Vrije Universiteit Brussel

Master thesis

Iterative typing

Author:
Laurent Christophe

Promotor:
Prof. Dr. Wolfgang De Meuter

Advisor:
Dr. Dries Harnie

August 20, 2012

Abstract

Types have been introduced into programming languages to increase safety and the abstraction level ; with typing, programmers do no longer directly manipulate byte strings but data of a certain type instead. Type systems must make a compromise between static safety and expressiveness ; static safety being the ability to detect errors based on a static code analysis and expressiveness being the ability to denote concepts. Traditionally, programming languages are separated into two classes: statically typed languages and dynamically typed languages. The first class emphasises early error detection and catches many errors during static code analysis. The second emphasises expressiveness but catches errors very lately, right before calls to built in functions. The gap between those two classes is considerable and we argue that the best of those two worlds can be combined. The compromise this thesis defends is called iterative typing. In essence, iterative typing is a static typing that delays as little checks as possible for runtime. When an uncertainty occurs during static code analysis, static typings typically make assumptions that allows to continue the analysis but inhibit the expressiveness of the language. In the same situation, iterative typing inserts annotations into the code so the uncertainty can be precisely resolved at runtime, when more information will be available. Iterative typing shines when type deduction depends on the executed branch of a conditional structure like it is the case for dispatch on types and deserialization patterns. As a validation of iterative typing we present three case studies of practical interest ; although the exposed solutions are implemented in a dynamic way, iterative typing is able to catch errors early on.

Résumé

Les types ont été introduits dans les langages de programmation pour augmenter le niveau de sécurité et d'abstraction ; grâce au typage, les programmeurs ne manipulent plus directement des chaînes de bits mais bien des données d'un certain type. Les systèmes de typages doivent faire un compromis entre la sécurité statique et l'expressivité ; la sécurité statique étant la capacité de détécter des erreurs sur base d'une analyse statique du code et l'expressivité étant la capacité à dénoter des concepts. Traditionnellement, les langages de programmation sont séparés en deux classes : les langages statiquement typés et les langages dynamiquement typés. La première classe privilégie la détection rapide d'erreurs durant une analyse statique du code. La seconde classe privilégie l'expressivité mais détecte les erreurs tardivement, durant l'appel de procédures primitives. La différence séparant ces deux classes est considérable et nous soutenons que le meileur de ces deux mondes peut être combiné. Le compromis que cette thèse propose est appellé typage itératif. Fondamentalement, le typage itératif est un typage statique qui retarde aussi peu de vérifications que possible pour l'exécution. Quand une incertitude apparaît durant une analyse statique du code, les typages statiques font des suppositions qui restreignent l'expressivité du langage mais permettent de poursuivre l'analyse statique. Dans la même situation, le typage itératif insert des annotations à l'intérieur du code pour que les incertitutes puissent être résolues durant l'exécution, lorsque plus d'informations seront disponibles. Le typage itératif montre l'etendue de ses capacités lorsque les types qui parcourent le programme sont influencés par des structures conditionelles comme c'est le cas pour la désérialisation d'objets. Comme validation du typage itératif, nous présentons trois analyses de cas ayant un intérêt pratique ; bien que ces solutions soient implémentées d'une manière dynamique, le typage itératif est capable de détecter des erreurs rapidement.

Acknowledgements

First I would like to thank professor Wolfgang Demeuter for promoting this thesis. When I came at his office, I asked him to work with a Lisp-like language and with type systems. Being an ULB student, he did not know anything of me and yet gave the perfect subject that allowed me to develop iterative typing. I would also like to thank him for his epic brain storming sessions (that left me brain dead for hours) that really sharped and recentralized my work. The second person I would like to thank is Dries Harnie. Dries was a gold mine of information concerning type systems and gave the general direction of this thesis. Without his advices, I would never been able to develop iterative typing. Beside helping me on the content of this thesis, he also helped me A LOT concerning its form. I thanks him for his patience and the time he gave me.

Special thanks to my mother and my grandparents which always supported me during my studies.

Contents

Chapter 1

Introduction

Type systems Types have been introduced into programming languages to increase safety and the abstraction level ; with type systems, programmers do no longer directly manipulate byte strings but data of a certain type instead. By increasing the abstraction level, type systems allow to define which operation can be applied on which data type. For instance type systems can prevent programmers to perform meaningless operations like mutliplying two byte sequences that represent strings. Typing is the discipline that provides that safety using sophisticated guarding systems. Type systems are mainly characterized by three criteria:

1. *Expressiveness*: the amount of concepts the type system is able to denote. Sometimes type systems inhibit the programmer's liberty ; for instance, many languages like Java do not allow the programmer to redefine the type of variables.

2. *Static safety*: the amount of errors detected based on a static code analysis. For instance, downcasts are a feature of Java that decrease its static safety.

3. *Error detection*: How early errors are detected is a prior concern since the cost of bug fix increases greatly as the development cycle goes on. Some languages like Java have IDEs that detect many errors as soon as they are written. On the opposite edge, some languages like Scheme detect errors at the last possible moment just before illegal calls of built-in procedures.

For any language expressive enough to have practical interest, it is not possible to be completely statically safe. A compromise needs to be done between those two criteria.

Static & dynamic typing Traditionally, languages are separated in two classes. The first class is called "dynamically typed languages" and emphasises expressiveness. In those languages, type systems do not restrict programmer's liberty but errors are detected lately ; as they happen. JavaScript and Scheme are both examples of dynamically typed languages. The second class is called "statically typed languages" and emphasises early error detection ; Java belongs to that class. Programs in those languages are statically analyzed which allows to detect many errors early on. Nevertheless, their expressiveness is restricted and they can not safely denote concepts like heterogeneous lists. People agree that dynamic typing is handy to quickly develop prototypes while statically typed languages are well suited to develop large and robust applications.

Problem statement The gap between dynamic typing and static typing is considerable ; programmers have few options between those extremes, between high expressiveness and high static safety. In particular, based on static code analysis, there is generally no clue to deduce which branch of a conditional structure will be actually executed. To resolve this uncertainty and carry forward the analysis, static typings typically assume both branches will be executed. Such assumption forces the branches to be type-compatible which greatly inhibits the expressiveness of the language. In this thesis, we challenge this modus operandi and argue it is possible to support dynamic events, like conditional structure with type-incompatible branches, and still being able to detect errors early on.

Contribution We developed a new typing technique called iterative typing. Iterative typing performs a static code analysis but considers that they are events, called dynamic events, that need runtime information to be analyzed precisely. When a dynamic event is encountered, annotations are inserted into the code so the type checking can be delayed to runtime when more information will be available. To illustrate our typing, we implemented in Scheme an interpreter of a practically usable Scheme subset.

Validation As validation we present in this thesis three case studies of practical interest.

The most direct application of iterative typing is deserialization, which is the use of a byte sequence to instantiate a data structure whose type depends on the byte sequence value. Iterative typing allows to detect a type mismatch right inside the reading procedure. Statically type languages can detect errors that early but require heavy use of type systems. On the other hand, iterative typing completely masks the type complexity involved in the pattern.

Iterative typing can also be used to propose an alternative of the common exception handling. In the presented alternative, the exception thrower knows whether throwing an exception will cause the application to stop or if the exception will be properly handled. In the first case, the exception thrower may decide it is preferable to return an usable result instead of stopping the whole application.

The last case study is a discussion about the interaction of iterative typing and side effects. In many cases, stopping the application when iterative typing predicts an error is just fine. But, sometimes it is preferable to perform side effects until normal Scheme applications would raise an error. This justifies the unsafe execution feature that allows the programmer to shut down iterative predictions for a while.

Road map The remainder of this thesis is structured as follows:

1. *Context*: We present a general introduction to type systems. We also compare in detail dynamic and static typing. At the end of the chapter we expose which problems iterative typing attempts to solve.

2. *Approach & Implementation*: We open this chapter by a discussion about case analysis on dynamic events. The algorithm based on such analysis would offer earliest error detection but would also grow exponentially with the amount of dynamic event. Iterative typing is then introduced as a succession of approximations of this case analysis.

3. *Evaluation*: In this chapter, we present three case studies to illustrate the possibilities and the limits of iterative typing. Those case studies are: deserialization, exception handling and a discussion about the interaction between iterative typing and side effects.

4. *Theory*: First we present a syntactic transformation of the used Scheme subset into a more suitable form for type systems. We then expose and justify the type system that lurks beyond iterative typing. Finally an iterative interpreter is exhibited and discussed in detail.

5. *Related work*: Here, two other typings that also blur the line between dynamic and static typing are presented and compared to iterative typing.

6. *Further work*: This chapter discusses three future iterative typing extensions: overloading using viability tests, polymorphic recursion and refinement of type deductions.

7. *Conclusion*: We open this chapter with a summary of this document, then we present our results and the further investigations we identified.

Chapter 2

Context

This chapter starts with a broad introduction of typing. The two main typing directions, static typing and dynamic typing, will then be exposed and compared in detail. At the end of the chapter we present the problems iterative typing attempts to resolve and where iterative typing is situated between dynamic and static typing.

2.1 A brief history of programming languages

Binary representation In the fifties the only available languages was the assembly ones that were machine dependent and directly manipulated bit words. Programmers were fully responsible to represent abstract data by bit sequences. The programs written in those languages were very remote from the human thinking and greatly polluted by low level burden like memory management. Also, those languages had very few guards making it easy to execute meaningless programs that can lead to memory corruptions.

Type as data representation The language C is very representative of the efforts made in the sixties to facilitate the programming task. C is a step towards abstraction. By providing build-in data type like `int` , `char` and `float` it attempts to free the programmer from the binary representation. In consequence, programs written in C are easier to write, easier to read, safer and more portable than assembly-like languages. Even if those programs are much safer than those written in assembly, it is still possible to execute unsound code because of features like unsafe cast (Figure 2.1).

High level languages Later on, relevant bug report and safety became a prior concern. Java, which appeared in 1995, is a widely used language that attempts to provide such characteristics. In those languages, only few very little used features are able to produce unsound executable code. But, sophisticated guarding systems are needed to ensure this safety which may significantly slow down the execution.

Figure 2.1: The effect on the memory of the statement: `(double*) "c"` . Firstly, a pointer to a character is created. Secondly, the data represented by this pointer is converted into a double precision number. As the binary representation of double precision numbers is bigger than the binary representation of characters, undefined data can be accessed and memory may be corrupted.

The applications written in those languages may still crash but in a more controlled and elegant way by throwing errors with relevant bug report. In Java for instance, castings are still possible, but they throw an exception when an invalid conversion occurs as in the program below:

```
Truck t = (Truck) new Object();
t.drive();                          // undefined behavior
```

```
java.lang.ClassCastException: java.lang.Object cannot be cast to Truck
```

Typing Typing is the discipline that studies those "sophisticated guarding systems" introduced in the previous paragraph. Typing techniques are characterized by:

1. *Expressiveness*: the amount of accepted sound executions.
2. *Safety*: the amount of rejected unsound executions.
3. *Error detection*: how early errors are detected and how relevant bug reports are.
4. *Performance*: the performance costs introduced by the guards.

We will see then that the perfect solution does not exist, compromises have to be done between those four criteria. Iterative typing is one of those compromises.

2.2 Expressiveness & Safety

Language A language is nothing more than a set, ordinary infinite, of symbol sequences called words or programs. A set of rule, called the syntax, is usually used to define the language boundary. To have any use, a language needs a semantic that associates meaning to its words. One can define multiple semantics for the same language ; for instance, `"3" + 1` has many reasonable meanings: the evaluation could return `4` , return `"31"` or simply fail.

Semantic One way to define the semantic of a program is to evaluate it inside any given context, in other words to explicit the relation between its input and its output. For instance we can define the semantic of the sentence "It is sunny today." by evaluating its truth value for any kind of weather. Formally, we define the semantic of a language as a function that takes a program and some inputs then returns a value as the result of the evaluation (Figure 2.2). A combination of a program and inputs is called configuration. One particular value is the error tag that indicates the semantic has failed. Notice that we can model side effects considering that the original memory is included in the input and that the transformed memory is included in the output[1].

Natural semantic It is a semantic that the language structure naturally suggests. For instance the natural semantic used in this thesis is the standard Scheme interpreter. The natural semantic allows us to define two configuration subsets and two program subsets (Figure 2.2):

- *Sound configuration*: causes the natural semantic to succeed (not return the error tag).
- *Unsound configuration*: causes the natural semantic to return the error tag.
- *Sound input relatively to a program*: causes the natural semantic to succeed (not return the error tag) for a given program.
- *Unsound input relatively to a program*: causes the natural semantic to return the error tag for a given program.
- *Sound program*: generates some sound configurations.
- *Unsound program*: generates only unsound configurations.

[1] In this section we cautiously avoid the halting problem, we suppose that all programs terminate. We could for instance suppose that after 10^{10} operations, the error tag is automatically thrown.

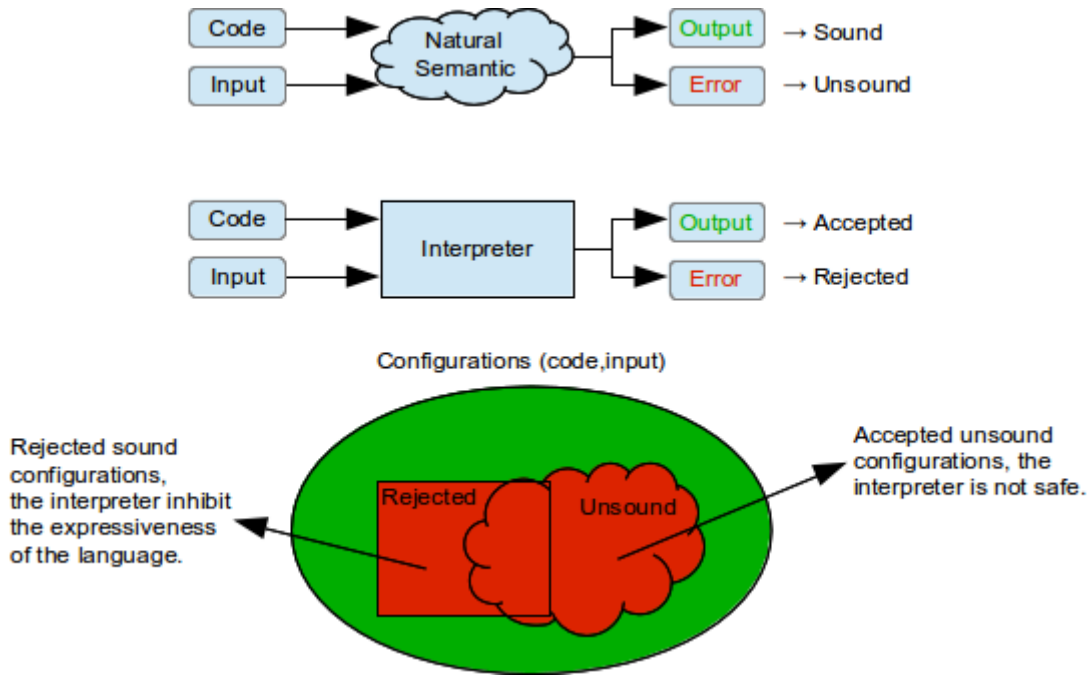

Figure 2.2: Given a programming language, we use two semantics: the one defined by the interpreter that is actually used and the natural one that represents an ideal.

Interpreter The interpreter defines the semantic actually used. For practical reasons, interpreters and natural semantics may vary. In this section we focus on failure discordances not differences between successful executions. In other words, we assert that when both semantics succeed, the result is the same. In the next paragraphs, we give examples about how interpreters and natural semantics can be different. Again, we use this semantic to define two configuration subsets (Figure 2.2):

- *Accepted configuration*: causes the interpreter to succeed (not throw the error tag).
- *Rejected configuration*: causes the interpreter to throw the error tag.

Expressiveness This paragraph deals with rejected sound programs which restrict the expressive power of the language. Here are some common Java restrictions:

1. *Variables have a fixed type.* The type of a variable is fixed at the declaration, it is not possible to change it after. Next we propose a Java program with a sound semantic, it creates a new banana and eats it, but it is rejected by standard Java interpreters:

```
Car a = new Car(); // a points to a new car
a = new Banana();  // a points to a new banana
a.eat();           // lets eat that banana
```

2. *Some of the context is ignored.* Generally the execution flow is not used to track errors on Java statements. The following program is globally sound even if separately some statements cause problems:

```
Object o = getSomething();
if(o.getClass() == Banana.class) {
    o.eat();                        // statement reached only when sound
}
if (false) {
    (new Banana()).drive();         // unreachable unsound statement
}
```

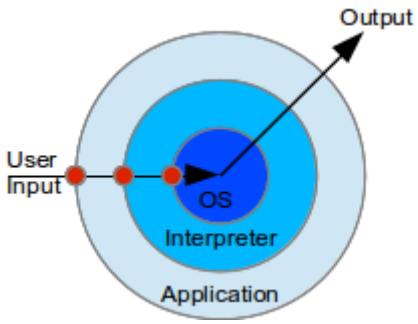

Figure 2.3: User inputs may be redundantly tested at each layer.

Safety This paragraph deals with accepted unsound programs. Low level languages like C allow developers to execute many unsound programs. When an unsound program is executed, either the OS layer detects an illegal operation and crashes the application, or no error is detected and the result is undefined. To bypass those restrictions introduced in the previous paragraph, high new level languages like Java also provide unsafe features that are loopholes for their safety. Here are two common unsafe features of C:

1. Unsafe casts:

```
int a = 1;
int* b = &a;              // get the address where 1 has been stored
double* c = (double*) b; // interpreter believe b points to a double
printf("%f",*c);          // try to print the double value stored in c
```

2. Direct memory management:

```
int* i = 123456; // erroneously consider i is an address to an integer
printf("%i",*i); // try to print the integer value stored in i
```

2.3 Error detection & Performance

Development cycle For a programmer, it is very important that errors are detected early in the development cycle. Indeed the cost to correct a bug right after code writing is negligible, when bugs in a deployed application may have catastrophic consequences. Nowadays, some IDEs like Eclipse constantly analyze the code and a lot of errors are detected as soon as they are written. Further we will see that it is not possible to catch all errors as early and that testing remains an indispensable step for software development.

Abstraction level Early error detection has an other advantage: it eases the debugging task. When an error is detected early, the error source is not difficult to spot and the fix is easier. Moreover, being close to the error source grants access to informations on the right abstraction level and improves the quality of bug reports.

The agenda sample To expose different kinds of error detections, we present a sample program that asks the user to access a random agenda:

```
import sun.misc.Unsafe;
import java.lang.reflect.Field;

public class Agenda {

    static public void main(String[] args) {
        Unsafe unsafe = getUnsafe();
        long[] year = createYear(unsafe);
```

```
        System.out.println("Please input a day and a month to consult the agenda...");
        java.util.Scanner scan = new java.util.Scanner(System.in);
        int day = scan.nextInt();
        int month = scan.nextInt();
        // Month guard
        if (month < 1 || month > 12) {
            System.out.println("Month must be between 1 (January) and 12 (December).");
        } else {
            System.out.print("On the " + day + "/" + month + " we have: ");
            System.out.println(unsafe.getByte(year[(month - 1)] + (day - 1)));
        }
    }

    // Fill randomly a byte matrix.
    // More precisely, year is a Java array of memory address.
    // Each element of year points to a 31 byte wide memory block.
    static public long[] createYear(Unsafe unsafe) {
        java.util.Random r = new java.util.Random();
        long[] year = new long[12];
        for(int month = 0; month < 12; month++) {
            year[month] = unsafe.allocateMemory(31);
            for (int day = 0; day < 31; day++) {
                unsafe.putByte(year[month] + day, (byte)(r.nextInt() % 125));
            }
        }
        return year;
    }

    // Returns an Unsafe instance for direct memory management
    static public Unsafe getUnsafe() {
        Unsafe unsafe = null;
        try {
            Field field = Unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            unsafe = (Unsafe) field.get(null);
        } catch (Exception e) {
            throw new RuntimeException("Unsafe use failed");
        }
        return unsafe;
    }

}
```

```
Please input a day and a month to consult the agenda...
19
8
On the 19/8 we have: -45
```

At the execution we can separate those kinds of error detections:

1. Application detection: the application detected that the user entered a wrong input

```
Please input a day and a month to consult the agenda...
16
13
Wrong user input: month must be between 1 and 12
```

2. Interpreter detection: the interpreter detected that the application asked for an illegal execution:

```
Please input a day and a month to consult the agenda...
5
Augustus
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:857)
    at java.util.Scanner.next(Scanner.java:1478)
    at java.util.Scanner.nextInt(Scanner.java:2108)
    at java.util.Scanner.nextInt(Scanner.java:2067)
    at Agenda.main(Agenda.java:12)
```

3. OS detection: the machine detected that the Java interpreter attempted to perform an illegal operation:

```
Please input a day and a month to consult the agenda...
1000000000
5
On the 1000000000/5 we have: #
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0xb6f8b6e6, pid=8515, tid=3061922624
#
# JRE version: 6.0_24-b24
# Java VM: OpenJDK Server VM (20.0-b12 mixed mode linux-x86 )
# Derivative: IcedTea6 1.11.3
# Distribution: Ubuntu 12.04 LTS, package 6b24-1.11.3-1ubuntu0.12.04.1
# Problematic frame:
# V  [libjvm.so+0x6486e6]  Unsafe_GetNativeByte+0x66
```

4. No detection: the application does not crash but an undefined result is perceived:

```
Please input a day and a month to consult the agenda...
33
5
On the 33/5 we have: 0
```

Those executions show how bug reports decline as errors are detected on low abstraction levels. As the abstraction level decreases, the reports have little meaning for the user and even for the programmer. Notice that in Java, as almost all the errors are caught in the two most external layers, we have to exploit very specific unsafe features to make the OS layer detect an error.

The client/server problem The problematic introduced in this section is well known to software engineers and can be formulated this way: "Should I consider the client knows what he is doing?". Often the answer is NO and the developer must test all the client inputs so it can prevent him to blow up the whole server. But when layered architectures are used, the pairs client/server are multiplied and those tests become redundant (Figure 2.3).

Performance cost As we saw, early error detection requires many redundant guards that can slow down the execution. If we analyze the programming language history from Assembly to Java, we see that constant efforts have been made to ease the programming task by providing abstraction and safety. At the beginning of the informatics era, performance was crucial and assembly languages proposed very low abstraction and few guards. Then human time became much more expensive than machine time and new languages privileged programming ease over performance.

2.4 Introduction to type theory

Typing Typing is a strategy to detect a large class of unsound configurations for safety purposes. The concerned configurations are formally defined by the provided guards but generally typing ensures that no operations are performed out of their definition domain. People agree to say typing ensures that executions have a meaning.

Type Typing techniques use types which enumerate a set of values along with operations that can be performed on them. For instance usual Java interpreters reject: `1 + "foo";` because `+` expects two arguments of type number and `"foo"` is of type string. The used type set is very dependent of the typing technique.

Dynamic typing Dynamic typing is a very direct typing technique. Each computed value contains a type tag, dynamic typing raises a type error when a primitive is called with arguments containing wrong type tags. Dynamic languages are examples where the semantic defined by the interpreter is very close to the natural one. In other words, dynamic typings usually does not inhibit the language expressiveness. Drawbacks of dynamic typing include late error detection and low performance.

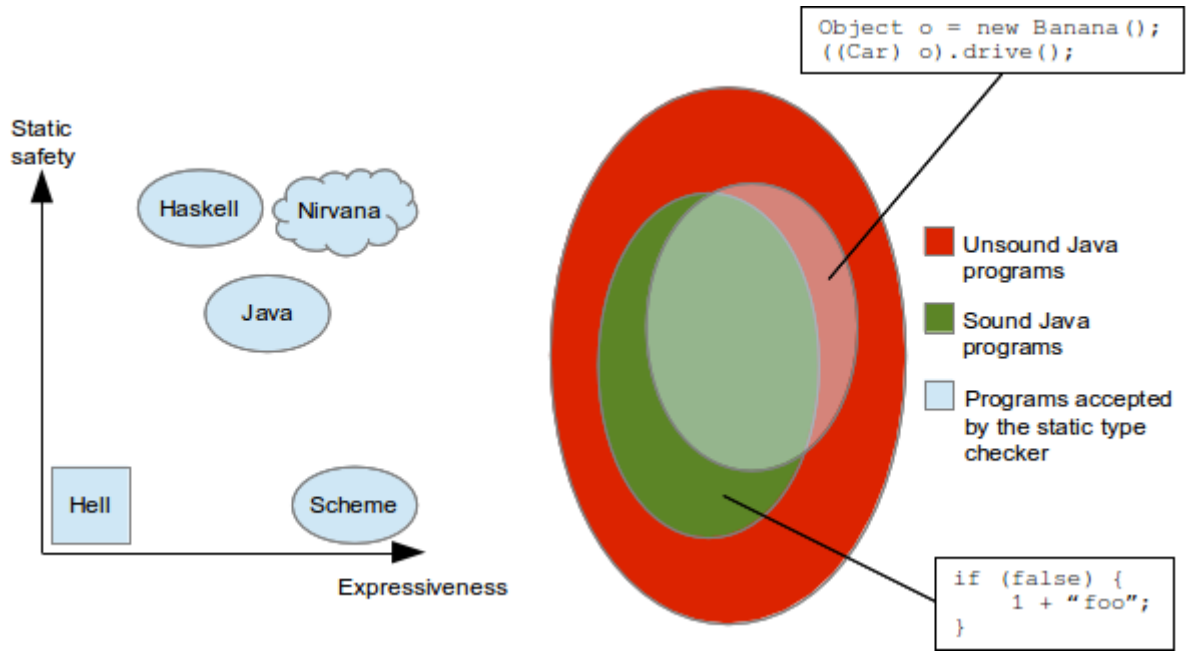

Figure 2.4: A compromise between expressiveness and static safety must be done. Scheme is a dynamic language, therefore it does not reject statically any program and is very expressive. On the opposite, Haskell is known to be very safe. Java is an interesting example of safe language with loopholes that increase expressiveness at the cost of some safety. Finally, the "Nirvana" would be to accept exactly the sound program which is not decidable for Turing-complete languages.

Static typing Static typing does not have to execute programs to detect errors ; based on a static code analysis, static typings are able to detect many errors. By detecting a lot of errors statically, many guards can be removed at the execution without damaging safety. The static type checker divides the language between the accepted programs and rejected programs (Figure 2.4). Statically typed languages are usually characterized by early error detection, high performance but restricted expressiveness.

Type system The type system is the core of the static type checker ; it formally describes the subset of the accepted programs. To be useful in practice, a type system must be decidable ; that means we must be able to separate the accepted program from the rejected one. It is easy to define undecidable type systems, for instance the type system that accepts exactly the sound programs is not decidable for any useful language in practice.

Static safety Static safety is related to the amount of unsound programs rejected during static code analysis. Static safety and safety are two different concepts, dynamic languages are statically unsafe because no program is statically rejected and yet they may be safe by catching all the errors at execution time. As type systems that exactly accept sound programs are not decidable, practical type systems either accept unsound programs or reject sound programs. The first option decreases the static safety and the second decreases the expressiveness (Figure 2.4).

Loopholes In most widely used type systems, some rules are dedicated to by-pass the other rules. In that case, we say that the type system has loopholes. Those holes aim to place the programmer over the static type checker and grant him the last word. Loopholes decrease static safety and increase expressiveness. Loopholes in high level languages like Java usually produce guards at runtime to ensure safety but other ones like C let the programmer execute unsound programs. Loopholes may be very powerful ; for instance, they can be used to express heterogeneous arrays in Java:

```
// Accepted unsound program
Object[] array = new Object[3];
```

```
array[0] = new Car();
array[1] = new Banana();
array[2] = new Car();
Car car = (Car) array[1];       //  Casting <=> Loophole
car.drive();                    // Unsound statement
```

Type annotations Static typing may need informations from the programmer to avoid ambiguous cases. The programmer may or must provide type indications for variables, parameters and expressions. In Java for instance, the annotations concern variables, parameters and return type:

```
public float circumference(Circle c) { // Annotated parameter and return
    float pi = 3.1416;                 // Annotated variable
    return 2 * pi * c.radius();        // Static type checker verify
}                                      // this is indeed a float
```

Type inference During static type checking, expressions are associated to types ; this task is called type inference. Early on, the amount of type annotations was so important that this task was trivial. But nowadays, languages tend to let programmers decide where they want to put type annotations ; this freedom complicates the inference task and sometimes type annotations are still required to avoid ambiguities. For instance in C#, the keyword `var` allows to automatically deduce the type of the initializer expression: `var x = "Yo";` is equivalent to `String x = "Yo";` .

Scripted languages & compiled languages Often, dynamically typed languages are directly executed, then they are called scripting languages. On the other hand, as statically typed programs must be analyzed anyway, they are often compiled before being executed. For instance Java programs are compiled into portable byte codes before execution.

Recapitulation In this section we saw how dynamic and static typing usually catch errors. We also introduced the important concept of static safety which is orthogonal to the expressiveness ; no typing can be entirely statically safe without inhibiting the language expressiveness. Finally some classic concepts in type theory: loopholes, type annotation and type inference.

2.5 Comparison between static and dynamic typing

There is a religious war since decades between the static typing defenders and the supporters of dynamic typing. The first ones appreciate the reliability and the organization of the static programs when the second ones highlight the expressiveness and the flexibility of dynamic languages. Static and dynamic typing have always cohabited (Figure 2.5) but before the explosion of Internet applications, only statically typed languages where widely used in practice. Indeed we will see that dynamic typing fits well the needs of web developers. The rest of this section is a succession of comparisons between static and dynamic typing.

Safety In a static typing, type errors are detected very early in the development process (some development platforms like Eclipse detect type errors as soon as the programmers writes it). When in dynamic typing, type errors are detected at the last possible moment ; type errors may occur during the use of a finished product. That is why statically typed languages are reputed more reliable. As a response, dynamic typing supporters may argue that testing can catch much more errors. But testing is a very expensive verification process that never reaches 100% coverage.

Blame In dynamic languages, type errors may be detected very deep inside a library because a call was made with the wrong types. As a result, the programmer gets a very irrelevant bug report like "true is not a number" spotted in some dark place he never saw. Generally, dynamic typing fails to blame the correct piece of code when a type error occurs. A lot of efforts are being made to get more relevant bug reports on those languages (Chapter 6).

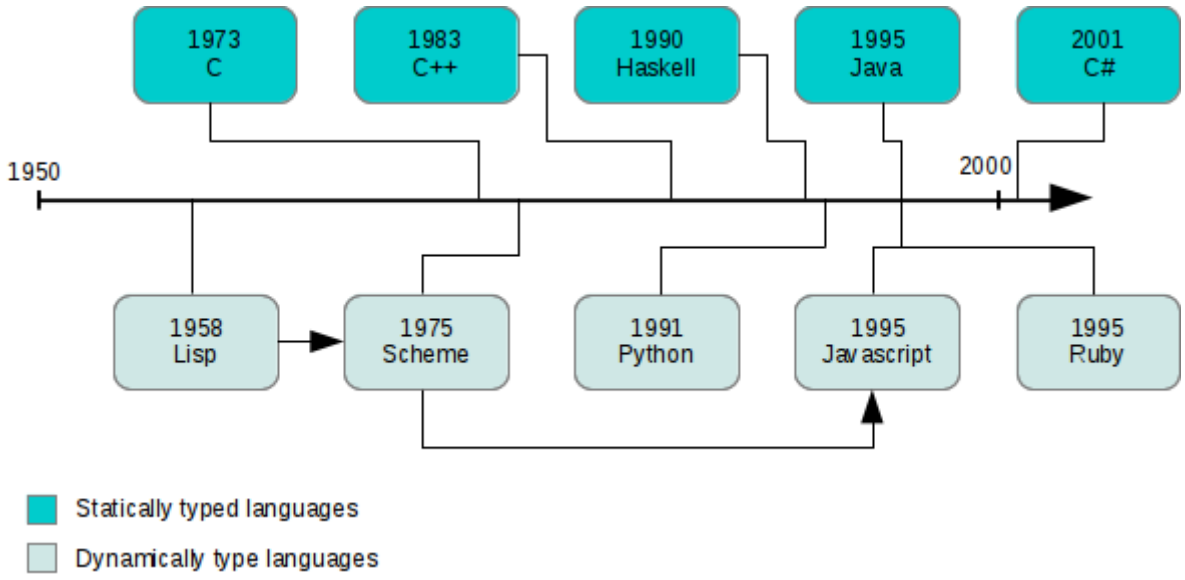

Figure 2.5: Apparition of some programming languages. JavaScript is an adaptation of Scheme which is itself an evolution of Lisp.

Expressiveness As said before, in a Turing-complete language it is impossible to separate the safe programs from the unsafe ones. In practice, no type system accepts all safe programs (the provided guaranties would be too weak) ; so they all have their limitations. Type systems tend to be improved to be able to express more and more concepts. Those refinements are a very difficult task and may easily lead to type systems without enough guards or unusable in practice. An example of such refinement is the genericity introduced in Java in 2004. Indeed before, the Java type system could not express the identity function that just returns its argument and casts had to be used:

```
public static Object identity(Object o) {
    return o;
}

public static void main(String[] args) {
    Banana b = (Banana) identity(new Banana()); // pollution cast, never fails
    b.eat();
    Car c = (Car) identity(new Car());          // pollution cast, never fails
    c.drive();
}
```

As this kind of program has practical interests, the Java type system has been improved with genericity:

```
public static T <T> identity(T t) {
   return t;
}

public static void main(String[] args) {
    identity(new Banana()).eat(); // no pollution cast
    identity(new Car()).drive();  // no pollution cast
}
```

Maintenance The last point can be analyzed from an other point of view: as static type checker forces the programs to be consistent and very organized they are easier to maintain. On the other hand, dynamic typing is so permissive that programs may quickly become inconsistent as the project grows. When a modification has to be done on a static programs, the static type checkers detect most of the spots where the code has to be updated.

	Static typing		Dynamic typing
	Not much inference	Hard inference	
Safety	+ Very early in the development process.		- At the last possible moment.
Blame	+ Relevant bug report.		- Irrelevant bug report spotted on bad places.
Expressiveness	- Limited by the type system.		+ No limitation compared to runtime semantic.
Maintenance (large project)	+ Forces structured and consistent programs.		- Modifications may lead to unsuspected errors.
Flexibility (small & medium project)	- Code updates are very concerned by type system restrictions.	= Some updates are concerned by types system restrictions.	+ Code updates only concern the semantic.
Performance	+ Less type check, efficient representation.		- Type checks at each primitive application, representations must carry types.
Documentation	+ Type signature is a good way to describe a procedure.	= Programmer dependent.	- Only rely on the procedure name.

Table 2.1: Pro and cons of dynamic typing and static typing (with a variable type inference presence). Dynamic typing is better for prototyping and static typing is better for large projects with a long life cycle.

Flexibility Again the last point may be turned around and because of type annotations, the purpose of most of those updates is to satisfy the type system without changing the semantics. So type annotation can be a burden when quick fixes must be done. In response of this argument, most statically typed languages develop type inference to let the programmer decide where he put type annotation.

Performance Because of the assumptions made on the accepted static programs, the static interpreters have to perform less type checks than the interpreters of dynamic languages. Moreover values of a statically typed language can be stored in a more efficient representation (the original motivation to introduce type in C).

Documentation A lot of things can be deduced from the type signature of a procedure ; while dynamic programs can only provide meaningful names as built-in documentation. This drawback can be countered by very complete documentations like in Perl.

Recapitulation In (Table 2.1) we summarize the main advantages and drawbacks of static typing (annotated or not) and dynamic typing. It comes out that static typing is good for large projects with a long life cycle when dynamic typing is good for prototyping. Static typing is used a lot on critical domain when dynamic typing is very used on domains where development speed is a prior concern like Internet applications development. As it appears that neither static typing nor dynamic typing are perfect a lot of investigations are currently performed to mix the best of those two worlds.

2.6 Iterative typing as a compromise

Mixing static and dynamic typing is currently a field heavily investigated. It is about performing as much static type checks as possible and leaving as little runtime checks as possible to ensure safety. By following this direction, we expose here what problem iterative typing attempt to solve. From now on, we suppose the reader is familiar with Scheme ; if not a gentle introduction can be found at (Annexe A)

Dynamic reading A much used primitive in Scheme is `read`, in this thesis we assume `read` may either return a boolean, a number or a string[2]. A common problem Scheme programmers encounter is late type error detections in case the return type of `read` has not been manually checked:

```
(define x (read))

;; Many lines later...

(* 2 pi x)
```

In the above code, if the `read` call did not return a number, the program raises an error at the last statement. We want to detect this error earlier, ideally right at `read` call, without reducing the expressiveness too much.

Dynamic conditional structure When a conditional structure modifies the type deduction depending on the branch actually executed, we say it is dynamic. Such dynamic behavior may have two aspects:

1. *Uncertain return*: depending on the executed branch, the evaluation of a `if` expression may return values of different types. The below Scheme program returns either an error message or the division result:

   ```
   (define (divide x y)
     (if (= y 0)
         "division by zero" ;; a string is returned
         (/ x y)))          ;; a number is returned
   ```

   The first branch returns a string and the second returns a number.

2. *Uncertain use*: depending on the executed branch, the use of variable is modified. The Scheme program below returns a convenient description of a `divide` result:

   ```
   (define (main x)
     (if (string? x)
         (concat "Error: " x)                    ;; x is used as a string
         (concat "Result: " (number->string x)))) ;; x is used as a number
   ```

   The first branch uses the given argument, `x` as a string and the second branch uses `x` as a number.

What static type systems do concerning conditional structure is assume both branches will be executed and so impose that they are type compatible. As a result static type system systematically reject dynamic conditional structure. We want to support such conditional structures and still be able to detect errors early.

Improve expressiveness The biggest drawback of static type systems is the expressiveness inhibition. A lot of investigations have been done to increase the expressiveness of statically typed language. Those efforts can be split into two directions:

1. *Pure static typing*: Complicate type systems to grab as many safe programs as possible. The Haskell type system is a successful result of this direction: it is believed safe and yet very expressive.

2. *Loopholes*: Intentionally introduce loopholes in the type system. Those loopholes decrease the static safety and require additional guards at runtime to ensure safety. Loopholes are the usual way to mix static and dynamic typing. This direction is currently under high investigation which has led to the creation of new typings like soft and gradual typings (Chapter 6).

Iterative typing The solution presented in this thesis, called iterative typing, follows the loophole direction. Iterative typing recognizes there are dynamic events, like `read` calls and dynamic conditional structures, that require runtime information to be analyzed precisely. Those events are accepted thanks to loopholes inside the type system and will generate annotations so that type checking can be fulfilled at runtime when more information are available.

[2]The read procedure actually return a s-expression but simple constants suffice to expose the problem we attempt to resolve.

2.7 Conclusion

In this chapter we exposed some of the most important concept of type theory. In particular we saw that it is not possible to design a type system that accepts exactly the sounds programs ; in practice, type systems must do a compromise between static safety and expressiveness. We then highlight the gap between static and dynamic typing and argued that errors in some very dynamic patterns could be detected much earlier than dynamic typed languages usually do.

Chapter 3

Approach & Implementation

This chapter is a gentle introduction to iterative typing. After this chapter, the reader should know what iterative typing is capable of and have some intuition about how iterative typing is implemented. In the first section we define an ideal iterative typing for Scheme without polymorphism or recursion. On the next section we expose the whole iterative framework as an approximation of the iterative ideal. The third section exposes how the core of iterative typing has been extended to support polymorphism. The next section is a discussion about the implementation of recursions in iterative typing. The chapter ends with the presentation of some features specific to iterative typing.

3.1 Iterative typing's ambition

In this section we reason about a non recursive subset of Scheme. This assumption will allow us to define an ideal algorithm for the iterative typing. This algorithm would never reject a rightful execution and would catch all the unsound combinations of program and input as soon as possible. This ideal motivates the whole iterative type system, it is the core idea of this thesis.

Inputs as a trace A trace is a sequence that represents all the inputs a program will receive. Without recursion, a Scheme program reads its input incrementally and we can assume the amount of required inputs is finite. As an example we present a simple program, its execution and the associate trace:

```
(display "What is your name?\n")
(define name (read))
(display "What is your age?\n")
(define age (read))
(display "Where do you live?\n")
(define place (read))
(define majority 18)
(if (< age majority)
    (display "You are too young...\n")
    (display (concat "Hello " name " of " place "!\n"))
;; Normally concat have an arity of 2 but for clarity reasons we use multiple arguments.
```

```
What is your name?
"Laurent"
What is your age?
22
Where do you live?
"Brussels"
Hello Laurent of Brussels!
```

The program asks the user some information and then displays a personalized message. In the above execution, the input trace is: `("Laurent" 22 "Brussels")` ; `"Laurent"` has been stored into the variable `name` , `22` into `age` and `"Brussels"` into `place` . This example will be used throughout the rest of this section and will be referred as the "name-age-place" program.

Criterion	Iterative ideal
Safety	All the errors are detected.
Expressiveness	No sound execution is rejected.
Error detection	As soon as possible with relevant bug report.
Peformance	The computation is probably very high due to the amount of trace to analyze.

Table 3.1: The iterative ideal maximize three out of four criteria concerning type systems.

Sound/Unsound traces An input trace is sound relatively to a program when the natural semantic does not return an error if the program is executed with that trace. Here, the natural semantic is given by a standard Scheme interpreter. On the other hand, unsound traces combined with the associated program cause the Scheme interpreter to raise an error. Below we present an unsound and a sound trace relatively to the name-age-place program with their associated execution:

1. The trace `(1234 22 "Brussels")` is unsound because `name` is used as a string in that execution:

```
What is your name?
1234
What is your age?
22
Where do you live?
"Brussels"
ERROR: concat: expects type <string> as 2nd argument,
       given: 1234; other arguments were: "Hello "
```

2. The trace `(1234 17 "Brussels")` is sound because the taken path (the else branch) does not force the name to be a string:

```
What is your name?
1234
What is your age?
17
Where do you live?
"Brussels"
You are too young...
```

The iterative ideal As the execution goes on, the head of the input trace is specified which restricts the number of traces that remain possible (Figure 3.1). If all the remaining traces are unsound, the ideal iterative would throw an error. So by definition, the iterative ideal throws an error as soon as possible, way before illegal operations are actually performed. This is the aim of iterative typing and the core idea of this thesis. The iterative ideal maximizes three of the four main criteria concerning type systems, the only drawback being the performance cost (Table 3.1). Indeed the amount of traces to analyze makes the iterative ideal unusable in practice. Lets illustrate this ideal with two examples concerning the "name-age-place" program:

- There is no sound trace that starts by `(1234 22)`. So if `1234` is entered for the name and `22` for the age, the ideal iterative would immediately throw an error with the following message: "If the first input is not a string then the second input must be a number smaller than 18".
- Again, the specification `("Laurent" "boum")` discriminates any sound trace. The iterative ideal would throw an error with the message: "The second input must be a number."

Computability of the ideal We give two direction to implement the iterative ideal:

- *Exhaustive testing*: The iterative ideal can be achieved by executing the program on each different trace. Obviously, the set of trace to analyze is way to large and this solution is not feasible.

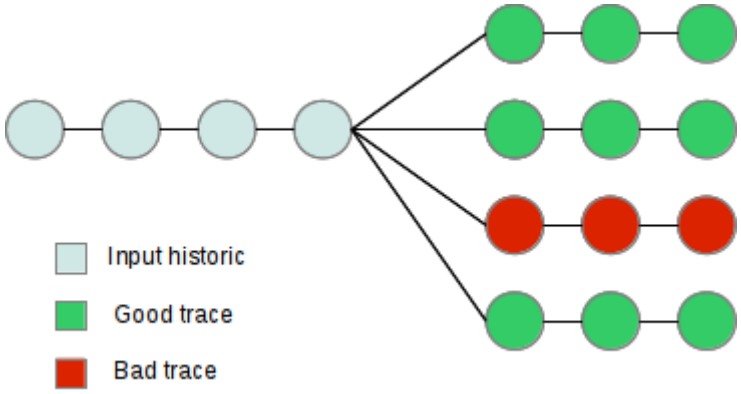

Figure 3.1: The available trace during an execution. As soon as no sound trace remains, the ideal iterative typing would throw an error.

Trace	Soundness
$(String, Boolean, String, then)$	Unsound trace: the second input must be a number.
$(String, Number, String, then)$	Sound trace.
$(Number, Number, String, else)$	Unsound trace: if the first input is not a string, the "then" branch must be executed.
$(Number, Number, String, then)$	Sound trace.

Table 3.2: Soundness of several traces for the "name-age-place" program.

- *Constraints on trace*: By performing a static code analysis we can deduce constraints that defines the set of sound traces. For the name-age-place program those constraints for any trace $(name, age, place)$ would be:

$$\begin{cases} typeOf(age) = Number \\ age > 18 \Rightarrow (typeOf(name) = typeOf(place) = String) \end{cases}$$

  Demonstrations that mix values and types to prove the safety of programs is the domain of dependent typing. Dependent typing is a very powerful tool but requires the programmer's help to provide those safety proofs which does not fit our problem statement[13].

So the first option is not feasible and the second is too complicated to use. At that point we, naturally conclude that some errors cannot be detected by iterative typing and reduce the working precision from values to types.

The type approximation In the type approximation, we work only with types instead of precise values. In consequence, the input traces become lists of types instead of lists of values. The drawback is that we do not have access to all values anymore. This is a problem when types depend on values ; the `if` statement can induce such dependency: for instance, the type of `(if x 1 "foo")` can, depending on the value of `x` , be either a number or a string. In consequence, we are forced to consider conditional structures as dynamic events just like inputs. For the "name-age-place" program, the trace $(String, Number, String, else)$ is sound and means that: a string was first entered, the second input was a number, the last input was a string and the else branch of the conditional structure was evaluated. We give some other examples in (Table 3.2).

Value errors The division by zero error is an example of value errors. The division operator is only defined for a subset of the number set, we say that zero is a gap in its definition. Generally value errors arise when an operator has an undefined behavior even when the given arguments have the rightful types.

An other common example of value error is the null pointer exception: the Scheme statement `(car null)` is type safe but unsound because the `car` operation is not defined for the null list. By working with type instead of values we can not detect value errors anymore.

Iterative typing Even if the type approximation considerably reduces the amount of traces to analyze, their number still exponentially grows with the amount of inputs and conditional structures. Exponential computation time algorithm are not practically acceptable and so iterative typing uses a system of constraint on traces instead of an exhaustive research. For the name-age-place programs, those constraints on the trace $(name, age, place, branch)$ would have been:

$$\begin{cases} age = Number \\ (branch = else) \Rightarrow (name = place = String) \end{cases}$$

Conclusion Typically, when static typings encounter an uncertainty, they make assumptions to carry on the static analysis. Those assumptions inhibit the expressiveness of the language and iterative typing drops constraints inside the code instead. Those annotation will be used at the execution to fulfill the type checking when more information will be available. By doing this, iterative typing can handle very dynamic code like the name-age-place program and is still able to detect error earlier than dynamic typing on more relevant place. The advantage of iterative typing is highlighted below by two executions of the "name-age-place" program:

1. Iterative typing accepts more sound executions than static typing:

   - Static execution:

```
Inter >> What is your name?
"Laurent"
What is your age?
17
Where do you live?
1234
TYPE ERROR: expects a string
```

   - Iterative execution:

```
Inter >> What is your name?
"Laurent"
What is your age?
17
Where do you live?
1234
You are too young...
```

2. Iterative typing detects type error sooner than dynamic typing on more relevant places:

   - Dynamic execution:

```
What is your name?
"Laurent"
What is your age?
"boum"
Where do you live?
"Brussels"
TYPE ERROR: > expects two arguments of type number give a string and a number
```

   - Iterative execution:

```
What is your name?
"Laurent"
What is your age?
"boum"
TYPE ERROR: expects a number
```

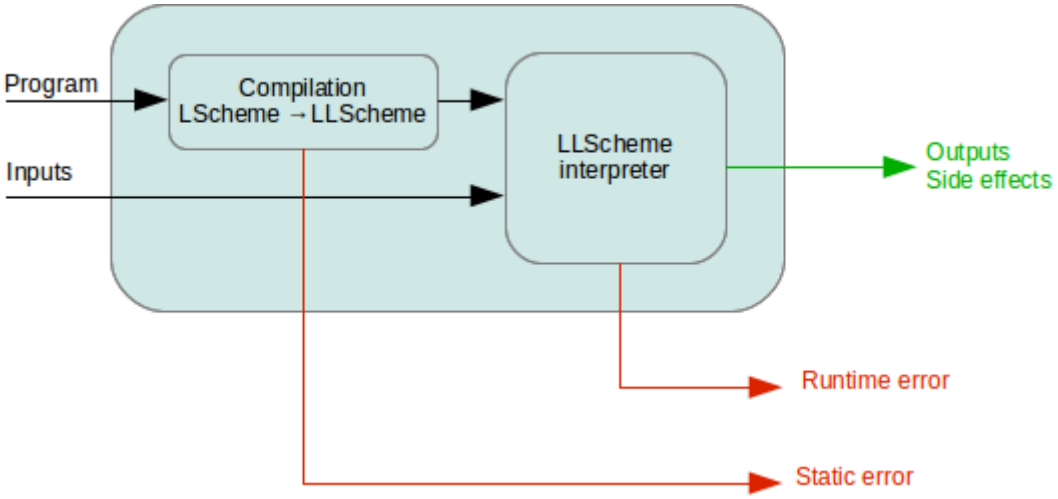

Figure 3.2: The atomic Scheme subset is compiled before being executed.

3.2 The iterative framework

Now we know more precisely what iterative typing attempts to achieve, we will present informally how it is implemented. Iterative typing performs a static typing that does not make any assumption when a dynamic event occurs (Figure 3.2). The static type checker performs as many checks as possible and puts annotations inside the code where a dynamic event occurs.

LScheme LScheme is a Scheme-like language that we will evaluate with an iterative interpreter. Beside continuation, LScheme supports all the Scheme core features. LScheme also includes some special forms specific to iterative typing that we will see further.

LLScheme LLScheme is the annotated version of LScheme that contains the constraints exposed before. Those constraints have the form `a => t` which means the type variable `a` is instantiated to `t`. The two kind of dynamic events that currently require annotations are:

1. *Input procedure*: The primitive `read` asks the user to enter some data, the returned constant can be of type boolean, number or string. The annotated call to `read` has the following form: `(sub [in => t]read)` which means that the input is expected to be of type `t`. Here is an example:

```
(define x ((sub [in => a] read)))       ; the input can be any constant
(define y ((sub [in => Number] read))) ; the input must be a number
(+ y 1)
```

2. *Conditional structure*: Each branch of the `if` expression contains an annotation of the form: `[cond => t, a1 => t1, ...]` which means that the type variable `cond` is instantiated to `t`, `a1` is instantiated to `t1`, etc. Usually, the type variable `cond` denotes the type of the whole conditional structure. The `if` statement is dynamic in two different ways:

   (a) The `if` statement can return two different types:

```
(if x
    [cond => Number] 1       ; the if returns a number
    [cond => String] "foo") ; the if returns a String
```

   (b) Depending on the taken branch, the type deduction differs:

```
(define x (({in => a} read)))                              ; x can be anything
(define y (({in => b} read)))
(if y
    [a => String, cond => String] (concat "yo" x) "ok" ; x must be a string
    [a => Number, cond => String] (+ 1 x) "ok")        ; x must be a number
```

Original constraints	Unified constraints

`a => d -> c`
`b => [c]`
`c => String`
	`a => d -> String`
	`b => [String]`
	`c => String`

`a -> b => String -> (c -> Boolean)`
`[d] => c`
`e -> c = Number -> [Number]`
	`a => String`
	`b => Number -> Boolean`
	`c => [Number]`
	`d => Number`
	`e => Number`

`a => b -> [Boolean]` `a -> Number => (Number -> [String]) -> Number)`	Failure: incompatible type `Boolean => String`
`a => b -> Boolean` `b => [b]`	Failure: infinite type on `a => [a] -> Boolean`

Table 3.3: On the left: the original constraints, on the right the resolved ones.

Compilation LScheme → LLScheme Though it is possible, the programmer is not supposed to write LLScheme annotations himself. Instead the iterative framework embeds a compiler that compiles LScheme to LLScheme by automatically inserting rightful annotations. This compilation is based on type inference and will be detailed in (Chapter 5).

Locus A locus is nothing more than the union of all the constraints generated during the execution by the previous dynamic events. Loci are used to keep track of the previous dynamic events and detect incompatibilities between constraints. The incorporation of new constraints inside the current locus is done via the unification algorithm [4]. We will not describe this algorithm here ; instead, we give some examples at (Table 3.3).

Interpretation of LLScheme The execution always starts with the empty locus which obviously contains no constraint. Then the locus is carried around and affected by dynamic events. This suggests that the locus follows exactly the execution flow and that compound procedures must manipulate the locus present during the application (Figure 3.3). Below we explain how the annotations of `read` and `if` affect the current locus:

- *Input procedure*: Given the annotated reading procedure: `(sub [in => a]read)` , if the input type was `b` the constraint `a => b` will be incorporated to the current locus.
- *Conditional structure*: The constraints added when a `if` statement is reached correspond to the constraints stored inside the executed branch. So when `(if x [a => String]"foo" [a => Number]1)` is reached, depending on the value of `x` , either `a => String` or `a => Number` will be incorporated.

The "name-age-place" example It is easy to insert the correct annotations inside the "name-age-place" program. Below we expose the annotated program and on (Figure 3.4) we graphically expose a possible execution.

```
(display "What is your name?\n")
(define name ((sub [in => a] read)))            ; No constraint on the first input
(display "What is your age?\n")
(define age ((sub [in => Number] read)))        ; Second input must be a number
(display "Where do you live?\n")
(define place ((sub [in => b] read)))           ; No constraint on the third input
(define majority 18)
(if (< age majority)
    ; The first and third input must be strings
    [a => String, b => String] (display (concat "Hello " name " of " place "!\n"))
     ; No constraint on the else branch
    [] (display "You are too young...\n"))
```

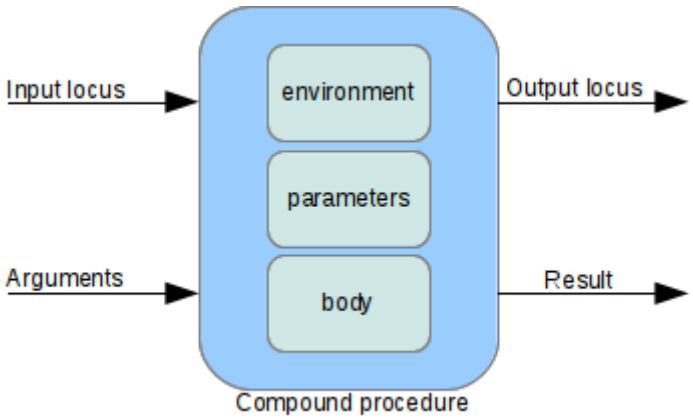

Figure 3.3: In iterative typing, procedures not only affect values but also the current locus. Unlike environments, loci are not encapsulated in compound procedures ; instead, it is the locus present at the application that is used.

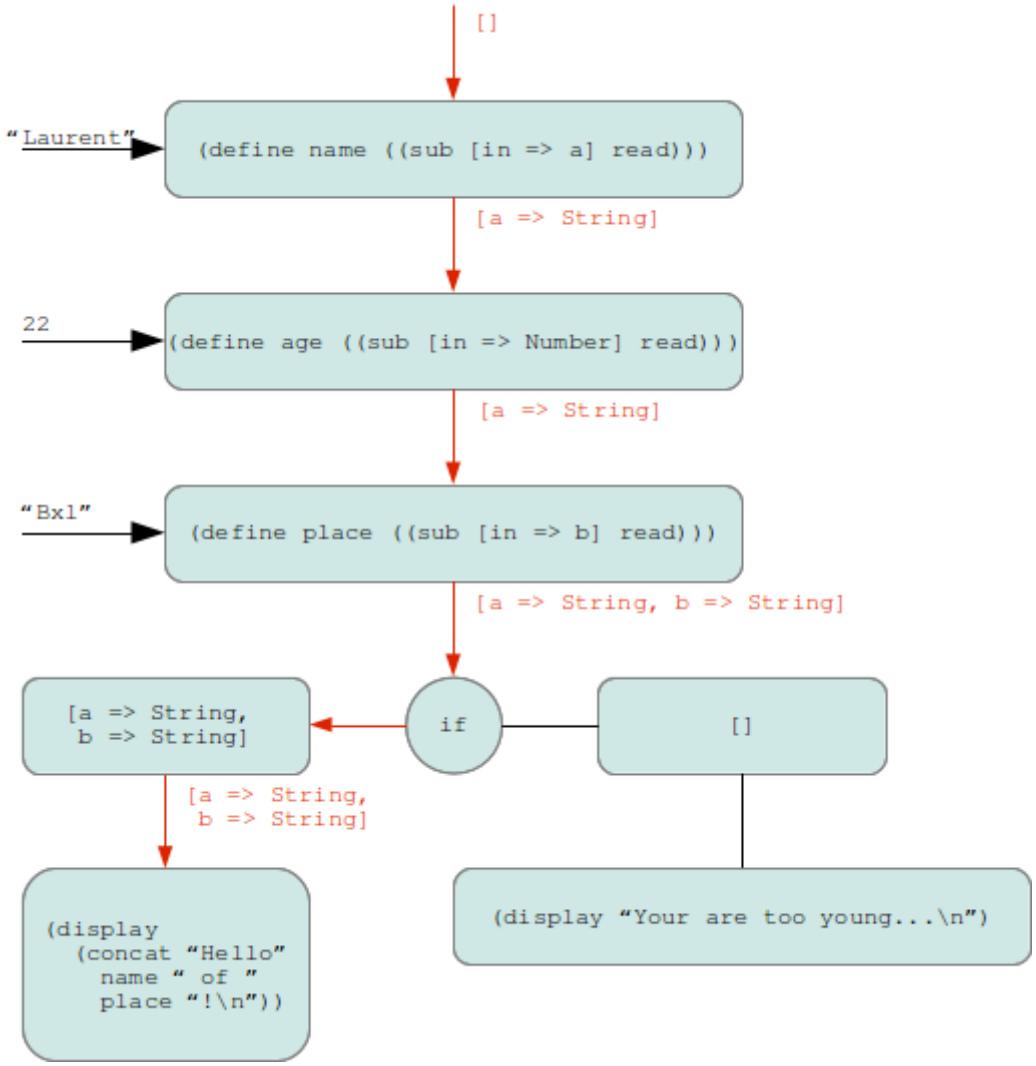

Figure 3.4: The evolution of the locus through the execution of the "name-age-place" program with the inputs `"Laurent"` , `22` and `"Bxl"`.

3.3 LScheme & polymorphism

The identity procedure A procedure is said polymorphic when it can be called with different types. For instance, in Scheme, the primitives that manipulate lists are polymorphic ; both `(cons 1 null)` and `(cons "foo" null)` are accepted. In dynamic typing, polymorphism support is automatically granted since the arguments of compound procedure are not type checks. On the other hand it is not trivial to support polymorphism in statically typed language. If Scheme did not support polymorphism, the following program would be rejected:

```
(define identity (lambda (x) x))
(identity 1)
(identity "foo") ; Static type error: Number cannot be converted to String
```

When it has a sound Scheme semantic:

```
1
"foo"
```

The map procedure While Scheme does not enforce it, programmers typically make and use homogeneous lists. Homogeneous lists are lists where each element has the same type. The map function is an example of a procedure that manipulates such lists:

```
(define map (lambda (f xs)
              (if (null? xs)
                  null
                  (cons (f (car xs)) (map f (cdr xs))))))

(define incr (lambda (x) (+ x 1)))
(define concat-hello (lambda (x) (string-append "hello " x)))

(map incr (cons 1 (cons 2 null)))                  ; map called with a list of number
(map concat-hello (cons "foo" (cons "bar" null))) ; map called with a list of string
```

Again, this program would fail without polymorphic support when it has a sound semantic:

```
(2 3)
("hello foo" "hello bar")
```

What comes out of those examples is that Scheme without polymorphism is not Scheme anymore. Polymorphism is a feature in static typing that can not be by-passed nowadays and therefore must be supported by iterative typing.

The polymorphic wrapper To simulate polymorphism the current iterative prototype uses a wrapper around monomorphic procedures (Figure 3.5). Essentially, this wrapper adds constraints before the monomorphic application and then releases them to allow a further call with different types. Practically polymorphic procedures are defined on `define` statement but the wrapper is created at each time the polymorphic procedure is referenced by an identifier.

Polymorphic calls All the information required by the wrapper is stored where the polymorphic procedure is pointed by an identifier. Indeed, each identifier that refers to a polymorphic value is annotated in LScheme. It is important to notice that those polymorphic annotations are safe and cannot transform the current locus into an incompatible set of constraints. So basically, type errors cannot appear at polymorphic calls and therefore polymorphic calls should not be considered as dynamic events. This behavior is in accordance with the iterative philosophy where type errors are spotted only where ambiguities arise.

A polymorphic example Here is an example that illustrates how the polymorphic wrapper works.

```
(define (evil x)
  (if x
      1
      "fool"))
(+ 1 (evil #t))
(concat "yo " (evil #f))
```

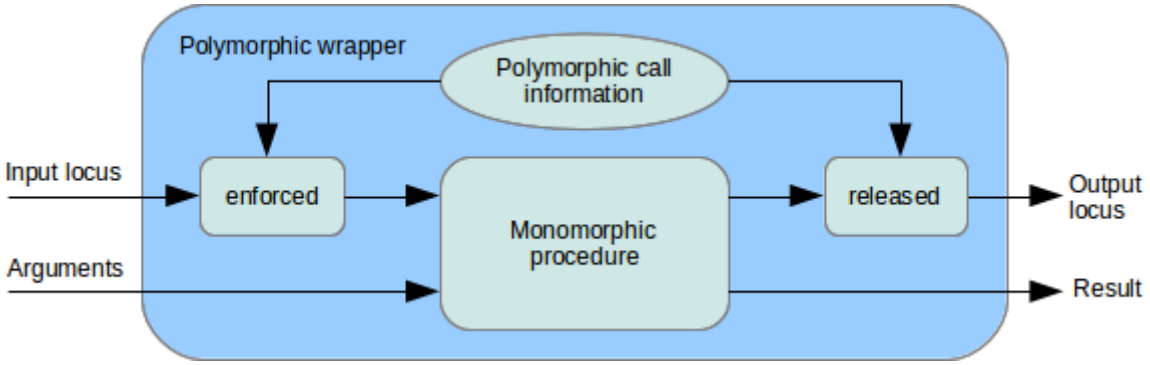

Figure 3.5: Polymorphism is simulated by a wrapper that modifies the locus before and after the monomorphic call.

The above program can be annotated into:

```
(define (evil x)
  (if x
      [a => Number] 1
      [a => String] "fool"))
(+ 1 ((sub [a => Number] evil) #t))
(concat "yo " ((sub [a => String] evil) #f))
```

So when the statement `(+ 1 ((sub [a => Number]evil) #t))` is reached, the current locus is empty and the constraint `a => Number` is added. Then we proceed to a monomorphic call of the compound procedure `evil`. The `then` branch being taken, the constraint `a => Number` is added and the current locus is still `a => Number`. Then when the monomorphic call returns we release the constraint `a => Number` and we get the empty locus again. The current locus being empty, we can evaluate the second call to `evil` without restriction.

Read as a polymorphic procedure Actually `(sub {in => t} read)` is not a special case but just a call to a polymorphic procedure. The `read` polymorphic procedure can indeed be implemented with primitive procedure `read-line` that returns the string the user just input:

```
(define (read)
  (let (input (read-line))
    (cond ([in => String]  (input-string? input)  (input->string input))
          ([in => Number]  (input-number? input)  (input->number input))
          ([in => Boolean] (input-boolean? input) (input->boolean input))
          ([in => void]     else                  (error "unknown type")))))
```

3.4 LScheme & recursion

Implementation Scheme provides many ways to create recursive processes ; one of them is based on the `define` special form. Indeed, in the definition `(define <name> (lambda (<params>) <body>))`, the identifier `<name>` can be used inside `<body>`. This is the mean we chose to support recursion in LScheme. To achieve recursion we need the environment inside the compound procedure to contain binding of `<name>` that points to the whole structure (Figure 3.6). Such implementation is the usual way to implement recursion and is not problematic when combined with iterative typing.

Monomorphic recursion So the statement `(define <name> (lambda (<params>) <body>))` creates a compound procedure that appears monomorphically inside `<body>` but polymorphically in the rest of the program. Such recursion is called monomorphic and the types that appear inside the body do not change during the whole recursion. The map function is an example of monomorphic recursion. Monomorphic recursion is powerful enough to fit the programmer's everyday needs and that is why recursion is almost always monomorphic in statically typed languages.

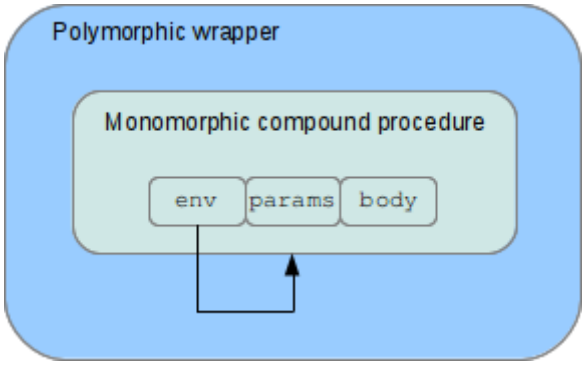

Figure 3.6: The environment of a recursive compound procedure contains a binding to itself.

Monomorphic recursion & iterative typing However, monomorphic recursion has particular consequence in iterative typing. Because the type that does not affect the outside world is fixed as well, some unfortunate restrictions appear. Lets present a simple loop that asks for an input and then displays it:

```
(define forever (lambda ()
                  (define x ((sub [in => a] read)))
                  ((sub [out => String] display) "input: ")
                  ((sub [out => a] display) x)
                  ((sub [out => String] display) "\n")
                  (forever)))
(forever)
```

Because the type of the input, `a` in the above program, has no consequence at all for the rest of the recursion, we naturally expect that the type of the input can be modified from one recursive call to an other. Well, this is currently not the case in iterative typing ; all the types involved in the recursion are fixed. As a result, the first input fixes the type of all the future inputs ; as soon as the user enters a value of a different type, an error is raised:

```
 1
input: 1
2
input: 2
3
input: 3
"BOUM"
TYPE ERROR
```

3.5 Iterative special forms

Until now, LScheme was nothing more than a subset of Scheme. Here we introduce three special forms that are specific to iterative typing. The first two special forms `if-static` and `unsafe` do not actually increase the Scheme semantic and just concern error detections. However, the last special form, `if-viable` is not expressible in Scheme and must be used very carefully.

Static conditional structure Iterative typing grants the programmer the power to choose whether a conditional structure should be statically checked or dynamically checked, as it was always the case so far. Static conditional structures, denoted by the special form `(if-static <pred> <then> <else>)` , are analyzed in the same way as pure static typings usually do. In particular, the branches must be type-compatible and no annotations are inserted during the compilation from LScheme to LLScheme. Programs that contain no dynamic conditional structure are statically type safe and although annotations on polymorphic calls still remain, the constraint system never raises an error. For instance, if we implement the classic conditional structure that returns two different types following the executed branch with `if-static` , then an error will be raised at compile time:

```
(define (evil x)
  (if-static x 1 "foo"))
```

```
Type error: Number is not compatible with String.
```

Unsafe execution By using the special form `(unsafe <body>)`, the programmer is able to perform an unsafe executions of the statements contained in `<body>`. More specifically, if a constraint incompatibility is detected inside `<body>` the iterative constraint systems is shutdown and the execution becomes unsafe.

```
(unsafe
 (define x ((sub [in => Number] read)))
 ((sub [out => String] display) "Still running...")
 (+ 1 x))
```

```
>> "bar"
Still running...
+: expects type <number> as 2nd argument, given: "bar"; other arguments were: 1
```

Without the wrapping inside `unsafe`, the above program would predict the error right inside the `read` procedure. The unsafe special form allows to execute the program until errors actually happen.

Viability test The special form `(if-viable <attempt> <else>)` allow to test if the executing `<attempt>` with the current dynamic information is safe. In other words, `<attempt>` is executed only if iterative typing does not predict an error for that execution. As this special form can not be expressed in Scheme it must be used with caution. Below we present a LLScheme program that uses the viability test:

```
(define x ((sub [in => a] read)))
(if-viable
 [a => Number] (+ x 1)
 [] "I cannot increment that shit...")
```

```
>> "foo"
I cannot increment that shit...
```

3.6 Recapitulation

We opened this chapter with the presentation of an algorithm that would produce an ideal behavior for iterative typing. Unfortunately we concluded such algorithm is not feasible and we introduced the real iterative behavior as an approximation of this ideal. We then presented the whole iterative framework and how it delays some static checks for runtime. Such delaying operates this way: annotations are inserted inside the code during a static code analysis ; at the executions those annotations generate constraints that are stored and carried through the computation flow. Afterwards, we exposed how iterative typing supports polymorphism by adding a wrapper around monomorphic procedures. The next section was a discussion about how monomorphic recursion can be restrictive for iterative typing. We closed the chapter by the presentation of some special forms specific to iterative typing.

Chapter 4

Evaluation

In this section we propose three case studies of practical interest that highlight the strengths and limits of iterative typing. The first case concerns reading procedure and deserialization patterns ; this application of iterative typing is the most direct one. The second case concerns exceptions management and how iterative typing could improve it. The last case is a discussion about early error detection and side effects.

4.1 Serialization

Introduction Serialization is a typical example where types are affected by dynamic information, which is a scenarop iterative typing is good at. Serialization is the process to convert a data structure into a format that can be stored or sent. For instance, serialization is much used on the Internet as a way to exchange informations. In this section we will focus on the reverse process which is called deserialization. Deserialization takes a byte sequence and returns something whose type depends on the input string value. For instance the byte sequence `"12"` will probably be deserialized into the integer `12` .

Element serializations Serialization may become a complicated process as pointers are used. Those references can be represented by an object graph (Figure 4.1). The serialization algorithm must then be processed on the whole graph and must be able to handle cycles and aliases. This complexity explains why most programming languages proposes facilities for serialization. For instance Scheme programmers use the `serializable-struct` keyword to define serializable structures when Java programmers use the `Serializable` interface. Below, we expose a list of students represented by two widely used formats:

1. XML which stands for "eXtensible Markup Language":

```
<students>
    <student>
        <name>Mehdi</name>
        <lastName>Benaissa</lastName>
    </student>
    <student>
        <name>Anna</name>
        <lastName>Smith</lastName>
    </student>
    <student>
        <name>Peter</name>
        <lastName>Jones</lastName>
    </student>
</students>
```

2. JSON wich stands for "JavaScript Object Notation" is a subset of JavaScript:

```
{
"students": [
{ "firstName":"Mehdi" , "lastName":"Benaissa" },
```

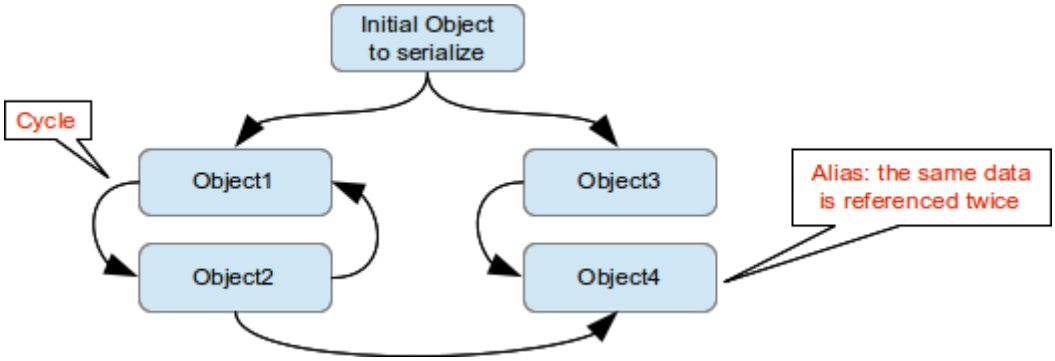

Figure 4.1: An object graph that contains a cycle and a alias. Cycles must be detected to prevent the serialization algorithm from looping ; aliases must be detected to prevent duplication of the same object.

```
{ "firstName":"Anna" , "lastName":"Smith" },
{ "firstName":"Peter" , "lastName":"Jones" }
]
}
```

Dynamic reading Dynamically typed languages do not mandate checks for data integrity before use. Typically, dynamic programmers try to apply methods and are satisfied when the program does not crash ; that is the duck typing philosophy: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck[1].". Below we propose a Python code that creates two classes `Duck` and `Person` that share some methods in common:

```
class Duck:
    def walk(self):
        print "Walk like a duck."
    def swim(self):
        print "Swim like a duck."
    def quack(self):
        print "Quaaaaaack!!!"

class Person:
    def walk(self):
        print "Walk like a baoss!"
    def swim(self):
        print "Swim like a baoss!"

duck = deserialize()
duck.walk()
duck.swim()
duck.quack()
```

In the case `deserialize()` returned an instance of the `Person` class, we get a very late error detection:

```
Walk like a baoss!
Swim like a baoss!
AttributeError: Person instance has no attribute 'quack'
```

The drawback of duck typing is the late error detection when data type are confused. For instance, lets say that a biologist reads gigabytes of data and assumes he received back a DNA object when he actually got a RNA object. Because those objects are very similar, he may use the RNA as a DNA for a while in a lot of very expansive computations without crashing the application. So this type error may be detected very late which will result in a great time loss for our biologist. Or even worse, the type error may not be detected at all leading to erroneous results.

[1] James Whitcomb Riley

Static reading On the other hand, most static programming patterns force to check data integrity as soon as possible. Those patterns usually catch erroneous input right after the deserialization. Static reading is much safer than dynamic reading but it supposes the programmer is aware of the type system and wants to spend time coding data integrity checks which may not be the case for our stressed biologist. In Java, objects can be serialized when they implement the `Serializable` interface. In that case, we can serialize and deserialize objects using respectively the methods `readObject` and `writeObject` on an `ObjectInputStream` instance. The result of `readObject` is an object and it needs to be cast before use. Below, we attempt to deserialize a banana object from a file named `banana.ser` for eating purposes:

```
try {
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream("banana.ser"));
    Object o = ois.readObject();
    Banana b = (Banana) o;
    b.eat();
    ois.close();
} catch(Exception e) {
    e.printStackTrace();
}
```

In case a `Banana` has not been deserialized, the casts `(Banana) o` would throw a runtime cast exception. By testing data integrity right after deserialization, we can avoid needless computations.

Context-aware reading If the reading method is aware of the type the invoker expects, type errors can be detected even during the deserialization. In that case the method can perform a sort of lazy reading where type related information is loaded before the whole parsing. For instance, lets suppose a biologist uses the following Scheme library to work with DNA and RNA objects:

```
;; DNA constructors
(define (DNA-from-sequence seq) ...)
(define (DNA-from-input in) ...)

;; RNA constructors
(define (RNA-from-sequence seq) ...)
(define (RNA-from-input in) ...)

;; tells if the input represents a DNA or a RNA
(define (read-type input) ...)

;; reads the input ; no guarantee on the result type
(define (unsafe-read input)
  (let ((type read-type))
    (cond ((equal? type DNA) (DNA-from-input input))
          ((equal? type RNA) (RNA-from-input input))
          (else (error "unknown type")))))

;; read the input ; if the call succeeds, the result is of type <expected-type>
(define (safe-read input expected-type)
  (if (equal? (read-type input) expected-type)
      (unsafe-read input)
      (error "the input has the wrong type")))

;; complement makes sens for DNAs and RNAs
(define (complement x) ...)

;; find-sequence makes sens only if x and y are both DNA or bot RNA
(define (contain? x y) ...)
```

Type information inside the serialization format need to be quickly accessible in order to decrease the computation time of `read-type`. The following XML sample `<DNA>AGCTTT...GG</DNA>` is a reasonable format. This library proposes two ways to deserialize DNA/RNA structures:

1. *Unsafe way*: potential type errors are detected after a lot of unnecesarry computations:

```
(define (unsafe-computation input)
  (let ((x (unsafe-read input)))              ;; no guarantee about the type of x
    (complement x)                            ;; expensive computation
```

```
    ...                                              ;; expensive computations
    (contain? x (DNA-from-sequence "AAGTTC")))       ;; unsafe call, x may be an RNA
  "end")
```

2. *Context-aware way*: potential type errors are detected as soon as possible inside the reading method:

```
(define (safe-computation input)
  (let ((x (safe-read input DNA)))                   ;; x is guaranteed to be a DNA
    (complement x)                                   ;; expensive computation
    ...                                              ;; expensive computations
    (contain? x (DNA-from-sequence "AAGTTC")))       ;; safe call, x is a DNA
  "end")
```

Iterative reading Using iterative typing, the unsafe use is actually equivalent to the safe use ; guards are automatically inserted. Indeed iterative typing can deduce that the second branch of `unsafe-read` is not viable inside the polymorphic call `(let ((x (unsafe-read input))) ...)` (Figure 4.2).

1. From the body of `DNA-from-sequence` , its type is inferred to $String \rightarrow DNA$.
2. (1) implies that `(DNA-from-sequence "AAGTTC")` is of type DNA.
3. From the body of `contain?` , its type is inferred to $\forall a.(a, a) \rightarrow Boolean$.
4. (2) and (3) imply that the variable `x` must also contain a value of type DNA.
5. (4) implies that `(unsafe-read input)` must be of type DNA.
6. From the body of `RNA-from-input` , its type is inferred to $String \rightarrow RNA$.
7. (5) and (6) implies that the second conditional branch is not viable because it returns a RNA when a DNA is required by the invoker.

As a result, if the input represents a RNA, the application will raise an error right after the predicate evaluation `(equal? type "RNA")` and not at the very end of the program. The user of the library does not have to care about types and still can produce applications that catch errors very early. This scenario was the motivating case for the iterative typing development. The attentive reader may try to reproduce the demonstration (Figure 4.2) from the below LScheme code:

```
;; DNA-from-sequence :: String -> DNA
(define (DNA-from-sequence seq) ...)
;; DNA-from-input :: String -> DNA
(define (DNA-from-input in) ...)

;; DNA-from-sequence :: String -> RNA
(define (ARN-from-sequence seq) ...)
;; DNA-from-input :: String -> RNA
(define (RNA-from-input in) ...)

;; read-type :: String -> String
(define (read-type input) ...)

;; unsafe-read :: \-/ a . String -> a
(define (unsafe-read input)
  (let ((type read-type))
    (cond ((equal? type "DNA") [a => DNA] (DNA-from-input input))
          ((equal? type "RNA") [a => RNA] (RNA-from-input input))
          (else (error "unknown type")))))

;; complement :: \-/ b . b -> b
(define (complement x)
  ...)

;; find-sequence :: \-/ c . (c,c) -> Boolean
```

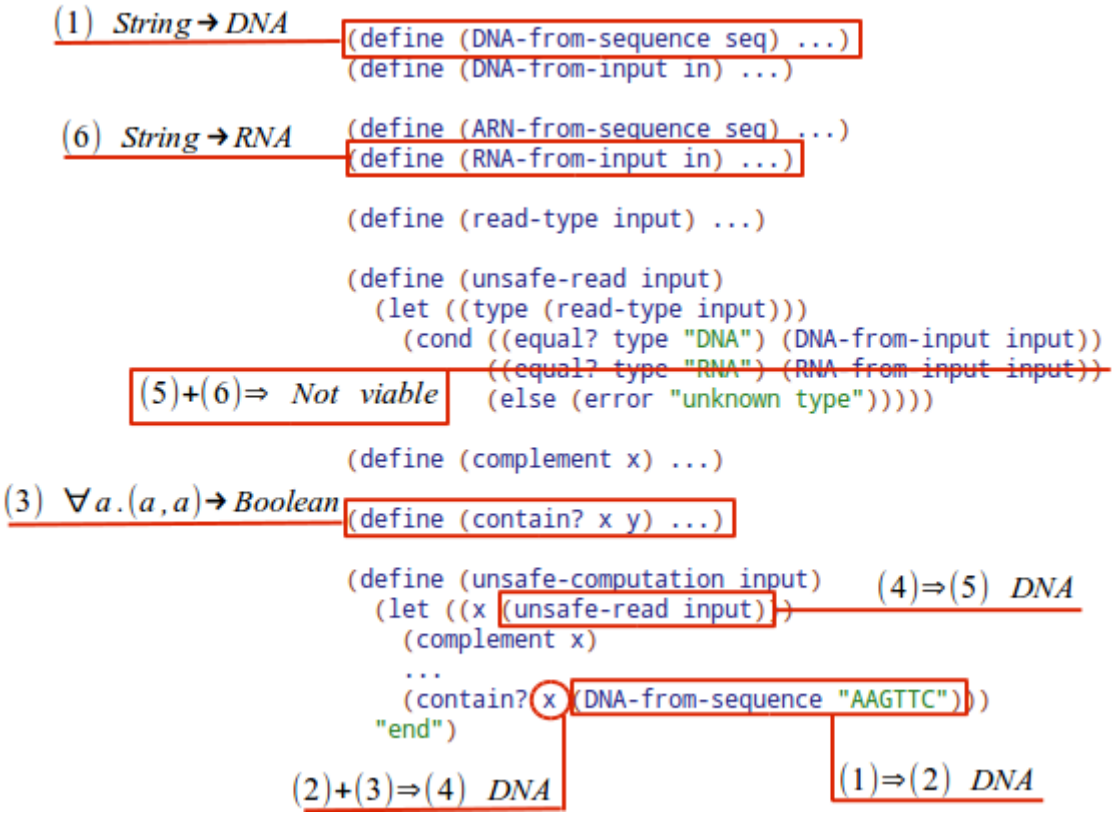

Figure 4.2: Succession of type deductions that demonstrates that the second conditional branch of the polymorphic procedure `unsafe-read` is not viable inside the call `(let ((x (unsafe-read input)))...)` . That call expects a `DNA` as result, not a `RNA` .

```
(define (contain? x y)
  ...)

;; unsafe-computation :: String -> String
;; x :: d
(define (unsafe-computation input)
  (let ((x ((sub [a => d] unsafe-read) input)))
    ((sub [b => d] complement) x)
    ...
    ((sub [c => d, c => DNA] contain?) x (DNA-from-sequence "AAGTTC"))
    "end))"
```

A real life example SBML stands for "Systems Biology Markup Language" it is an XML-like language that is used to encode biologic reactions. SBML provides a real world example of large size objects that are serialized in form where type informations are easily accessible. Indeed SBML documents may be very large and even contain references to biologic data base to load complicated protein structures. Nevertheless very early in the model description, one can define the attribute `id` of the tag `mode` that can be used to predict the kind of model the SBML document describes. Below, we give a cut example of a SBML document exposed inside the official documentation[9]:

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level3/version1/core" level="3" version="1">
    <!-- The model represented by this document is dimerization -->
    <model id="dimerization" substanceUnits="item" timeUnits="second"
                             volumeUnits="litre" extentUnits="item">
        <listOfUnitDefinitions>
        ...
```

```
        </listOfUnitDefinitions>
        <listOfCompartments>
        ...
        </listOfCompartments>
        <listOfSpecies>
        ...
        </listOfSpecies>
        <listOfReactions>
        ...
        </listOfReactions>
    </model>
</sbml>
```

The reading procedure for SBML documents would have the following form in LScheme:

```
;; readSBML :: \-/ a . string -> a
(define (readSBML docSBML)
  (let (id ((getSBML "model" "id" docSBML)))
    (cond ([a => facilitated_ca_diffusion] (equal? id "facilitated_ca_diffusion") ...)
          ([a => lotkaVolterra_tranport]   (equal? id "lotkaVolterra_tranport")   ...)
          ([a => dimerization]             (equal? id "dimerization")             ...)
          ([a => String]                   else                    "Unknown model"))))
```

As for the DNA-RNA case, type errors would be detected right inside the reading procedure at the case analysis.

4.2 Computation context

Introduction Sometimes values alone do not suffice to represent the result of a computation. Additional informations, called computation context, may be needed to precisely define how the computation happened. One evident context is the failure context that indicates the computation did not end properly. But other contexts may be useful like confidence indices, relative errors, etc. When those contexts are represented by different types, iterative typing can detect very early the production of a wrong context.

Failure context When a computation cannot be performed normally, the failure can be handled with different patterns:

1. *Error marker*: when the procedure detects an illegal configuration it directly returns a marker that must be interpreted as a failure. This is the old-fashion error handling where the invoker must systematically test the result to verify the computation succeed. In case of failure the invoker may either handle it or return an error marker. The following Java code illustrates such pattern with the classic division by zero error:

```
// code error for division by zero
static double DIVISION_BY_ZERO = Double.MIN_VALUE;

// divide performs an integer division
// It fails when it receives 0 as divisor
public static double integerDivision(int x, int y) {
    if (y == 0) {
        return DIVISION_BY_ZERO;
    } else {
        return x / y;
    }
}

// randomDivision lets the invoker handle the division by zero case
public static double randomDivision() {
    Random r = new Random();
    int x = r.nextInt();
    int y = r.nextInt();
    return integerDivision(x,y);
}

// safeMain handles the potential division by zero
```

```
public static void safeMain() {
    double d = randomDivision();
    if (d == DIVISION_BY_ZERO) {
        System.out.println("Unfortunately 0 has been picked as a divisor");
    } else {
        System.out.println(2 * d);
    }
}

// unsafeMain return an inconsistent result in division by zero case
public static void unsafeMain() {
    System.out.println(2 * randomDivision());
}
```

Lets notice that in statically typed languages there is no clear distinction between the error marker and regular return values since they must share the same type. In consequence, the error marker could be confused with a correct result.

2. *Exceptions*: high level languages usually propose native structures to handle errors. Here, we present the exception handling proposed by Goodenough[6] which is very common. Semantically, this pattern is mainly equivalent to the error marker one but facilities are provided to separate exceptions generation from exceptions handling. The following Java code performs the same actions as the first one but in a more concise and elegant way:

```
// divide fails if it receives 0 as divisor.
public static int integerDivision(int x, int y) throws ArithmeticException {
    if (y == 0) {
        throw new ArithmeticException();
    } else {
        return x / y;
    }
}

// randomDivision lets the invoker handle division by zero cases
public static int randomDivision() throws ArithmeticException {
    Random r = new Random();
    int x = r.nextInt();
    int y = r.nextInt();
    return integerDivision(x,y);
}

// main handle the potential division by zero
public static void safeMain() {
    try {
        System.out.println(2 * (randomDivision()));
    } catch (ArithmeticException e) {
        System.out.println("Unfortunately 0 has been picked as a divisor");
    }
}

// uncaught exception that may cause the application to stop
public static void unsafeMain() throws ArithmeticException {
    System.out.println(2 * (randomDivision()));
}
```

3. *Persistent computation*: to conserve library use as simple as possible, some languages like Matlab often return usable result in case of failure. For instance `1/0` in the Matlab console gives `Inf` which can be used as a number.

Iterative marker Currently iterative typing does not support exceptions. Nevertheless the marker pattern can be implemented very successfully. Thanks to dynamic conditional structures the type of the error marker may be different from the one of the regular values. That way, the use of the marker will produce a type error instead of an inconsistent result. As usual that error will be detected as soon as the failure is detected by the dynamic conditional structure:

```
;; return null in division by zero case
(define (division x y)
  (if (= y 0)
      "division by zero" ; error detection on unsafe use
      (/ x y)))

;; let the invoker handle the division by zero case
(define (randomDivision)
  (define x (randomNumber))
  (define y (randomNumber))
  (division x y))

;; handle the division by zero case
(define (safe-main)
  (define x (randomDivision))
  (if (null? x)
      (display "Unfortunately 0 has been picked as a divisor")
      (display (* 2 x))))

;; force randomDivision to return a number
(define (unsafe-main)
  (define x (randomDivision))
  (display "dummy string")
  (display (* 2 x)))
```

1. *Scheme execution*: in case of division by zero, a type error is raised at `* 2 x` which is still better than displaying an inconsistent result:

```
dummy string
*: expects type <number> as 2nd argument, given: null; other arguments were: 2
```

2. *Iterative execution*: in case of division by zero, a type error is raised right after failure detection at the conditional structure of `division` ; we get:

```
Unification failed on [String => Number]
```

Marker-Persistent pattern Iterative typing allows to combine the persistent computation and the error marker pattern through the `if-viable` experimental feature. `(if-viable <attempt> <alt>)` only executes `<attempt>` when it is viable according to iterative typing.

```
(if-viable <attempt>
           <alt>)
;; Semantically equivalent ;;
(if <attempt viable?>
    <attempt>
    <alt>)
```

The procedure that detects an error can, thanks to `if-viable` , know if the error marker will be handled or if it will produce a type error later on. Knowing this, it may want to adjust its strategy and return a usable result instead to prevent the whole application to stop:

```
;; create-matrix :: [[Number]] -> Matrix
(define (create-matrix s) ...)

;; det :: Matrix -> Number
(define (det m) ...)

;; add-matrix :: (Matrix,Matrix) -> Matrix
(define (add-matrix m1 m2) ...)

;; inv :: \-/ a . Matrix -> a
(define (inv m)
  (if (= (det m) 0)
      (if-viable "zero determinant"
                 infinite-matrix)
      <matrix inversion>))
```

```
;; safe :: Matrix -> String
(define (safe m)
  (define inv-m (inv m))
  (if (string? inv-m)
      (display inv-m)
      (display (add-matrix inv-m m)))
  "end")

;; unsafe :: Matrix -> String
(define (unsafe m)
  (display (add-matrix (inv m) m))
  "end")
```

1. *Safe use*: because `safe` tests the result of `inv`, the `inv` procedure detects that is has been call in a safe context. In the invoker handles the error marker pattern, `inv` can safely return the null pointer:

```
> (safe (create-matrix (list (list 1 2 3) (list 1 2 3) (list 4 5 6))))
"m cannot be inversed"
```

2. *Unsafe use*: because `unsafe` does not test the result of `inv` before calling `add-matrix`, the `inv` procedure detects that it has been called in an unsafe context. In that case, `inv` knows that returning `null` will produce a type error and that the invoker did not implement the error marker pattern. As a result the persistent computation pattern is used:

```
> (unsafe (create-matrix (list (list 1 2 3) (list 1 2 3) (list 4 5 6))))
[Inf Inf Inf, Inf Inf Inf, Inf Inf Inf]
```

We highlight the point that all the consequences of the `inspect` special form have not been investigated so far ; a more elaborate presentation of `inspect` can be found at (Chapter 7).

4.3 Side effects

The leitmotiv of iterative typing is to detect errors earlier without changing the Scheme semantic. This is a praise-worthy goal but detecting errors earlier may actually modify the semantic if we consider side effects.

Side effects A procedure is said to have side effects when it can interact with the outside world with other means than its result. Below we give an implementation of the map procedure, this procedure is free of side effect ; its result has to be used otherwise the computation is useless:

```
(define (map f xs)
  (if (null? xs)
      null
      (cons (f x) (map f (cdr xs)))))

(begin
  (map (lambda (x) (+ x 1)) (list 1 2 3)) ; useless computation
  "done")
```

Below we give the definition of a procedure that computes the square of its argument and stores the result inside the variable `*global*` :

```
(define *global* null)
(define (side-effect-square x)
  (set! *global* (* x x))
  "done")

(begin
  (side-effect-square 2) ; usable computation result can be retrieved
  "done")
```

In this section we make a distinction between internal side effects that are cleared when the application exits and external side effects that remain after the application exits. Internal side effects are mainly represented by assignments like `set!`. External side effects may involve the file system or the console with `display`.

Semantic modification By detecting errors earlier, iterative typing also stops the program earlier. This may prevent some side effects to happen:

```
(define (launch-da-rocket x)
  (if <pred>
      (begin
        ...
        (display "Launch da rocket!!!")
        ...
        (+ x 1))
      <alt>))
(launch-da-rocket "foo")
```

1. *Scheme execution*: when the above program is executed by a standard Scheme interpreter, the rocket is launched then the application raises a type error:

   ```
   Launch da rocket!!!
   +: expects type <number> as 1st argument, given: "foo"; other arguments were: 1
   ```

2. *LScheme execution*: when the above program is executed by an iterative Scheme interpreter, the application raises the type error before launching any rocket:

   ```
   Unification failed on [String => Number]
   ```

Notice that internal side effects are not concerned by this discussion since the application will exits anyway. As exposed below, this semantic variation may have good and bad implications.

Positive scenario Many processes may leave the computer in an undesirable state if not terminated properly. This is the case for resource allocation, a common pattern when several instances want to manipulate the same data:

1. The instance waits its turn to use the resource.
2. When the instance obtains the permission, it is the only one allowed to manipulate the resource.
3. Once its job on the resource is done, the instance releases the resource to allow another instance to use the resource.

If the resource is not released, other instances may never access it which can be catastrophic. Below we present a Scheme code where iterative early error detections allow to prevent the resource to be locked for ever:

```
(define (write x)
  (if <pred>
      (begin                              ; LScheme error detection
        (display "lock resource")
        (write resource (concat x "yo")) ; Scheme error detection
        (display "unlock resource"))
      <alt>))
(write 1)
```

1. *Scheme execution*: the permission has been aquired ; after the application exits, the resource cannot be manipulated anymore:

   ```
   lock resource
   concat: expects type <string> as 1st argument, given: 1; other arguments were: "yo"
   ```

2. *LScheme execution*: the permission has not been acquired ; after the application exits, the resource may still be manipulated:

   ```
   Unification failed on [String => Number]
   ```

Negative scenario Even if it seems natural to decrease the impact of a program that will crash anyway, sometimes we do want to execute the program as far as possible. For instance, programmers typically use log files to debug application. Those files contain information about the program execution and it is bad if some traces are missing. Iterative typing may reduce the amount of provided information:

```
(define (write-log x)
  (if <pred>
      (begin                                    ; LScheme error detection
        (write "Log start")
        ...
        (write (concat "x is a" (type-of x)))
        (write (concat x "yo"))                 ; Scheme error detection
        ...)
      <alt>))
```

"Unsafe" saves the day The above scenario demonstrates that it can be useful to grant programmers the ability to shut down iterative typing for a while. This has been achieved with the implementation of the `(unsafe <body>)` special form ; if a type error is detected early inside `<body>` the iterative prediction is switched off instead of raising an error. As a result, LScheme execution inside the `unsafe` special form has the same semantics as the pure Scheme execution:

```
(define (launch-rocket x y)
  (if x
      (begin (display "Launch rocket!!!")
             (+ x 1))
      (concat x "foo")))
```

1. *LScheme execution*: the application crashes before launching any rocket:

   ```
   (launch-rocket #t "bar")
   ```

   ```
   Unification failed on [String => Number].
   ```

2. *Unsafe execution*: the rocket is launched before the application crashes ; the error is caught by the underlying Scheme instead of the iterative interpreter:

   ```
   (unsafe (launch-rocket #t "bar"))
   ```

   ```
   "Launch rocket!!!"
   + expects type <number> of as 1st argument, given: "bar"; other arguments were: 1
   ```

Recapitulation The combination of early error detection and external side effects cause iterative typing to modify the original Scheme semantic. In some cases, this can be an issue and that is why iterative typing provides the `unsafe` special form that allows to shut down early error detection and to execute code with the exact Scheme semantic.

4.4 Conclusion

In this chapter we saw three case studies. The first case study is about early error detection for deserialization patterns. The second case study exhibits how iterative typing can implement a mix of marker and persistent patterns for exceptions management. The last case study is a discussion about early error detection and side effect. The second case study is an application of the `if-viable` special form and the third case study is an application of the `unsafe` special form. Those case studies highlight the fact that iterative typing completely hides the type complexity from the programmer. As a result, very unskilled programmers can implement relatively safe dynamic programs. A part of the scientific community uses dynamic languages to easily develop applications, those scientists could use iterative typing as a mean to detect errors earlier without changing theirs habits.

Chapter 5

Theoretical basis

At this point, the reader must have a precise idea about what iterative typing is capable of and some intuition about how it is implemented. The first part of this chapter provides a strong base for iterative typing. The second part goes deep down the iterative typing implementation and, according the theoretical base, justify most implementation choices.

5.1 Type set

In this section we introduce very standard concepts [12][10][18] that will be used in the entire chapter.

Monotypes The monotype set is an infinite set based on type variables and type constructors. We use τ, ξ, γ to respectively range over monotypes, type variables and type constructors. A monotype is either a type variable or a constructed type ; formally the type set is recursively defined as $\tau = \xi \mid \gamma\ \tau\ \ldots\ \tau$. A common interpretation of type variable is that they can represent any type. Type constructors accept a fixed number of monotype, this number being the arity of the type constructor, and return a new kind of monotype.

Primitive type Zero-arity type constructors construct primitive types. Each primitive type represents a set of semantic values whose elements can be represented as constants inside a program. The primitive types we will use are $String$, $Boolean$ and $Number$. We assume the existence of a function $typeOf$ that returns the primitive type of the given constant:

- $typeOf(\text{"}foo\text{"}) = String$
- $typeOf(123) = Number$

Compound type constructors Those type constructors expect at least one monotype to construct new kind of types. By referencing other monotypes those constructors can be nested without theoretical limit:

1. *Functional type constructor*: Values of type $\tau \to \tau'$ are procedures that, once they get a value of type τ, returns value of type τ'. For simplicity reasons, in this part of the thesis we will consider that procedures exactly accept one single argument ; so the arity of $\to$ is exactly two. Nevertheless we will see that this restriction does not reduce the expressive power of the language. Here are a bunch of $\mathcal{L}$ procedure definitions with one of their associated type:
   - $(\lambda\ x\ (if(<\ x\ 0)\ \text{"}pos\text{"}\ \text{"}neg\text{"})) : Number \to String$
   - $(\lambda\ x\ x) : \xi \to \xi$
   - $(\lambda\ f\ (f\ \text{"}foo\text{"})) : (String \to \xi) \to \xi$
2. *List type constructor*: values of type $[\tau]$ are lists whom each element is of type τ. Here is some L expressions that return lists:

- $(list\ 1\ 2\ 3)\ :\ [Number]$
- $(cons\ "foo"\ (cons\ "bar"\ null))\ :\ [String]$

3. *Pair type constructor*: values of type $(\tau_1.\tau_2)$ are pairs whom first element is of type τ_1 and whom second element is of type τ_2. Here is some L expressions that return lists:
   - $(pair\ 1\ "foo")\ :\ (Number.String)$
   - $(pair\ (pair\ 1\ "bar")\ 2)\ :\ ((Number.String).Number)$

Polytypes Polytypes are monotypes whose some type variables have been linked ; we use σ to range over the set of polytypes. The formal recursive definition of the polytypes is: $\sigma = \tau \mid \forall\xi.\tau$. Polytypes introduce the concept of scoping for type variables ; for instance in the polytype: $\forall\xi.\xi \to Number$, outside ξ are dissociated from the inside ξ that appear in $\xi \to Number$. Below, we give some examples of expressions with their polytype:

- $(\lambda\ x\ x)\ :\ \forall\xi.\xi \to \xi$
- $(\lambda\ f\ (f\ 1))\ :\ \forall\xi.(Number \to \xi) \to \xi$
- $(\lambda\ f\ (\lambda\ x\ (f\ x)))\ :\ \forall\xi_1,\xi_2.((\xi_1 \to \xi_2) \to \xi_1) \to \xi_2$

$\forall\xi_1,\xi_2,\ldots.\tau$ is an abusive notation to denote $\forall\xi_1.\forall\xi_2.\ldots.\tau$.

Free type variables Informally, the free type variables of a type represent the type variables that appear inside the type but are not linked to it:

$$\begin{cases} free(\xi) = \{\xi\} \\ free(\gamma\ \tau_1\ \ldots\ \tau_k) = \bigcup_{i=1}^{k} free(\tau_i) \\ free(\forall\xi.\sigma) = free(\sigma) - \{\xi\} \end{cases}$$

Here are the free variable sets of some types:

- $free(\forall\xi.\xi \to \xi') = \{\xi'\}$
- $free(\forall\xi.\xi') = \{\xi'\}$
- $free(\forall\xi.String \to \xi) = \{\}$
- $free(\xi \to String) = \{\xi\}$

Instantiations Instantiations are mappings from type variables to monotypes written $[\xi_i \mapsto \tau_i]$. The mapping only affects the free type variables of a type. Two examples of instantiation are given below:

- $[\xi \mapsto String]\,\xi \to \xi = String \to String$
- $[\xi_1 \mapsto (\xi \to String), \xi_2 \mapsto Number]\,\xi_1 \to (\xi_2 \to Boolean) = (\xi \to String) \to (Number \to Boolean)$
- $[\xi_1 \mapsto String, \xi_2 \mapsto Number]\,\forall\xi_1.\xi_1 \to \xi_2 = \forall\xi_1.\xi_1 \to Number$

Equivalence The equivalence on monotype is reduced to the syntactical identity. That is: two monotypes are equivalent if and only if their symbolic representation is identical. The equivalence on polytype is a little bit more elaborated. Basically polytype equivalence is similar to the syntactical equivalence up to the name of the linked type variables. Again we skip the formal definition and simply provide some examples:

- $\xi \to String = \xi \to String$
- $\xi_1 \to String \neq \xi_2 \to String$
- $\forall\xi_1.\xi_1 \to String = \forall\xi_2.\xi_2 \to String$

Primitive types (I)	
ι	$Boolean$
	$Number$
	$String$

Monotypes (M)	
τ	ι
	ξ
	$\tau \to \tau$
	$[\tau]$
	$(\tau.\tau)$

Polytypes (P)	
π	$\forall \xi.\tau$
	$\forall \xi.\pi$

Types (T)	
σ	τ
	$\forall \xi.\sigma$

Table 5.1: Given an infinite set of type variable (X, ranged with ξ) this table defines recursively the sets: primitive types, monotypes, polytypes, and types. The following relation can be easily verified: $I \subset M \subset P \subset T$. Primitives types are the zero arity type constructors. Monotypes are combinations of type constructors and type variables. Polytypes must contain at least one link before their mono-part. The type set is the union of the monotype set and the polytype set.

Subtype partial order We do not defined any order relation on types ; as a result, given two types we can only say if they are equivalent or not. On the other hand, we do provide a partial order relation for type schemes. Intuitively, the partial order $\sqsubseteq$, named subtype relation[1], ranks type schemes following their specificity. Formally σ is a subtype of σ' means that it exists an instantiation on σ' linked type variables that unifies the type part of σ and σ':

$$\begin{cases} \forall \xi\sigma \sqsubseteq \forall \xi.\sigma' \Leftrightarrow (\sigma = \forall \xi.\sigma') \vee (\exists [\xi \mapsto \tau].\sigma \sqsubseteq [\xi \mapsto \tau]\sigma') \\ \sigma \sqsubseteq \tau \Leftrightarrow \sigma = \tau \end{cases}$$

As $\sqsubseteq$ is not complete partial order, many type scheme pairs like $\forall \xi.\xi \to String$ and $\forall \xi.\xi \to String$ are not comparable. Here are some examples:

- $\xi \to String \sqsubseteq \xi \to String$
- $\forall \xi.\xi \to \xi \sqsubseteq Number \to Number$
- $\forall \xi.\xi \to \xi \sqsubseteq \xi \to \xi$

Alpha-renaming This technique has been took from the lambda calculus. Alpha renaming is about avoiding name collision by renaming linked variables. After alpha-renaming, we can say that two variables denote the same entity if and only if they share the same name. Below, we present some alpha-renaming:

- $\forall \xi.\forall \xi.\xi \to String \xrightarrow{\alpha} \forall \xi_1.\forall \xi_2.\xi_2 \to String$
- $(\forall \xi.\xi \to \xi_1, \forall \xi.\xi_1 \to \xi) \xrightarrow{\alpha} (\forall \xi_2.\xi_2 \to \xi_1, \forall \xi_3.\xi_1 \to \xi_3)$

Recapitulation The different type sets are summarized in (Table 5.1). We then saw the following concepts for type schemes:

- Free variables, $free(\sigma)$: links capture type variables.
- Instantiation, $[\xi_i \mapsto \tau_i]\,\sigma$: mapping of the free type variables contained in σ.
- Equivalence, $\sigma_1 = \sigma_2$: syntactic equivalence up to the symbols used for the linked variables.
- Subtype relation: $\sigma_1 \sqsubseteq \sigma_2$: require equivalence after instantiation of some of σ_2 linked type variables.
- Alpha renaming: by renaming the linked variables of type schemes we can avoid name collisions.

[1]The reader should not confuse this relation with the concept of subtyping in object oriented paradigm ; here subtyping is used in the sens of [16]

Name	Atomic Scheme		$\mathcal{L}$	
Constant	`<EXPR>`	`c`	e	c
Identifier		`x`		x
Abstraction		`(lambda (<PARAM>) <EXPR>)`		$(\lambda\ x\ e)$
Application		`(<EXPR> <EXPR>)`		$(e\ e)$
Binding		`(let ((x <EXPR>)) <EXPR>)`		$(let\ x\ e\ e)$
Conditional structure		`(if <EXPR> <EXPR> <EXPR>)`		$(if\ e\ e\ e)$

Table 5.2: The equivalence of each $\mathcal{L}$ expression in Scheme.

5.2 The $\mathcal{L}$ language

$\mathcal{L}$ is a representation of an atomic subset of Scheme which is much more suitable to be expressed in type systems. As argued in this section, despite being atomic $\mathcal{L}$ has about the same expressive power as Scheme without assignments and continuations. The semantic of Scheme will be skipped since it is exposed at (Annexe A). Because the transformation from atomic Scheme to $\mathcal{L}$ is exposed in this section, we skip the semantic of $\mathcal{L}$ as well.

Statement elimination To keep type systems as simple as possible, we must eliminate the concept of statement. Statements can be easily replaced by nested `let` expression:

- *Definition statement*: Without assignment, removing definition also allow to remove side effect on the environment.

$$[\![\ \texttt{(define x <bind>) <rest>}\ ]\!] = \texttt{(let ((x } C[\![\ \texttt{<bind>}\ ]\!] \texttt{)) } C[\![\ \texttt{<rest>}\ ]\!] \texttt{)}$$

  - $C[\![\ \texttt{(define pi 3.14) (* 2 pi)}\ ]\!] = \texttt{(let ((pi 3.14)) (* 2 pi))}$
  - $C[\![\ \texttt{(define pi 3.14) (define e 2.71) (* pi e)}\ ]\!] = \texttt{(let ((pi 3.14)) (let ((e 2.71)) (* pi e)))}$

- *Other statement*: Here a dummy name `_` is used ; it should never be referenced.

$$C[\![\ \texttt{<expr> <rest>}\ ]\!] = \texttt{(let ((\_ } C[\![\ \texttt{<expr>}\ ]\!] \texttt{)) } C[\![\ \texttt{<rest>}\ ]\!] \texttt{)}$$

  - $C[\![\ \texttt{(display "foo") (+ 1 2)}\ ]\!] = \texttt{(let ((\_ (display "foo"))) (+ 1 2))}$
  - $C[\![\ \texttt{(define pi 3.14) (display "foo") (* 2 pi)}\ ]\!] = \texttt{(let ((pi 3.14)) (let ((\_ (display "foo"))) (+ 1 2)))}$

Fixing the arity The second difficulty we prefer to avoid is the freedom about procedure arity. With the fellowing syntactic transformations we can simulate the behavior of multiple argument procedure with only procedure of one argument:

- *Constant procedure*: This case correspond to the definition of a procedure that takes no arguments. We introduce an artificial argument whose symbol is `_` and whose value is a constant of a specific type `#_` :

$$\begin{cases} C[\![\ \texttt{(lambda () <body>)}\ ]\!] = \texttt{(lambda (\_) } C[\![\ \texttt{<body>}\ ]\!] \texttt{)} \\ C[\![\ \texttt{(<abs>)}\ ]\!] = \texttt{ (} C[\![\ \texttt{<body>}\ ]\!] \texttt{ \#\_)} \end{cases}$$

  - $C[\![\ \texttt{(lambda () "foo")}\ ]\!] = \texttt{(lambda (\_) "foo")}$
  - $C[\![\ \texttt{(f)}\ ]\!] = \texttt{(f \#\_)}$

- *Single argument procedure*: No syntactic transformation:

$$\begin{cases} C[\![\ \texttt{(lambda (x) <body>)}\ ]\!] = \texttt{(lambda (x) } C[\![\ \texttt{<body>}\ ]\!] \texttt{)} \\ C[\![\ \texttt{(<abs> <arg>)}\ ]\!] = \texttt{ (} C[\![\ \texttt{<abs>}\ ]\!] \ \ C[\![\ \texttt{<arg>}\ ]\!] \texttt{)} \end{cases}$$

- *Multiple argument procedure*: Multiple argument procedures can be transformed into nested mono argument procedure ; such transformation is called curryfication:

$$\begin{cases} C[\![\texttt{(lambda (x1 <tail>) <body>)}]\!] = \texttt{(lambda (x\_1)}\ C[\![\texttt{(lambda (<tail>) <body>)}]\!] \texttt{)} \\ C[\![\texttt{(<abs> <init> <last>)} = \texttt{(}\ C[\![\texttt{(<abs> <init>)}]\!]\ C[\![\texttt{<last>}]\!] \texttt{)} \end{cases}$$

  - $C[\![\texttt{(lambda (x1 x2 x3) x3)}]\!] = (\lambda\ x_1\ (\lambda\ x_2\ (\lambda\ x_3\ x_3)))$
  - $C[\![\texttt{(f 1 2 3)}]\!] = (((f\ 1)\ 2)\ 3)$

Variable shadowing We suppose the compilation from the atomic Scheme to $\mathcal{L}$ alpha-renames all the variables so we do not have to care about variable shadowing in $\mathcal{L}$. Below we give a program that tests if two numbers are divisible:

```
(define x 10)
(define (divide? (x y)) (= (modulo x y) 0))
(devide? x 5)
```

After compilation we get:

```
(let ((x1 10))
  (let ((divide? (lambda (x1) (lambda (y)
                                ((= ((modulo x1) y)) 0)))))
    ((divide? x1) 5)))
```

Recapitulation What we attempt to do here is to split a difficult problem into easier problems. The original problem was to express a type system for Scheme without assignments nor continuations which is quite hard. The equivalent easier problems are compilation from Scheme to $\mathcal{L}$ and expression of the type system for $\mathcal{L}$. The $\mathcal{L}$ language is free of statements, multiple-argument procedure, variable shadowing and syntactic sugar.

5.3 Hindely-Milner type system

In this section we describe a typical type system rule by rule after introducing assumption set, judgment and derivation

Assumptions set Assumptions, notated A, are sets whose elements have the form: $x : \sigma$. In other words, A associate types to identifiers ; it is basically an environment that carries types instead of values. We extend the concept of free variables to assumptions: the set of the free type variables of an assumption set correspond to the union of all the free variable that appear in those assumptions. Formally we have:

$$free(A) = \bigcup_{x_i:\sigma_i \in A} (free(\sigma_i)$$

Below we give some examples of assumptions instantiation:

- $free(\{x : \xi_1 \to String, y : \xi_2\}) = \{\xi_1, \xi_2\}$
- $free(\{x : \xi_1 \to String, y : \forall \xi_2.\xi_2 \to String\}) = \{\xi_1\}$

Judgment Judgment have the following form: $A \vdash e : \sigma$ which intuitively means that the assumptions A do not contradict that e is of type σ. Here are some example of judgments that holds:

- $\{\} \vdash "foo" : String$
- $\{\} \vdash (\lambda\ x\ x) : \xi \to \xi$
- $\{\} \vdash (\lambda\ x\ x) : \forall \xi.\xi \to \xi$
- $\{x : Number\} \vdash (\lambda\ y\ (+\ x\ y)) : Number \to Number$

We extend the concept of instantiation to judgments: an instantiation on a judgment corresponds to the instantiation on all its types. Here are some judgment instantiations:

- $[\xi \mapsto Number]\,\{x : \xi \to String, y : \xi\} \vdash y : \xi = \{x : Number \to String, y : Number\} \vdash y : Number$
- $[\xi_1 \mapsto \xi, \xi_2 \mapsto String]\,\{x : \xi_1, y : [\xi_2], z : \forall \xi_1.\xi_1 \to \xi_1\} \vdash x : \xi_1 = \{x : \xi_1, y : [String], z : \forall \xi_1.\xi_1 \to \xi_1\} \vdash x : \xi$

Type instantiation on judgment may seem odd for readers used to type system but it will become very handy to describe the iterative type system. In particular, the rule dedicated to conditional structures make heavy use of this instantiation.

Inference rule & derivation An inference rules indicates that given some premise schemes the judgment scheme holds:

$$\frac{premise_1 \qquad ... \qquad premise_N}{conclusion}$$

In the presented type systems premises are either assertions that must be verified either judgments. Derivations are tree of inference rules in which all the leaves are assertions. A judgment holds if and only if it exists a derivation that ends with that judgment. Inference rules are declarative, that is they do not tell how to construct derivations but given a derivations it is trivial to verify if it respects the inference rules.

Leaves They are only two rules that does not have to be justified by other rules:

- ID: the type of an identifier is defined by the current assumption set:

$$\text{ID}\ \frac{x : \sigma \in A}{A \vdash x : \sigma}$$

- CONST: the type of a constant is supposed to be directly deductible through the previously introduced function $typeOf$:

$$\text{CONST}\ \frac{typeOf(c) = \gamma}{A \vdash c : \gamma}$$

Procedure type deductions Two rules are dedicated to handle type deductions that involve procedures:

- ABS: Given a procedure $(\lambda\ x\ e)$, if the assumption $x : \tau$ allows to deduce $e : \tau'$ that means the procedure will return a value of type τ' if it receives a value of type τ. This is the intuitive interpretation of $\tau \to \tau'$:

$$\text{ABS}\ \frac{A \cup \{x : \tau\} \vdash e : \tau'}{A \vdash (\lambda\ x\ e) : \tau \to \tau'}$$

- APP: When we give a value of type τ' to a procedure of type $\tau' \to \tau$ we expect to receive a value of type τ:

$$\text{APP}\ \frac{A \vdash e : \tau' \to \tau \qquad A \vdash e' : \tau'}{A \vdash (e\ e') : \tau}$$

Here is derivation that justifies the judgment $\{\} \vdash ((\lambda\ x\ x)\ 1) : Number$ with the four rules introduced so far:

$$\text{APP}\ \frac{\text{ABS}\ \dfrac{\text{ID}\ \dfrac{x : Number \in \{x : Number\}}{\{x : Number\} \vdash x : Number}}{\{\} \vdash (\lambda\ x\ x) : Number \to Number} \qquad \text{CONST}\ \dfrac{typeOf(1) = Number}{\{\} \vdash 1 : Number}}{\{\} \vdash ((\lambda\ x\ x)\ 1) : Number}$$

Polymorphism support From here we move away from the λ calculus to support polymorphism:

- GEN: At any time, if the assumptions does not contain the type variable ξ we can generalize the current deduction $A \vdash e : \sigma$ to $A \vdash e : \forall \xi . \sigma$:

$$\text{GEN}\ \frac{A \vdash e : \forall \xi_{1..i} . \tau \qquad \xi \notin free(A)}{A \vdash e : \forall \xi, \xi_{1..i} . \tau}$$

- SUB: this the inverse rule of GEN: at any time we can make the current deduction more specific:

$$\text{SUB}\ \frac{A \vdash e : \sigma \qquad \sigma' \sqsubseteq \sigma}{A \vdash e : \sigma'}$$

Polymorphic recursive let There are many ways to introduce recursion, some use the fix point, others use a dedicated rules. We chose to support both polymorphism and recursion through *let* expressions. Given (*let x bind body*) the first think to do is to deduce the type of *bind* with monomorphic recursion. Monomorphic recursion suppose that both x and *bind* share the same type which can be expressed by the following judgment: $\{x : \tau\} \vdash bind : \tau$. After that deduction, we should generalize the type of *bind* to σ if we want that x appear polymorphically inside *body*. The overall type is given by type deduction of *body* knowing that x is of type σ. So the body of a *let* allows to enter polymorphically into the binding ; once inside the binding, monomorphic recursion is performed.

$$\text{LET}\ \frac{A \cup \{x : \tau\} \vdash e : \sigma \qquad A \cup \{x : \sigma\} \vdash e' : \tau' \qquad \tau \sqsubseteq \sigma}{A \vdash (let\ x\ e\ e') : \tau'}$$

Conditional structure Both branches are analyzed the same way and must return the same type ; such modus operandi considerably reduces the expressiveness of $\mathcal{L}$.

$$\text{COND}\ \frac{A \vdash e : \tau \qquad A \vdash e_1 : \tau' \qquad A \vdash e_2 : \tau'}{A \vdash (if\ e\ e_1\ e_2) : \tau'}$$

Recapitulation We first defined the concept of assumptions, judgments and derivations. We also exposed a usual type system based on Hindley-Milner work (Table 5.3).

5.4 The $\mathcal{L}$ type system

The complete type system of $\mathcal{L}$ is shown at (Table 5.4). Many rules remain intact, this section justifies the modifications done on COND and GEN.

LCOND rule The iterative LCOND rule is much more permissive than the static one, it summarizes the essence of iterative typing. LCOND rule makes that the type deduction of each branch does not affect each other and the rest of the deduction. Lets analyze two different conditional structures:

- Conditional branches can return different types. Here is the proof that the classic evil conditional structure, $(if\ x\ 1\ "foo")$, is accepted by the $\mathcal{L}$ type system:

$$\text{ID}\ \frac{x : B \in \{x : B\}}{\{x : B\} \vdash x : B \quad *}$$

$$\text{CONST}\ \frac{typeOf(1) = N}{[\xi \to N]\,\{x : B\} \vdash 1 : \xi \quad **}$$

$$\text{CONST}\ \frac{typeOf("foo") = S}{[\xi \to S]\,\{x : B\} \vdash "foo" : \xi \quad ***}$$

$$\text{LCOND}\ \frac{* \qquad ** \qquad ***}{\{x : B\} \vdash (if\ x\ 1\ "foo") : \xi}$$

$$\text{ID}\ \frac{\{x:\sigma\} \in A}{A \vdash x:\sigma} \qquad \text{CONST}\ \frac{typeOf(c) = \gamma}{A \vdash c:\gamma}$$

$$\text{ABS}\ \frac{A \cup \{x:\tau\} \vdash e:\tau'}{A \vdash (\lambda\ x\ e):\tau \to \tau'} \qquad \text{APP}\ \frac{A \vdash e:\tau' \to \tau \qquad A \vdash e':\tau'}{A \vdash (e\ e'):\tau}$$

$$\text{GEN}\ \frac{A \vdash e:\sigma \qquad \xi \notin free(A)}{A \vdash e:\forall\xi.\sigma} \qquad \text{SUB}\ \frac{A \vdash e:\sigma \qquad \sigma' \sqsubseteq \sigma}{A \vdash e:\sigma'}$$

$$\text{LET}\ \frac{A \cup \{x:\tau\} \vdash e:\sigma \qquad A \cup \{x:\sigma\} \vdash e':\tau' \qquad \tau \sqsubseteq \sigma}{A \vdash (let\ e\ e'):\tau'}$$

$$\text{COND}\ \frac{A \vdash e:\tau \qquad A \vdash e_1:\tau' \qquad A \vdash e_2:\tau'}{A \vdash (if\ e\ e_1\ e_2):\tau'}$$

Table 5.3: A Hindley-Milner based type system.

$$\text{ID}\ \frac{\{x:\sigma\} \in A}{A \vdash x:\sigma} \qquad \text{CONST}\ \frac{typeOf(c) = \gamma}{A \vdash c:\gamma}$$

$$\text{ABS}\ \frac{A_x \cup \{x:\tau\} \vdash e:\tau'}{A \vdash (\lambda\ x\ e):\tau \to \tau'} \qquad \text{APP}\ \frac{A \vdash e:\tau' \to \tau \qquad A \vdash e':\tau'}{A \vdash (e\ e'):\tau}$$

$$\text{LGEN}\ \frac{A \vdash e:\forall\xi_1,\ldots.\tau \to \tau' \qquad \xi \notin free(A)}{A \vdash e:\forall\xi,\xi_1,\ldots.\tau \to \tau'} \qquad \text{SUB}\ \frac{A \vdash e:\sigma \qquad \sigma' \sqsubseteq \sigma}{A \vdash e:\sigma'}$$

$$\text{LET}\ \frac{A_x \cup \{x:\tau\} \vdash e:\sigma \qquad A_x \cup \{x:\sigma\} \vdash e':\tau' \qquad \tau \sqsubseteq \sigma}{A \vdash (let\ e\ e'):\tau'}$$

$$\text{LCOND}\ \frac{A \vdash e:\tau \qquad [\xi' \mapsto \tau']A \vdash e':\tau''' \qquad [\xi'' \mapsto \tau'']A \vdash e'':\tau'''}{A \vdash (if\ e\ e'\ e''):\tau'''}$$

Table 5.4: The $\mathcal{L}$ type system.

- The identifier use can be encapsulated inside each branch. For instance the $\mathcal{L}$ code $(if\ x\ (incr\ y)\ (length\ y))$ is accepted by the $\mathcal{L}$ type system. This code, following the value of x, either considers y as a number and increments it or considers y as a string and evaluates its length. Here is the proof that this code is accepted by $\mathcal{L}$ type system:

$$\text{ID}\ \frac{x : B \in \{x : B, y : N, incr : N \to N, length : S \to N\}}{\{x : B, y : N, incr : N \to N, length : S \to N\} \vdash x : B} \quad *$$

$$\text{ID}\ \frac{y : N \in \{x : B, y : N, incr : N \to N, length : S \to N\}}{\{x : B, y : N, incr : N \to N, length : S \to N\} \vdash y : N} \quad (1)$$

$$\text{ID}\ \frac{incr : N \to N : \{x : B, y : N, incr : N \to N, length : S \to N\}}{\{x : B, y : N, incr : N \to N, length : S \to N\} \vdash incr : N \to N} \quad (2)$$

$$\text{APP}\ \frac{(1) \qquad (2)}{[\xi \to N]\,\{x : B, y : \xi, incr : N \to N, length : S \to N\} \vdash (incr\ y) : N} \quad **$$

$$\text{ID}\ \frac{y : S \in \{x : B, y : S, incr : N \to N, length : S \to N\}}{\{x : B, y : S, incr : N \to N, length : S \to N\} \vdash y : S} \quad (3)$$

$$\text{ID}\ \frac{length : S \to N : \{x : B, y : N, incr : N \to N, length : S \to N\}}{\{x : B, y : S, incr : N \to N, length : S \to N\} \vdash length : S \to N} \quad (4)$$

$$\text{APP}\ \frac{(3) \qquad (4)}{[\xi \to S]\,\{x : B, y : \xi, incr : N \to N, length : S \to N\} \vdash (length\ y) : N} \quad ***$$

$$\text{LCOND}\ \frac{* \qquad ** \qquad ***}{\{x : B, y : \xi, incr : N \to N, length : S \to N\} \vdash (if\ x\ (incr\ y)\ (length\ y)) : N}$$

Hindley-Milner type variables When a Hindley-Milner derivation contains free type variables, each instantiation of those type variables applied on that derivation produces a new valid derivation. Below we give a general derivation for the identity procedure and then instantiate its free type variable ξ to S:

$$\text{APP}\ \frac{\text{ID}\ \dfrac{x : \xi \in \{x : \xi\}}{\{x : \xi\} \vdash x : \xi}}{\{\} \vdash (\lambda\ x\ x) : \xi \to \xi} \xrightarrow{[\xi \mapsto S]} \text{APP}\ \frac{\text{ID}\ \dfrac{x : \xi \in \{x : S\}}{\{x : S\} \vdash x : S}}{\{\} \vdash (\lambda\ x\ x) : S \to S}$$

We can say that the rules of the Hindley-Milner type system consider type variables as marker that can be replaced by any monotype. This status is verified by the Hindley-Milner type inference where type variables represent unspecified type which wait for an instantiation.

Iterative type variables In the $\mathcal{L}$ type system, the status of type variable is damaged. In particular, the rule LCOND produces invalid derivations when instantiations are applied on entire derivations.

Below, we illustrate this fact with the traditional $(if\ x\ 1\ "foo")$:

$$\text{LCOND}\ \frac{\text{ID}\ \dfrac{x:B \in \{x:B\}}{\{x:B\} \vdash x:B} \quad \text{CONST}\ \dfrac{typeOf 1 = N}{[\xi \mapsto N]\{x:B\} \vdash 1:\xi} \quad \text{CONST}\ \dfrac{typeOf "foo" = S}{[\xi \mapsto S]\{x:B\} \vdash "foo":\xi}}{\{x:B\} \vdash (if\ x\ 1\ "foo"):\xi}$$

$$\xrightarrow{[\xi \mapsto S]}$$

$$\text{LCOND}\ \frac{\text{ID}\ \dfrac{x:B \in \{x:B\}}{\{x:B\} \vdash x:B} \quad \text{CONST}\ \dfrac{typeOf 1 = N}{\{x:B\} \vdash 1:S} \quad \text{CONST}\ \dfrac{typeOf "foo" = S}{\{x:B\} \vdash "foo":S}}{\{x:B\} \vdash (if\ x\ 1\ "foo"):S}$$

The first instantiation is valid for the $\mathcal{L}$ type system but its instantiation is not because of the judgment $\{x:B\} \vdash 1:S$. So, inside $\mathcal{L}$ type system, type variables do not longer represent any monotype. Instead, type variables should be considered as an unknown specific types. Above, the judgment $\{x:B\} \vdash (if\ x\ 1\ "foo"):\xi$ should be understand as: "the derivation did not succeed to find the exact type of $(if\ x\ 1\ "foo")$; it can be anything". It is interesting to notice how a small modification of the type system radically affects our comprehension of types.

The generalization rule Due to the new status of type variables, we want to generalize only functional types. Lets analyze the following Scheme code:

```
(define x (if <pred> 1 "foo"))
(incr x)   ; x must be a number
(length x) ; x must be a string
```

This code obviously contains a static type error since `x` can not be a number and a string at the same time. If we do allow generalization on non functional types the type of the expression `(if <pred> 1 "foo")` can be inferred to $\forall\xi.\xi$ and the above code will be accepted with two successive instantiation of `x` . Here is the undesirable derivation of $A \vdash (if\ x\ 1\ "foo"):\forall\xi.\xi$:

$$\text{GEN}\ \frac{\text{LCOND}\ \dfrac{\text{ID}\ \dfrac{x:B \in \{x:B\}}{\{x:B\} \vdash x:B} \quad \text{CONST}\ \dfrac{1 :: N}{[\xi \to N]\{x:B\} \vdash 1:\xi} \quad \text{CONST}\ \dfrac{typeOf("foo") = S}{[\xi \to S]\{x:B\} \vdash "foo":\xi}}{\{x:B\} \vdash (if\ x\ 1\ "foo"):\xi}}{\{x:B\} \vdash (if\ x\ 1\ "foo"):\forall\xi.\xi}$$

Direct context We already saw that LCOND rule allows to encapsulate type deduction inside conditional branched. In this paragraph we will see that outside world still continue to affect the type deduction inside the conditional branches. If this statement was not verified, the following $\mathcal{L}$ code $(incr(if\ x\ 1"foo"))$[2] would be accepted which is not desirable since the second branch $"foo"$ will always cause a type error. Below, we give a proof by contradiction that judgments of the form $\{incr:N \to N, A\} \vdash (incr\ (if\ x\ 1\ "foo")):N$ can not be derived in the $\mathcal{L}$ type system:

1. As the outmost expression is an application we are forced to use the APP rule:

$$\text{APP}\ \frac{\text{ID}\ \dfrac{incr:N \to N \in \{incr:N \to N, A\}}{\{incr:N \to N, A\} \vdash incr:N \to N} \qquad \{incr:N \to N, A\} \vdash (if\ x\ 1\ "foo"):N}{\{incr:N \to N, A\} \vdash (incr\ (if\ x\ 1\ "foo")):N}$$

2. We then have to justify a judgment of the form: $\{incr:N \to N, A\} \vdash (if\ x\ 1\ "foo"):N$. As the outmost expression is a if we are forced to use the LCOND. This rule requires the justification of the following judgments:

   (a) $\{incr:N \to N, A\} \vdash x:\tau$

[2] $incr$ denotes the increment procedure which is of type: $Number \to Number$.

Name	$\mathcal{L}$		$\mathcal{LL}$	
Constant	e	c	e	c
Identifier		x		x
Abstraction		$(\lambda\ x\ e)$		$(\lambda\ x\ e)$
Application		$(e\ e)$		$(e\ e)$
Binding		$(let\ x\ e\ e)$		$(let\ x\ e\ e)$
Conditional structure		$(if\ e\ e\ e)$		$(if\ e\ i\ e\ i\ e)$
Subtyping				$(sub\ \sigma\ \tau\ e)$

Table 5.5: Grammar of the $\mathcal{LL}$ language, x ranges on identifiers, i ranges on instantiations, τ ranges on monotypes and σ ranges on polytypes.

(b) $[\xi_i \to \tau_i]\{incr : N \to N, A\} \vdash 1 : N$

(c) $[\xi'_i \to \tau_i]\{incr : N \to N, A\} \vdash "foo" : N$

Lets focus on the third one: it is a constant so we have to use the CONST rule. That rule requires the verification of the assertion $typeOf("foo") = N$ which is false since $"foo"$ is of type string. By contradiction we can deduce judgments of the form $\{incr : N \to N, A\} \vdash (incr\ (if\ x\ 1\ "foo")) : N$ can not be derived in the $\mathcal{L}$ type system.

Remote context We will now see that type deductions on different conditional branches are independent. As an illustration, we present a program with two sequential conditional structures and its type derivation:

```
((lambda (x)
   (if y
       (incr x)
       (length x))) (if z
                        1
                        "foo"))
```

$$\text{ABS}\ \frac{\dots\ \dfrac{\text{Derived on paragraph LCOND rule.}}{\{x : \xi, y : B, z : B, incr : N \to N, length : S \to N\} \vdash (if\ y\ (incr\ x)\ (length\ x)) : N}}{\{y : B, z : B, incr : N \to N, length : S \to N\} \vdash (\lambda\ x\ (if\ y\ (incr\ x)\ (length\ x))) : \xi \to N} \quad *$$

$$\dots\ \frac{\text{Derived on paragraph LCOND rule.}}{\{y : B, z : B, incr : N \to N, length : S \to N\} \vdash (if\ z\ 1\ "foo") : \xi} \quad **$$

$$\text{APP}\ \frac{\overset{*}{\qquad}\quad\overset{**}{\qquad}}{\{y : B, z : B, incr : N \to N, length : S \to N\} \vdash ((\lambda\ x\ (if\ y\ (incr\ x)\ (length\ x)))\ (if\ z\ 1\ "foo")) : N}$$

This derivation show that the type deduction of the conditional branches inside the argument did not affect the type deduction of the conditional branches inside the body of the procedure.

Recapitulation In this section we introduced the $\mathcal{L}$ type system. We exposed in detail how the rule LCOND affects derivation and the status of type variables. Because of that new status, we justified the rule LGEN and explained why we want to accept only function as candidate for polymorphism.

5.5 The $\mathcal{LL}$ language

Conditional structure The $\mathcal{L}$ conditional structure should be interpreted exactly the same way of the LLScheme conditional structure. In the $\mathcal{L}$ expression: $(if\ e_{pred}\ i_{then}\ e_{then}\ i_{else}\ e_{else})$, e_{pred} represents the predicate, i_{then} the locus of the then branch, e_{then} the then branch itself, i_{else} the locus of the else branch and e_{else} the else branch itself.

$$\text{ID}\ \frac{\{x:\sigma\} \in A}{A \vdash x:\sigma} \qquad \text{CONST}\ \frac{typeOf(c) = \gamma}{A \vdash c:\gamma}$$

$$\text{ABS}\ \frac{A_x \cup \{x:\tau\} \vdash e:\tau'}{A \vdash (\lambda\ x\ e):\tau \to \tau'} \qquad \text{APP}\ \frac{A \vdash e:\tau' \to \tau \qquad A \vdash e':\tau'}{A \vdash (e\ e'):\tau}$$

$$\text{LGEN}\ \frac{A \vdash e:\forall \xi_1, \ldots . \tau \to \tau' \qquad \xi \notin free(A)}{A \vdash e:\forall \xi, \xi_1, \ldots . \tau \to \tau'} \qquad \text{LLSUB}\ \frac{A \vdash e:\sigma \qquad \tau \sqsubseteq \sigma}{A \vdash (sub\ \sigma\ \tau\ e):\tau}$$

$$\text{LET}\ \frac{A_x \cup \{x:\tau\} \vdash e:\sigma \qquad A_x \cup \{x:\sigma\} \vdash e':\tau' \qquad \tau \sqsubseteq \sigma}{A \vdash (let\ e\ e'):\tau'}$$

$$\text{LLCOND}\ \frac{A \vdash e:\tau \qquad [\xi' \mapsto \tau']A \vdash e':\tau''' \qquad [\xi'' \mapsto \tau'']A \vdash e'':\tau'''}{A \vdash (if\ e\ [\xi' \mapsto \tau']\ e'\ [\xi'' \mapsto \tau'']\ e''):\tau'''}$$

Table 5.6: The $\mathcal{LL}$ type system. The modifications done on the $\mathcal{L}$ type system are just meant to handle code annotations.

LLCOND rule This rule (Table 5.6) has exactly the same behavior the LCOND but it verifies the instantiations made on each branch are present inside the code. Below we present the static checking of the annotated $(if\ x\ 1\ "foo")$:

$$\text{LLCOND}\ \frac{\text{ID}\ \dfrac{x:B \in \{x:B\}}{\{x:B\} \vdash x:B} \qquad \text{CONST}\ \dfrac{typeOf(1):N}{\{x:B\} \vdash 1:N} \qquad \text{CONST}\ \dfrac{typeOf("foo"):S}{\{x:B\} \vdash "foo":S}}{\{x:B\} \vdash (if\ x\ [\xi \mapsto N]\,1\,[\xi \mapsto S]\,"foo"}$$

We stress the point that the rules presented at (Table 5.6) denote a type system and not the description of an algorithm responsible to insert code annotations.

Subtype expression In the theoretical part of this thesis, we preferred to put explicitly the polymorphic type and the instantiated monotype instead of the instantiation. But, $(sub\ \sigma\ \tau\ x)$ has exactly the same semantic as `(sub <locus> <id>)` ; both those annotation are used to implement polymorphic wrappers around monomorphic procedures.

LLSUB rule Again, the rule LLSUB has the same behavior as LSUB, it just verifies the code annotations are coherent with the deduced types. Below we present a derivation of the annotated polymorphic call $(id\ 1)$ where id is the identity procedure:

$$\text{APP}\ \frac{\text{LLSUB}\ \dfrac{\text{ID}\ \dfrac{id:\forall\xi.\xi \to \xi \in \{id:\forall\xi.\xi \to \xi\}}{\{id:\forall\xi.\xi \to \xi\} \vdash id:\forall\xi.\xi \to \xi} \qquad N \to N \sqsubseteq \forall\xi.\xi \to \xi}{\{id:\forall\xi.\xi \to \xi\} \vdash (sub\ (\forall\xi.\xi \to \xi)\ (N \to N)\ id):N \to N} \qquad \text{CONST}\ \dfrac{typeOf(1) = N}{\{id:\forall\xi.\xi \to \xi\} \vdash 1:N}}{\{id:\forall\xi.\xi \to \xi\} \vdash ((sub\ (\forall\xi.\xi \to \xi)\ (N \to N)\ id)\ 1):N}$$

5.6 Locus

Loci are nothing more than instantiations that are used in a very particular way. In iterative typing, the importance of that use fully justifies a dedicated name. In this section we provide a locus interface that will be used by the $\mathcal{LL}$ interpreter and the $\mathcal{L} \to \mathcal{LL}$ compiler exposed in the following sections.

Robinson unification Unification is a well known algorithm much used in type theory and particularly in Hindley-Milner type deduction. In this paragraph we will expose informally this algorithm, a complete formal presentation can be found at [14]. Two types are unified by an instantiation if it equalizes those two types. Here are some examples:

- $[\xi \mapsto String]\, unifies(\xi, String)$
- $[\xi_1 \mapsto Number, \xi_2 \mapsto \xi]\, unifies(\xi_1 \to \xi, Number \to \xi_2)$

The unification algorithm U, given a set of pairs of types, either fails or returns an instantiation that unifies each pair of types. Here are some application of the unification algorithm

- $U\ \{(\xi, Number), (String \to \xi, String \to Number)\} = [\xi \mapsto Number]$
- $U\ \{(\xi, Number), (String \to \xi, String \to Boolean)\} = fail$
- $U\ \{(\xi_1, Number), (String \to \xi_2, String \to Boolean)\} = [\xi_1 \mapsto Number, \xi_2 \to Boolean]$

Locus unification Loci are instantiations, as such they can be seen as a set of pairs of type. The unification of two loci, notated $\odot$, is given by applying Robinson algorithm on the union of all the locus pair; formally we have:

$$l_1 \odot l_2 = U\ (l_1 \cup l_2)$$

Locus enforcement We first define the operator $\triangleright$ that returns an instantiation on linked variables that unifies monotype parts of two types:

$$U(\tau, \tau') = [\xi_i \mapsto \tau_i]\{\xi_i\} \subset \{\xi_i'\}\ \} \Rightarrow \tau \triangleright \forall\{\xi_i'\}.\tau = [\xi_i \mapsto \tau_i]$$

Then we define the locus enforcement, notated $\oplus$, as the unification of the locus pairs and instantiation pairs:

$$l \oplus (\sigma, \tau) = l \odot (\tau \triangleright \sigma)$$

Closure The closure of a set of type variable A relatively to a locus l is the maximal subset of A such that the constraints inside the locus do not contain type variable out of the closure or do not contain type variable in the closure. Formally we have:

$$\left.\begin{array}{l} A_i \subseteq A \\ \forall p \in l.(free(p) \subseteq A_i) \vee (free(p) \cap A_i = \varphi) \end{array}\right\} \Rightarrow l \oslash A = \max_i A_i$$

The empty set and the set that contain all the type variables are closures for any locus.

Locus releasing When releasing a locus we must care to not remove constraints that contain not linked type variables. The releasing operator, notated $\ominus$ is characterized by the following relations:

$$\left\{\begin{array}{l} p \in l \wedge (free(p) \subseteq A) \Rightarrow p \notin (l \ominus A) \\ p \in l \wedge (free(p) \nsubseteq A) \Rightarrow p \in (l \ominus A) \\ p \notin l \Rightarrow p \notin (l \ominus A) \end{array}\right.$$

5.7 $\mathcal{LL}$ evaluator

In this section we present a $\mathcal{LL}$ evaluator written in Scheme. The evaluator is not completely implemented and we assume we have access to an interface to manipulate loci and dictionary.

Interfaces The presented evaluator is not totally implemented and we suppose the below interfaces are provided:

1. *Locus interface*: We suppose the locus operator defined in the previous are implemented:
   - `(define (locus:union locus1 locus2) ...)` : unification of two locus, implements the $\odot$ locus operator.
   - `(define (locus:enforce locus polytype monotype) ...)` : locus enforcement, implements the $\oplus$ locus operator.
   - `(define (locus:release locus polytype) ...)` : locus release, implements the $\ominus$ locus operator.
2. *Environment interface*: We supposed we implemented environments as well. As $\mathcal{LL}$ is free of side effect, we can model environments as simple non mutable dictionaries:
   - `(define (dico:add env id value) ...)` : returns a new environment with an updated `id` binding.
   - `(define (dico:lookup env id) ...)` : returns the value associated with `id` inside `env` .

Values The values manipulated by the evaluator are:

- *Constants*: either Scheme strings, Scheme numbers or Scheme booleans.
- *Primitive procedures*: a Scheme procedures that manipulate constants.
- *Monomorphic procedures*: classical structures to represent compound procedures:

```
(struct monoproc (env param body) #:mutable)
```

  This structure must be mutable so it can refer to itself through the environment to support recursion.

- *Polymorphic procedure*: a monomorphic procedure wrapped during the evaluation of a subtype expression:

```
(struct polyproc (polytype monotype monoproc))
```

  `polytype` corresponds to the general type scheme of the polymorphic procedure like $\forall\xi.\xi \to \xi$ for the identity function. `monotype` corresponds to the instantiated monotype inside a `let` body. `monoproc` is the normal compound procedure this polymorphic procedure wraps.

Structures Evaluation procedures return a combination of value and locus. To improve readability we define a dedicated structure for those return values:

```
(struct lval (locus value))
```

Utility The only utility procedure used is `tagged-list` that tells if the given expression is a list that starts with the given tag:

```
(define (tagged-list? expr tag)
  (and (list? expr) (not (null? expr)) (eq? (car expr) tag)))
```

Overall evaluation Expression evaluations start with a case analysis that dispatches the evaluation to specific procedures like `eval-const` :

```
(define (eval expr env locus)
  (cond ((tagged-list? expr 'lambda) (eval-abs expr env locus))
        ((tagged-list? expr 'let)    (eval-let expr env locus))
        ((tagged-list? expr 'if)     (eval-if expr env locus))
        ((tagged-list? expr 'sub)    (eval-sub expr env locus))
        ((symbol? expr)              (eval-id expr env locus))
        ((not (list? expr))          (eval-const expr env locus))
        (else                        (eval-app expr env locus))))
```

Evaluation procedures takes an expression, an environment and a locus ; they returns a modified locus and a value.

Simple evaluations In this paragraph we expose the specific evaluations that are very simple and do not affect the locus:

- *Constant evaluation*

```
(define (eval-const expr env locus)
  (lval locus expr))
```

- *Identifier evaluation*: the binding inside the environment is returned:

```
(define (eval-id expr env locus)
  (lval locus (dico:lookup env expr)))
```

- *Abstraction evaluation*: unlike the environment, the locus is not stored inside the compound procedure representation. That suggests the application will be performed with a future locus not the current one. This is expected since we want the locus to follow exactly the computation flow.

```
;; (lambda <param> <body>)
(define (eval-abs expr env locus)
  (let ((param (car (cdr expr)))
        (body  (car (cdr (cdr expr)))))
    (lval locus (monoproc env param body))))
```

Conditional evaluation Although being the core of iterative typing, the evaluation of conditional structures is actually fairly simple. First we extract all the informations contained inside the expression. Then we evaluate the predicate and get back both a modified locus and a value. Following that value, we unify the predicate locus with the then locus or the locus and evaluate the then or the else branch with the unified locus (Figure 5.1):

```
;; (if <pred> <then-locus> <then> <else-locus> <else>)
(define (eval-if expr env locus)
  (let* ((pred       (car (cdr expr)))
         (then-locus (car (cdr (cdr expr))))
         (then       (car (cdr (cdr (cdr expr)))))
         (else-locus (car (cdr (cdr (cdr (cdr expr))))))
         (else       (car (cdr (cdr (cdr (cdr (cdr expr)))))))
         (pred-lval  (eval pred env locus))
         (pred-locus (lval-locus pred-lval))
         (pred-value (lval-value pred-lval)))
    (if (eq? pred-value #f)
        (eval else env (locus:union locus else-locus))
        (eval then env (locus:union locus then-locus)))))
```

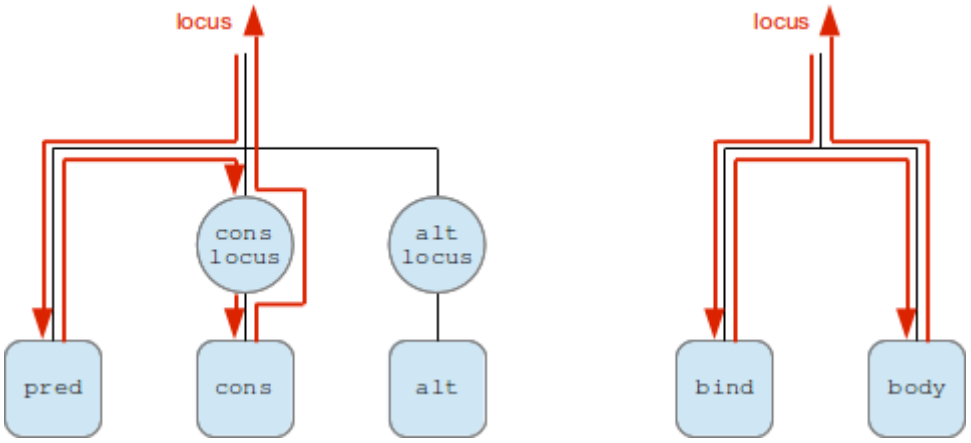

Figure 5.1: On the left, the locus flow inside an evaluation of a conditional structure whom predicate has not been evaluated to `#f` . On the right, the locus flow inside an evaluation of a binding.

Let evaluation First we extract all the information contained inside the expression. Then we evaluate the binding and get back both a modified locus and a value. Now we have the binding value we can extend the environment. In case the binding is a procedure definition, we must support recursion and update the environment of the returned monomorphic procedure. The body of the let can now be evaluated with the extended environment and the modified locus:

```
;; (let <param> <bind> <body>)
(define (eval-let expr env locus)
  (let* ((param      (car (cdr expr)))
         (bind       (car (cdr (cdr expr))))
         (body       (car (cdr (cdr (cdr expr)))))
         (bind-lval  (eval bind env locus))
         (bind-value (lval-value bind-lval))
         (bind-locus (lval-locus bind-lval))
         (ext-env    (dico:add env param bind-value)))
    (when (tagged-list? (car (cdr (cdr expr))) 'lambda)
      (set-monoproc-env! bind-value ext-env))
    (eval body ext-env bind-locus)))
```

Subtyping evaluation This rule allows to instantiate polymorphic procedures. After evaluating the supposed polymorphic procedure it returns a `poly` structure made of information contained inside the given expression:

```
;; (sub <polytype> <monotype> <proc>)
(define (eval-sub expr env locus)
  (let* ((polytype (car (cdr expr)))
         (monotype (car (cdr (cdr expr))))
         (proc (car (cdr (cdr (cdr expr)))))
         (proc-lval (eval proc env locus))
         (proc-locus (lval-locus proc-lval))
         (proc-value (lval-value proc-lval)))
    (lval proc-locus (polyproc polytype monotype proc-value))))
```

Application evaluation First the argument is evaluated then a case analysis on the procedure is performed:

```
;; (<proc> <arg>)
(define (eval-app expr env locus)
  (define (apply-prim proc-value arg-value locus) ...)
  (define (apply-mono proc-value arg-value locus) ...)
  (define (apply-poly proc-value arg-value locus) ...)
  (let* ((proc       (car expr))
```

```
        (arg        (car (cdr expr)))
        (proc-lval  (eval proc env locus))
        (proc-locus (lval-locus proc-lval))
        (proc-value  (lval-value proc-lval))
        (arg-lval   (eval arg env proc-locus))
        (arg-locus  (lval-locus arg-lval))
        (arg-value  (lval-value arg-lval)))
   (cond ((procedure? proc-value) (apply-prim proc-value arg-value arg-locus))
         ((monoproc? proc-value) (apply-mono proc-value arg-value arg-locus))
         ((polyproc? proc-value) (apply-poly proc-value arg-value arg-locus))
         (else (error "cannot apply")))))
```

1. *Primitive*: Primitive procedure does not modify the locus (Figure 5.2), we simply wrap the result of the primitive application inside a `lval` structure:

```
(define (apply-prim proc-value arg-value locus)
  (lval locus (proc-value arg-value)))
```

2. *Monomorphic*: This application is very similar to the application of standard Scheme compound procedure (Figure 5.2), we only have to give the current locus to perform the body evaluation:

```
(define (apply-mono proc-value arg-value locus)
  (eval (monoproc-body proc-value)
        (dico:add env (monoproc-param proc-value) (arg-value))
        locus))
```

3. *Polymorphic*: This application actually implements the polymorphic wrapper of monomorphic procedure (Figure 5.2), we must enforce the locus before the monomorphic application and release it after:

```
(define (apply-poly proc-value arg-value locus)
  (let* ((monoproc       (polyproc-monoproc proc-value))
         (polytype       (polyproc-polytype proc-value))
         (monotype       (polyproc-monotype proc-value))
         (enforced-locus (locus:enforce locus polytype monotype))
         (lval-mono      (apply-mono monoproc arg-value enforced-locus)))
    (lval (locus:release (lval-locus lval-mono) polytype)
          (lval-value lval-mono))))
```

5.8 Compilation $\mathcal{L} \rightarrow \mathcal{LL}$

Objectives The compiler $\mathcal{L} \rightarrow \mathcal{LL}$ has been developed following those two main objectives:

1. The type deduction inside a conditional branch should never affect the type deduction out of that branch.
2. The type generalization of a procedure should concern all the type variables introduced during the body compilation that does not affect the external type variables.

The first objective is a translation of the iterative leitmotiv which is to avoid making restrictive assumptions during the static code analysis. The second objective maximize the constraints releasing after a polymorphic call. Lets analyze the following $\mathcal{LL}$ pseudo code that did not generalized `b` for the polymorphic procedure `useless-read` :

```
(define (useless-read x)
  (sub (\-/in.()->in) (()->b) read)
  x)

((sub (\-/a.a->a) (Num->Num) useless-read) 1)
((sub (\-/a.a->a) (Str->Str) useless-read) "foo")
```

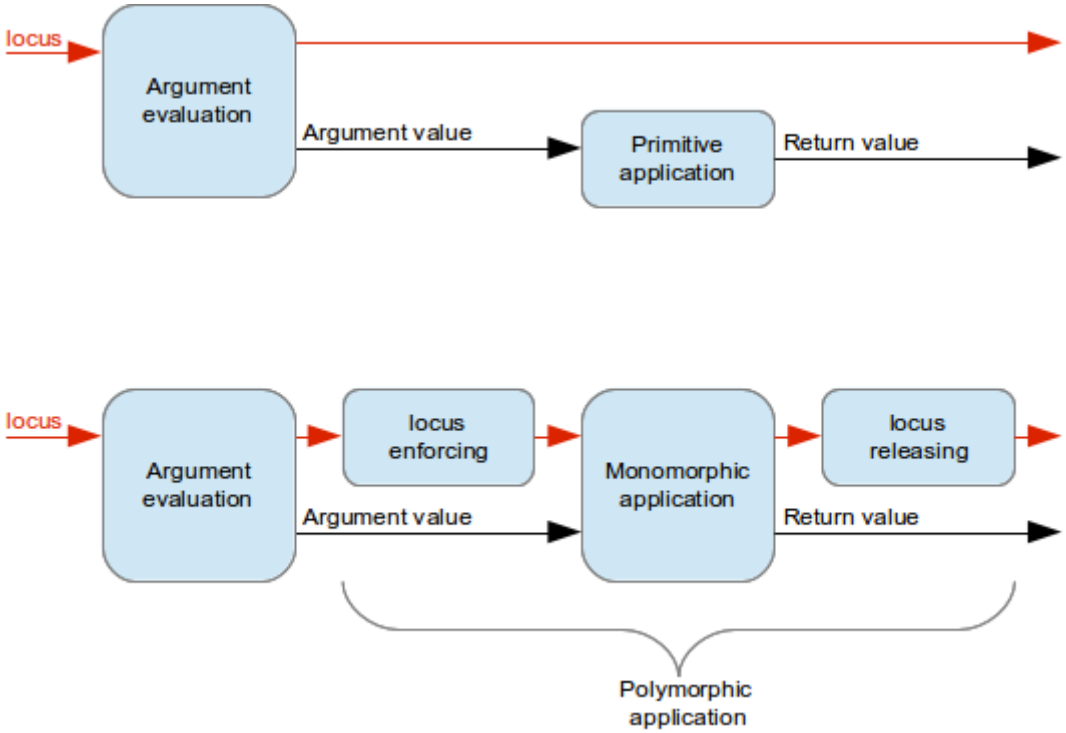

Figure 5.2: The evolution of the locus and the value during the three different kind of applications: primitive, monomorphic and polymorphic.

The type of `useless` has been inferred to `-/a.a->a` . As a result, the constraint on `b` is not released after the first polymorphic call and the second input must have the same type as the first one. This is not a correct behavior for polymorphic procedures. The problem is avoided if the type of `useless` is inferred to `-/a,b.a->a` . This example highlights the necessity to link all the type variables introduced inside the body of a polymorphic procedure that does not affect the external type variables.

Markers The second objective could be fulfilled by storing the generated variables during the compilation. However, this solution does not scale well with the delayed compilation of conditional structure. Instead of using explicit lists we preferred to use markers to define sets of linked variables. In consequence, the polytype $\forall a_1, \ldots, a_N.\tau$ is now represented by: $\forall a.\tau$ which means that all the type variables marked with a are linked. In other words, when two type variables share the same marker that means they are linked into the same polytype.

Interfaces

- *Type interface*: Beside markers this type interface implements very standard concepts concerning types:
  1. `(define (type:type-of constant) ...)` : implementation of $typeOf$ function.
  2. `(define (type:make-poly marker monotype) ...)` : creates a polytype from a marker and a monotype.
  3. `(define (type:poly? type) ...)` : tells if the given argument is a polytype.
  4. `(define (type:sub polytype) ...)` : returns a generic instance of the given polytype.
  5. `(define (type:make-abs arg-type res-type) ...)` : the functional type constructor: $\rightarrow$.
  6. `(define (type:abs-arg type) ...)` : returns the argument type of a function type.
  7. `(define (type:abs-res type) ...)` : returns the result type of a function type.

8. `(define (type:make-marker!) ...)` : returns a never-used marker.
9. `(define (type:make-var! marker) ...)` : returns a never-used type variable containing the given marker.

- *Locus interface*:
  1. `(define (locus:make-empty) ...)` : returns a locus without any constraint.
  2. `(define (locus:union locus1 locus2) ...)` : classic locus unification
  3. `(define (locus:add locus type1 type2) ...)` : unification of a locus and the new constraint `[type1 => type2]` .
  4. `(define (locus:close! locus marker) ...)` : create a closure inside the given locus for the given marker. The marker of some variable may be mutated in the process.

$$\texttt{[a1 => b2, b3 => String]} \xrightarrow{\texttt{close! a}} \texttt{[a1 => a2, b3 => String]}$$

  5. `(define (locus:project locus ass) ...)` : instantiate all the free variables contained in the assumptions following the given locus.
- *Dictionary interface*: Same interface as for the environment in the $\mathcal{LL}$ evaluator.

Structures To improve readability, we dedicate a structure for the return values of the compilers:

```
(struct idiom (type expr locus))
```

Overall compilation As for the interpretation, the overall compilation start with a case analysis that dispatches the compilation to specific procedures like `compile-const` :

```
(define (compile expr ass marker)
  (cond ((tagged-list? expr 'lambda) (compile-abs expr ass marker))
        ((tagged-list? expr 'let)    (compile-let expr ass marker))
        ((tagged-list? expr 'if)     (compile-if expr ass marker))
        ((symbol? expr)              (compile-id expr ass marker))
        ((not (list? expr))          (compile-const expr ass marker))
        (else                        (compile-app expr ass marker))))
```

Constant compilation No constraints are generated and so the empty locus is returned. The constant remains intact and its type is given by `type:type-of` which implements the $typeOf$ function:

```
;; CONST => CONST
(define (compile-const expr ass marker)
  (idiom (type:type-of expr)
         expr
         (locus:make-empty)))
```

Abstraction evaluation We assume that the type of the argument is an arbitrary type variable which produces the assumption: `param:arg-var` . `ext-ass` is the union of this new assumption with the old ones. The body is then compiled with `ext-ass` and `marker` :

```
;; (lambda <param> <body>) => (lambda <param> <compiled-body>)
(define (compile-abs expr ass marker)
  (let* ((param ((car (cdr expr))))
         (body ((car (cdr (cdr expr)))))
         (arg-var (type:make-var! marker))
         (ext-ass (dico:add ass param arg-var))
         (body-idiom (compile body ext-ass marker)))
    (idiom (type:make-abs arg-var (idiom-type body-idiom))
           (list 'lambda param (idiom-expr body-idiom))
           (idiom-locus body-idiom))))
```

Application compilation We compile separately the procedure and the argument. The overall type is given by the return type of the procedure. The overall locus is given by the unification of the procedure locus, the argument locus and the constraint that links the expected argument with the actual argument:

```
;; (<proc> <arg>) => (<compiled-proc> <compiled-arg>)
(define (compile-app expr ass marker)
  (let* ((abs (car (expr)))
         (arg (car (cdr expr)))
         (abs-idiom (compile abs ass marker))
         (arg-idiom (compile arg ass marker)))
    (idiom (type:abs-res (idiom-type abs-idiom))
           (list (idiom-expr abs-idiom) (idiom-expr arg-idiom))
           (locus:add (locus:union (idiom-locus abs-idiom) (idiom-locus arg-idiom))
                      (type:abs-arg (idiom-type abs-idiom)) (idiom-type arg-idiom)))))
```

Identifier compilation We first lookup for type associated to the identifier inside the assumption set. Then we differentiate the case when the binding is a monotype or polytype:

```
;; SYMBOL => SYMBOL
;;        => (<polytype> <monotype> SYMBOL)
(define (compile-id expr ass marker)
  (define (compile-poly-id polytype) ...)
  (define (compile-mono-id monotype) ...)
  (let ((bind (dico:lookup ass expr)))
    (if (type:poly? bind)
        (compile-poly-id bind)
        (compile-mono-id bind))))
```

- *Monotype binding*: An idiom is returned containing the monotype, the symbol and an empty locus (no constraints are generated):

```
(define (compile-mono-id monotype)
  (idiom monotype
         expr
         (locus:make-empty)))
```

- *Polytype binding*: Again, no constraints are generated ; both the polytype and a generic instance the polytype are stored inside a *sub* expression:

```
(define (compile-poly-id polytype)
  (let ((monotype (type:sub polytype marker)))
    (idiom monotype
           (list 'sub polytype monotype expr)
           (locus:make-empty))))
```

Binding compilation We first extract the different parts of the let expression and then perform a case analysis on whether the binding is an abstraction or not:

```
;; (let <param> <bind> <body>) => (let <param> <compiled-bind> <compiled-body>)
(define (compile-let expr ass marker)
  (define (compile-let-abs param bind body) ...)
  (define (compile-let-rest param bind body) ...)
  (let ((param (car (cdr expr)))
        (bind  (car (cdr (cdr expr))))
        (body  (car (cdr (cdr (cdr expr))))))
    (if (tagged-list? bind 'lambda)
        (compile-let-abs param bind body)
        (compile-let-rest param bind body))))
```

1. *Other binding*: The binding is not a procedure definition, the expression (*let x bind body*) is equivalent to ((*lambda x body*) *bind*):

```
(define (compile-let-rest param bind body)
  (let* ((bind-idiom (compile bind ass))
         (ext-ass (dico:add ass param (idiom-type bind-idiom)))
         (body-idiom (compile body ext-ass)))
    (idiom (idiom-type body-idiom)
           (list 'let param (idiom-expr bind-idiom) (idiom-expr body-idiom))
           (locus:union (idiom-locus bind-idiom) (idiom-locus body-idiom)))))
```

2. *Procedure binding*: The binding is a procedure and the we must handle recursion and polymorphism.

   - `bind-marker` : a never-use marker to create type variables inside the binding.
   - `bind-var` : the type the binding perceive from itself. Since the binding perceive itself as a monomorphic type, the recursion is monomorphic.
   - `mono-ext-ass` : the assumption set inside which the binding will be compiled.
   - `bind-idiom` : the result of the binding compilation with the extended assumptions `mono-ext-ass` and the new marker `bind-marker` .
   - `bind-monotype` : the type of the binding returned by the compilation.
   - `bind-polytype` : the generalization of `bind-monotype` .
   - `bind-locus` : the locus generated by the binding compilation with the additional constraint `bind-var => bind-monotype` .
   - `poly-ext-ass` : the assumption set inside which the body will be compiled.
   - `body-idiom` : the result of the body compilation with the extended assumption `poly-ext-ass` and the old marker `marker` .

   The general type of the expression is monotype returned by the body compilation. The compiled expression is: `(let <param> <compiled-bind> <compiled-body>)` . The total locus is the union of `bind-locus` and the locus returned by the body compilation:

```
(define (compile-let-abs param bind body)
  (let* ((bind-marker (type:make-marker!))
         (bind-var (type:make-var! bind-marker))
         (mono-ext-ass (dico:add ass param bind-var))
         (bind-idiom (compile bind mono-ext-ass bind-marker))
         (bind-monotype (idiom-type bind-idiom))
         (bind-polytype (type:make-poly marker bind-monotype))
         (bind-locus (locus:add (idiom-locus bind-idiom) bind-var bind-monotype))
         (poly-ext-ass (dico:add ass param bind-polytype))
         (body-idiom (compile body poly-ext-ass marker)))
    (locus:close! bind-locus marker)
    (idiom (idiom-type body-idiom)
           (list 'let param (idiom-expr bind-idiom) (idiom-expr body-idiom))
           (locus:union bind-locus (idiom-locus body-idiom)))))
```

Conditional structure compilation The compilation of conditional structure is first delayed inside a structure that contains useful information like the current marker and the current environment:

```
;; (if <pred> <then> <else>)
;;   => (delay <type-var> <marker> <ass> <compiled-pred> <then> <else>)
(define (compile-if expr ass marker)
  (let* ((pred (car (cdr expr)))
         (then (car (cdr (cdr expr))))
         (else (car (cdr (cdr expr))))
         (pred-idiom (compile pred ass))
         (var (type:make-var! marker)))
    (idiom var
           (list 'delay var marker ass (idiom-expr pred-idiom) then else)
           (idiom-locus pred-idiom))))
```

The idea that lurks behind that delayed compilation is simple: the direct context must be completely analyzed before we can analyze each branch separately (Figure 5.3). By doing this we can ensure both branch are compatible with the direct context. This would not be the case if the syntax tree was constructed purely depth first.

Delay compilation This is the actual compilation of the conditional structure. Given the locus that characterizes the direct context we search for delayed conditional structures:

```
(define (finalize tree locus)
  (cond ((tagged-list? tree 'delay) (finalize-delay tree locus))
        ((list? tree) (map (lambda (tree) (finalize tree locus)) tree))
        (else tree)))
```

Once a delayed conditional structure has been found we first extract all the information. Then we incorporate the given locus, that represents the direct context, inside the stored assumption set. We then evaluate and finalize separately each branch:

```
;; (delay <var> <marker> <ass> <compiled-pred> <then> <else>)
;; => (if <compiled-pred> <then-locus> <compiled-then> <else-locus> <compiled-else>)
(define (finalize-delay delay locus)
  (let* ((var           (car (cdr delay)))
         (marker        (car (cdr (cdr delay))))
         (ass           (car (cdr (cdr (cdr delay)))))
         (compiled-pred (car (cdr (cdr (cdr (cdr delay))))))
         (then          (car (cdr (cdr (cdr (cdr (cdr (cdr delay)))))))
         (else          (car (cdr (cdr (cdr (cdr (cdr (cdr (cdr delay))))))))
         (projected-ass (locus:project locus ass))
         (then-idiom    (compile then projected-ass marker))
         (then-locus    (locus:add (idiom-locus then-idiom) var (idiom-type then-idiom)))
         (then-expr     (finalize (idiom-expr then-idiom) then-locus))
         (else-idiom    (compile else projected-ass marker))
         (else-locus    (locus:add (idiom-locus else-idiom) var (idiom-type else-idiom)))
         (else-expr     (finalize (idiom-expr else-idiom) else-locus)))
    (list 'if compiled-pred then-locus then-expr else-locus else-expr)))
```

5.9 Conclusion

We opened this chapter with some very common concepts in type theory. We then introduced a classic language to deal with type system ; we called it $\mathcal{L}$. We also demonstrated that $\mathcal{L}$, beside assignments, has the same expressiveness as LScheme. Afterwards, an Hindley-Milner type system was introduced as a strong theoretical base for iterative type system. The next section exposed the $\mathcal{L}$ type system as a slightly modified Hindley-Milner type system. Then the $\mathcal{LL}$ language, which is the annotated version of $\mathcal{L}$, and its type system were presented. The following section defined some operations on loci. Finally we had everything in hands to expose the compilation $\mathcal{L} \rightarrow \mathcal{LL}$ and the evaluation of $\mathcal{LL}$ expressions.

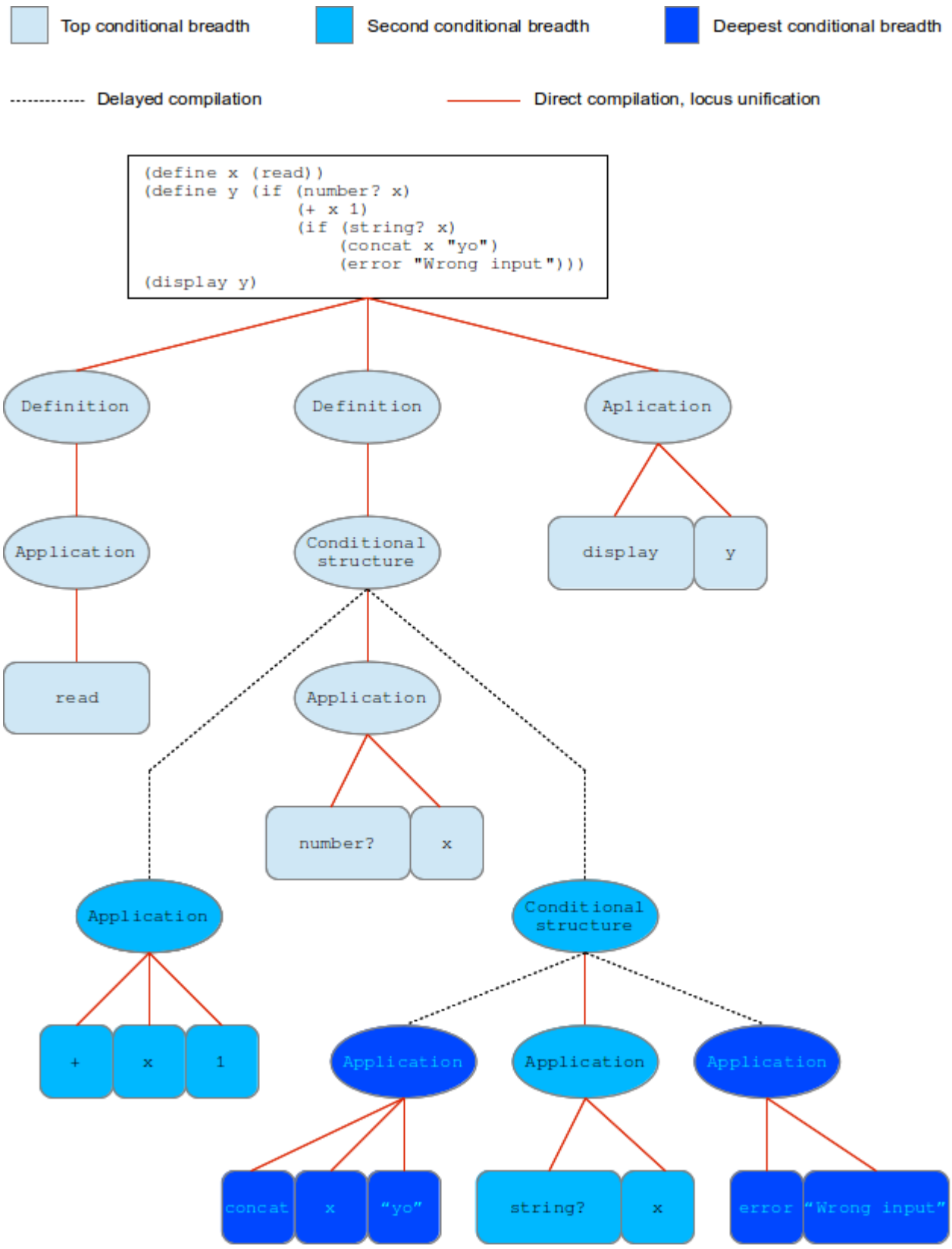

Figure 5.3: The syntax tree as constructed by the compiler $\mathcal{L} \to \mathcal{LL}$ for the upper program. Each color represents a different conditional breadth. The current breadth is totally compiled before jumping to deeper breadth. Inside a breadth, all the generated loci are unified.

Chapter 6

Related Works

In this chapter, we will introduce soft and gradual typing, two newly created typings that attempt to mix static and dynamic typings. We then compare iterative typing with those researches.

6.1 Soft Typing

Introduction This section is strongly inspired from [18] and [19]. Soft typing is an attempt to increase the static safety of dynamic programs. Soft typing first performs a static type inference. When an primitive application can not be proven safe, a runtime check is inserted. Soft typing can be seen as a dynamic typing that statically performs as many type checks as possible.

Soft type system Soft typing uses an elaborate type system based on union types, a typing discipline that can express that a variable can have multiple types at the execution. For instance $Number|Null$ is the type of an expression that may either be of type number or be the null constant. This expressive power allows soft typing to handle many common Scheme patterns. The specificity of the soft type system is encapsulate by two rules about applications (Figure 6.1). The first application rule, labeled APP, concerns the applications that have been proven safe. The second application rule, labeled CHECK-APP, concerns unsafe applications that must be checked at runtime. When CHECK-APP is not used at all, the program is statically type safe.

Example Here is the the `flatten` function, it takes a tree and return list of all the leaves:

```
(define (flatten xs)
  (cond ((null? xs) null)
        ((pair? xs) (append (flatten (car xs)) (flatten (cdr xs))))
        (else (list xs))))
(flatten '(1 (2 3) 4))
```

```
'(1 2 3 4)
```

The authors of [19] present the above program as a success story because it is statically type safe for soft typing. In comparison, iterative typing cannot support such program because it uses heterogeneous lists.

Criticisms Although able to remove many runtime checks from Gabriel benchmarks, which standard benchmarks to validate a lisp implementation, soft typing suffer from severe criticisms:

$$[\text{APP}]\ \frac{A \vdash e_1 : (\tau_2 \to \tau_1) \cup \varphi \qquad A \vdash e_2 : \tau_2}{A \vdash (app\ e_1\ e_2) : \tau_1} \qquad [\text{CHECK-APP}]\ \frac{A \vdash e_1 : (\tau_2 \to \tau_1) \cup \tau_3 \qquad A \vdash e_2 : \tau_2}{A \vdash (check - app\ e_1\ e_2) : \tau_1}$$

Table 6.1: The two application rules of the soft type system. The left one concerns safe applications, the right one concerns unsafe applications that must be checked at the execution.

$$\frac{A \vdash e_1 : ?\qquad A \vdash e_2 : \tau_2}{A \vdash (e_1\ e_2) : ?} \qquad \frac{A \vdash e_1 : \tau \to \tau' \qquad A \vdash e_2 : \tau_2 \qquad A \vdash \tau \sim \tau_2}{A \vdash (e_1\ e_2) : \tau}$$

Figure 6.1: Two gradual rules about applications that show how dynamic type is handled.

1. *Unnecessary runtime checks*: The proposed type system does not succeed to infer the precise type of some much used procedures like `map` . As a result every `map` call will generate runtime checks even those who are trivially safe like `(map (lambda (x) (+ x 1)) (list 1 2 3))` .
2. *Expressiveness restriction*: Though very permissive, the soft type system still rejects some well typed programs. In particular, the `if` expressions are statically analyzed ; the two branches must be type compatible.

6.2 Gradual Typing

Introduction This section is strongly inspired from[16]. Gradual typing is meant to perform a smooth transition from scripts to static programs. So, as soft typing, gradual typing also combines static and dynamic typing. But their approach are radically different. When soft typing provides unsafe primitive application, gradual typing provides a so called dynamic type that always pass static checks. This type is annotated ? and represents values that can be anything at the execution. Variables annotated with such tag will generate casts and as many runtime checks.

Consistency relation The consistency relation, notated ~, replaces the subtyping relation usually used in type systems. Saying that a type must never fail a static checks is equivalent to say it must be compatible with any other type. This justifies the following axioms:

$$? \sim \tau \qquad \tau \sim ? \qquad \tau \sim \tau \qquad \frac{\tau_1 \sim \tau_2 \qquad \tau_1' \sim \tau_2'}{\tau_1 \to \tau_1' \sim \tau_2 \to \tau_2'}$$

Gradual type system Gradual typing is based on the simply typed lambda calculus [1]. But type annotations are not mandatory ; the dynamic type is inserted every where annotations are missing. In other word: "$(\lambda\ x\ e)$" is a syntactic sugar for "$(\lambda\ (x : ?)e$". As explain before, the gradual type system use the consistency relation instead of the classic subtype relation. In (Figure 6.1) are exposed the most representative rules of gradual typing.

Cast insertion Everywhere applications statically involve the dynamic type, casts are inserted. For instance the wrapping of the increment procedure:

$$(\lambda\ (x)\ (increment\ x)) \xrightarrow{compilation} (\lambda\ (x : ?)\ (increment\ \langle Number \rangle x))$$

The annotation $(x : Number)$ would have prevented the insertion of the cast $\langle Number \rangle x$.

Compilation Gradual typing provide an algorithm that automatically insert type annotations so the programmer does not have to manually sort the variables that should be statically checked and the variable that should be dynamically checked.

Result

1. The execution of compiled program (with inserted casts) can not lead to an uncaught type error.
2. If a program free of the dynamic type is entirely statically checked. In other words, if such program pass the static then it is type safe.
3. After a simple wrapping manipulation, all the programs are accepted. The gradual typing can be as expressive as a pure dynamically typed language based on the lambda calculus.

6.3 Comparison with iterative typing

Runtime checks Lets consider a simple Scheme code, that asks the user a radius the compute the circumference and the surface of the associated circle:

```
(define radius        (read))
(define pi            3.14)
(define circumference (* 2 (* pi radius)))
(define surface       (* pi (* radius radius)))
```

Each exposed typing techniques insert runtime checks in its own way:

1. *Soft typing compilation* The return type of `read` is an union of string, number, etc. Therefor any primitive call that involves `radius` must be checked at runtime:

   ```
   (define radius        (read))
   (define pi            3.14)
   (define circumference (* 2 (CHECK-MULT pi radius)))
   (define surface       (pi (CHECK-MULT radius radius)))
   ```

   The uncertainty on `radius` led to two unsafe primitive applications. If the user entered a wrong input, the error is detected at the `circumference` definition.

2. *Gradual typing compilation* The return type of `read` is the dynamic type. Therefor any applications, that involves `radius` and requires non dynamic types as argument, generates runtime cast on `radius` :

   ```
   (define radius:?             (read))
   (define pi:Number            3.14)
   (define circumference:Number (* 2 (pi <Number>radius)))
   (define surface:Number       (* pi (<Number>radius <Number>radius)))
   ```

   The checks that ensure `radius` is a number is performed thrice. If the user did not enter a number, the error is detected at the first use, at the `circumference` definition.

3. *Iterative typing compilation* Iterative typing detect `read` should return a number, only one check is required at the `read` call:

   ```
   (define radius        ((in => Number) read))
   (define pi            3.14)
   (define circumference (* 2 (* pi radius)))
   (define surface       (* pi (* radius radius)))
   ```

   The type of `radius` is only checked once right at the read primitive call.

As we can see, Iterative typing have the ability to summarize and move upward the checks inserted by soft and gradual typing. As a result, less runtime checks are performed and they are performed earlier on more relevant places.

Expressiveness The cost of having less runtime checks on better places is a restricted expressiveness. Here are examples of valid Scheme programs that are not currently accepted by iterative typing:

- *Polymorphism*: polymorphic use of argument inside a procedure:

  ```
  (define (use-poly f)
    (f 1)
    (f "foo"))
  (use-poly (lambda (x) x))
  ```

  ```
  "foo"
  ```

- *Heterogeneous lists*: manipulation of heterogeneous lists:

```
(define (use-hetero xs)
  (cond ((null? xs) null)
        ((string? (car xs)) (cons (concat "yo " (car xs)) (use-hetero (cdr xs))))
        ((number? (car xs)) (cons (* 2 (car xs))) (use-hetero (cdr xs)))
        (else (error "non handeled type"))))
```

```
'("yo foo" 2 "yo bar" 4 6)
```

Type tag Gradual and soft typing can both be seen as dynamic typing which perform as many checks as possible during a static code analysis. Iterative typing goes the other way around, it is a static typing that delays as few checks as possible at runtime. Soft and gradual typings both use type tags to perform runtime checks during the interpretation while the iterative interpreter don't need them to ensure safety. The only places where the schemes primitives like `number?` and `string?` occur are inside the implementation of LLScheme primitives. For instance the LLScheme primitive `string?` calls the Scheme primitive `string?` .

6.4 Conclusion

Soft and gradual typings both propose an interesting mix between static and dynamic typings. Iterative typing succeed to summarize soft and gradual runtime checks and move them on an upper level. The cost having a earlier error detection is a diminished expressiveness.

Chapter 7

Future Works

7.1 Overloading

This section is about how simulate overloading behavior with the `if-viable` iterative special form. A lot of investigations are needed to understand all the consequences of `if-viable` special form. One consequence that makes the `if-viable` very dangerous is that it introduces a new semantic that does not exist in Scheme, so this extended iterative typing is not only about detecting errors earlier but also express new concepts.

Overloading Overloading is a very important feature of most modern programming languages that allows creating several methods with the same name which differ from each other in the type of inputs and the output of the function. That is, the concrete body is defined by the calling context at compile time. In the following Java program, the rectangle constructor is overloaded which allows creating rectangles from two points or with two integers:

```
class Rectangle () {
    private Point a;
    private Point b;
    // Construct a rectangle defined by the given points.
    Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }
    // Construct a rectangle with one of its vertices located on the origin.
    Rectangle(int width, int height) {
        a = new Point(0,0);
        b = new Point(width, height);
    }
}
```

When overloading on inputs is present on most mainstream languages, output overloading is less common because it produces ambiguities. One notable exception being Haskell that does support output overloading trough type classes[11]. Lets suppose Java supports output overloading and we want to simplify the use of the `Random` class, we would then write the following code:

```
int random() {
    (new Random()).nextInt();
}

float random() {
    (new Random()).nextFloat();
}

long random() {
    (new Random()).nextLong();
}

void test() {
```

```
    System.out.println(random()); // which random body???
}
```

Indeed, the use of `random` inside `test` is ambiguous and that is why output overloading, although handy, is often avoided.

Viability tests The iterative `if-viable` must not be confused with the standard `try` constructions. The `try` construction performs code until an exceptions is raised and call the exception handler. On the other hand, iterative typing does not execute any code, it just tests if the first branch is viable inside the current context. In other words, `if-viable` is semantically equivalent to an iterative `if` containing a test about the viability of the first branch as a predicate:

```
(if-viable <attempt>
           <alternative>)
```

Is equivalent to:

```
(if <attempt viable?>
    <attempt>
    <alternative>)
```

Iterative typing With early error detection, iterative typing is able to grab some information about the calling context. This is achieved be proposing several bodies inside a `try` structure, iterative typing will loop through those proposition until it find one that is viable:

```
(define (<name> <args>)
  (if-viable <body1>
             ...
             <bodyN>))
```

If iterative typing reaches the last proposition, it will be executed even if not viable. Notice that this naturally resolves the ambiguities usually introduced by output overloading.

Knowing your arguments For input overloading, the iterative `if-viable` structure is close to regular `try` structures. That is, the viability of the proposition itself is tested not its future consequences:

```
(define pimp-my-arg (lambda (x)
                      (if-viable (concat x "bunta")
                                 (+ x 3))))
(pimp-my-arg 1)
(pimp-my-arg "gama ")
```

```
4
"gama bunta"
```

Knowing your result For output overloading, the iterative `if-viable` has a different behavior than regular `try` structures. That is, the future consequences of the proposition is tested:

```
(define pimp-my-result (lambda ()
                (if-viable "party"
                           1)))
(+ (pimp-my-result) 2)
(concat (pimp-my-result) " rocking")
```

```
3
"party rocking"
```

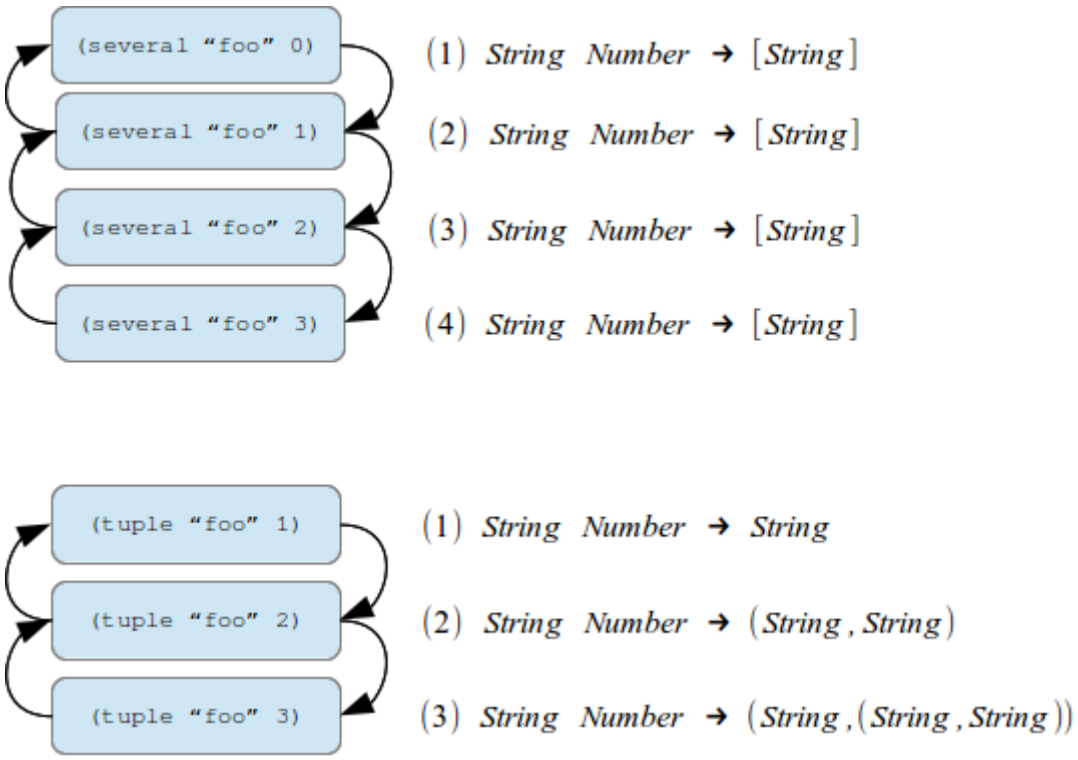

Figure 7.1: In a monomorphic recursion, the recursive call are always perform on the same type. In a polymorphic recursion the type may vary from one recursive call to an other.

7.2 Polymorphic recursion

Monomorphic recursion For now, iterative typing implements monomorphic recursion. Inside a monomorphic recursive computation, types are not allowed to change: each recursive call accepts the same argument and result type. The factorial mathematic function is a typical example of monomorphic recursion ; the recursive procedure always takes a number as argument and return a number as well:

```
;; fac :: Number -> Number
(define (fac n)
  (if (= n 0)
      1                       ; return a number
      (* n (fac (- n 1))))) ; return a number
```

It is also possible to produce polymorphic procedure that use monomorphic recursion. Below we propose a Scheme program that defines a procedure which creates a list containing `n` times the data `x` :

```
;; several :: \-/ a . a Number -> [a]
(define (several x n)
  (if (= n 0)
      null                              ; return a polymorphic constant
                                        ; that is interpreted as [a]
      (cons x (several x (- n 1))))) ; return [a]
```

This procedure is polymorphic because it accepts the call `(several 0 10)` but also the call `(several "foo" 10)` . Nevertheless, the type variable a is defined only once at the top level call and never change during the recursive computation. If `(several "foo" 3)` is invoked, the procedure `several` will always have the type $String\ Number \rightarrow [String]$ (Figure 7.1).

Polymorphic recursion On the opposite way, polymorphic recursion does allow types to change inside the recursive computation. Generally, type inference for polymorphic recursion is undecidable[7]. So it is not possible to statically decide whether a polymorphic recursion is type safe or not. Below,

we define the `tuple` procedure which is somehow similar to the `several` procedure but create a tuple instead of a list:

```
;; tuple :: \-/ a . a Number -> ???
(define (tuple x n)
  (if (= n 1)
      x                              ; return a
      (pair x (tuple x (- n 1))))) ; return (a,???)
```

Through this polymorphic recursion a nested pair is created. To known the nesting depth and the precise type return by a `tuple` call, we actually need to execute the code. As Scheme is Turing complete this execution may never return, as is the case for the call `(tuple "foo" 0)` .

Heterogeneous list We can interpret heterogeneous lists as nested pairs that end with the `null` value. Below we give a small Scheme program that manipulate heterogeneous lists that way:

```
;; evil-list :: \-/ a . Number -> a
(define (evil-list n)
  (cond ((<= n 0) null)
        ((prime? n) (cons "prime" (evil-list (- n 1))))
        (else (cons n (evil-list (- n 1))))))

;; evil-use :: \-/ a . a -> String
(define (evil-use xs)
  (display "\n")
  (cond ((null? xs) (display "TOUCH DOWN!!!"))
        ((prime? (length xs)) (begin (display (string-append (car xs) " length"))
                                     (evil-use (cdr xs))))
        (else (begin (display (* 2 (car xs)))
                     (evil-use (cdr xs)))))
  "end")

(evil-use (evil-list 6))
```

```
12
prime length
8
prime length
prime length
prime length
TOUCH DOWN !!!
```

For instance the call `(evil-list 4)` return a value of type `(Number,(String,(String,(String,Null))))` which is very hard to compute statically. By staying at the first depth level of the polymorphic recursion iterative typing can deduce useful information without executing the code. Below we present the compilation into LScheme of the above program ; as we can see all the informations are present to perform a recursive type deduction at runtime:

```
;; evil-list :: \-/ a . Number -> a
(define (evil-list n)
  (cond ([a => NULL] (<= n 0) null)
        ([a => (String,b)] (prime? n)
                           (cons "prime" ((sub [b => <next-a>] evil-list) (- n 1))))
        ([a => (Number,b)] else
                           (cons n ((sub [b => <next-a>] evil-list) (- n 1))))))

;; evil-use :: \-/ a . a -> String
(define (evil-use xs)
  (display "\n")
  (cond ([a => Void] (null? xs) (display "TOUCH DOWN!!!"))
        ([a => (String,b)] (prime? (length xs))
                           (begin (display (concat "yo " (car xs)))
                             ((sub [b => <next-a>] evil-use) (cdr xs))))
        ([a => (Number,b)] else
                           (begin (display (+ 1 (car xs)))
                           ((sub [b => <next-a>] evil-use) (cdr xs)))))
  "end")
```

```
((sub [a => c] evil-use) ((sub [a => c] evil-list) 4))
```

Recursive type serialization Many types are naturally recursive. For instance the elements of a list can themselves be lists. In fact every generic types like list $[a]$, pairs (a, b), function $a \to b$ etc are recursive in the sens they can include each other without theoretical depth limitation. The deserialization of such type have not been exposed in (Section 4.1) because it require a recursive polymorphic parser which iterative typing does not currently support. Lets say we use a XML format that serializes the value `(("abc",123),"ABC")` into:

```
<pair>
    <fst>
        <pair>
            <fst>
                <string>abc</string>
            </fst>
            <scd>
                <number>123</number>
            </scd>
        </pair>
    </fst>
    <scd>
        <string>ABC</string>
    </scd>
<pair>
```

The LScheme recursive polymorphic parser would be:

```
;; rec-read :: \-/ a . String -> a
(define (rec-read input)
  (define (type (get-top-level-tag input)))
  (cond ([a => (b,c)] (equal? type "pair") (cons ((sub [b => <next-a>] rec-read)
                                                  (get "fst" input))
                                                 ((sub [c => <next-a>] rec-read)
                                                  (get "scd" input))))
        ([a => String] (equal? type "string" (get "string" input)))
        ([a => Number] (equal? type "number") (string->number (get "number" input)))
        (else (error "unknwon type"))))
```

As usual, LScheme polymorphic call carries the contextual type information:

```
(define x ({String -> ((a,Number),b)} rec-read xml-string))
(+ 1 (cdr (car x)))
```

Those information will make the application crash as soon a type mismatch is detected during the recursive parsing.

Iterative monomorphic recursion Because even types that do not affect the outside world are fixed as well, iterative monomorphic recursion produces undesirable constraints. Lets consider the following code that defines the factorial function with an input request at each call:

```
(define (input-fac n)
  (read)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))
(input-fac 6)
```

The return type of `read` can only be set once. Which is very disturbing since this input does not affect the execution at all.

1. The user has first entered a boolean, he must continue to enter booleans to avoid dummy unification failure:

```
>> #t
>> #f
>> 3
Unification failed on [Boolean => Number]
```

2. The user has first entered a string, he must continue to enter strings to avoid dummy unification failure:

```
>> "iter 1"
>> "iter 2"
>> "iter 3"
>> "iter 4"
6
```

Iterative polymorphic recursion Iterative typing is well suited to support polymorphic recursion since it considers conditional branch as dynamic event which return a type variable. Therefore the code analysis stops at the recursion top level, and code execution is not required for type inference. At the execution, the locus would store information to deduce the precise type returned by the polymorphic recursion will return. Nevertheless implementation of polymorphic recursion poses a technical challenge and so far, two approaches have been investigated:

1. *Modification of the locus structure*: for now, loci are just unordered sets of type constraint ; mirroring the recursion stack could be a solution.

2. *Put version on type variables*: by introducing a version system for type variable, we could differentiate the type variables located on the same place but inside a different recursive call.

7.3 Refinement of type deduction

Introduction For now, dynamic events totally encapsulate type constraints when some deduction could be applied outside the event. Effectively there are many scenarios where some constraints of two conditional branches can be merged and moved on a upper level. Below we give two examples that show the current limit of iterative type inference:

1. *Output deduction*: the type of conditional structure is always inferred to a type variable. Below we expose a conditional structure that always returns a string, particularity that iterative typing does not exploit for the moment:

```
(define (output-gross)
  (if <pred>
      "foo"
      "bar"))
(+ (output-gross) 1)
```

   The type of `output-gross` is currently inferred to $\forall a.() \to a$ when $() \to String$ would have been more accurate. Such vagueness causes iterative typing to miss a static error at `(+ (output-gross) 1)`.

2. *Input deduction*: type constraints generated inside a conditional branch stay inside that branch. Below we expose a conditional structure that always threat `x` as a number:

```
(define (input-gross x)
  (if <pred>
      (+ x 1)
      (+ x 2))
  "end")
(input-gross "foo")
```

   The type of `input-gross` is currently inferred to $\forall a.a \to String$ when $Number \to String$ would have been more accurate. Such vagueness causes iterative typing to miss a static error at `(input-gross "foo")`.

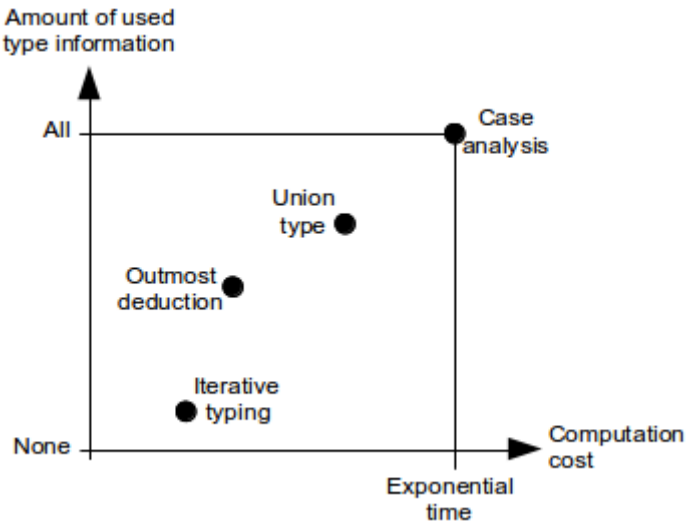

Figure 7.2: A compromise must be done between type information given by conditional structures and computation cost. Iterative typing propose very few guarantees. Outmost deduction is a must have as it does not complicate the execution algorithm. Union typing [19] could have a great potential but we did not investigated viability of such refinement. A complete case analysis is not practically feasible as its computation time grows exponentially with the amount of conditional structures.

Compromise As more and more information are exploited, the complexity of the algorithm increase. In general, a compromise has to be done between accuracy and computation cost (Figure 7.2). For now, iterative typing only guarantees the branches of a computation structure are all viable inside the direct context.

Outmost deduction Without changing the type system we could detect when types share a common outmost part. Lets consider the following Scheme code and its iterative type deduction:

```
;; concat-all :: [String] -> String
(define (concat-all xs)
  (if (null? xs)
      ""
      (string-append (car xs) (concat-all (cdr xs)))))

;; add-all :: [Number] -> Number
(define (add-all xs)
  (if (null? xs)
      0
      (+ (car xs) (add-all (cdr xs)))))

;; yo :: \-/ a . a -> [String]
(define (yo x)
  (if <pred>
      (concat-all x)
      (add-all x)
   "end"))
```

The type of `yo` has been inferred to $\forall a.a \rightarrow String$ when $\forall a.[a] \rightarrow String$ would have been more accurate. Indeed in any case, `x` is used as a list and a call like `(yo #t)` would have no chance to succeed. This type errors would have been detected by a more precise type deduction. As outmost deduction would not affect the type system, only the compilation algorithm would be complicated by such improvement which makes us believe outmost deduction is a must have on iterative typing.

Union typing As exposed below, union type could describes very well the semantic of dynamic conditional structures even if some static errors remain uncaught:.

1. *Mono-union*: When only one type is affected by the conditional structure, union type is a great success:

```
(define (yo)
  (if <pred>
      "foo"
      1))
(+ (yo) 1)       ; accepted
(concat (yo) 2) ; accepted
(and (yo) #f)    ; rejected
```

   Iterative-union typing would infer the type of `yo` to $() \rightarrow String|Number$ and so static errors like `(and (yo) #f)` would be detected.

2. *Poly-union*: when multiple types are affected through the conditional structure, some static errors are not detected:

```
(define (evil x y)
  (if <pred>
      (begin (+ x 1)
             (concat y "bar"))
      (begin (+ y 1)
             (concat x "bar")))
  #t)
;; All calls are accepted
(evil 1 2)          ; static error
(evil "foo" "bar") ; static error
(evil 1 "bar")
(evil "foo" 2)
```

   Iterative-union typing would infer the type of `evil` to: $(String|Number, String|Number) \rightarrow Boolean$ when argument types cannot actually be the same ; calls like `(evil 1 2)` would be erroneously accepted.

Union typing could significantly complicate the algorithm at the execution. The viability of such improvement needs further investigations.

Case analysis As presented in (Chapter 3), we could detect all static type errors by performing a complete case analysis. Such an algorithm is easy to implement through a recursive procedure that calls itself twice each time a conditional structure is encountered. That way, we could investigate all possible execution traces. Nevertheless the computation time of such an algorithm grows exponentially with the amount of conditional structure which is not practically acceptable. Furthermore, in case of polymorphic recursion, such computation never ends.

Predicative information So far we only discussed how informations contained in conditional branches can filter to external type deductions. But predicates give type information as well. For instance the following procedure is obviously free of type error:

```
(define (guarded-operation x)
  (cond ((number? x) (+ x 1))
        ((string? x) (concat "yo" x))
        (else "errhh, dunno what to do...")))
```

For now, iterative does not use guard at all and some checks are redundant:

```
;; guarded-operation :: \-/ a,b . a -> b
(define (guarded-operation x)
  (cond ([a => Number, b => Number] (number? x) (+ x 1))
        ([a => String, b => String] (string? x) (concat "yo" x))
        ([b => String]              else        "errhh, dunno what to do...")))
```

For instance, the guard `a => Number` of the first branch is useless since that branch can only be executed if `x` is effectively a $Number$.

Chapter 8

Conclusion

8.1 Summary

Context Traditionally, languages are separated in two classes. The first class is called "dynamically typed languages". Usually those languages are very expressive but they detect errors as they happen. Scheme and JavaScript are both dynamically typed. The second class is called "statically typed languages" ; Java belongs to that class. Programs in those languages are statically analyzed which allows detecting many errors early on. Nevertheless their expressiveness is restricted and they can not denote concepts like heterogeneous lists. It is agreed that dynamic typing is handy to quickly develop prototypes while statically typed languages are well suited to develop large and robust applications. In this thesis, we developed a typing technique called iterative typing that aims to combine the best of those two worlds. In other words, iterative typing is meant to provide high expressiveness and still be able to detect errors early on.

Implication On a very atomic language, earliest error detection could be obtained by performing a case analysis on dynamic events. This analysis would allow to detect errors as soon as possible when the current succession of dynamic events will, in any case, lead to an error later on. Nevertheless such algorithm is not compatible with recursion and grows exponentially with the number of dynamic events which is not theoretically and practically acceptable. Iterative typing tries to approach the case analysis behavior by dropping constraints wherever dynamic events occur. The input language is called LScheme and is very close to Scheme ; the output language is called LLScheme which is the annotated version of LScheme. As the execution goes on, dynamic uncertainties are resolved and new constraints are generated. Iterative typing carries around all the previously generated constraints and verifies the type compatibility at each new constraint generation.

Validation As validation we present in this thesis three case studies of practical interest.

The most direct application of iterative typing is deserialization, that is the use of a byte sequence to instantiate a data structure whose type depends on the byte sequence value. Iterative typing allows to detect a type mismatch right inside the reading procedure. Statically typed languages can detect errors that but require heavy use of type systems. On the other hand, iterative typing completely masks the type complexity involved in the pattern.

Iterative typing can also be used to propose an alternative of the common exception handling. In the presented alternative, the exception thrower knows whether throwing an exception will cause the application to stop or if the exception will be properly handled. On the first case, the exception thrower may decide it is preferable to return an usable result instead of stopping the whole application.

The last case study is a discussion about the interaction of iterative typing and side effects. On many cases, stopping the application when iterative typing predicts an error is just fine. But, sometimes it is preferable to perform side effects until normal Scheme applications would raise an error. This justifies the unsafe execution feature that allows the programmer to shut down iterative predictions for a while.

Theory A new language $\mathcal{L}$ was introduced as a representation of an atomic subset of LScheme. To keep the theoretical part of this thesis as simple as possible many features were bypassed in that part. In particular, the discussed atomic subset of LScheme is free of assignments. Even simplified, LScheme is still not well suited to be described in type systems. In consequence, two new languages: $\mathcal{L}$ and $\mathcal{LL}$ were developed to mirror LScheme and LLScheme. This approach is validated by exhibiting the syntactical transformations that allows to compile LScheme to $\mathcal{L}$. The theoretical presentation of iterative typing is articulated around three points. First, type systems for $\mathcal{L}$ and $\mathcal{LL}$ are both exposed and justified. Then we present a Scheme interpreter of $\mathcal{LL}$. Finally we present a Scheme compiler from $\mathcal{L}$ to $\mathcal{LL}$ which is based on Hindley-Milner type inference[12].

Related work Blurring the line between dynamic typing and static typing is not a recent topic. Many investigations have been performed in that sense which has led to the appearance of soft typing and gradual typing. Soft typing allows to specify whether an application should be checked during static code analysis or during the execution[18]. Gradual typing allows to define dynamic variables that always pass static type checks but require further checks at runtime[16]. We can say that those typings are dynamic typings that are partially statically type checked, while iterative typing is a static typing that is partially dynamically type checked. In other words, gradual and soft typing start with the status of dynamic typing and move as many type checks as possible to static code analysis. When iterative typing starts with the status of static typing and delays as little type checks as possible for the execution.

8.2 Contributions

In this thesis, we developed a new typing technique called iterative typing. Iterative typing can be seen as a static typing that delays as little type checks as possible to the execution. This approach is new, usually the efforts made to combine static and dynamic advantages go the other way around: it is about dynamic typing that performs as many checks as possible during an additional static code analysis phase. As an illustration of that difference, soft and gradual typing explicitly use type tags to ensure safety while iterative typing does not. To apply our typing, we created two new languages called LScheme which is very close to Scheme and LLScheme which is the annotated version of LScheme. We also implemented in Scheme a compiler from LScheme to LLScheme and an interpreter of LLScheme. Iterative typing does not accept as many programs as most dynamically typed languages but it is still able to implement very dynamic patterns as exhibited in (Chapter 4). The main advantage of iterative typing is the early error detection ; as shown in (Chapter 3) iterative typing is not remote from earliest error detection.

8.3 Future work

Viability testing The `if-viable` LScheme special form is very powerful and allows the programmer to have an insight inside iterative predictions. `(if-viable <prop> <alt>)` can be understood as: if the execution of `<prop>` will not produce type errors then execute `<prop>` otherwise execute `<alt>` . For instance, this special form can be used to simulate overloading on both inputs and output types. Nevertheless, this feature is highly experimental and increases the Scheme semantic. So far, no investigations have been performed to evaluated the consequences of such feature.

Polymorphic recursion We argue that it is possible to support polymorphic recursion inside an iterative framework. This would greatly increase the expressive power of iterative typing. For instance heterogeneous lists could be simulated by tuples manipulated by polymorphically recursive functions. Polymorphic reading procedures would allow to instantiate value of nested types like $[(String, [Number])]$ and would be a great application for iterative typing as well. The expected nested type would be deconstructed at each polymorphic recursive call and type errors would be detected at the earliest possible moment, even before the whole type would be known.

Type deduction refinement The presented LScheme to LLScheme compiler is still a prototype. Some effort should be done to clean the inserted constraints and refine the type deduction. In particular,

when the branches of a conditional structure are partially compatible, some checks could be moved on a upper level which is not currently performed.

Algorithm correctness Finally although the presented $\mathcal{L}$ and $\mathcal{LL}$ type systems are a strong theoretical basis, the validations of the compilation and the evaluation algorithm are still missing. In particular we would like to demonstrate that iterative typing never call primitives with erroneous types, and that when both iterative evaluation and normal evaluation succeed the results are the same.

Appendix A

Scheme

Introduction Scheme is a minimalist yet extremely powerful language which can develop most of the concepts and paradigms supported by the actual programming languages. Those characteristics make Scheme a very suitable language for teaching and experimenting ; that is why we choose Scheme as the working language of this thesis. Scheme is an evolution of Lisp and, as Lisp, is dynamically typed. As the language evolve, revised report on Scheme (abbreviated RnRS) are published to standardize the language specifications. The working language of this thesis is Racket which is based on the most recent released: R6RS. Nevertheless we will only expose in this section an atomic subset of Racket and explain the most important concepts of Scheme. This section is important for three reasons:

1. The iterative typing is implemented for a Scheme subset.
2. The language used for this implementation is Scheme itself (more precisely Racket).
3. The semantic of the Scheme syntax will be widely used in the technical part of this document.

Grammar A Scheme program is nothing more than a S-Expression. A S-Expression is either an atom or a list of S-Expression surrounded by parenthesizes. To attach a specific semantic to a S-Expression keywords are used in front of it, those S-Expressions are called special forms. Here is the explained syntax of the presented Scheme subset of (the formal grammar is presented at (Table A.1)):

1. *Constant*: either a boolean, a number, a character, a string, or a symbol (see the paragraph "Atomic data" for more insight).
2. *Identifier*: a symbol that is interpreted as its associated value inside the current environment.
3. *Procedure declaration* `(lambda (<PARAMETERS>) <BODY>)` : the keyword `lambda` is used to produce a data structure that can be used as a procedure (see the paragraph "Procedure representation" for more insight). For instance, the increment function is declared like this: `(lambda (x) (+ x 1))` .
4. *Application* `(<PROCEDURE> <ARGUMENTS>)` : In Scheme, applications are represented by by S-Expressions in prefix notation (see the paragraph "Application" for more insight). Here is an arithmetic example: `(+ 1 (* 2 3))` is evaluated to `7` .
5. *Global binding* `(define <PARAMETER> <EXPRESSION>)` : extends the current environment by the binding: `[<PARAMETER>,<EVALUATED-EXPRESSION>]` , the rest of the expression in this level will be evaluated with that new environment.

```
(define pi 3.14)
(define radius 5)
(define circumference (* 2 pi radius))
```

6. *Local binding* `(let (<BINDINGS>) <BODY>)` : extends the current environment with the provided bindings then evaluate the body with the new environment.

```
(let ((pi 3.14)
      (identity (lambda (x) x)))
  (identity pi))
```

```
3.14
```

7. *Data structure declaration* `(struct <NAME> [<PARENT-NAME>](<FIELDS>))` : provides a constructor and accessors for the specified fields ; the accessors of the parent data structure can be used on it:

```
(struct hero (name health))
(struct wizard hero (mana))
(define gandalf (wizard "Gandalf" 10 100))

(hero-health gandalf)
(wizard-mana gandalf)
```

```
10
100
```

8. *Assignment* `(set! <IDENTIFIER> <EXPRESSION>)` : updates, inside the current environment, the binding of `<IDENTIFIER>` . The old value and the new value must not have necessary the same type as in static languages.

9. *Conditional structure* `(if <PREDICATE> <CONSEQUENT> <ALTERNATIVE>)` : if the predicate is evaluated to false then the evaluation of alternative is returned ; otherwise the evaluation of the consequent is returned (every that is not false is considered as true). For instance `(if #t (+ 1 2) "foo")` is evaluated to `3` .

10. *Conjunction* `(and <ARGUMENTS>)` : the arguments are evaluated until one of them is not true then the result is false. True is returned when all argument are true. This special evaluation rule allows to use guards like: `(and (pair? x) (equal? (car x) 0))` so the second predicate which requires that `x` is a pair is only evaluated when `x` is indeed a pair.

11. *Disjunction* `(or <ARGUMENTS>)` : the arguments are evaluated until one of them is true then the result is true. False is returned when no arguments are true. Again this special form can be used as a guard and `(or #t (error "boum"))` will not produce an error.

Atomic data Although some of those data can be split (for instance a string can is an array of character), we will threat them as atom in the rest of this document. Here is the most used primitive data in Scheme:

- Boolean: The true constant is written `#t` and the false constant: `#f` . Scheme provide many tests but the ones most generals are: `eq?` that compares pointers and `equal?` that compares values.
- Number: We do not need to work with specific number representation so we only introduce the generic type number instead of the traditional integer, float, etc. Scheme is high level enough to automatically use the suitable representation.
- Character: a simple unicode character, for instance the constant 'A' is written: `#x0041` or simply `#\A` .
- String: a fixed-length array of characters. As usual a string can constructed with double quote like this: `"foo"` .
- Symbol: symbols are declared with a single quote like `'foo` ; the quote procedure has many other complex uses but we will not see them here. In substance, symbols are nothing more than immutable string. But they are implemented in such a way than two symbols with the same character content share the same pointer. So the result of `(eq? 'foo 'foo)` is `#t` when the result of `(eq? "foo" "foo")` is `#f` .

Variable	Substitution	Explanation	Example
<EXPR>	*boolean*	Boolean constant	#t , #f
	numeral	Numeral constant	1
	character	Character constant	#\A
	string	String constant	"foo"
	symbol	Identifier	x
	(lambda (<PARAMS>) <EXPRS>)	Procedure declaration	(lambda (x) (+ x 1))
	(<EXPR> <EXPRS>)	Application	(+ 1 2)
	(define <PARAM> <EXPRS>)	Global binding	(define identity (lambda (x) x))
	(let (<BINDS>) <BODY>)	Local bindings	(let ((pi 3.14)) (* 2 pi 5))
	(struct <PARAM> <[PARAM]> (<PARAMS>))	Data structure declaration	(struct rect (width height))
	(set! <PARAM> <EXPR>)	Assignment	(set! pi 3.1416)
	(if <EXPR> <EXPR> <EXPR>)	Conditional structure	(if (< x 0) "neg" "pos")
	(and <EXPRS>)	Logical conjunction	(and (pair? x) (car x))
	(or <EXPRS>)	Logical disjunction	(or #t (error "boum"))
<EXPRS>	<EXPR> <EXPRS>	Expression sequence	
	ϵ		
<PARAM>	*symbol*	Formal parameter	
<[PARAM]>	PARAM	Optional parameter	
	ϵ		
<PARAMS>	<PARAM> <PARAMS>	Parameter sequence	
	ϵ		
<BINDS>	(<PARAM> <EXPR>) <BINDS>)	Binding sequence	
	ϵ		

Table A.1: Grammar of the presented Scheme subset.

- Procedure of aridity n: Scheme does not precisely type a procedure but it still makes a distinction between two procedures of different aridity. This allows to throw an error when the number of argument inside an application does not match the aridity of the procedure. It is possible to defines procedure for an undefined number of argument but we will not expose it.

Compound data With the keyword `struct`, the programmer can create its own compound data structures. Nevertheless Scheme provide a primitive compound data structure that is powerful enough to represent any other compound data structure. That structure is the pair ; a pair is a couple of values "glued" together. Pairs are manipulated with the following primitives:

- `cons <first> <second>` : returns a pair compound with `<first>` and `<second>`.
- `car <pair>` : returns the first element of `<pair>`.
- `cdr <pair>` : returns the second element of `<pair>`.
- `set-car! <pair> <new-value>` : updates the first element of `<pair>` without changing the pointer of the whole pair.
- `set-cdr! <pair> <new-value>` : updates the second element of `<pair>` without changing the pointer of the whole pair.

Pairs are said mutable ; that is: without changing the pointer of the whole pair, we can either modify the pointer toward the first element or the pointer toward the second element.

Simply linked list As said before: pairs can represent any data structure, including simply linked list. Indeed in Scheme, lists are represented as nested pairs ending by the null constant written `null` [1]. Here is an example that constructs a simply linked list and access to a it:

```
(define list (cons 1 (cons #\A (cons #f (cons "foo" null)))))
```

```
> list
'(1 #\A #f "foo")
> (car list)
1
> (cdr list)
'(#\A #f "foo")
> (car (cdr list))
#\A
```

To ease the list processing, Scheme provides utility procedures:

- `list <elem1> <elem2> ...` : constructs a linked list from the given objects.
- `list? <object>` : tests if the given object is nested pair ending by `null`.
- `null? <object>` : tests if the given object is the null constant `null`.

Here is a program that computes the length of a list using those facilities:

```
(define (length xs)
  (if (null? xs)
      0
      (+ 1 (length (cdr xs)))))
```

```
> (length (list 1 2 3))
3
```

[1] Usually, this constant is represent through the quotation procedure: `'()`, but as iterative typing does not support quotations we will use the identifier `null`

Procedure status We already know that the keyword `lambda` is used to create a data structure representing a compound procedure. But Scheme goes further and grants the full status of first citizen to procedure. That means:

- Procedures may be named by variables.

```
(define incr (lambda (x) (+ x 1)))
```

- Procedures may be passed as arguments to procedures.

```
(define double-apply (lambda (f x) (f (f x))))
```

```
> (double-apply incr 3)
5
```

- Procedures may be returned as the results of procedures.

```
(define incr (lambda (x) (+ x 1)))
(define double (lambda (x) (* 2 x)))
(define compose (lambda (f g)
                  (lambda (x)
                    (g (f x)))))
```

```
> ((compose double incr) 3)
7
```

- Procedures may be included in data structures:

```
(define fs (list (lambda (x) (+ x 1))
                 (lambda (x) (+ x 2))
                 (lambda (x) (+ x 3))))
```

```
> ((car (cdr fs)) 3)
5
```

Function that manipulates other procedures are called higher order function and they are pretty much used every where in Scheme. One very famous higher order procedure is the map function which applies a procedure to each member of a list ; here is its implementation:

```
(define map (lambda (f xs)
              (if (null? xs)
                  null
                  (cons (f (car xs)) (map f (cdr xs))))))
```

```
> (map (lambda (x) (* 2 x)) (list 1 2 3))
'(2 4 6)
```

Procedure representation We must here make a difference between two kinds of procedure: the primitive ones defined in the Scheme implementation and the compound ones defined with the help of the `lambda` keyword. As it depend on the implementation, there is little to say about the first ones ; the second ones are more interesting. When the `lambda` special form is invoked, no further evaluation is performed, instead a data structure containing the parameters, the body and the current environment is returned.

Procedure application The evaluation of an application and the primitive case and in the compound case is detailed in respectively (Table A.2) and (Table A.3). Concerning application, two points must be highlighted:

1. Body is evaluated with environment where the procedure is defined NOT in the current environment where the application occurred.

	General case	`(cons "foo" (+ 1 2))`
1	All the argument are evaluated	The list `("foo" 3)` is received
2	Checks are performed concerning the number of arguments and their type.	As `cons` expects two arguments of any type the computation continue.
3	Depending on the called primitive, works is done on the representations.	A pointer containing the pair `("foo" . 3)` is returned.

Table A.2: Description of primitive procedure application: the general case and an example.

	General case	`define pi 3.14`
		`(define circ (lambda (rad) (* 2 pi rad)))`
		`(circ (+ 1 2))`

1	The compound procedure is evaluated returning a list of symbols (the formal parameters), an environment (here represented as a list of pair) and a list of expression (the body).	• Parameters: `(rad)` . • Environment: `((pi . 3.14))` . • Body: `((* 2 pi rad))`
2	All the arguments are evaluated.	The list `(3)` is received.
3	Checks are performed concerning the number of argument but NOT concerning their type.	The procedure `circ` specify one parameter and one argument was given, the computation can continue.
4	The environment carried by the procedure is extended for each couple "parameter", "evaluated-environment".	The extended environment is: `((pi . 3.14) (rad 3))` .
5	All the expression of the body are evaluated ; the evaluation of the last expression is the result of the application.	`(2 * pi * rad)` is evaluated with the environment `((pi . 3.14) (rad 3))` . `18.84` is the result of the application.

Table A.3: Description of compound procedure application: the general case and an example.

2. The applicative order is used which means thats argument are all evaluated once before computing the result. This example shows that the arguments are only evaluated once even when they are used multiple times inside the body:

```
(define *counter* 0)
(define incr! (lambda () (set! *counter* (+ *counter* 1))))
((lambda (x) x x) (incr!)) ;; only evaluated once, used twice
*counter*
```

```
1
```

IO actions In this thesis we will only be interested by two IO actions:

1. `display` : print on screen the string representation of the given object.

2. `read` : ask the user to input something. The input can be any "pure" data like number, string, tree, etc but not procedure.

Tail recursion In Scheme there is no built-in control flow like the traditional `for` and `while` ; instead Scheme emulate them with recursion. For most other languages, recursion is synonym with inefficiency because the frame stack increases at each recursive call. This is not necessary the case with Scheme. When the recursion call is the last statement of the body, Scheme destroys the current frame[2] before performing the recursion. Such optimization is called tail recursion and allows to keep the stack frame constant. We illustrate this with the factorial function:

1. Recursive definition of the factorial function:

```
;; Very close to the mathematic definition.
(define fac-rec (lambda (n)
                  (if (equal? n 0)
                      1
                      (* n (fac-rec (- n 1)))))) ; Recursion not in tail position,
                                                 ; the frame still contains useful
                                                 ; informations.
```

```
> (fac-rec 6)
720
```

2. Iterative definition of the factorial function:

```
;; Less elegant than the recursive definition.What is your name?
1234
What is your age?
17
You are too young...
(define fac-iter (lambda (n i)                        ; (n,i) are state variables.
                   (if (equal? n 0)
                       i
                       (fac-iter (- n 1) (* n i))))) ; Recursion in tail position,
                                                     ; the frame does not contain
                                                     ; any useful information.
```

```
> (fac-iter 6 1)
720
```

[2]Indeed no useful informations are stored in this frame instead all the informations must be stored in the recursive arguments.

Continuations The most specific characteristic of Scheme is the full support of continuations. Continuations are the mothers of all the modern control flow mechanisms, as the rest of scheme it is a very elegant and minimalist way to express a lot of concepts. In Scheme, any expression has a future that wait for the result of the expression ; this future is called continuation. Essentially, continuations are lambdas that wait for the result of an expression to produce the rest of the computation. The continuation of an expression is accessible with the primitive `call/cc` [3] ; this procedure expects a lambda and will execute it giving the current continuation:

1. The future is invoked, the given lambda never returns:

```
;; The future of the call/cc special
;; form is to be displayed on screen
(display (call/cc (lambda (cont)
           (cont 1)          ; future invocation
           (error "boum")))) ; never reached
```

```
1
```

2. The future is not invoked, the continuation is implicitly called with result of the given lambda:

```
(display (call/cc (lambda (cont)
           0))) ; equivalent to (cont 0)
```

```
0
```

[3] The original name of this procedure is `call-with-current-continuation` .

Bibliography

[1] H.P. Barendregt, W. Dekkers, and R. Statman. Typed lambda calculus. *Handbook of Mathematical logic*, pages 1091–1132, 1977.

[2] M. Bayne, R. Cook, and M.D. Ernst. Always-available static and dynamic feedback. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 521–530. IEEE, 2011.

[3] G. Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*. Citeseer, 2004.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[5] M. Fagan. *Soft typing: an approach to type checking for dynamically typed languages*. PhD thesis, Citeseer, 1992.

[6] J.B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[7] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, 1993.

[8] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, pages 1–23, 2011.

[9] M. Hucka, F.T. Bergmann, S.M. Keating, J.C. Schaff, and L.P. Smith. The systems biology markup language (sbml): Language specification for level 3 version. 2010.

[10] Jukka Lehtosalo and David J. Greaves. Genericty, nominal inheritance and gradual typin. Technical report, University of Cambridge Computer Laboratory.

[11] M. Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2011.

[12] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[13] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. *Exploring New Frontiers of Theoretical Informatics*, pages 437–450, 2004.

[14] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[15] J. Siek and W. Taha. Gradual typing for objects. *ECOOP 2007–Object-Oriented Programming*, pages 2–27, 2007.

[16] J.G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[17] R. Wolff, R. Garcia, E. TanterX, and J. Aldrich. Gradual featherweight typestate. Technical report, CMU-ISR-10-116R, 2010.

[18] A.K. Wright. *Practical soft typing*. PhD thesis, Citeseer, 1994.

[19] A.K. Wright and R. Cartwright. A practical soft type system for scheme. In *ACM SIGPLAN Lisp Pointers*, volume 7, pages 250–262. ACM, 1994.