Chapter 1

MISC

1.0.1 Why don't we want polymorphic let?

$(letevil(lambda(x)(ifx1"yo"))(+(evil\#t)(evil\#f)))$ $evil(lambda(x :: bool)(ifx1"yo") :: b) :: Bool \to b$
We loose dynamicity otherwise dico plumbery.

1.0.2 Execution & free variables

Linked variable describe a type: $(lambda(x)x) :: \forall a.a \to a$; when free variables represent an incertainty: $(if(eq?x0)1"yo") :: a$ or $(lambda(x)x :: a) :: \forall a.a \to a$: in the body of the lambda, a represent an incertainti, we don't know what is the type of the argument. So during execution we carry around a generic instantiation which link all the free variable to concrete type (concrete type = type without free variable).

1.0.3 Pure lambda calculus

We have to complicate the language in order to accept some important construction of the lambda calculus (cf fix & let).

Chapter 2

Actual work

2.1 Standard monomorphic type system

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \tag{2.1}$$

(2.2)

(2.3)

(2.4)

Chapter 3

Introduction

Chapter 4

Context

Quest to find errors as soon as possible.

Chapter 5

Technic

5.1 Introduction

5.2 Input

5.2.1 Semantic

- B_i is a CPO of primitive semantic value. The minium element of B_i is named $\bot_i$. For instance $B_{byte} = \{0, 1, 2, 3, 4, 5, 6, 7\}$; $\bot_{byte} = 0$.
- $B_0 = T = \{true, false, \bot_T\}$ where $\bot_T \leq true$ and $\bot_T \leq false$.
- $W = \{wrong\}$ is the singleton of the failure semantic.
- $V = B_0 + ... + B_N + F + W$ where $F = V \to V$ is the set of the functions from V to V. The minimum element of V is named $\bot_V$.

5.2.2 Language

E	x
	$[EE]$
	$[\lambda x.E]$
	$[ifEEE]$
	$[letxEE]$

5.2.3 Interpretation

5.3 Output

5.3.1 Semantic

Let $\tilde{W} = W + \{wr\tilde{o}ng\}$ the extended failure semantic. Further we will see that $wrong$ represents a static failure when $wr\tilde{o}ng$ represents a dynamic failure. The semantic of the output language is given by: $\tilde{V} = B_0 + ... + B_N + F + \tilde{W}$ (so $\tilde{V} = V + \{wr\tilde{o}ng\}$).

5.3.2 Language

E	$\{x\tau\}$
	$\{[EE]\tau\}$
	$\{[\lambda x.E]\tau\}$
	$\{[ifE\rho E\rho E]\tau\}$

5.3.3 Interpretation

Chapter 6

YOOO

E	$x\tau$
	$[EE]\tau$
	$[\lambda x.E]\tau$
	$[ifEpEpE]$

- $inter(x\tau, env) = env(x)$
- $inter([e_1 e_2]\tau, env) = cond$

Chapter 7

Iterative typing

7.1 Semantic

7.2 Type

7.3 Language

7.3.1 Type system (well typed word)

7.4 An algorithm to find well typed word

Chapter 8

Related workds

8.1 Interative typing

8.2 Hindley Milner rule system

$$\frac{x : \sigma \in \Gamma \qquad \sigma \leq \tau}{\Gamma \vdash x : \tau} \tag{8.1}$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 e_1 : \tau'} \tag{8.2}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \tag{8.3}$$

$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma, x : \Gamma'(\tau) \vdash e_1 : \tau'}{\Gamma \vdash let x = e_0 in e_1 : \tau'} \tag{8.4}$$

8.3 Soft Typing

P	num nil cons T T -¿ N
Ps	P Ps ϵ
N	(not num) (not nil) (not cons) (not -¿)
T T	+ Ps + Ps X
Ts	T Ts ϵ

- Soft typing
- Gradual typing [**?**]

Chapter 9

Idea, implication

9.1 Extended language

$G : \underline{Env \to Bool}$

E	x
	$[EE]$
	$[EE\tau]$
	$[if\ E\ gE\ g\tau]$
	$[\lambda x.E]$
	$[let\ x = E\ in\ E]$
C	$[E]$

Chapter 10

Evaluation

Chapter 11

Conclusion

Chapter 12

Annexe

12.1 Complte partial order

Here is the definition of partial order set (poset):

$$poset(A, \leq) := \begin{cases} \forall a \in A.a \leq a \\ \forall a_1, a_2 \in A.(a_1 \leq a_2 \cup a_2 \leq a_1) \Rightarrow a_1 = a_2 \\ \forall a_1, a_2, a_3 \in A.(a_1 \leq a_1 \cup a_2 \leq a_3) \Rightarrow a_1 \leq a_2 \end{cases}$$

A complete partial order is a poset in which each directed subset has a supremum.

$$cpo(A, \leq) := \begin{cases} poset(A, \leq) \\ \forall \alpha \subseteq A.(\alpha \neq \varphi \cup \forall x, y \in \alpha \exists z \in \alpha.(x, y \leq z)) \Rightarrow \exists z \in A.(\forall x \in \alpha z \leq x) \end{cases}$$

12.2 Hindley Milner

12.2.1 Semantic

Semantic values A program deals with values, here we are about to define the set of all those value.

- B_i is a CPO of primitive semantic value. The minium element of B_i is named $\perp_i$. For instance $B_{byte} = \{0, 1, 2, 3, 4, 5, 6, 7\}$; $\perp_{byte} = 0$.
- $B_0 = T = \{true, false, \perp_T\}$ where $\perp_T \leq true$ and $\perp_T \leq false$.
- $W = \{wrong\}$ is the singleton of the failure semantic.
- $V = B_0 + ... + B_N + F + W$ where $F = V \to V$ is the set of the functions from V to V. The minimum element of V is named $\perp_V$.

Then we define three relations:

1. $\triangleright \in V \times \{B_0, ..., B_N, F, W\} \to T$:
   - $v \in D \Rightarrow v \triangleright D = true$
   - $(v \notin D) \cup (v = \perp_V) \Rightarrow v \triangleright D = \perp_T$
   - $(v \notin D) \cup (v \neq \perp_V) \Rightarrow v \triangleright D = false$
2. $| \in V \times \{B_0, ..., B_N, F, W\} \to V$
   - $v \in D \Rightarrow v|D = v$
   - $v \notin D \Rightarrow v|D = \perp_d$
3. $cond \in T \times V \times V \to V$

- $cond(true, v, v') = v$
- $cond(fase, v, v') = v'$
- $cond(\bot_T, v, v') = \bot_V$

12.2.2 Syntaxic

Lets Id be the set of the identifiers ; x ranges over Id. Lets Exp be the language generated by the grammar:

E	x
	$[E\ E]$
	$[if\ E\ E\ E]$
	$[\lambda x.E]$
	$[let\ x\ =\ E\ in\ E]$

12.2.3 Link

Environement The environement function allow to link identifier to semantic value: $Env = Id \to V$. We can modify an environement with the function: $\{/\} \in Env \times Id \times V \to Env$ such that:

- $env\{v/x\}(x) = v$
- $x \neq \tilde{x} \Rightarrow env\{v/\tilde{x}\}(x) = env(x)$

Semantic function $inter : Exp \to Env \to V$: the semantic function is actually an interpreter.

- The interpretation of an identifier is its mapping over the environement.

$$inter(x, env) = env(x)$$

- The interpretation of an application is the result of the function discribed by the first expression given the argument described by the second expression. The interpretation is poluted by checks to ensure the first expression represent indeed a function from V to V and to ensure the evaluation of the argument has not failed.

$$inter([e_1\ e_2], env) = cond \quad \begin{array}{ll} inter(e_1, env) \triangleright F & \\ \quad cond & \begin{array}{l} inter(e_2, env) \triangleright W \\ \quad wrong \\ \quad (inter(e_1, env)|F)(inter(e_2, env)) \end{array} \\ \quad wrong & \end{array}$$

- $$inter([if\ e_1\ e_2\ e_3], env) = cond \quad \begin{array}{ll} inter(e_1, env) \triangleright T & \\ \quad cond & \begin{array}{l} inter(e_1, env)|T \\ \quad inter(e_2, env) \\ \quad inter(e_3, env) \end{array} \\ \quad wrong & \end{array}$$

- $$inter([\lambda x.e], env) = \lambda v.inter(e, env\{v/x\})$$

- $$inter([let\ x = e_1\ in\ e_2], env) = cond \quad \begin{array}{l} inter(e_1, env) \triangleright W \\ \quad wrong \\ \quad inter(e_2, env\{inter(e_1, env)/x\}) \end{array}$$

12.2.4 Type

Monotype For each B_i we have a primitive type ι_i. ι is used to range over $\{\iota_0, ..., \iota_N\}$. Here is the grammar that generate the monotype language:

μ	ι
	$\mu \to \mu$

- $v : \iota_i \Leftrightarrow (v = \bot_V \cap v \in B_i)$
- $v : \mu_1 \to \mu_2 \Leftrightarrow (v = \bot_V \cap ((\tilde{v} : \mu_1) \Rightarrow (v|F)(\tilde{v}) : \mu_2))$

Some values has no monotype ($wrong$) and some others have mutliple monotypes ($\bot_T$, $[\lambda x.x]$). We can proove that:

- $\forall v_1, v_2 \in V.(v_1 : \mu \cap v_1 \leq v_2) \Rightarrow v_2 : \mu$
- $(X \text{directed subset of} V) \Rightarrow (\forall x \in X.???dafuq$

Type Now we introduce a set of variable. α, β and γ are used to range over the type variables (we call type that contain variables polytype). Here is the grammar that generate the type language (which is ranged by τ):

τ	ι
	α
	$\tau \to \tau$

$\tau_1 \leq \tau_2$ means that we can obtain ρ_1 by replacing some of the variables of τ_2 with types. We can now lift the type belongship to polytype: $v : \tau \Leftrightarrow \forall \mu \leq \tau . v : \mu$. Directly we have:

- $(v : \tau \cap \tilde{\tau} \leq \tau) \Rightarrow v : \tilde{\tau}$
- $(v : \tau_1 \to \tau_2 \cap)$
- *JavaScript*: multi-paradigm and weakly typed language which has first call procedure and that is used as the Internet assembly language. Which make it the world most used language. Although *JavaScript* took a lot of name and convention from *Java* those two languages are somehow unrelated, *JavaScript* is far more inspired from *Scheme*.
- *Python*: multi-paradigm language most successfully used to create prototypes.
- *Perl*: high-level, general purpose language ; initially developed as a scripting language for Unix.
- *Smalltalk*: pure reflexive object-oriented language developed in the seventies, heavily used nowadays thanks to web application frameworks like *Seaside*.
- *Ruby*: multi-paradigm reflexive language combine *Smalltalk* feature with a syntax inspired of *Perl*. Also heavily used again through a web application framework: *Ruby on Rail*.
- *Scheme*: dialect of Lisp developed in the seventies by the MIT that considers procedure as first citizen and support current continuation. This language is known to provide great power within a minimalist design which make it suitable for learning purpose and experimentations.

In the guarding process, typing use a set whom each element (named type) is associate to a subset of the semantic domain. In term of typing we would say that `+` expects two numbers and returns a number ; the execution `1 + "foo"` would then produce a type error because `"foo"` is a string (is associate to the type string). The type set construction is dependent of the used typing technique.

Typing Typing is about providing guards for a large class of bad configuration. Informally typing is the discipline that ensure a program's execution has a meaning and does not apply operations on irrelevant data structures.

In this chapter we will introduce the typing discipline, two of its main stream and a comparison between those. Then we will expose the working language of this thesis.

12.3 Element of typing

Introduction Firstly, through model simulation, we define the concepts of well-behaved execution and program safety. Then we introduce the type concept and two way to perform typing: dynamically and statically. Please notice that this presentation is very personal and informal.

Language, Semantic & error A language is specified by its grammar and its interpretation relatively to a semantic. The semantics regroup the values manipulates by the language, the interpretation function returns a semantic value from an expression and an environment. One important semantic values the "error" value which is used each time an unwanted configuration shows up. Here is a piece of code that perfectly respects the Java grammar but produces differents kind of errors:

Notice that the concept of error is interpretation dependent, for instance `1 + "foo";` does not produce an error in Javascript specification. When a program does not never produce an error we say it is safe.

Typing Is a strategy the provide guards for a large class of errors. Type errors are formally defined by the provided guards but generally type safety ensure that no operation are performed out of its definition domain. We can say that a typing ensure that an execution has a meaning. In the guarding process, typing use a set whom each element (named type) is associate to a subset of the semantic domain. In term of typing we would say that `+` expects two numbers and returns a number ; the execution `1 + "foo"` would then produce a type error because `"foo"` is a string (is associate to the type string). The type set construction is dependent of the used typing technique.

Dynamic typing One very direct way to provide type safety is to add type flag on any computed value. Then type checks are performed before applying any primitive operation:

```
x = "foo"; // x points to bytes with the string type flag
x = 1;     // x points to bytes with the number type flag
x + 1;     // as both the values of x and 1 have the number
           // type flag, we are sure x + 1 is type safe
```

In dynamic typing, values are typed and variable can point to values of any type.

Static typing Static typing is about easing the interpreter to ensure well-behavior. So before being executed, programs pass through a static type checker that either accept or reject the programs, the purpose of this pass is to provide assumptions for the interpretation. For instance the following Java programs is accepted by the static type checker, but it uses casting which involve the interpreter:

```
Object[] array = new Object[3]; // array that can contain anything
array[0] = new Car();
array[1] = new Patato();
array[2] = new Car();
Car car = (Car) array[1];       // runtime error
car.drive();                    // undefined behavior
```

Static type checker attach type to variables and more generally to most expressions ; each variable has a fixed type and it is not possible to change it. The static type checkers are based on a set of syntactic rules named type system.

Type Obvious error in symbolic programming may be hard to detect symbolic programming After compilation to machine code, obvious error may be hard to detect. For instance both for instance both "1" Typing use the concept of type which represent the a subset of

; for instance $1 +$ "foo" and $1/0$ will result in an error in most interpreters. Errors can be detected in two places: the application of a primitive and the evaluation of a grammatical rule

The operators inside that semantic are generally not defined for the value of the domain ; for instance

Hardwares only manipulate bit words through very low level operator like bit-adder. If we want to use this model to simulate the behavior of higher level model we first need to translate its value in term of bit words.

Model simulation We define a model (V, O), as a set of values and a set of operators that manipulate those values. Lets defined two models:

1. The algebra model: $\mathcal{A} = (\mathbb{N}^{100} \cup \{\top, \bot\}, \{+, \vee\})$. This model manipulates boolean and ranged natural numbers trough the operators $+ : \mathbb{N}^{100} \times \mathbb{N}^{100} \to \mathbb{N}^{100}$ and $\vee : \{\top, \bot\} \times \{\top, \bot\} \to \{\top, \bot\}$.
2. The binary model: $\mathcal{B} = (\{0,1\}^N, \{+, jump, push, ...\})$. The binary model manipulates bit words through some low level operators. We present this model as the interface provided by some hardware.

If we want our hardware working on the $\mathcal{A}$ model, we need to simulate the behavior of $\mathcal{A}$ with $\mathcal{B}$. The first step is to represent each value of $\mathcal{A}$ by values of $\mathcal{B}$. Then the $\mathcal{A}$ operators can be defined as combinations of $\mathcal{B}$ operators. But how can we translate the definition domain of the $\mathcal{A}$ operators? If we do not answer this question, the computation $1 + \bot$ may either crash, return an undefined result or produce an unsuspected side effect. Typing is about providing guard to prevent such computation ; an execution that does not produce such computation is said well behaved. A program that does only produce any well-behaved execution is said to be safe (effectively with IO actions, a program may produce a lot of different execution). Does two concept are very language dependent ; for instance $1 +$ "3" may be well-behaved in some languages (because an errors is thrown or an implicit cast is performed) and may be ill-behaved in some others.

Type Types attach meaning to binary representations, each specific domain of the simulated model is granted by a type which is used to ensure that the higher level operations are performed on rightful values. In the precedent model, both 1 and $\bot$ has binary representations but the one of 1 has the type *Number* and the one of $\bot$ has the type *Boolean*. In order to ensure well-behaved execution, typing technique deduce types. The construction of the type set is very dependent of the used typing technique.

Dynamic typing One very direct way to provide safety is to add type flag on any computed value. Then type checks are performed before applying any primitive operation:

```
x = "foo"; // x points to bytes with the string type flag
x = 1;     // x points to bytes with the number type flag
x + 1;     // as both the values of x and 1 have the number
           // type flag, we are sure x + 1 is well-behaved
```

In dynamic typing, values are typed and variable can point to values of any type. The interpreter is the only responsible to ensure well-behavior. Insofar as dynamic typing reject all ill-behaved executions, all the programs are safe.

Static typing Static typing is about easing the interpreter to ensure well-behavior. So before being executed, programs pass through a static type checker that either accept or reject the programs, the purpose of this pass is to provide assumptions for the interpretation. For instance the following Java programs is accepted by the static type checker, but it uses casting which involve the interpreter:

```
Object[] array = new Object[3]; // array that can contain anything
array[0] = new Car();
array[1] = new Patato();
array[2] = new Car();
Car car = (Car) array[1];       // runtime error
car.drive();                    // undefined behavior
```

Static type checker attach type to variables and more generally to most expressions ; each variable has a fixed type and it is not possible to change it. The static type checkers are based on a set of syntactic rules named type system.

Introduction As we known, a program without any extern interaction is deterministic and it is well-typed or ill-typed. But that class of program does hardly anything useful (as they return the same result at each execution), it is the IO actions that add practical interest to programs. IO actions may change the values that are manipulated from on execution to an other. That freedom combined with conditional structures allows to vary the applied set of instruction. So IO actions and conditional structure are complementary and give all the dynamic flavor to languages [Table 12.1].

	Conditional structures	IO actions	IO actions and conditional structure
Description	Execution is deterministic, nothing can change from one execution to an other.	Executions carry around different values and possibly types but the same instructions are executed each time.	Executions carry around different values and possibly types, different sets of instructions may be executed.
Restriction	Programs are reduced to calculators, there is not input just a result.	Recursion is impossible (no stopping condition).	

Table 12.1: Complementary of IO actions and conditional structures.

Input as a trace

Bad starting

Wet dream

A first compromise

Wet dream As we do not introduce time, we can model the inputs as a trace. We can model dynamic program as boxes that for a given trace may:

1. Terminate without error
2. Terminate with a type error.
3. Terminate with a value error.

Given a program, computing the set of all the "good" trace that make it terminate without errors. So at the moment, the user goes out of the "good" traces the execution is stopped. If the set of the "good" trace is empty, the program is simply rejected. This would be the best scenario in term of expressiveness, safety and early detection.

A first compromise Even if the set of value is practically finite, it is not realist to analyze each trace of values. So we must regroups the values. If those groups are defined by predicate we go to the dependent typing discipline which is out of the range of this thesis. If we split values into set we fall into a normal type semantic. As we are working with type we do no have access to value anymore. So conditional structure becomes dynamic event and we must analyze both branches (on specific case, type may suffice but we work on the general case, this can be an improvement) and we cannot detect value error anymore. For this series of dynamic event there is no type errors (but possibly value errors).

```
(let ((x (read))
      (y (read)))
  (if (equal? x y)
      (+ x x)                ; then branch
      (string-append y y))) ; else branch
```

Here are the following viable event sequence:

- $(String, String, else)$
- $(Number, Number, then)$
- $(Number, String, then)$
- $(Number, String, else)$

With refined static analysis we would know that if the `read` returned different type, the *then* event cannot happen.

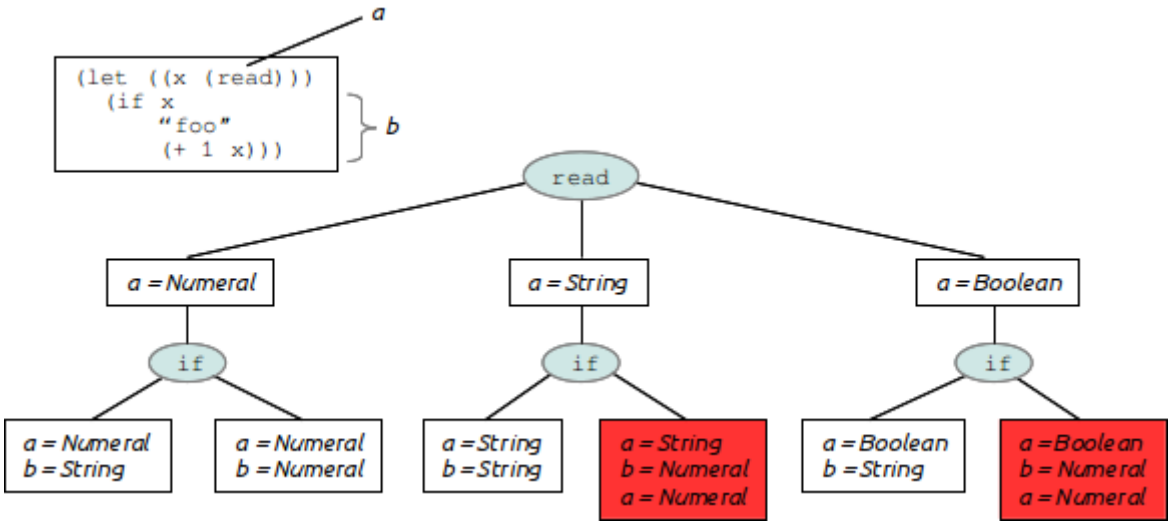

Figure 12.1: The different paths that may take the execution of a dynamic program.

The iterative compromise The first compromise let us an exponential growth on both read and if which is not acceptable [Figure 12.1]. In price of a little bit more computation at runtime, the iterative typing propose almost the same feature. The idea is this: knowing the precedent dynamic event we can know of the current event is about to lead the trace out of the viable one.

```
(let ((x (read))
      (y (read)))
  (if (equal? x y)
      (+ x x)                ; then branch
      (string-append y y))) ; else branch
```

Here are the following viable event sequence:

- First `read` : no constraint
- Second `read` : no constraint
- `if` : if predicate is satisfied, the first `read` must have returned a *Number*. If the predicate is not satisfied then the second `read` must have returned a *String*.

We detect errors a little bit latter, with the first compromise if both `read` would returned a *Boolean* we would directly detect the error (without waiting the next event).

Compilation So iterative typing is capable to tell at each dynamic event if the program will produce a type error until the next event. To achieve this, a variant of the Hindley-Milner type inference is used.

- The type of a dynamic event (either `if` or `read`) is a type variable.
- The type variable introduced on `read` call are stored on the identifier.
- The constraints done inside a conditional branch are stored in the head of the branch.
- Lets take back our last example:

```
(let ((x (read))
      (y (read)))
  (if (equal? x y)
      (+ x x)
      (string-append y y)))
```

- Following our inferencer it would be compiled in something like this:

Constructor	Arity	$\mathcal{L}$ expression	An associated monotypes
$String$	0	$"foo"$	$String$
$Boolean$	0	$true$	$Boolean$
$Number$	0	3.14	$Number$
$\rightarrow$	2	$(\lambda\ (x)\ x)$ $(\lambda\ (x)\ (if\ (<\ x\ 0)\ "pos"\ "neg"))$	$\xi \rightarrow \xi$ $Number \rightarrow String$
[]	1	$(cons\ 1\ (cons\ 2\ (cons\ 3\ null)))$ $null$	$[Number]$ $[String]$
(.)	2	$(pair\ 1\ "foo")$	$(Number.String)$

Table 12.2: The different type constructor currently implemented in the iterative prototype. Notice that the action of associate types to expressions has not been defined, the examples are just meant to develop an intuitive interpretation of monotypes.

```
(let ((x (read [a]))
      (y (read [b])))
  (if (equal? x y)
      [a=Numeral]
        (+ x x)
      [b=String ]
        (string-append y y)))
```

Interpretation We must keep a trace of the precedent events to decide the viability of the current event. Those informations are stored inside a set of type constraint that we carry trough the execution. Such a set is called locus.

- For the inputs `1` and `"foo"` the compiled program returns `"foofoo"` . Here is the evolution of the locus through the execution:

```
(let ((x (read [a]))          ; [a=Numeral]
      (y (read [b])))         ; [a=Numeral, b=String]
  (if (equal? x y)
      [a = Numeral]
        (+ x x)
      [b = String ]           ; [a=Numeral, b=String, b=String]
        (string-append y y))) ; [a=Numeral, b=String]
```

- For the inputs `1` and `2` the compiled program returns a type error. Here is the evolution of the locus through the execution:

```
(let ((x (read [a]))          ; [a=Numeral]
      (y (read [b])))         ; [a=Numeral, b=Numeral]
  (if (equal? x y)
      [a = Numeral]
        (+ x x)
      [b = String ]           ; [a=Numeral, b=Numeral, b=String]
        (string-append y y))) ; TYPE ERROR
```

An other point of view As the inference goes on, it introduce type variable and as much freedom degrees. When an application is analyzed, type constraint are generated that restrict the number of type the type variables may takes. We store those constraint trough the program ans we just must ensure at least one instantiation is possible. The word locus comes from the fact that it define a "place" where type variable may be instantiated.

Chapter 13

Garbage

We should demonstrate that any program can be derivated, can only crash at locus intersections or unsafe primitives like car divide,...

13.1 The $\widetilde{\mathcal{L}}$ type system

Judgment The judgments of the $\widetilde{\mathcal{L}}$ type system are little bit more elaborated than the ones of $\mathcal{L}$. Judgments have to form: $A|L \vdash e : \sigma$, intuitively it means: from the assumptions A and relatively to the locus L we can deduce that the expression e has the type σ. This dichotomy of the hypotheses (A is transformed to $A|L$) allows us to introduce scopes for generated type constraints. And that is exactly what we need to express the conditional structures in an iterative way.

Overview The entire $\widetilde{\mathcal{L}}$ type system is presented at [Table **??**]. In the rest of this section we will justify each rule of this type system.

MONO-ID This rule is fundamental, it allow to freely distribute a type between assumptions and locus. The big diffence between assumptions and loci is that informations stored on the assumptions rely on the program to be relaxed

Here is two derivation that end with: $x : String$

$$\text{MONO-ID}\ \frac{x : String \in \{x : String\}}{\{x : String\}|\{\} \vdash x : String} \quad \Big| \quad \text{MONO-ID}\ \frac{x : \xi \in \{x : \xi\}}{\{x : \xi\}|\{\xi.String\} \vdash x : String}$$

Although those derivation end up with the same conclusion, the left one is much strict than the right one. That is because the assumptions

CONST and ASSIGN Those rules are fairly simple are represent a direct adaption of the $\mathcal{L}$ rules.

ABS and APP The abstraction and application rule are very similar to the ones of $\mathcal{L}$, they just carry a locus around. That is an important point, loci are not affected by abstraction neither applications. In comparison, it is those rules that carry most of the essence of gradual typing and soft typing.

COND The rule about the conditional structure motivate the dichotomy of the usual set of assumption. In essence: all the constraints stored inside the locus of a branch does not affect the rest of the inference. When all the information stored inside the assumptions must be shared by all the parts. In [Figure 13.1] we present a derivation that illustrate the behavior of the COND rule.

POLY-LET Given the binding judgment $A|L \vdash e : \tau \to \tau'$, this rule first identity a suitable set of type variables $\xi_{1..i}$ to split the locus L. The first part $L \backslash \xi_{1..i}$ concerns the polymorphic bind and will be stored inside the polymorphic let expression. The second part $L/\xi_{1..i}$ is the locus in which the body of the let expression must be evaluated. This rule summarize the rules LET and GEN of the $\mathcal{L}$ type system, we

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

$$\text{ID}\ \frac{\{x:\sigma\} \in A}{A \vdash x:\sigma} \qquad \text{SUB}\ \frac{A \vdash x:\sigma \qquad \tau \sqsubseteq \sigma}{A \vdash (\sigma\ \tau\ x):\tau}$$

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

$$\text{CONST}\ \frac{c :: \gamma}{A \vdash c:\gamma} \qquad \text{ASSIGN}\ \frac{A \vdash e:\tau \qquad A \vdash x:\tau}{A \vdash (set\ x\ e):\tau}$$

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

$$\text{COND}\ \frac{A \vdash e:\tau \qquad \{\xi'_i \mapsto \tau'_i\}A \vdash e':\tau''' \qquad \{\xi''_i \mapsto \tau''_i\}A \vdash e'':\tau'''}{A \vdash (if\ e\ \{\xi'_i \mapsto \tau'_i\}\ e'\ \{\xi''_i \mapsto \tau''_i\}\ e''):\tau'''}$$

Table 13.1: The $\widetilde{\mathcal{L}}$ type system.

$$\text{APP}\ \frac{\text{ID}\ \frac{f:Number \to \xi_{if} \in \{f:Number \to \xi_{if}, x:\xi\}}{\{f:Number \to \xi_{if}, x:\xi\} \vdash f:Number \to \xi_{if}} \qquad \text{CONST}\ \frac{1 :: Number}{\{f:Number \to \xi_{if}, x:\xi\} \vdash 1:Number}}{\{f:Number \to \xi_{if}, x:\xi\} \vdash (f\ 1):\xi_{if}} \quad (1)$$

$$\text{APP}\ \frac{\text{ID}\ \frac{f:String \to \xi_{if} \in \{f:String \to \xi_{if}, x:\xi\}}{\{f:String \to \xi_{if}, x:\xi\} \vdash f:String \to \xi_{if}} \qquad \text{CONST}\ \frac{"foo" :: String}{\{f:String \to \xi_{if}, x:\xi\} \vdash "foo":String}}{\{f:String \to \xi_{if}, x:\xi\} \vdash (f\ "foo"):\xi_{if}} \quad (2)$$

$$\text{ABS}\ \frac{\text{ABS}\ \frac{\text{COND}\ \frac{\text{CONST}\ \frac{x:\xi \in \{f:\xi', x:\xi\}}{\{f:\xi', x:\xi\} \vdash x:\xi} \qquad (1) \qquad (2)}{\{f:\xi', x:\xi\} \vdash (if\ x\ \{\xi' \mapsto Number \to \xi_{if}\}\ (f\ 1)\ \{\xi' \mapsto String \to \xi_{if}\}\ (f\ "foo")):\xi_{if}}}{\{f:\xi'\} \vdash (\lambda x.(if\ x\ \{\xi' \mapsto Number \to \xi_{if}\}\ (f\ 1)\ \{\xi' \mapsto String \to \xi_{if}\}\ (f\ "foo"))):\xi \to \xi_{if}}}{\{\} \vdash (\lambda f.(\lambda x.(if\ x\ \{\xi' \mapsto Number \to \xi_{if}\}\ (f\ 1)\ \{\xi' \mapsto String \to \xi_{if}\}\ (f\ "foo")))):\xi' \to (\xi \to \xi_{if})}$$

Table 13.2: The $\widetilde{\mathcal{L}}$ program: $(\lambda f.(\lambda x.(if\ x\ \{\xi' \mapsto Number \to \xi_{if}\}\ (f\ 1)\ \{\xi' \mapsto String \to \xi_{if}\}\ (f\ "foo"))))$ is type safe.

$$\text{MONO-ID}\ \frac{f : \xi \to \xi \{f : \xi \to \xi, x : \xi'\}}{\{f : \xi \to \xi, x : \xi'\} | \{\xi.Number\} \vdash f : Number \to Number} \qquad \text{CONST}\ \frac{1 :: Number}{\{f : \xi \to \xi, x : \xi'\} | \{\xi.Number\} \vdash 1 : Number}$$
$$\text{APP}\ \frac{}{\{f : \xi \to \xi, x : \xi'\} | \{\xi.Number\} \vdash (f\ 1) : Number} \quad (1)$$

$$\text{MONO-ID}\ \frac{f : \xi \to \xi \in \{f : \xi \to \xi, x : \xi'\}}{\{f : \xi \to \xi, x : \xi'\} | \{\xi.String\} \vdash f : String \to String} \qquad \text{CONST}\ \frac{\text{"}foo\text{"} :: String}{\{f : \xi \to \xi, x : \xi'\} | \{\xi.String\} \vdash \text{"}foo\text{"} : String}$$
$$\text{APP}\ \frac{}{\{f : \xi \to \xi, x : \xi'\} | \{\xi.String\} \vdash (f\ \text{"}foo\text{"}) : String} \quad (2)$$

$$\text{CONS}\ \frac{\text{MONO-ID}\ \dfrac{x : \xi' \in \{f : \xi \to \xi, x : \xi'\}}{\{f : \xi \to \xi, x : \xi'\} | \{\} \vdash x : \xi'} \qquad (1) \qquad (2)}{\{f : \xi \to \xi, x : \xi'\} | \{\} \vdash (if\ x\ \{\xi.Number, \xi''.Number\}\ (f\ 1)\ \{\xi.String, \xi''.String\}\ (f\ \text{"}foo\text{"}))}$$

Figure 13.1: The constraints generation of the $\widetilde{\mathcal{L}}$ type system on the initial $(if\ x\ (f\ 1)\ (f\ \text{"}foo\text{"}))$ expression.

$U\ \{(\xi, \delta\ \tau_{1..k}), tail\}\ i = r$ • $\xi \in free(\tau_{1..k}) \Rightarrow r = fail$ • $\xi \notin free(\tau_{1..k}) \Rightarrow r = U\ tail\ i \cup (\xi, i\ \delta\ \tau_{1..k})$	When the first type is a variable and the second a type variable, we first ensure the variable does not appear into the compound type. Indeed, constraint: $(\xi, Number \to \xi)$ describe an infinite type that the unification algorithm does not support. Once the guard passed, we basically move the constraint to the instantiation.
$U\ \{(\delta\ \tau_{1..k},, \xi), tail\}\ i = U\ \{(\xi, \delta\ \tau_{1..k}), tail\}\ i$	We reverse the pair to jump in the first case.
$U\ \{(\delta_1\ \tau_{1..k}, \delta_2\ \tau_{1..l}), tail\}\ i = r$ • $\delta_1 \neq \delta_2 \vee k \neq l \Rightarrow r = fail$ • $\delta_1 = \delta_2 \wedge k = k \Rightarrow r = U\ \{(\tau_1, \tau_2), \ldots, (\tau_k, \tau_l), tail\}\ i$	If the compound types are compatible, monotypes are combined two by two to create new constraints.
$U\ \phi\ i = i$	All the constraint has been moved to the instantiation, the algorithm return the instantiation.

Table 13.3: Description of the unification algorithm. Notice that the unification is generally called without initial instantiation ; in this case, the empty instantiation is used.

had to do this combination because the linked variable had to be known to analyze a polymorphic let expression.

13.2 Locus

Introduction A locus is mapping from type variable to monotype, but a type variable that appears in a substitution cannot be substituated itself.

Robinson's Unification The unification is a well known algorithm much used in type theory. From a given set of monotype constraints it either fails or returns an instantiation that unify all the given constraints [Table 13.3]. Actually, instantiations are just a specific kind of locus where or left member are type variables that does never appear on right member. Here is an example of unification:

$$U \left\{ \begin{array}{l} \xi_1 = (String \to \xi_2) \to \xi_4 \\ \xi_2 = \xi_3 \\ \xi_3 = Number \end{array} \right\} = \left\{ \begin{array}{l} \xi_1 = (String \to Number) \to \xi_4 \\ \xi_2 = Number \\ \xi_3 = Number \end{array} \right\}$$

Adding some constraints A very direct way to include type constraints into locus is to perform an unification of the whole new set of constraint. Formally we have: $L + \tau.\tau' = U\ L \cup \tau.\tau'$; an example is presented at [Figure 13.2].

Closure We use the closure concept to support polymorphism. A set of type variable is close inside a locus when they are not related with any other type variables. The formal definition of the closure test $\square$ is:

$$\xi_{1..j} \square L \coloneqq \xi_{1..j} \subseteq (free\ L) \wedge \forall c \in L.(free\ c) \cap \xi_{1..j} = \phi \vee (free\ c) \subseteq \xi_{1..j}$$

$$\left\{ \begin{array}{c} \xi_1 = String \to \xi_2 \\ \xi_3 = \xi_2 \to Number \end{array} \right\} + \xi_1 = \xi_4 \to Boolean = \left\{ \begin{array}{c} \xi_1 = String \to Boolean \\ \xi_2 = Boolean \\ \xi_3 = Boolean \to Number \\ \xi_4 = String \end{array} \right\}$$

Figure 13.2: The constraint addition seen as a unification on the new set of constraint.

$$\left\{ \begin{array}{rcl} \xi_1 & . & String \\ Number \to String & . & \xi_2 \to \xi_1 \\ \xi_3 & . & Boolean \end{array} \right\}. \Bigg\| \left\{ \begin{array}{ccc} \xi_1 & . & String \\ \xi_2 & . & Number \\ \xi_3 & . & Boolean \end{array} \right\}$$

Figure 13.3: On the left the original locus, and on the right the same locus after unification. The closures before unification are: ϕ, $\{\xi_1, \xi_2\}$, $\{x_3\}$ and $\{\xi_1, \xi_2, \xi_3\}$ when the closures after unification are any subset of $\{\xi_1, \xi_2, \xi_3\}$

In [Figure 13.3] we present an example that shows the closure are affected by the unification.

Locus separation A direct application concerning closure, is that they allow to properly separate constraints. We define two operators on locus:

1. An operator $\backslash$ that collects all the constraints containing only the given type variable: $L \backslash \xi_{1..j} = \{c \in L | (free\ c) \subseteq \xi_{1..j}\}$
2. An operator $/$ that collects all the constraints containing none of the given type variable: $L / \xi 1..j = \{c \in L | (free\ c) \cap \xi_{1..j} = \phi\}$

By definition, if $\xi_{1..j}$ is a closure on L we have a clean separation: $L = L \backslash \xi_{1..j} \cup L / \xi_{1..j}$.

13.3 Compilation $\mathcal{L} \to \widetilde{\mathcal{L}}$

13.4 Conclusion

13.5 The $\mathcal{L}$ type system

Transformation We transform $\mathcal{L}$ to a language easier to express the rules of the type system.

13.6 The $\widetilde{\mathcal{L}}$ type system

Demonstrate when a locus fail it fil untill the end

The iterative type set The concrete type used in iterative typing is a recursive set described in [Table 13.4]. Concrete type are not powerful enough to express the most general type of simple expression like the A type variable is a market meant to be substituted by any other type.

Number	`Number`	`1 :: Number`
String	`String`	`"foo" :: String`
Boolean	`Boolean`	`#t :: Boolean`
Pair	`(x . y)`	`(cons 1 #t) :: (Number . Boolean)`
List	`[x]`	`(cons "foo"(cons "bar"null)) :: [String]`
Procedure	`(x1 ... x2 -> y)`	`+ :: (Number Number -> Number)`

Table 13.4: The recursive set of concrete type ; `x` and `y` denotes concrete type.

e	x	Identifier
	$(lambda\ (x)\ e)$	Abstraction
	$(e\ e)$	Application
	$(let\ x\ (x)\ e\ e)$	Binding
	$(if\ e\ e\ e)$	Conditional structure

Table 13.5: Grammar of the $\widetilde{\mathcal{L}}$ language. x is used to range over X which is is the infinite set of the identifiers.

13.7 The $\mathcal{L}$ language

13.7.1 Semantic

$$
\begin{array}{rcl}
V & | & B_0 = \{true, false\} \\
 & | & \dots \\
 & | & B_N \\
 & | & W = \{ERROR\} \\
 & | & F = V \to V
\end{array}
$$

- $B_0, \dots B_N$ is a list of set of primitive semantic value.
- $B_0 = W = \{w\}$: singleton of the semantic failure.
- $B_1 = \{true, false\}$: booleans.
- $W = \{w\}$ singleton of semantic failure.
- $V = B_0 \cup ... \cup B_N \cup F$ where $F = V \to V$

13.7.2 Grammar

$\mathcal{L}$ is very similar the an atomic Scheme. This language express lambda terms with two non fondamental forms: let and if. Those forms may be represented by only lambda expressions but they are usefull for the compilation $\mathcal{L} \to \widetilde{\mathcal{L}}$. The grammar is exposed at [Table 13.5].

13.7.3 Interpretation

Environment The environment is a function that associate something to an identifier, $c_A \in X \to A$. Three kind of environements will be used:

- Interpretation of $\mathcal{L}$: $c_V \in X \to V$.
- Compilation of $\mathcal{L} \to \widetilde{\mathcal{L}}$: $c_\Sigma \in X \to \Sigma$.
- Interpretation of $\widetilde{\mathcal{L}}$: $c_{\widetilde{V}} \in X \to \widetilde{V}$.

Normally it can happen that an environment does not contain any binding for an identifier, this usually corresponds to a syntactic error and we suppose that the given expressions does not contain any syntactic error. If this disturbs you, we can simply give a default binding to any identifier like: $global\ \xi = 0$. We define a function that adds a binding to an environment: $\oplus$, this function is defined as fellow:

$I\ [\![x]\!]\ c = c\ x$	The semantic function returns the binding of the identifier inside the given envrionment.
$I\ [\![(lambda\ (x)\ e)]\!]\ c = \lambda arg.(I\ [\![e]\!]\ (c \oplus \{x, arg\}))$	A lambda abstraction is returned, when an argument is given, the environement is extented before evaluating the body of the procedure.
$I\ [\![(e_1\ e_2)]\!]\ c = v$ • $v_1 = I\ [\![e_1]\!]\ c$ • $v_2 = I\ [\![e_2]\!]\ c$ • $\begin{cases} (v_1 \notin F) \vee (v_2 = w) & \Rightarrow \quad v = w \\ v_1 \in F) \wedge (v_2 \neq w) & \Rightarrow \quad v = v_1\ v_2 \end{cases}$	We simply checks the values of the suposed abstraction and the argument, then we apply the argument on the abstraction. This is the only rule that may produce an error.
$I\ [\![(if\ e_1\ e_2\ e_3)]\!]\ c = v$ • $v_1 = I\ [\![e_1]\!]\ c$ • $v_2 = I\ [\![e_2]\!]\ c$ • $v_3 = I\ [\![e_3]\!]\ c$ • $\begin{cases} v_1 = w & \Rightarrow \quad v = w \\ v_1 = true & \Rightarrow \quad v = v_1 \\ v_1 \notin \{true, w\} & \Rightarrow \quad v = v_2 \end{cases}$	Following the value of the predicate (the first expression) we execute the second expression or the third one. If the evaluation of the predicate returned an error it is simply forwarded.
$I\ [\![(let\ x_1\ (x_2)\ e_2\ e_1)]\!]\ c = I\ [\![((lambda\ (x_1)\ e_1)\ (lambda\ (x_2)\ e_2))]\!]\ c$	The let expression is syntactic sugar for $\widetilde{\mathcal{L}}$. It can be seen as "*let name be value inside body*" where *value* is a procedure declaration.

Table 13.6: Semantic function

- $(c \oplus \{x, v\})\ x' = v'$
  - $\begin{cases} x = x' & \Rightarrow \quad v = v' \\ x \neq x' & \Rightarrow \quad v' = c\ x' \end{cases}$

Note that we consider that primitives values like number, strings, boolean are all identifiers present in the global scope. So inside an expression "foo" is consider as an identifier and c_V "foo" refers to the semantic value of "foo" which the string "foo". That "trick" allows us to keep the $\mathcal{L}$ grammar as simple as possible.

Semantic function I is the semantic function, that associate a semantic value to an $\mathcal{L}$ expression and en environment. $I \in (\mathcal{L} \times (X \to V)) \to V$. A complete description of the semantic function can be found at [Table 13.6].

Semantic error In a pure lambda calculus there is only lambda terms and the evaluation can never fail. The only failure produced by the interpretation itself correpond to case were the first term of an application is not a function. All the other failure are produced by the primitive. For instance the increment function should return an error when the given argument is not a number. The division function should return an error when the second argument is zero etc.

δ	β
	$\delta \to \delta$

π	ξ
π	β
π	$\pi \to \pi$

σ	$\forall \xi.\sigma$
	π

Table 13.7: Recursive definition of the monotypes (Δ), polytypes (Π) and type schemes (Σ). Those definitons use ξ that ranges over an infinite set of type variables Ξ and β that range over $\{\beta_0, \ldots, \beta_N\}$; β_i is associate to B_i.

13.8 The $\widetilde{\mathcal{L}}$

13.8.1 Type

Introduction Type describe

Monotype Concrete type mimic the structure of the semantic [Table 13.7]. We define the function : that tests if a semantic value can be associate to a monotype:

- $v : \beta_i \Leftrightarrow v \in B_i$
- $v : \delta_1 \to \delta_2 \Leftrightarrow \forall v' : \delta_1.((v\ v') : \delta_2 \vee (v\ v') = w)$

A single semantic value may have multiple concrete types (possibly zero).

Polytype Monotype are strictly larger to concrete type, the embed the concept of type variable. We do not extend the definition of : to polytype (as it is done for Hindely Milner inference [**?**]). Intuitively a type variable represent an undefined type.

Type scheme This extension is required to support polymorphism.

Instantiation This function takes a poly type and replaces all the binded type variable by new ones $\circlearrowleft \in \Sigma \to \Pi \times \Xi^N$. This function returns the monotype, the new variables and a locus that map the old variables to the new ones. Here is its definition:

- $\circlearrowleft\ \sigma = f\ \sigma\ \top\ \phi$
  - $f\ (\forall \xi.\sigma)\ \gamma\ \psi = f\ \sigma\ (\gamma \oplus \{(\xi, \xi')\})\ (\psi \cup \{\xi'\})$
  - $f\ \pi\ \gamma\ \psi = (g\ \pi, \gamma, \psi)$
    - $g\ \xi = \gamma\ \xi$
    - $g\ \beta = \beta$
    - $g\ (\pi_1 \to \pi_2) = (g\ \pi_1) \to (g\ \pi_2)$

13.8.2 Locus

Introduction A locus defines a what concrete type can take the type variables introduced by the Hindley Milner inference. Each type variable represents a new degree of freedom and a locus allow to reduce the overall freedom. When we start the inference, the locus allow every thing ; as we perform unifications, we produce constraints that restrict the place (locus) of available types. When the locus reach the zero solution (that means there is no instantiation of the variables that allow to satisfy all the constraints) the inferentiation fail. A locus binds type variables to type $\Gamma = \Xi \to \Sigma$. $\top$ represent a locus without binding, so $\forall \xi \in \Xi.(\top\ \xi = \xi)$. $\bot$ is the failure locus, $\forall \xi \in \Xi.(\bot\ \xi) = \beta_0$.

Remove This function remove all the constraints over the given set of type variables: $\ominus \in \Gamma \times \Xi^N \to \Gamma$.

- $(\gamma \ominus \psi)\ \xi = \pi$
  - $\begin{cases} \xi \in \psi & \Rightarrow \quad \pi = \xi \\ \xi \notin \psi & \Rightarrow \quad \pi = \gamma\ \xi \end{cases}$

e	x	Monomorphic call
	$[x\ \gamma]$	Polymorphic call
	$(lambda\ (x)\ e)$	Abstraction
	$(e\ e)$	Application
	$(let\ x\ [\psi\ (x)\ e]\ e)$	Polymorphic binding
	$(if\ e\ [\gamma\ e]\ [\gamma\ e])$	Conditional structure

Unify Robinson algorithm to unify:$\mathcal{U} \in (\Pi \times \Pi)^N \to \{(Xi, \Pi)^M, fail\}$ [**?**]. We can easily define $U \in (\Pi \times \Pi)^N \to \Gamma$.

- $(U\psi)\ \xi = \pi$
  - $$\begin{cases} (\mathcal{U}\ \psi) = fail & \Rightarrow \quad \pi = \beta_0 \\ (\mathcal{U}\ \psi) \neq fail \wedge (\xi, \pi_i) \in (\mathcal{U}\ \psi) & \Rightarrow \quad \pi = \pi_i \\ (\mathcal{U}\ \psi) \neq fail \wedge (\xi, \pi_i) \notin (\mathcal{U}\ \psi) & \Rightarrow \quad \pi = \xi \end{cases}$$

Intersection This function compound the constraints inside two locuses: $\odot \in \Gamma \times \Gamma \to \Gamma$.

- $\gamma_1 \odot \gamma_2 = \gamma$
  - $\forall \xi, \pi.((\xi, \pi) \in \chi_1 \Leftrightarrow \gamma_1\ \xi = \pi)$
  - $\forall \xi, \pi.((\xi, \pi) \in \chi_2 \Leftrightarrow \gamma_2\ \xi = \pi)$
  - $\gamma = U\ (\chi_1 \cup \chi_2)$

$$\gamma_1 \odot \gamma_2 = \gamma_2 \odot \gamma_1$$
$$(\gamma_1 \odot \gamma_2) \odot \gamma_3 = \gamma_1 \odot (\gamma_2 \odot \gamma_3)$$
$$\gamma \odot \bot = \bot$$
$$\gamma \odot \top = \gamma$$

13.8.3 Semantic

- $\widetilde{B}_0 = \widetilde{W} = \{w, \widetilde{w}\}$
- $\widetilde{F} = \widetilde{V} \times \Gamma \to \widetilde{V} \times \Gamma$
- $\widetilde{V} = \widetilde{B}_0 \cup \ldots \cup B_N \cup \widetilde{F}$

13.8.4 Grammar

13.8.5 Compilation $\mathcal{L} \to \widetilde{\mathcal{L}}$

13.8.6 Interpretation

Why do we have to carry the locus around?

```
((lambda (x) (if #f
                 (+ x 1)                ;; x used as a numeral
                 (str-append x "yo"))) ;; x used as a string
   (read))                             ;; need to carry the locus to know
                                       ;; the actual type of x
```

$C\ [\![x]\!]\ c = (\tau, \top, e, \psi)$ • $(\tau, \gamma, \psi) = \circlearrowleft\ (c\ x)$ • $\begin{cases} \psi = \phi & \Rightarrow \quad e = [\![x]\!] \\ \psi \neq \phi & \Rightarrow \quad e = [\![\gamma\ x]\!] \end{cases}$	If the binding is a polytype we return a polymorhpic call that store the maping of the new variables to the old ones. Otherwise we simply return the identifier (monomorphic call).
$C\ [\![(lambda(x)\ e)]\!]\ c = (\xi \to \pi, \gamma, [\![(lambda\ (x)\ e')]\!], \psi)$ • $(\pi, \gamma, e', \psi) = C\ [\![e]\!]\ (c \oplus \{(x, \xi)\})$	This is a standard Hindley Milner inference: we simply compile the body with the binding (x, ξ) where ξ is a fresh variable.
$C\ [\![(e_1\ e_2)]\!]\ c = (\xi, \gamma, [\![(e_1' e_2')]\!], \psi_1 \cup \psi_2)$ • $(\tau_1, \gamma_1, e_1', \psi_1) = C\ [\![e_1]\!]\ c$ • $(\tau_2, \gamma_2, e_2', \psi_2) = C\ [\![e_2]\!]\ c$ • $\gamma = \gamma_1 \odot \gamma_2 \odot (U\ \{(\tau_1, \tau_2 \to \xi)\})$	This case is also very close to the Hindley Milner inference but we can return the fresh variable ξ because its binding is store to the returned locus.
$C\ [\![(let\ x_1\ (x_2)\ e_2\ e_1)]\!]\ c = (\tau_1, \gamma, [\![(let\ x_1\ [\psi\ (x_2)\ e_2]\ e_1)]\!], \psi_1)$ • $(\tau_2, \gamma_2, e_2', \psi_2) = C\ [\![e_2]\!]\ (c \oplus \{(x_1, \xi_1), (x_2, \xi_2)\})$ • $\gamma_2' = \gamma_2 \odot (U\ \{(\xi_2 \to \tau_2, \xi_1)\})$ • $\psi = \psi_2 \cup \{\xi_1, \xi_2\}$ • $\sigma = \forall \xi \in \psi . (\gamma_2'\ \xi_1)$ • $(\tau_1, \gamma_1, e_1', \psi_1) = C\ [\![e_2]\!]\ (c \oplus \{(x_1, \sigma)\})$ • $\gamma = \gamma_1 \oplus (\gamma_2' \ominus \psi)$	First we evaluate the abstraction body (e_2) with a recursive monomorphic call $((x_1, \xi_2))$. Then we transform the monotype to a polymorhpic type σ and we evaluate the let body (e_1) with the binding $((x_1, \sigma))$. Those step are also very inspired from the Hindley Milner type inference.
$C\ [\![(if\ e_1\ e_2\ e_3)]\!]\ c = (\xi, \gamma_1, [\![(if\ e_1'\ [\gamma_2'\ e_2']\ [\gamma_3'\ e_3'])]\!], \psi_1 \cup \psi_2 \cup \psi_3)$ • $(\tau_1, \gamma_1, [\![e_1']\!], \psi_1) = C\ [\![e_1]\!]\ c$ • $(\tau_2, \gamma_2, [\![e_2']\!], \psi_2) = C\ [\![e_2]\!]\ c$ • $(\tau_3, \gamma_3, [\![e_3']\!], \psi_3) = C\ [\![e_3]\!]\ c$ • $\gamma_2' = \gamma_2 \odot (U\ \{(\xi, \tau_2)\})$ • $\gamma_3' = \gamma_3 \odot (U\ \{(\xi, \tau_3)\})$	The conditional structure is very specific, first we evaluate all the expressions. Then we store the loci generated by the consequent and the alternative and we return the locus returned by the predicate. So the consequent and the alternative does not affect the outside inference.

$\widetilde{I}\ [\![x]\!]\ c\ \gamma_{in} = (c\ x, \gamma_{in})$	Monomorphic call: return the binding and does not touch the locus.
$\widetilde{I}\ [\![[\gamma_{poly}\ x]]\!]\ c\ \gamma_{in} = (\lambda\gamma.\lambda arg.(c\ x)\ (\gamma \odot \gamma_{poly})\ arg, \gamma_{in})$	Polymorphic call: create a lambda term; when performed, the given locus is extended by the mapping from the new variables to the old variable.
$\widetilde{I}\ [\![(lambda(x)\ e)]\!]\ c\ \gamma_{in} = (\lambda\gamma.\lambda arg.\widetilde{I}\ [\![e]\!]\ (c \oplus \{(x, arg)\})\ \gamma, \gamma_{in})$	Abstraction: identic to the original language but must give a locus to the body (not γ_in)
$\widetilde{I}\ [\![(e_1\ e_2)]\!]\ c\ \gamma_{in} = (v, \gamma_2)$ • $(v_1, \gamma_1) = \widetilde{I}\ [\![e_1]\!]\ c\ \gamma_{in}$ • $(v_2, \gamma_2) = \widetilde{I}\ [\![e_2]\!]\ c\ \gamma_1$ • $\begin{cases} v_1 = w \vee v_2 = w & \Rightarrow \quad v = w \\ v_1 \neq w \wedge v_2 \neq w \wedge (v_1 = \widetilde{w} \vee v_2 = \widetilde{w}) & \Rightarrow \quad v = \widetilde{w} \\ v_1 \notin (\widetilde{W} \cup \widetilde{F}) \wedge v_2 \notin \widetilde{W} \wedge \gamma_2 \neq \bot & \Rightarrow \quad v = w \\ v_1 \notin (\widetilde{W} \cup \widetilde{F}) \wedge v_2 \notin \widetilde{W} \wedge \gamma_2 = \bot & \Rightarrow \quad v = \widetilde{w} \\ v_1 \in F \wedge v_2 \notin \widetilde{W} & \Rightarrow \quad v = v_1\ v_2 \end{cases}$	Application: very close to the original language, have to carry around the locus and test $\widetilde{w}$.
$\widetilde{I}\ [\![(let\ x_1\ [\psi\ (x_2)\ e_2]\ e_1)]\!]\ c\ \gamma_{in} = \widetilde{I}\ [\![e_1]\!]\ (c \oplus \{(x_1, poly)\})\ \gamma_{in}$ • $abs = \lambda\gamma.\lambda arg.\widetilde{I}\ [\![e_2]\!]\ (c \oplus \{(x_2, arg), (x_1, abs)\})\ \gamma$ • $poly = \lambda\gamma.\lambda arg.(v, \gamma' \ominus \psi)$ • $(v, \gamma') = abs\ \gamma\ arg$	Polymorphic binding: first we define recurively and monomorphically the procedure. Then we create a lambda term that remove all the binding of the linked type variables after the body evlauation.
$\widetilde{I}\ [\![(if\ e_1\ [\gamma_2\ e_2]\ [\gamma_3\ e_3])]\!]\ c\ \gamma_{in} = (val, \gamma)$ • $(v_1, \gamma_1) = \widetilde{I}\ [\![e_1]\!]\ c\ \gamma_{in}$ • $\begin{cases} v_1 \in \widetilde{W} & \Rightarrow \quad (v, \gamma) = (v_1, \gamma_1) \\ v_1 = true & \Rightarrow \quad (v, \gamma) = \widetilde{I}\ [\![e_2]\!]\ c\ (\gamma_1 \odot \gamma_2) \\ v_1 \notin (\widetilde{W} \cup \{true\}) & \Rightarrow \quad (v, \gamma) = \widetilde{I}\ [\![e_3]\!]\ c\ (\gamma_1 \odot \gamma_3) \end{cases}$	Conditional structure: first we evaluate the predicate then we get sure that the next branch is safe be performing a locus intersection.

13.9 Compilation $\mathcal{L}$

13.10 Safty and correctness

- Safety: the evaluation of a compiled $\mathcal{L}$ expression cannot lead to an uncaught error.

$$\forall e \in \mathcal{L}.(C\ e\ g_\Sigma) = (\widetilde{e}, \gamma) \Rightarrow (\widetilde{I}\ \widetilde{e}\ g_{\widetilde{V}}\ \gamma) \neq w$$

- Correctness: when the evaluation of a compiled $\mathcal{L}$ expression succeed, the result is equal to the evaluation of the original expression.

$$\forall e \in \mathcal{L}.((C\ e\ g_\Sigma) = (\widetilde{e}, \gamma)) \wedge (\widetilde{I}\ \widetilde{e}\ g_{\widetilde{V}}) \neq \widetilde{w} \Rightarrow (\widetilde{I}\ \widetilde{e}\ g_{\widetilde{V}}) = (I\ e\ g_V)$$

Chapter 14

garbage2

Addition This function add a binding to type variable and update all the existing binding. $\otimes \in \Gamma \times (\Xi \times \Pi) \to \Gamma$. $(\gamma \otimes (\xi, \pi))\ \tilde{\xi} = (\gamma\ \tilde{\xi}) \oslash (\xi, \pi)$. When $\otimes$ is performed, it perform a substitution to all the bindings, so all the type variables that appear in a binding does not have bindings themself.

Substitution This function apply a subsitution to a monotype, $\oslash \in \Pi \times (\Xi \times \Pi) \to \Pi$. Here is its definition:

- $\beta \oslash (\xi, \pi) = \beta$
- $\xi_1 \oslash (\xi_2, \pi) = \xi$

  $$\_ \begin{cases} \xi_1 = \xi_2 & \Rightarrow \quad \xi = \pi \\ \xi_1 \neq \xi_2 & \Rightarrow \quad \xi = \xi_1 \end{cases}$$

- $(\pi_1 \to \pi_2) \oslash (\xi, \pi) = (\pi_1 \oslash (\xi, \pi)) \to (\pi_2 \oslash (\xi, \pi))$

$U\ \phi\ \gamma = \gamma$	Unification terminated
$U\ (\{(\xi_1, \xi_2)\} \cup \psi)\ \gamma = \tilde{\gamma}$ • $\begin{cases} \xi_1 = \xi_2 & \Rightarrow \\ \xi_1 \neq \xi_2 \wedge (\gamma\ \xi_1 = \xi_1) & \Rightarrow \\ \xi_1 \neq \xi_2 \wedge (\gamma\ \xi_1 \neq \xi_1) & \Rightarrow \end{cases}$	When two type variables have to be unified, there is three case. When the variables are equals, there is no effect. When there is no binding for ξ we bind it to ξ_2. When there is a binding for ξ we unify ξ_2 to that binding.
$U\ (\{(\xi, \beta)\} \cup \psi)\ \gamma = \tilde{\gamma}$ • $\begin{cases} \gamma\ \xi = \xi & \Rightarrow \\ \gamma\ \xi \neq \xi & \Rightarrow \end{cases}$	
$U\ (\{(\xi, \pi_1 \to \pi_2)\} \cup \psi)\ \gamma = \tilde{\gamma}$ • $\begin{cases} \gamma\ \xi = \xi & \Rightarrow \\ \gamma\ \xi \neq \xi & \Rightarrow \end{cases}$	
$U\ (\{(\beta_1, \beta_2)\} \cup \psi)\ \gamma = \tilde{\gamma}$ • $\begin{cases} \beta_1 = \beta_2 & \Rightarrow \\ \beta_1 \neq \beta_2 & \Rightarrow \end{cases}$	
$U\ (\{(\beta, \pi_1 \to \pi_2)\} \cup \psi)\ \gamma = \bot$	
$U\ (\{(\beta, \xi)\} \cup \psi)\ \gamma = U\ (\{(\xi, \beta)\} \cup \psi)\ \gamma$	
$U\ (\{(\pi_1 \to \pi_2, \tilde{\pi}_1 \to \tilde{\pi}_2)\} \cup \psi)\ \gamma = U\ (\{(\pi_1, \tilde{\pi}_1), (\pi_2, \tilde{\pi}_2)\} \cup \psi)\ \gamma$	
$U\ (\{(\pi_1 \to \pi_2, \xi)\} \cup \psi)\ \gamma = U\ (\{\xi, \pi_1 \to \pi_2\} \cup \psi)\ \gamma$	
$U\ (\{(\pi_1 \to \pi_2, \beta)\} \cup \psi)\ \gamma = \bot$	

Table 14.1:

Chapter 15

Correctness of the iterative typing

15.1 Monomorphic type inference

$$\frac{x : \tau \in \Gamma \qquad \tau \leq \tau'}{\Gamma \vdash x : \tau'} \tag{15.1}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \tag{15.2}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \to \tau'}{\Gamma \vdash (e_1\ e_2) : \tau'} \tag{15.3}$$

Type safety If there exists a derivation that ends with: $\Gamma \vdash e : \tau$ then $inter(env, e)$ is does not produce a type error (where for all x: $env(x)$ return a value of type $\Gamma(x)$).

15.1.1 Basic type inference

1. $W(\Gamma, x) = (Id, \Gamma(x))$
2. $W(\Gamma, [\lambda x.e]) = (S, \tau)$
   - $(S, \tau') = W(\Gamma + \{x : a\}, e)$
   - $\tau : S(a) \to \tau'$
3. $W(\Gamma, [e_1\ e_2]) = (S, \tau)$
   - $(S_1, \tau_1) = W(\Gamma, e_1)$
   - $(S_2, \tau_2) = W(S_1(\Gamma), S_1(e_2))$
   - $V = U(S_2(\tau_1), \tau_2 \to a)$
   - $S = S_1.S_2.V$
   - $\tau = V(a)$

Soudness If $W(\Gamma, e)$ succeed with (S, τ) then there exists a derivation which end with: $S(\Gamma) \vdash e : \tau$. Because the type system is type safe, we ensure that every expression

Completness If there exists a derivation which end with: $\Gamma \vdash e : \tau$ then there exists S such that: $W(S(\Gamma), e) = (S, \tau)$.