i <- l to u do
      sum = (+) sum ((*) A[i] B[i])
     finally sum }};
```

whereas their system simply passes a single method dictionary and uses its methods instead,

```
typeof vip = (Num? *0) . (array *0) -> (array *0) -> *0;

def vip' numD A B =
  { sum = zero numD;
    (l,u) = bounds A
   in
    {for i <- l to u do
      sum = add numD sum (mul numD A[i] B[i])
     finally sum }};
```

Their system also opens up interesting possibilities for inheritance and sharing of methods among classes giving them a somewhat object-oriented flavor. We will not go into any more details of their system and only mention that practical experience with Haskell should provide more data as to the usefulness of *classes* and the efficiency of the generated code.

Chapter 5

Typing Non-Functional Constructs

So far, we were working with a completely functional language; there was no notion of a store in the dynamic semantics (refer section 2.3.3), and no expressions involving locations or assignments to locations; we only had environments and value bindings to variables. In this chapter we will discuss the typing issues involved in extending our language with imperative or **non-functional** constructs such as pointers, assignments, and updatable data-structures.

The primary motivation behind incorporating non-functional constructs in the language is to facilitate the use of a wide variety of conventional techniques and algorithms based on such constructs within the framework of our polymorphic type system as described in chapter 2. Another pragmatic concern is that very often it is possible to achieve much higher execution efficiency for the same algorithm by using imperative constructs. This is usually the case while manipulating large data-structures (*e.g.*, arrays) where an *in-place* update is much more efficient than copying the whole data-structure in order to modify a small portion of it.

The problem of type-checking non-functional constructs has been studied in depth for the ML language by Tofte [57] and our discussion is based on that research. We will not go into the details of how to extend the Hindley/Milner type system to incorporate these constructs since that has been excellently dealt with in Tofte's thesis. Our approach will be to first describe the non-functional constructs in Id *via* examples. Then, we will discuss some theoretical and pragmatic issues involved in their polymorphic typing, taking examples from both Id and ML. We will briefly describe the solution presented by Tofte, his description of the solution by Damas in [19] and subsequent refinements by David MacQueen as implemented in Standard ML of New Jersey [2, 3]. Finally, we will present the approach taken in Id, which is a refinement of these earlier works.

5.1 Non-Functional Constructs in Id

Id has two classes of non-functional objects, **I-Structures** and **mutex** objects [9, 48]. We will not concern ourselves with the run-time dynamic semantics of these objects except for the fact that they behave like storage locations, in that they can be allocated without specifying their contents, be assigned to, or be queried for their contents[1]. For type-checking purposes, they are similar to the "`ref`" construct of ML which generates a storage location pointing to an object that can later be dereferenced for its value or assigned a new value.

We will informally describe the use of some non-functional constructs in Id and the associated requirements they place on the type-checker[2]. Consider the following Id example called *wavefront*[3] taken from [9],

```
{ A = I_matrix ((1,m),(1,n));
  {For i <- 1 to m do
     A[i,1] = 1 };
  {For j <- 2 to n do
     A[1,j] = 1 };
  {For i <- 2 to m do
     {For j <- 2 to n do
        A[i,j] = A[i-1,j] + A[i-1,j-1] + A[i,j-1] }}
 in
   A }
```

The function `I_matrix` allocates and returns a fresh, uninitialzed I-structure matrix with type `(I_Matrix *0)`, given its 2-dimensional bounds. Note that the type of the elements of the matrix is not known at this point. This type has to be obtained *via* the assignment operation occuring elsewhere in the program. Moreover, the type-checker must make sure that all locations of the matrix are filled with elements of the same type, since arrays are meant to be homogeneous. The problem is non-trivial since allocation of an array can be widely separated from its assignment in the program text.

It is possible to write the above example functionally using special array specification syntax called *array comprehensions*[4]. Here, we simply specify an array of values all together. There is

[1]The reader should note here that the dynamic semantics of I-structures and mutex objects is very different from conventional storage locations. But here, we are interested only in their static semantics.

[2]All our examples deal with I-structure objects, similar examples can be constructed involving mutex objects.

[3]The function `I_matrix` dynamically allocates an uninitialized I-structure matrix. This is subsequently filled in three independent regions. The left and the upper boundary are initialized to 1 and the computation of the remaining elements depend upon their neighbors to the left and above. Given the parallel semantics of Id, all loop iterations execute in parallel, filling the matrix from the top left corner to the bottom right in a wave-like fashion, hence the name *wavefront*.

[4]The footnote on page 119 explains the syntax of array comprehensions. Non-strict semantics of Id allows the use of an array before it is fully specified.

no notion of assignable locations, the value of the array at each index is specified at the time of creation once and for all.

```
{ A = {matrix ((1,m),(1,n))
        | [i,1] = 1 || i <- 1 to m
        | [1,j] = 1 || j <- 2 to n
        | [i,j] = A[i-1,j] + A[i-1,j-1] + A[i,j-1]
                    || i <- 2 to m & j <- 2 to n };
 in
   A }
```

It is meaningless to assign a value to a location of a functional array just as it is meaningless to assign a value to the integer 1. The type-checker is able to detect such errors, since functional arrays have a different type from non-functional arrays which are assignable. Even though we keep the two kinds of arrays separate, for the sake of simplicity and efficiency, we may want to implement the functional arrays using assignable I-structure arrays. We will come back to this point in section 5.3.

We show another example to illustrate that a non-functional implementation may considerably improve the efficiency of an otherwise functional operation. Consider the problem of appending two lists; a recursive, functional version is shown below.

```
def append nil l2 = l2
  | append (x:xs) l2 = x:(append xs l2);
```

The above function recursively calls itself while traversing down the first list, implicitly recording each of its elements in a separate activation. At the end of the list, it pops the stack of activations one by one, *cons*-ing the elements over the second list. This is a very expensive implementation as it requires as many procedure invocations as the length of the first list. A tail-recursive version shown below, is more efficient but still allocates too many *cons*-cells, because it has to reverse the first list before *cons*-ing up its elements onto the second list, so that its elements are added in the right order.

```
def append l1 l2 = loop (reverse l1) l2;

def loop nil l = l
  | loop (x:xs) l = loop xs (x:l);
```

Instead of these functional solutions, below we show an implementation using a technique in Id known as "open lists"[5]. The idea here is to be able to grow a list at its end rather than at its front. There is no functional way to do this. Using open lists, we can efficiently copy the

[5]Refer [6, 48] for details and more examples of this technique.

first list as we traverse it down in a loop and stick the second list at its end in order to *close* it and make it into a valid functional list[6]. Only as many *cons*-cells are allocated in the process as necessary in copying the first list.

```
Def append l1 l2 =
  { header = OLcons _ _ ;
    current_cell = header;
    last_cell = {while not (nil? l1) do
                   next_cell = OLcons _ _ ;
                   element = head l1;
                   assign next_cell (OLcons element _);
                   assign current_cell (OLcons _ next_cell);
                   next l1 = tail l1;
                   next current_cell = next_cell
        Finally current_cell };
    assign last_cell (OLcons _ l2);
    result = tail header;
   in result };
```

The function `OLcons` allocates a fresh I-structure *cons*-cell which initially has the type `(list *0)`. Again, the type of the list element to be stored in the *cons*-cell is not known at the time of this allocation. This is determined implicitly *via* the `assign` operation over that *cons*-cell. Moreover, the type-checker must make sure that lists constructed in this manner are homogeneous and further `assign` operations on its *cons*-cells are disallowed because we intend the `append` operation to return functional lists.

It should be clear that in general, such storage allocating functions destroy the referential transparency of the program and hence render it non-functional. A simple example is shown below.

```
e1 = (i_array (0,1)),(i_array (0,1))
```

is not equivalent to

```
e2 = { a = i_array (0,1)
      in
       a,a }
```

The expression `e1` allocates two separate arrays, returning a pointer to each in a tuple, whereas `e2` attempts to eliminate the common subexpression `(i_array (0,1))` and allocates

[6]This example uses assignable I-structure *cons*-cells created *via* the allocator function `OLcons`. The special identifier "_" (underscore) indicates that the particular field of the allocated *cons*-cell is initially empty. The binary operation `assign` is the equivalent of the assignment operation in this case. It stores the specified values from its second argument pattern into the *cons*-cell given by the first argument. Assignment into a particular slot of the cell may be avoided by using the special identifier "_" in the pattern specification. The `next`-ified identifier bindings within the body of the loop specify the values of those identifiers for the next iteration of the loop.

only one array, returning a tuple of pointers that are aliased to the same array. Such a transformation does not affect the meaning of functional programs (say, if numbers were present instead of I-structure arrays), because functional programs are referentially transparent. But the above pair of expressions have different meanings as shown by their use in the following simple context.

```
def context (x,y) = { x[1] = 1
                      in
                       y[1] };
```

`context e1` $\Longrightarrow$ *Undefined!*
`context e2` $\Longrightarrow$ `1`

Referencial transparency is a very useful property for compiler optimizations, so we would like to preserve it for the rest of the program even if some parts of it are non-functional. Therefore our goal is two-fold,

1. Non-functional constructs are desirable because there may be some algorithms that are inherently non-functional or their functional versions do much worse in execution speed as the `append` example shows. Therefore, we must provide sound type inference rules for them in our language.

2. We should be able to restrict the scope of visibility of these constructs to clearly demarcated non-functional regions, so that functional optimizations can proceed unhindered in the rest of the program.

5.2 Issues in Typing Non-Functional Objects

In this section, we will discuss the issues involved in inferring sound typings for non-functional constructs such as I-structures in Id or the `ref` construct in ML. We will informally describe some extensions to the Hindley/Milner type system that are proposed in the literature in order to take care of these constructs.

5.2.1 Store and Store Typings

The basic conceptual shift in going from a functional view of the world to an imperative one is to expand the notion of *expressions manipulating values* to that of *expressions manipulating values and objects*. Values stand only for the mathematical entities they denote, while there is a well defined notion of an instance of an object separate from its contents or value. This corresponds to a location in the run-time store where this object may be found. This gives rise to a sharp difference between the functional typing mechanism where we assign a type to every value (or an expression that evaluates to a value), and the imperative world where we have to assign a type to an object (or an expression that evaluates to an object) which may not yet have a value associated with it. The problem in having objects with unknown or updatable values is that we must make sure that values (or other objects) that ever get stored into that object must all be of the same type. Otherwise, we may easily generate unsound typings and run into run-time type errors. The following example from Tofte's thesis [57] chapter 3, page 31, illustrates this problem.

```
let r = ref (λx.x)
 in (r := λx.x+1; (!r) true)
```

`ref` creates an assignable storage location that is initialized to the identity function, but is later overwritten with the successor function. Therefore its later use ("`!`" is the dereferencing operator) in an application to the boolean `true` is a type error. But this may go undetected if we assign the type $\forall\alpha.\alpha \rightarrow (\alpha\ \mathit{ref})$ to `ref` and allow generalization of the type $(\beta \rightarrow \beta)\ \mathit{ref}$ assigned to `r` in the usual manner in a `let`-binding.

The problem is that the location created by the use of `ref` is a single object and can have at most one type. Assigning types to address locations in the store is what Tofte calls a *Store Typing*. And, he goes on to show that generalization over type variables that are free in the current store typing gives rise to unsound type inference and should not be permitted.

5.2.2 Approximating the Store Typing

The difficulty in preventing such generalizations is that the store and the store typing come into existence only at run-time. So we need a conservative approximation of the store typing for the purpose of static type analysis in order to figure out what type variables *may* be free in the store typing.

The type checker must also know when new type variables are introduced in the store typing (and possibly, when some type variables may be removed from it). This means that there has to be some indication that evaluating a particular expression will cause allocation of new objects, possibly with free type variables embedded in their type, and therefore may involve adding new type variables to the store typing. For example, the evaluation of `ref` itself does not expand the store, it is the evaluation of its application to another expression that creates a new reference and has to be recorded in the store typing. Again, we will have to make a conservative approximation of the actual run-time behaviour of the expression.

Different authors have taken different positions in this matter. We will outline three schemes by means of examples and describe our position in the next section.

Approximation I

Tofte [57] makes a broad syntactic classification of expressions into two categories, applications and let-expressions termed as **expansive**, and identifiers and λ-abstractions termed as **non-expansive**. The idea is that only expansive expressions can ever lead to allocation of an object, which in turn implies an expansion of the store typing. Moreover, he classifies the type variables into two categories as well, **imperative** (meta-variable u) and **applicative** (meta-variable t) type variables. Storage allocating functions bind and return imperative type variables, while the usual functional definitions use applicative type variables. Thus, the built-in storage allocator in ML `ref` has a type $\forall u.u \rightarrow (u\ \mathit{ref})$. Using the above extension to the Hindley/Milner typing framework, Tofte derives typings of the usual form,

$$TE \vdash e\ :\ \tau$$

The key idea is that while typing an expression "`let` $x = e_1$ `in` e_2" using the LET-rule, the imperative type variables in the type of e_1 that would otherwise be closed, are not allowed to be generalized if expression e_1 is expansive. This is the conservative guess Tofte makes in order to recognize a type variable entering the store typing. Thus, if expression e_1 is the identifier `ref` or a λ-abstraction with imperative type variables in its type signature (possibly due to occurrences of `ref` inside it), it can still be generalized since those expressions are taken to be non-expansive, but an application of `ref` to another expression or its use in another let-expression can not be generalized since it is deemed expansive.

Coming back to the problematic example above, the type of the let-bound identifier `r` will be inferred as $(u \rightarrow u)\ \mathit{ref}$ and will not be generalized since the application `ref (λx.x)` in the binding is rightly believed to be expansive. Therefore, the first statement will unify the type of the location `r` to be $(\mathit{int} \rightarrow \mathit{int})\ \mathit{ref}$ and the system will detect the type error in the second expression while applying the contents of `r` to a *bool*.

Approximation II

Damas, in his thesis [19], takes a slightly different position[7]. He generates typings of the form,

$$TE \vdash e : \sigma * \Delta \tag{5.1}$$

where, σ is a type-scheme and Δ is the set of types of all objects to which new references may be created as a side effect of evaluating e. This is his approximation for the set of types that may occur in the store typing. We will call this set the **immediate store types**. In addition, each function type-scheme carries a list of types Δ', that denotes an upper bound for the types of objects to which new references will be created as a side effect of applying that function. We will call these types **future store types**. At each application of such a function, fresh instances of its future store types are added to the set of immediate store types signifying the fact that new objects are allocated in the store when this function is invoked, expanding the corresponding store typing. λ-abstractions simply convert the immediate store types of its body into future store types, again denoting the fact that the definition itself does not create any new objects but its application will. Finally, generalization over type variables occurring in the immediate store types is prohibited, which guarantees the soundness of the type inference system.

One immediate consequence of this simple two-level partitioning is that all embedded future store types in curried λ-abstractions are pushed up to the first λ-abstraction. Thus, the very first application will cause all of them to be added to the set of immediate store types, even if the actual allocation of the storage object does not occur until further applications. We will come back to this point later.

In Damas' system, `ref` is typed as,

$$TE \vdash \texttt{ref} : (\forall \alpha . \alpha \rightarrow \alpha \; \mathit{ref} \; * \; \{\alpha\}) \; * \; \{\} \tag{5.2}$$

The immediate store types are empty, denoting the fact that merely mentioning `ref` does not allocate any new references, but the future store types are non-empty signifying that each application of `ref` will indeed cause expansion of the store and the store typing.

Damas' system makes finer distinctions among expressions in deciding which expressions allocate objects and when that allocation occurs, than Tofte's system which uses a simple

[7] Our account of Damas' solution is based on its lucid exposition by Tofte in [57], chapter 6.

syntactic classification to achieve that. So it should not surprise us if Damas' system admits programs that Tofte's system rejects. The following example shows this.

```
let f = let x = 1
         in
          λy.!(ref y)
 in
  f 1, f true
```

Tofte's system does not generalize over the inferred type $(u \rightarrow u)$ for `f` because the inner let-expression is deemed to be expansive. While in Damas' system, the λ-abstraction returned by the inner let-expression does not have any immediate store types but has a non-empty future store type list. This rightly permits multiple instantiations in the following tuple expression, creating two different reference locations that store `1` and `true` respectively.

Approximation III

Even though Damas' system attempts to predict and propagate information about the point of allocation of store objects that may be hidden under λ-abstrations, it performs poorly with curried function definitions. It knows only about a single level of λ-nesting that hides future store types from immediate store types. It does not distinguish between curried functions where the store object is created after the very first application or in subsequent applications. The somewhat contrived Id example shown below makes this distinction clear.

```
def store a i v = { a[i] = v;
                   in
                    v };

def f1 = { a = i_array (1,1);
          in
           store a };                % N -> *0 -> *0

def f2 i = { a = i_array (1,1);
            in
             store a i };            % N -> *0 -> *0

def f3 i v = { a = i_array (1,1);
              in
               store a i v };        % N -> *0 -> *0
```

The definitions `f1`, `f2`, and `f3`, all have the same type. The difference lies in the time of allocation of their I-structure array cells. In case of `f1`, this allocation has already occurred at the time of definition of `f1` and the corresponding type variable `*0` has already entered the store typing. Therefore, `f1` can subsequently be used only in one way. In case of `f2` and `f3`, the allocation occurs after one and two applications respectively, and till that happens, each of them can be instantiated over different types without the danger of unsound type inference since each instantiation will eventually generate a new fresh array cell.

Following an idea due to David MacQueen that is currently used in Standard ML of New Jersey [2, 3], we may add this flexibility to Damas' system as follows. Each type variable inside a type expression is associated with a natural number called its *rank*. This identifies the depth of λ-nesting of the given type variable after which it will enter the set of immediate store types. This is exactly the number of λ-abstractions suspending the creation of a storage object in the associated expression. For each application to the current expression, the rank of its type variables is reduced by 1 and those with rank equal to the current level of λ-nesting are added to the set of immediate store types. This gives a much more fine-grain control over what type variables may enter the store typing and at what time. In this scheme, an applicative type variable will have the rank of infinity.

This system effectively generalizes the concept of future store types of Damas' system to arbitrary levels of λ-nestings. Unfortunately, very little documentation is available about its implementation, and to our knowledge, an in-depth theoretical analysis has not been carried out yet.

5.3 Our Approach in Id

5.3.1 Typing Open Structures

Id, with non-functional arrays and open lists (and indeed other open constructs such as algebraic types and tuples as proposed in its latest extension in [48]) has exactly the same flavor of typing complexity as ML's `ref` construct. The currently implemented mechanism in the Id Compiler was developed independently by the author under the guidance of Prof. R.S. Nikhil and was later found to be similar to Damas' system described in the last section.

One major difference between our system and Damas' system is that we infer types for expressions while he infers type-schemes as given by typing 5.1[8]. This permits him to pass around a function type-scheme with non-empty future store types, without pushing any of them to immediate ones. The instantiation of the type scheme and the expansion of the immediate store types occur together when this function is used in an application. The typing 5.2 for the

[8]Refer back to figures 2.4 and 2.5 in section 2.3.3 for a comparison of the two inference systems.

`ref` identifier is an example of such inference. But in our system, a type-scheme is instantiated as soon as it is referred, even as in a simple variable expression (see figure 2.4). So the future store types are made immediate much earlier. For example, using Damas' notation for our scheme, each occurrence of `ref` will be typed as,

$$TE \vdash \texttt{ref} : (\alpha \to \alpha\ ref)\ *\ \{\alpha\} \tag{5.3}$$

which instantiates its type-scheme and pushes its future store types to immediate ones prohibiting further generalizations over α. Indeed, this is even weaker than Tofte's scheme where such generalizations are permitted because variables are treated as non-expansive. Thus, the following example that simply renames the storage allocating function `i_array` is rejected by our system even though both Tofte's and Damas' system accept it.

```
{ allocator = i_array ;
 in
  allocator bounds, allocator bounds }
```

The astute reader will note that such programs can be made to type-check within our scheme by using the original allocator function at all places.

Our current system is clearly not very satisfactory and we plan to refine it in the direction of the solution offered by David MacQueen by taking into account the current level of λ-nesting. This will not only take care of the above problem but also improve over the schemes of Tofte and Damas. Even though this system appears to be substantially more complicated than simple syntactic divisions made by Tofte, there are non-trivial, interesting examples where this system wins over the other two. The following example shows this:

```
def make_i_array bounds f =
   { a = i_array bounds;
     (l,u) = bounds
    in
      {for i <- l to u do
         a[i] = f i
       finally a }};
...
memoize_mybounds = make_i_array mybounds;
...
squares = memoize_mybounds square;
...
evens? = memoize_mybounds even?;
```

`make_i_array` is a library function that memoizes the function `f` within the given `bounds`. In an application program, we may deal with different arrays with the same bounds, so it is logical to specialize the memoizing function once (`memoize_mybounds`) instead of carrying the bounds all over the program. The problem here is that we have to associate the allocation of a fresh array with each application of `memoize_mybounds` and not with the initial application of `make_i_array`. This is possible only if we record and propagate the fact that the array allocation inside `make_i_array` is under 2 levels of λ-nesting and will not take place until after two applications. Neither Tofte's nor Damas' system is capable of doing this.

5.3.2 Closing Open Structures

Another important concern is efficient implementation of functional data-structures. We gave examples of array comprehensions earlier. These generate functional arrays whose allocation and value is specified together. Clearly, the compiler will have to deal with them independently because of their more lenient typing mechanism. Each such construct adds to the syntactic complexity of the language and it would be nice if we can express them in terms of a small kernel language consisting of low level constructs. We will now examine some issues arising out of non-functional implementations of such functional constructs. In the following discussion, *open* objects refer to any kind of assignable data-structures and functions (with arbitrary, but finite λ-nesting) that manipulate them.

In Id, array comprehensions are *desugared* into I-structure array allocations and loops that fill them according to the comprehension clauses. A similar strategy is adopted for list comprehensions[9] and other list manipulating functions (such as `append` shown in section 5.1), all of which are implemented using open lists. The important point here is that, once constructed, these arrays and lists are guaranteed to be used only functionally. Therefore, we should be able to *close* the type of these open data-structures in order to regain the polymorphism lost due to their non-functional implementation. In Tofte's terminology, we wish to convert an imperative type variable into an applicative one whenever it is safe to do so.

The following example shows what we mean. The function `make_array` returns a functional array. We intend to implement it using a slight variation of our `make_i_array` function above.

```
def make_array bounds f =
  { (l,u) = bounds
    in
      {array bounds
```

[9]List Comprehensions are similar to array comprehensions in structure and mechanism but generate lists instead. See [46, 48] for details.

```
          | [i] = f i || i <- l to u }};
```

is implemented as

```
def make_array bounds f =
  { (l,u) = bounds;
    in
      { a = i_array bounds;
        {for i <- l to u do
           a[i] = f i };
       in
         (read_only a) }};
```

`read_only` is a special type coercion operator that has the effect of converting an I-structure array to a functional array. In particular, it accomplishes two important tasks. First, it actually coerces the type `(I_array *0)` to `(Array *0)`. This enables the type-checker to identify subsequent assignments to that array as type-errors since only I-structure arrays are deemed assignable. Second, it lifts the polymorphic typing restrictions on the type variables of such structures rendering them truely functional. In other words, it substitutes applicative type variables for imperative ones in the types of these structures. Of course, this must be done with care so as not to generate unsound typings involving non-functional constructs, which we set out to correct in the first place.

Both the above mentioned tasks are compile-time (specifically, type-checking time) operations and do not affect the compiled code. In the particular example shown above, the coercion is safe since we do not allow assignments to functional arrays and nobody has a handle on the I-structure alias of the array after the coercion is done. This last condition is our guarantee for generating only sound typings involving the closed array, because the array cannot be assigned to anymore. In compiler generated desugaring, we may be able to ensure that the I-structure array does not *escape* the block of code in which it is allocated and hence can be safely coerced to a functional array outside it, but in general, enforcing this condition may require a lot of compiler analysis.

5.3.3 Comments

Finally, we would like to point out that incremental typing of non-functional constructs using our incremental book-keeping mechanism described in chapter 3 is only slightly more complex. In particular, we will have to maintain a few more compilation properties, such as which type variables are imperative, and what is their rank, etc. We even allow globally open data-structures–they will be treated as global constants, possibly containing imperative type variables with rank 0. This is more general than the position taken in Standard ML of New Jersey, where top-level reference data-structures are forced to be grounded (no type variables allowed) as soon as they are allocated. A more detailed description of this scheme is beyond the scope of this thesis.

Chapter 6

Conclusion

In this thesis, we described the mechanism of type inference in the programming language Id, as well as its other features that have a direct bearing upon the type system. We summarize our results below.

6.1 Summary

We used the Hindley/Milner type inference mechanism as a basis for the type system in Id. First, we described a deterministic set of type inference rules and the standard inference algorithm available in the literature that infers sound and complete types for complete programs. We also showed a translation of the Id Kernel language to the abstract mini-language used in the algorithm.

Then, we described our main research contribution, an **incremental** type inference mechanism for Id. First, we established a general framework for incremental computation and maintenance of compile-time properties in the light of frequent additions or editing of the program. Then, we gave a detailed account of how to extend the Hindley/Milner type inference algorithm to work in this incremental framework and demonstrated a proof of its soundness and completeness with respect to the original type system. We evaluated the performance of this system and then discussed optimizations to improve upon it.

We also described other features of the language Id that are important from a type inference perspective. We presented a systematic scheme to permit overloading of identifiers in the language, describing rules for their type inference and then discussing several mechanisms for their translation into actual code. We compared and evaluated two strategies in particular, parameterization and specialization. Finally, we touched upon the type inferencing issues for polymorphic non-functional constructs of Id. We outlined three schemes, each giving more flexibility and polymorphism to non-functional constructs than the previous one, while still maintaining the soundness of type inference.

6.2 Future Work

At several places in the thesis we hinted at the possibility of future research work. We summarize these areas below.

6.2.1 Extensions to the Language and the Type System

The Hindley/Milner type system has been extremely successful and popular type inference system for modern languages, and therefore it was our natural choice while designing the type system for Id. But recently, there have been further advancements that strengthen this system non-trivially by permitting a limited amount of polymorphism even for λ-bound variables [30]. It may be worthwhile to examine and evaluate the benefits of upgrading the type system to this level.

On another dimension, we would like to suggest extensions to the already rich repertoire of types and data-structures in Id. One possible direction for extension is to introduce modules, functors, and structures in Id like those in ML [37]. Id already has abstract data types and an associated notion of information hiding, but structures and parameterized modules (functors) are more powerful features geared towards programming in the large and disciplined code sharing.

Another area worth exploring is to introduce inclusion polymorphism, subtyping and inheritence among objects and object-classes, in the language as a step towards object-oriented languages. A lot of activity has been going on in this area recently and we are watching the field in anticipation. This will definitely be a big change in the typing philosophy and will necessitate a reexamination of the basic type system as well as its incremental variants.

6.2.2 The Incremental System

We presented a general framework of incremental property maintenance in section 3.1.2. It is possible and probably desirable to do a theoretical characterization of the incremental properties that can be maintained in our system, the syntactic structure of the property domains, and showing that our incremental scheme will work for any property that can be computed by a monotonic function over these domains.

The correspondence theorem 3.12 shown in section 3.6 is valid only for the incremental system where no editing was allowed. For the editing case, we only presented what extra properties we may have to maintain. A likely direction of reserach would be to concretize the requirements and the effect of editing more carefully and modify the proof of soundness of the incremental type inference to account for it.

In section 3.8 we analyzed the worst-case complexity of our incremental system. In our experience, the incremental system performed very well on an average and it was our feeling that the worst-overhead cases are not very common and easily avoided. In particular, we felt that the size of the strongly connected components of identifiers is usually very small and the number of recompilations can be kept within limits by intelligent scheduling. A more detailed empirical evaluation with real compilation data is necessary in order to substantiate these hypotheses.

6.2.3 Overloading Analysis

We mentioned at the end of section 4.2 that the compiler needs to be smart in order to detect and prevent circular instance declarations. Preventing cycles in our set of instance rewrite rules will guarantee termination of the overloading resolution process and make the compiler more robust. We would like to incorporate such criteria into our compiler.

A major topic that we side-stepped in overloading translation is the compiler analysis required to decide which translation strategy to use. Various economic factors need to be considered before a compiler can decide whether to parameterize or to specialize or to generate some mixture of the two. The compiler can utilize hints from the user as well as generate its own estimate of the cost of either of these strategies. It is also possible to employ general techniques for partial evaluation to automatically convert parameterized code into specialized one. We have not presented any scheme for this analysis and this clearly presents possibilities for further reserach.

We also need to explore less ambitious schemes that have low run-time and compile-time overhead costs, and weigh them against the costs and the benefits of implementing the complete proposal. We are currently investigating one such scheme in Id, where we allow overloading of only predefined identifiers and generate code with no run-time overheads.

6.2.4 Non-Functional Type Inference

Our current implementation for typing non-functional constructs in Id is sound but is very weak. One of our immediate concerns is to extend it to incorporate David MacQueen's idea of typing on the basis of the depth of λ-nesting. We also plan to generalize the concept of open structures, as defined in section 5.3.2, to tuples and algebraic types.

Another important task is to clearly demarcate the functional and the non-functional regions in the program and have a type-safe interface between the two. To go a step further, an important topic for research would be to analyze when is it safe to close an open structure so that it can be passed from a non-functional region to a functional region of the program. This analysis is important because closed structures possess more polymorphism and permit many compiler optimizations that can not be performed on open structures. This analysis will be useful in verifying annotations supplied by the user as well as validating efficient non-functional implementations of functional data-structures.

6.3 Concluding Remarks

Finally, we would like to point out that our work on incremental type inferencing has grown out of a need for a rich type system for interactive program development environments such as that for Id [50]. A rich type system is a valuable tool in developing large, robust, and reliable software. But in association with interactive environments, it requires careful engineering, so as not to inhibit their interactive and incremental nature.

The Hindley/Milner type inference system is a rich, polymorphic, and tractable system that offers a reasonably unrestricted programming style without compromising efficiency or robustness. Our incremental scheme further adapts this system for the interactive environment of Id and fits into a general framework for incremental information management that is interesting in its own right. Our work provides a sound theoretical basis to incremental type management along with its efficient implementation.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] Andrew W. Appel and David B. MacQueen. A Standard ML Compiler. Distributed along with the Standard ML of New Jersey Compiler, 1989.

[3] Andrew W. Appel and David B. MacQueen. *Standard ML Reference Manual.* Princeton University and AT&T Bell Laboratories, Preliminary edition, 1989. Distributed along with the Standard ML of New Jersey Compiler.

[4] Zena M. Ariola and Arvind. P-TAC: A Parallel Intermediate Language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures, London, UK*, pages 230–242, September 1989. Also: CSG Memo 295, MIT Laboratory for Computer Science 545 Technology Square, Cambridge, MA 02139, USA.

[5] Arvind and David E. Culler. Dataflow Architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986.

[6] Arvind, Steve Heller, and Rishiyur S. Nikhil. Programming Generality and Parallel Computers. In *Fourth International Symposium on Biological and Artificial Intelligence Systems, Trento, Italy*, September 1988. Also: CSG Memo 287, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.

[7] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. Technical Report CSG Memo 271, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, June 1988. An earlier version appeared in Proceedings of the PARLE Conference, Eindhoven, The Netherlands, Springer-Verlag LNCS Volume 259, June 15-19, 1987.

[8] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. Id Nouveau Reference Manual, Part II: Operational Semantics. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[9] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[10] Lennart Augustsson. A Compiler for Lazy ML. In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming, Austin, Texas*, pages 218–227, August 1984.

[11] Lennart Augustsson. Compiling Pattern Matching. In *Proceedings of the 1985 Workshop on Implementations of Functional Languages, Goteborg, Sweden*. Chalmers University of Technology, February 1985.

[12] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1984.

[13] Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. The Semantics of Second-Order Lambda Calculus. *Information and Computation*, 85:76–134, 1990.

[14] Rod M. Burstall, David B. MacQueen, and Don Sanella. Hope: an Experimental Applicative Language. In *Proceedings of the Lisp Conference, Stanford*, pages 136–143. ACM, August 1980.

[15] L. Cardelli. Compiling a Functional Language. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 208–217, August 1984.

[16] L. Cardelli. Basic Polymorphic Typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[17] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4), December 1985.

[18] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A Simple Applicative Language: Mini-ML. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 13–27, August 1986.

[19] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, Department of Computer Science, 1985.

[20] L. Damas and R. Milner. Principle Type Schemes for Functional Programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[21] S. I. Feldman. Make – A Program for Maintaining Computer Programs. In *Unix Programmer's Manual Supplementary Documents Volume 1*. 4.3 Berkeley Software Distribution, April 1986.

[22] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, September 1987.

[23] J. Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. These D'Etat, Université de Paris VII, 1972.

[24] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*, chapter 2. Springer-Verlag, 1979.

[25] Robert Harper, Robin Milner, and Mads Tofte. The Definition of Standard ML Version 2. Technical Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, August 1988. (also published as CSR-274-88).

[26] R. Hindley. The Principle Type Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[27] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[28] G. Huet. *Resolution d'equations dans les langages d'ordre 1,2,...,ω*. PhD thesis, Université de Paris VII, 1976.

[29] Paris C. Kanellakis and John C. Mitchell. Polymorphic Unification and ML Typing. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 105–115, January 1989.

[30] A.J. Kfoury and J. Tiuryn. Type Reconstruction in Finite Rank Fragments of the Second-Order λ-Calculus. Technical Report 89-011, Boston University, October 1989.

[31] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. An Analysis of ML Typability. Technical Report 89-009, Boston University, October 1989.

[32] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type-checking in the presence of polymorphic recursion. Technical report, Boston University, 1989.

[33] J.W. Klop. Term Rewriting Systems. Technical report, Center for Mathematics and Computer Science, Amsterdam, The Netherlands, September 1985.

[34] J.W. Klop. Term Rewriting Systems: A Tutorial. Technical report, Center for Mathematics and Computer Science, Amsterdam, The Netherlands, 1987. Also published in the Bulletin of the European Association for Theoretical Computer Science, 32, 1987.

[35] Kevin Knight. Unification: A Multidisciplinary Survey. *ACM Computing Surveys*, 21(1), March 1989.

[36] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[37] David MacQueen. Modules for Standard ML. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 198–207, August 1984.

[38] Harry G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 382–401, January 1990.

[39] Lambert Meertens. Incremental Polymorphic Type Checking in B. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, January 1983.

[40] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[41] Robin Milner. A Proposal for Standard ML. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 184–197, August 1984.

[42] Robin Milner. The Standard ML Core Language. *Polymorphism*, II(2), October 1985.

[43] A. Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.

[44] Rishiyur S. Nikhil. Practical Polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures, Nancy, FRANCE*, volume 201 of *Lecture notes in Computer Science*. Springer-Verlag, September 1985.

[45] Rishiyur S. Nikhil. Id Nouveau Reference Manual, Part I: Syntax. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[46] Rishiyur S. Nikhil. Id (Version 88.1) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.

[47] Rishiyur S. Nikhil. Overloading. For internal circulation in the Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA 02139, May 1988.

[48] Rishiyur S. Nikhil. Id Version 90.0 Reference Manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 1990.

[49] Rishiyur S. Nikhil and Arvind. *Programming in Id: A Parallel Programming Language*. Book in preparation, 1990.

[50] Rishiyur S. Nikhil, P. R. Fenstermacher, J. E. Hicks, and R. P. Johnson. *Id World Reference Manual*. Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, revised edition, November 1989.

[51] J. Ophel. AIMLESS: A Programming Environment for ML. Technical Report TR-CS-88-20, Australian National University, 1988.

[52] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[53] J. C. Reynolds. Towards a Theory of Type Structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[54] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[55] D. Scott. Data types as lattices. *Siam Journal of Computing*, 5(3):522–587, 1976.

[56] J. E. Stoy. *Denotational Semantics*. MIT Press, Cambridge, MA, 1977.

[57] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, Department of Computer Science, 1988. Also published as ECS-LFCS-88-54.

[58] Ian Toyn, Alan Dix, and Colin Runciman. Performance Polymorphism. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 325–346. Springer-Verlag, 1987. Proceedings of the FPCA Conference held in Portland, Oregon, 1987.

[59] Kenneth R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986.

[60] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures, Nancy, FRANCE*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1985.

[61] David A. Turner. An Overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, December 1986.

[62] Kevin S. Van Horn. Functional Language Implementations Survey, March 1990. Message on the Functional Programming Mailing List.

[63] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 60–76, January 1989.