Polymorphic typing of an algorithmic language

Xavier Leroy

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports de Recherche

N°1778

Programme 2

Calcul symbolique, Programmation
et Génie logiciel

POLYMORPHIC TYPING OF AN ALGORITHMIC LANGUAGE

Xavier LEROY

Octobre 1992

Polymorphic typing of an algorithmic language

Xavier Leroy[1]

Abstract

The polymorphic type discipline, as in the ML language, fits well within purely applicative languages, but does not extend naturally to the main feature of algorithmic languages: in-place update of data structures. Similar typing difficulties arise with other extensions of applicative languages: logical variables, communication channels, continuation handling. This work studies (in the setting of relational semantics) two new approaches to the polymorphic typing of these non-applicative features. The first one relies on a restriction of generalization over types (the notion of dangerous variables), and on a refined typing of functional values (closure typing). The resulting type system is compatible with the ML core language, and is the most expressive type systems for ML with imperative features proposed so far. The second approach relies on switching to "by-name" semantics for the constructs of polymorphism, instead of the usual "by-value" semantics. The resulting language differs from ML, but lends itself easily to polymorphic typing. Both approaches smoothly integrate non-applicative features and polymorphic typing.

Typage polymorphe d'un langage algorithmique

Xavier Leroy

Résumé

Le typage statique avec types polymorphes, comme dans le langage ML, s'adapte parfaitement aux langages purement applicatifs, mais ne s'étend pas naturellement au trait principal des langages algorithmiques: la modification en place des structures de données. Des difficultés de typage similaires apparaissent avec d'autres extensions des langages applicatifs: variables logiques, canaux de communication, manipulations de continuations. Ce travail étudie (dans le cadre de la sémantique relationnelle) deux nouvelles approches du typage polymorphe de ces constructions non-applicatives. La première repose sur une restriction de l'opération de généralisation des types (la notion de variables dangereuses), et sur un typage plus fin des valeurs fonctionnelles (le typage des fermetures). Le système de types obtenu reste compatible avec le noyau applicatif de ML, et se révèle être le plus expressif parmi les systèmes de types pour ML plus traits impératifs proposés jusqu'ici. La seconde approche repose sur l'adoption d'une sémantique "par nom" pour les constructions du polymorphisme, au lieu de la sémantique "par valeur" usuelle. Le langage obtenu s'écarte de ML, mais se type très simplement avec du polymorphisme. Les deux approches rendent possible l'interaction sans heurts entre les traits non-applicatifs et le typage polymorphe.

[1] École Normale Supérieure and INRIA Rocquencourt, projet Formel

Foreword

This document is the English translation of the author's doctoral dissertation, *Typage polymorphe d'un langage algorithmique*, defended at University Paris VII on June 12, 1992.

Introduction

When I was a child, I was told that Santa Claus came in through the chimney, and that computers were programmed in binary code. Since then, I have learned that programming is better done in higher-level languages, more abstract and more expressive. I have also learned that programmers sometimes make errors, and therefore that advanced programming languages should not only convey the programmer's thoughts, but also allow the automatic checking for certain inconsistencies in programs.

The most popular of these consistency checks is called static typing. It consists in detecting a large family of errors: the application of operations to objects over which they are not defined (the integer addition to a boolean and a character string, for instance). This is achieved by grouping the objects handled by the program into classes: the types, and by abstractly simulating the execution at the level of types, following a set of rules called the type system.

The strength of static typing is that it guarantees the absence of type errors in the programs it accepts. The weakness of static typing is that it rejects some programs that are in fact correct, but too complex to be recognized as such by the type system used. From this tension follows the search for more and more expressive type systems [14, 66], an endless quest to which the work presented here belongs.

In this quest, the apparition of the concept of polymorphism has been a major breakthrough. Polymorphism supports the static typing of generic functions: pieces of programs that can operate on data of different types [83, 58]. For instance, a sorting algorithm can work on arrays of any type, as long as a suitable ordering function between the array elements is provided. These generic functions are important, since they can naturally be reused without modification in many programs. Without polymorphism, static typing prohibits this reuse; with polymorphism, static typing supports this reuse, while checking its correctness with respect to the types.

Polymorphic type systems fit seamlessly within purely applicative programming languages: those languages where variables cannot be assigned to, and data structures cannot be modified in place [47, 94, 38]. In contrast, in this dissertation, I concentrate on non-purely applicative languages such as the algorithmic languages from the Algol family: Pascal, Ada, Modula-3, and especially ML. These languages support variable assignment and in-place modification of data structures. These two features are difficult to typecheck with polymorphic types. Mutable data structures naturally admit simple typing rules; these rules are sound in monomorphic type systems, but turn out to be unsound in the presence of polymorphic types. This is illustrated by the

following example, written in ML[1]:

```
let r = ref [] in
    r := [1];
    if head(!r) then ... else ...
```

The naive application of the typing rules leads to assign the polymorphic type $\forall\alpha.\,\alpha$ `list ref` to the variable `r` in the body of the `let` construct. This type allows `r` to be considered with type `int list ref`, and therefore to store the list containing the integer 1 in `r`. This type also allows `r` to be considered with type `bool list ref`, hence we can treat `head(!r)` (the first element of the list contained in `r`) as a boolean. The `if` construct is therefore well-typed. However, at run-time, `head(!r)` evaluates to the value 1, which is not a boolean. This example demonstrates that modifying a polymorphic object in-place can invalidate static typing assumptions, hence compromise type safety: after [] has been replaced by [1] in r, the typing assumption `r` : $\forall\alpha.\,\alpha$ `list ref` no longer holds.

As illustrated above, a polymorphic type system must restrict the use of polymorphic mutable objects. We must not be able to use a given mutable object inconsistently, that is, with two different types. That's what the type system must ensure. Several type systems are known to achieve this result [64, 99, 21, 3, 40, 100]; but they turn out to be too restrictive in practice, rejecting as erroneous many correct programs that are useful in practice. To palliate these deficiencies, I was led to design new polymorphic type systems for mutable structures, to which the main part of this dissertation is devoted. To illustrate these generalities, take as example the simplest of all restrictions that prevent inconsistent use of mutable objects: the restriction consisting in requiring all mutable objects to be created with closed, monomorphic types, that is, types containing no type variables. This is the approach taken in early ML implementations, such as the Caml system [19, 99]. The resulting type system is obviously sound. Unfortunately, it prohibits generic functions that create mutable structures. This is a major limitation in practice, as shown by the following example. Assume defined an abstract type τ `matrix` for matrices with elements of type τ, admitting in-place modification. The following function is useful in many programs:

```
let transpose_matrix m1 =
    let m2 = new_matrix(dim_y(m1), dim_x(m1), matrix_elt(m1,0,0)) in
    for x = 0 to dim_x(m1) - 1 do
      for y = 0 to dim_y(m1) - 1 do
        set_matrix_elt(m2, x, y, matrix_elt(m1,y,x))
      done
    done;
    m2
```

[1] The examples given in this Introduction assume that the reader has some notions of ML. I use the syntax of the CAML dialect [19, 51]; the readers used to SML will add `val` and `end` after `let`. Here is a survival kit for the readers unfamiliar with ML. Most syntactic constructs read like English. [] stands for the empty list, $[a]$ stands for the list with one element a, and $[a_1; \ldots; a_n]$ stands for the n-element list $a_1 \ldots a_n$. The type τ `list` is the type of the lists whose elements have type τ; the empty list [] has type τ `list` for all types τ. References are indirection cells that can be modified in-place — in other terms, arrays of size 1. `ref`(a) allocates a fresh reference, initialized to the value a. $!r$ returns the current contents of reference r. $r := a$ replaces the contents of the reference r by a.

Clearly, this function can operate over matrices of any type: its natural type is α `matrix` $\rightarrow$ α `matrix` for all types α. However, the Caml type system does not allow this type for this function. That's because the function creates a matrix `m2` with a non-closed type (α `matrix`). The function `transpose_matrix` is not well-typed in Caml unless its parameter `m1` is restricted (by a type constraint) to a particular monomorphic type such as `int matrix`. We thus obtain a function that transposes only integer matrices. To transpose floating-point matrices, the function must be duplicated just to change the type constraint — as if we were programming in a monomorphic language like Pascal. We are therefore prevented from providing a library of generic functions operating on mutable matrices. The same holds for many well-known data structures (hash tables, graphs, balanced binary trees, B-trees), whose efficient implementation requires in-place modification of the structure: we cannot implement these structures once and for all by a library of generic functions.

To overcome this limitation, other polymorphic type systems have been proposed for mutable objects, that are more precise, but also more complex than the one considered above. These more precise type systems are able to type many generic functions over mutable structures. A well-known example is the Standard ML type system [64, 63, 71], that relies on the notion of "imperative variables" [92, 93], and its extension to "weak variables", as in the Standard ML of New Jersey implementation [3]. With this system, the generic functions over matrices, hash tables, etc. are given slightly restricted polymorphic types, called weakly polymorphic types, that are in practice general enough to support the reuse of these functions. However, serious deficiencies of this system appear when we start constructing other generic functions on top of these weakly polymorphic functions.

Weakness number one: weakly polymorphic functions do not interfere well with full functionality. For instance, the two code fragments below are not equivalent:

```
let makeref1 = function x → ref x in ...
let makeref2 = (function f → f)(function x → ref x) in ...
```

The `makeref1` function is weakly polymorphic; the `makeref2` function is monomorphic. Inserting an application of the identity function can therefore change the type of a program; this is strange. Other oddities show up when we partially apply higher-order functions to weakly polymorphic functions:

```
let mapref1 = map (function x → ref x) in ...
let mapref2 = function l → map (function x → ref x) l in ...
```

The `mapref1` function is monomorphic; the `mapref2` function is weakly polymorphic. Hence, the SML type system is not stable under eta-reduction; once again, this is strange.[2] The two oddities mentioned above reveal deep theoretical flaws in the design of the Standard ML type system; however, they do not cause serious programming difficulties, since they can be circumvented by local transformations over programs.

This is not the case for the second major weakness of the Standard ML typing for imperative constructs: generic functions that utilize mutable structures internally, to hold intermediate results,

[2] In the core ML type system, as in most known type systems, if an expression a has type τ under some assumptions, and if a eta-reduces to a', then a' also has type τ. This is not so in SML, since `mapref2` eta-reduces to `mapref1`.

cannot be assigned fully polymorphic types. There is no simple way to circumvent this weakness. This is a serious deficiency: the internal use of mutable structures is required for the efficient implementation of many generic functions. Example: the well-known algorithms over graphs (depth-first or breadth-first walk, determination of the connected components, topological sort, etc.) often allocate an array, stack or queue of vertices [88]. Here is an example of generic function written in this style:

```
let reverse1 l =
    let arg = ref l in
    let res = ref [] in
    while not is_null(!arg) do
        res := cons(head(!arg), !res);
        arg := tail(!arg)
    done;
    !res
```

This function reverses lists. It makes use of two local, private references. Here is another function that reverses lists, without using references this time:

```
let reverse2 l =
    let rec rev arg res =
        if null(arg) then res else rev (cons(head(arg),res)) (tail(res))
    in rev l []
```

Those two functions are exactly equivalent. Yet, the Standard ML type system assigns to `reverse1` a less general type than the one assigned to `reverse2`: the type of `reverse1` is weakly polymorphic, preventing `reverse1` from being applied to the polymorphic empty list for instance, while the type of `reverse2` is fully polymorphic, allowing `reverse2` to be applied to lists of any type. Most polymorphic functions that use `reverse1`, directly or indirectly, will be assigned a weakly polymorphic type, less general than if `reverse2` were used instead. Hence we cannot replace `reverse2` by `reverse1` in a library of functions over lists, even if those two functions have exactly the same semantics. This goes against the excellent principles of modular programming [70] that Standard ML precisely aims at supporting [55]: an implementation issue: the programming style (imperative vs. purely applicative), interferes with one of the specifications: the type.

As demonstrated above, Standard ML and its variants have not succeeded in their attempt to combine polymorphic typing and imperative constructs: they allow programming either generically, or in the imperative style, but not both. It is delicate to discuss the importance of this fact without engaging in the controversy between tenants of purely applicative programming ("everything else is messy") and defenders of the imperative style ("nothing else works"). Here, I take the viewpoint of a programmer who looks for an algorithmic language — a language that can directly express most known algorithms — for large-scale programming. Most algorithms are described in the literature by pseudo-code, mixing the declarative style ("let x be a vertex of the graph with minimal degree") and the imperative style ("take $E \leftarrow E \setminus \{x\}$ and iterate until $E = \emptyset$"). Translating these algorithms in a purely applicative language without increasing their complexity is often difficult. (The *Journal of Functional Programming* devotes a column, *Functional pearls*, to this kind of problems.) There are many efficient algorithms for which no purely applicative formulations are

known [75]. A huge translation work remains to be done before purely applicative languages can be considered as algorithmic languages.

Large-scale programming is made easier by decomposing the program in "autonomous" modules: modules that communicate as few information as possible with the remainder of the program, following a strict protocol. With mutable structures, each module can contain status variable, maintained internally without cooperation from the client modules. In a purely applicative language, this status information must be communicated to the clients (the outside world), and it is the client's responsibility to correctly propagate them. This is clearly error-prone.

From these two standpoints, I conclude that imperative features are required for algorithmic programming, in the current state of our knowledge. The fact that the polymorphic typing of these features is problematic therefore means that it remains to prove that an algorithmic language can actually benefit from polymorphic typing. One of the goals of this dissertation is to make such a proof, by proposing type systems where polymorphism interacts much better with the imperative programming style, without compromising type safety.

As further evidence that the issues raised by the polymorphic typing of mutable structures are not anecdotal, it should be pointed out that similar problems appear when attempting to apply a polymorphic type discipline to a number of other features from various non-purely applicative languages. I shall mention three such features: logical variables, communication channels, and continuation objects.

Logical variables are the basis of most of the "logical" programming languages; they can also be found in some proposals for unifying logical and applicative languages [89, 74, 1]. They allow to compute over objects with initially unknown parts, which are gradually reconstructed by accumulation of constraints during the program execution. Thanks to logical variables, the programmer can avoid to explicitly formulate a resolution strategy for these constraints.

Communication channels support the exchange of data between processes executing concurrently. They are essential to many calculi of communicating processes [59, 37]. (Here, I consider higher-order calculi, where the channels themselves are first-class values.) This extension of applicative languages supports the description of distributed algorithms. Also, some apparently sequential problems can be implemented more elegantly as several communicating processes.

Finally, continuation objects allow programs to capture and to manipulate the current state of their evaluation, and hence to define their own control structures, such as interleaving mechanisms (coroutines) or non-blind backtracking strategies specially designed for the problem at hand. Continuation objects can be found in Scheme [76], and in some implementations of ML [25].

I would have detailed the typing issues raised by logical variables, communication channels and continuations if I did not have to repeat the discussion of mutable structures almost word per word. The three extensions above can be typed in the same way as references or arrays. The resulting type system is sound without polymorphism; it becomes unsound when polymorphism is added without restrictions. In the case of logical variables and channels, it is easy to convince oneself of this fact, on examples similar to the first example in this Introduction. Just as references, logical variables can be updated in-place — only once, actually — once created. Just as references, channels allow transmitting data from a writer (an expression) to a reader (a context) that are not adjacent in the

source code. Hence, we immediately encounter the same problems as with mutable structures. For continuations, it is harder to make the connection with mutable structures: the naive polymorphic typing of continuations had long been believed sound (a "proof" of this fact was even published), until the first counterexamples appeared. The counterexamples are much more complicated than those for references; they rely on the fact that continuations allow restarting the evaluation of an expression in a context that does not correspond exactly to the context in which it was typed.

We are therefore faced by three useful extensions of purely applicative languages: logical variables, communication channels and continuations, that are not directly related to the imperative programming style, yet raise the same typing issues as mutable data structures. We can of course apply the known type systems for references to these three extensions. For instance, the Standard ML system has been applied to continuations [3, 101] and to channels [81]; the CAML system has been applied to logical variables [48]. As one could expect, the weaknesses of these systems reappear immediately, making it difficult, if not impossible, to write generic functions utilizing either logical variables, or channels, or continuations. As a consequence, it remains to establish that the following three language families can actually benefit from polymorphic type systems: 1- the attempts at integrating logic and functional programming based on logical variables, 2- the calculi of communicating processes that take channels as values, and 3- the manipulations of continuations as values. That's a fairly wide range of languages, and that's why this dissertation is not limited to proposing type systems for references: I also show that they successfully apply to communication channels and to continuation objects.[3]

The thesis I defend herein is that polymorphic typing can profitably be applied to a number of features from non-applicative languages. As we shall see in the remainder of this work, this extension of the field of polymorphic typing is not easy: large-scale modifications are required, either in the type system or in the semantics of some constructs. I argue that such is the price to pay to take polymorphism and its benefits out of the ghetto of purely applicative languages.

Outline

The remainder of this dissertation is organized as follows. The first chapter presents the core ML language, as a small applicative language equipped with Milner's polymorphic type system. I formalize the semantics and the typing rules for this language in the relational semantics framework, and I show the usual properties of soundness of the type system with respect to the evaluation rule, and existence of a principal type.

Chapter 2 enriches the applicative language from chapter 1 with three kinds of first-class objects: references, channels and continuations. References model mutable data structures; channels, the communication of data between concurrent processes; continuations, non-local control structures. I give the modified evaluation rules for the three extensions, and I show that their natural typing rules are not sound with respect to these evaluation rules, because of polymorphic types.

[3] The application to logical variables has been studied by Vincent Poirriez [74], for an older version of one of my systems. In the present dissertation, mutable structures, channels and continuations are studied in isolation. I cannot foresee any difficulty with combining mutable structures and continuations, as in the Scheme language. It is hard to give sensible semantics to the other combinations (mutable structures and channels, continuations and channels).

Chapter 3 presents a new polymorphic type system for references, channels and continuations. This system differs from Milner's system by extra restrictions over type generalization (certain type variables cannot be generalized: the variables occurring in dangerous position), on the one hand, and on the other hand by a refined typing of functional values, called closure typing. I first informally present this system and show that it correctly supports the imperative programming style. I then formalize the typing rules, and show their soundness with respect to the evaluation rules for references, for channels and for continuations. Finally, I prove the principal type property for this system.

In chapter 4, I demonstrate that the previous system is not completely satisfactory, since it rejects some "pure" programs that are well-typed in ML, and I describe a refinement of this system that avoids this difficulty. This refinement requires a fairly heavy technical apparatus; that's why I have chosen to present first the simpler system in chapter 3, for pedagogical purposes, even if it is subsumed by the refined system in chapter 4. I prove again the principal type property and the soundness with respect to the evaluation rules, and I show that this system is a conservative extension of ML.

Chapter 5 attempts to evaluate the contribution of this work from a practical standpoint. I compare the proposed type systems with other systems from the literature, on the one hand from the standpoint of expressiveness (is that useful, correct program recognized well-typed?), and on the other hand from the standpoint of modular programming (how difficult is it to specify types in module interfaces?)

Chapter 6 shows that the difficulties encountered with the polymorphic typing of references, channels and continuations are essentially specific to the ML semantics for generalization and specialization (the two constructs that introduce polymorphism), which I call the "by-value" semantics for polymorphism: if polymorphism is given alternate semantics, which I call the "by-name" semantics, the naive typing rules for references, channels and continuations are now sound. This is the main technical result in this chapter, and it suggests that a language with polymorphism by name could be an interesting alternative to ML.

Readings

Here are some possible paths through this dissertation. They all assume some familiarity with the ML language, as provided by the first chapters of [71, 51], or by [34, 19], and also with the ideas from relational semantics [73, 42]. To get an overview of the problem, it suffices to read chapter 1 (especially sections 1.1 and 1.3) and chapter 2 (sections 2.1.2, 2.2.2 and 2.3.2 can be skimmed over). To get an idea of the solutions I propose, add sections 3.1 and 4.1, chapter 5, and section 6.1. The reader interested in soundness and principality proofs should read chapters 1, 2, 3 and 6 in their entirety. The careful reading of chapter 4 is recommended to experts only.

Conventions

In this dissertation, the author uses "I" to express his opinions, viewpoints, methodological choices and results obtained, all personal matters that should not be hidden behind the editorial "we". "We" denotes the reader and the author, joining hands for pedagogical purposes. The original

French for this work makes heavy use of the French impersonal pronoun "on", to state claims that are universally true. These have also be translated by "we".

To avoid heavy style, formal definitions are run into the text, instead of being displayed like propositions. Definitions are however emphasized by setting in SMALL CAPITALS the word being formally defined.

Some paragraphs are marked "Context". They briefly describe alternate approaches, and situate my work with respect to them. This provides the knowledgeable reader with some landmarks, and the curious mind with some pointers to the literature. Bourbaki fans will consider these paragraphs useless.

Chapter 1

A polymorphic applicative language

The full ML language is too complex to allow easy formal reasoning about it. This chapter introduces a small applicative language with a polymorphic type discipline, that presents the main features of ML, while remaining much simpler. This language is the basis for the extensions studied in the next chapters.

The present chapter contains nothing that has not been already written several times [58, 22, 21, 92, 17, 64, 63, 93]. Nonetheless, I attempt to give a self-contained presentation with few prerequisites. This presentation introduces most of the notations and proof techniques that are widely used in the remainder of this work.

1.1 Syntax

The EXPRESSIONS of the language (written a, possibly with a prime or a subscript) are the elements of the term algebra generated by the following grammar:

a	::=	cst	constant
		x	identifier
		$op(a)$	primitive application
		f `where` $f(x) = a$	recursive function
		$a_1(a_2)$	application
		`let` $x = a_1$ `in` a_2	the `let` binding
		(a_1, a_2)	pair construction

In this grammar, x and f range over an infinite set `Ident` of IDENTIFIERS. Expressions of the form cst are CONSTANTS, taken from a set `Cst` that we do not specify here, for the sake of generality. For instance, `Cst` might contain integer numbers, the true values `true` and `false`, character strings, etc.

In the expressions $op(a)$, op ranges over a set of OPERATORS `Op` which is also left unspecified. Typically, `Op` contains the usual arithmetic operations over integers, the comparisons between integers, the two projections `fst` and `snd`; even the usual conditional constructs such as `if`...`then`...`else`... can be treated as operators.

The expression f `where` $f(x) = a$ defines the function whose parameter is x and whose result is the value of a. The identifier f is the internal name for the function. Inside the expression a, the identifier f is considered bound to the function being defined. The defined function can therefore be recursive. In the examples below, we write $\lambda x.\, a$ for non-recursive functions. This is an abbreviation for f `where` $f(x) = a$, where f is any identifier that does not appear in a.

Context. The toy languages studied in the literature usually do not feature recursion. As a consequence, they are unrealistic: it is hard to write interesting examples in these languages. Also, they often possess properties that do not hold for any "real" programming language: "all well-typed programs terminate", for instance. Moreover, it is not easy to add recursion as an afterthought: the proofs, and sometimes even the formalism, must be entirely revised. That's why I have chosen to consider a recursive language from the beginning.

In the ML language, recursion is presented as an extension of the `let` construct: `let rec`, `let val rec`, `let fun` depending on the dialects. I prefer to present recursion as an extension of λ-abstraction; this way, it is apparent that only functions can be defined recursively, and not any value. (In ML, extra restrictions are required to rule out definitions such as `let rec x = x + 1`, that cannot be executed.) In Standard ML syntax, f `where` $f(x) = a$ is written `let fun` $f\ x = a$ `in` f `end`; in Caml syntax, `let rec` $f\ x = a$ `in` f, or f `where rec` $f\ x = a$. □

Example. Here are two syntactically correct expressions of the language:

$$\texttt{let fact} = \texttt{f where f(n)} = \texttt{ifthenelse}(\leq(\texttt{n},0),(1,\times(\texttt{n},\texttt{f}(-(\texttt{n},1))))) \texttt{ in fact}(2)$$

$$\texttt{let double} = \lambda\texttt{f}.\,\lambda\texttt{x}.\,\texttt{f}(\texttt{f}(\texttt{x})) \texttt{ in double}(\lambda\texttt{x}.\,+(\texttt{x},\texttt{true}))(1)$$

For the sake of readability, we take the liberty to use the familiar infix notation for binary and ternary operators, as well as the usual precedence and associativity rules. We therefore write the expressions as:

$$\texttt{let fact} = \texttt{f where f(n)} = \texttt{if n} \leq \texttt{0 then 1 else n} \times \texttt{f(n} - \texttt{1) in fact}(2)$$

$$\texttt{let double} = \lambda\texttt{f}.\,\lambda\texttt{x}.\,\texttt{f}(\texttt{f}(\texttt{x})) \texttt{ in double}(\lambda\texttt{x.x} + \texttt{true}))(1)$$

□

We write $\mathcal{I}(a)$ for the set of the FREE IDENTIFIERS of expression a. The binding constructs are `let` and `where`. The precise definition of $\mathcal{I}$ is:

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{I}(cst) & = & \emptyset & \mathcal{I}(x) & = & \{x\} \\
\mathcal{I}(op(a)) & = & \mathcal{I}(a) & \mathcal{I}(f \texttt{ where } f(x) = a) & = & \mathcal{I}(a) \setminus \{f, x\} \\
\mathcal{I}(a_1(a_2)) & = & \mathcal{I}(a_1) \cup \mathcal{I}(a_2) & \mathcal{I}(\texttt{let } x = a_1 \texttt{ in } a_2) & = & \mathcal{I}(a_1) \cup (\mathcal{I}(a_2) \setminus \{x\}) \\
\mathcal{I}(a_1, a_2) & = & \mathcal{I}(a_1) \cup \mathcal{I}(a_2) & & &
\end{array}$$

1.2 Semantics

We are now going to give a meaning to the language expressions, by defining an evaluation mechanism that maps expressions to the results of their evaluations. To do so, we use the formalism of relational semantics. It consists in defining, by a system of axioms and inference rules, a predicate between expressions and results, the EVALUATION JUDGEMENT, telling whether a given expression can evaluate to a given result.

Context. This method is known as "structured operational semantics" (SOS) in Northern Great Britain [73], and as "natural semantics" in Southern France [42]. In contrast with denotational semantics, relational semantics require no complex mathematical framework; it also remains closer to direct execution on a computer [16]. Other mathematically simple approaches include rewriting semantics (for the extensions in next chapter, simple rewriting does not suffices, rewriting rules with contexts are required [26, 27, 101]), and also translating expressions into code for an abstract machine [46, 18]. □

Actually, the evaluation judgement depends on extra arguments, that represent the context in which evaluation takes place. For the language considered in the present chapter, only one such extra argument is required: the evaluation environment, that records the bindings of identifiers to values. The evaluation judgement is therefore $e \vdash a \Rightarrow r$, read: "in the evaluation environment e, the expression a evaluates to the result r".

1.2.1 Semantic objects

The time has come to define precisely the various kinds of objects that appear in the semantics: the RESULTS, the VALUES, and the EVALUATION ENVIRONMENTS. These semantic objects range over the term algebra defined by the grammar below:

Results:	r	::=	v	normal result (a value)
		|	`err`	error result
Value:	v	::=	cst	base value
		|	(v_1, v_2)	value pair
		|	(f, x, a, e)	functional value (closure)
Environments:	e	::=	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	

In environment expressions e, we assume the identifiers $x_1 \ldots x_n$ to be distinct. Here and elsewhere, the term $e = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ is interpreted as a partial mapping with finite domain from identifiers to values: the mapping that associates v_i to x_i, for all i from 1 to n, and that is undefined on the other identifiers. We improperly call e a finite mapping from identifiers to values. The empty mapping is written $[\,]$. We write $\mathrm{Dom}(e)$ for the DOMAIN of e, that is $\{x_1, \ldots, x_n\}$, and $\mathrm{Codom}(e)$ for its RANGE, that is $\{v_1, \ldots, v_n\}$. If x belongs to $\mathrm{Dom}(e)$, we write $e(x)$ for the value associated to x in e. Finally, we define the extension of e by v in x, written $e + x \mapsto v$, by:

$$
\begin{aligned}
[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] + x \mapsto v &= [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, x \mapsto v] \\
&\quad \text{if } x \notin \{x_1, \ldots, x_n\} \\
[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] + x \mapsto v &= [x_1 \mapsto v_1, \ldots, x_{i-1} \mapsto v_{i-1}, x \mapsto v, x_{i+1} \mapsto v_{i+1}, \ldots, x_n \mapsto v_n] \\
&\quad \text{if } x = x_i
\end{aligned}
$$

Hence, we have as expected

$$
\begin{aligned}
\mathrm{Dom}(e + x \mapsto v) &= \mathrm{Dom}(e) \cup \{x\} \\
(e + x \mapsto v)(x) &= x \\
(e + x \mapsto v)(y) &= e(y) \text{ for all } y \in \mathrm{Dom}(e),\ y \neq x.
\end{aligned}
$$

1.2.2 Evaluation rules

Once these definitions are set up, we are able to define the evaluation judgement $e \vdash a \Rightarrow r$. The definition is presented as a set of axioms and inference rules. Directions for use: an axiom P allows to conclude that proposition P holds; an inference rule such as

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{P}$$

allows to prove that the conclusion P holds as soon as all premises $P_1 \ldots P_n$ have been proved. Axioms and rules can contain (meta-) variables. These variables are implicitly quantified universally at the beginning of each rule.

$$e \vdash cst \Rightarrow cst \qquad \frac{x \in \mathrm{Dom}(e)}{e \vdash x \Rightarrow e(x)} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x \Rightarrow \texttt{err}}$$

The axiom on the left states that a constant evaluates to itself. The middle rule states that an identifier evaluates to the value bound to it in the environment e, provided that this identifier does belong to the domain of e. Otherwise, there is no sensible value for the expression x. In this case, the right rule therefore returns the response "an error has occurred", denoted by `err`.

$$e \vdash (f \ \texttt{where} \ f(x) = a) \Rightarrow (f, x, a, e)$$

A function evaluates to a CLOSURE: an object that combines the unevaluated body of a function (the triple f, x, a) with the environment e at the time the function was defined.

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e \vdash a_2 \Rightarrow v_2}{e \vdash (a_1, a_2) \Rightarrow (v_1, v_2)} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}}$$

A pair expression normally evaluates to a pair of values, in the obvious way. However, if the evaluation of one of the two pair components causes an error (such as evaluating an unbound identifier), then the evaluation of the whole pair also returns an error.

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e + x \mapsto v_1 \vdash a_2 \Rightarrow r_2}{e \vdash \texttt{let} \ x = a_1 \ \texttt{in} \ a_2 \Rightarrow r_2} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash \texttt{let} \ x = a_1 \ \texttt{in} \ a_2 \Rightarrow \texttt{err}}$$

A `let` binding evaluates its first argument, then associates the value obtained to the bound identifier, and evaluates its second argument in the environment thus enriched. The result obtained is also the result for the whole `let` expression. In the case where the evaluation of the first argument causes an error, the evaluation of the `let` expression immediately stops on an error.

$$\frac{e \vdash a_1 \Rightarrow (f, x, a_0, e_0) \qquad e \vdash a_2 \Rightarrow v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \Rightarrow r_0}{e \vdash a_1(a_2) \Rightarrow r_0}$$

The rule for applications $a_1(a_2)$ is the most complex. The expression a_1 must evaluate to a closure (f, x, a_0, e_0). The argument a_2 must evaluate to a value v_2. We then evaluate the function body a_0 from the closure in the environment e_0 from the closure, after having enriched e_0 by two bindings: the function parameter x is bound to the value v_2 of the argument; the function name f is bound to the closure (f, x, a_0, e_0) itself. The latter binding ensures that the evaluation of a_0, which can refer to f, will find the right value for f in the environment.

$$\frac{e \vdash a_1 \Rightarrow r_1 \qquad r_1 \text{ does not match } (f, x, a_0, e_0)}{e \vdash a_1(a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash a_1(a_2) \Rightarrow \texttt{err}}$$

Here are the two error cases for function application. The application is $a_1(a_2)$ is meaningless if a_1 evaluates in a value other than a closure. Example: $1(2)$. The leftmost rule returns `err` in this case. The rightmost rule ensures that the response `err` propagates through applications: if the evaluation of the argument part stops on an error, the evaluation of the whole application also stops on an error, even if the function does not use its argument.

It remains to give evaluation rules for each of the primitives in the set `Op`. As an example, here are the rules for integer addition, and for the conditional construct:

$$\frac{e \vdash a \Rightarrow (cst_1, cst_2) \qquad cst_1 \in \texttt{Int} \qquad cst_2 \in \texttt{Int} \qquad cst_1 + cst_2 = cst}{e \vdash +(a) \Rightarrow cst}$$

$$\frac{e \vdash a \Rightarrow r \qquad r \text{ is not } (cst_1, cst_2) \text{ with } cst_1 \in \texttt{Int} \text{ and } cst_2 \in \texttt{Int}}{e \vdash +(a) \Rightarrow \texttt{err}}$$

$$\frac{e \vdash a_1 \Rightarrow \texttt{true} \qquad e \vdash a_2 \Rightarrow r_2}{e \vdash \texttt{if } a_1 \texttt{ then } a_2 \texttt{ else } a_3 \Rightarrow r_2} \qquad \frac{e \vdash a_1 \Rightarrow \texttt{false} \qquad e \vdash a_3 \Rightarrow r_3}{e \vdash \texttt{if } a_1 \texttt{ then } a_2 \texttt{ else } a_3 \Rightarrow r_3}$$

$$\frac{e \vdash a_1 \Rightarrow r_1 \qquad r_1 \notin \texttt{Bool}}{e \vdash \texttt{if } a_1 \texttt{ then } a_2 \texttt{ else } a_3 \Rightarrow \texttt{err}}$$

Here and elsewhere, we associate to these rules a relation between environments, expressions and values, written $e \vdash a \Rightarrow v$, which is the least defined relation that satisfies the inference rules. In the following, we frequently use the following characterization of this relation: (e, a, v) are in relation by $\cdot \vdash \cdot \Rightarrow \cdot$ if and only if there exists a finite derivation of the judgement $e \vdash a : v$ from the axioms, by repeated applications of the inference rules [73].

Context. Introducing the `err` response and the rules that return this response (the so-called "error rules") is the traditional answer to the problem of distinguishing erroneous programs from non-terminating programs: without error rules, when an expression does not admit any (finite) evaluation derivation, we do not know whether this is due to the fact that at some point no regular evaluation rule match (case of a run-time type error), or to the fact that the only possible derivations are infinite (case of a non-terminating program).

Besides complicating the semantics, the error rules put unfortunate constraints over the evaluation strategy — which should not interfere at this level of description. For instance, the two error rules for pairs proposed above:

$$\frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}}$$

imply the parallel evaluation of the two pair components ("angelic non-determinism"), since we must return an error in all cases where one component does not terminate while the other causes an error. To allow sequential implementation, we must break the symmetry between the two error rules, and take for instance:

$$\frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_1 \Rightarrow v_1 \qquad e \vdash a_2 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}}$$

But, then, we have imposed the evaluation order for the pair: left to right. Alternate, more recent approaches that do not require error rules avoid this difficulty. These approaches make use of richer notions of derivations: partial derivations [32], or possibly infinite derivations [20].

This indirect specification of the evaluation order through error rules is less of a problem for statically-typed languages: when we evaluate well-typed terms only, the error cases never occur at run-time; hence, we can adopt an evaluation strategy that does not respect the error rules. That's why I indulged in non-sequential error rules for pairs and function applications: these rules are simpler and more elegant than the sequential rules. □

1.3 Typing

We are now going to equip the language with a type system. The typing rules associate types (more precisely, type expressions) to expressions, in the same way as the evaluation rules associate evaluation results to expressions. We use Milner's type system [58, 22], which is the basis for the type systems of the ML language and several functional languages (Miranda, Hope, Haskell). The distinctive feature of Milner's system is polymorphism, taking into account the fact that an expression can belong to several different types.

1.3.1 Types

We assume given a set `TypBas` of BASE TYPES (such as `int`, the type of integers, and `bool`, the type of truth values), and an infinite set `VarTyp` of TYPE VARIABLES.

ι	$\in$	`TypBas`	$= \{\texttt{int}; \texttt{bool}; \ldots\}$	base types
α, β, γ	$\in$	`VarTyp`		type variables

We define the set `Typ` of TYPE EXPRESSIONS (or simply TYPES), with typical element τ, by the following grammar:

τ	$::=$	ι	base type
	$\mid$	α	type variable
	$\mid$	$\tau_1 \rightarrow \tau_2$	function type
	$\mid$	$\tau_1 \times \tau_2$	product type

We write $\mathcal{F}(\tau)$ for the set of type variables that are free in the type τ. This set is defined by:

$$\begin{aligned}\mathcal{F}(\iota) &= \emptyset \\ \mathcal{F}(\alpha) &= \{\alpha\} \\ \mathcal{F}(\tau_1 \to \tau_2) &= \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) \\ \mathcal{F}(\tau_1 \times \tau_2) &= \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2)\end{aligned}$$

1.3.2 Substitutions

The substitutions considered here are finite mappings from type variables to type expressions. They are written φ, ψ.

$$\text{Substitutions:} \quad \varphi, \psi \ ::= \ [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$$

A substitution φ naturally extends to an homomorphism of type expressions, written $\overline{\varphi}$, and defined by:

$$\begin{aligned}\overline{\varphi}(\iota) &= \iota \\ \overline{\varphi}(\alpha) &= \varphi(\alpha) \text{ if } \alpha \in \mathrm{Dom}(\varphi) \\ \overline{\varphi}(\alpha) &= \alpha \text{ if } \alpha \notin \mathrm{Dom}(\varphi) \\ \overline{\varphi}(\tau_1 \to \tau_2) &= \overline{\varphi}(\tau_1) \to \overline{\varphi}(\tau_2) \\ \overline{\varphi}(\tau_1 \times \tau_2) &= \overline{\varphi}(\tau_1) \times \overline{\varphi}(\tau_2)\end{aligned}$$

From now on, we do not distinguish anymore between φ and its extension $\overline{\varphi}$, and we write φ for both.

The following property shows the effect of a substitution over the free variables of a type.

Proposition 1.1 (Substitution and free variables) *For all types τ and all substitutions φ, we have:*

$$\mathcal{F}(\varphi(\tau)) = \bigcup_{\alpha \in \mathcal{F}(\tau)} \mathcal{F}(\varphi(\alpha))$$

Proof: straightforward structural induction over τ. □

1.3.3 Type schemes

We define the set `SchTyp` of TYPE SCHEMES, with typical element σ, by the following grammar:

$$\sigma ::= \forall \alpha_1 \ldots \alpha_n . \tau \qquad \text{type scheme}$$

In this syntax, the quantified variables $\alpha_1 \ldots \alpha_n$ are treated as a set of variables: their relative order is not significant, and they are assumed to be distinct. We do not distinguish between the type τ and the trivial type scheme $\forall . \tau$, and we write τ for both. We identify two type schemes that differ only by a renaming of the variables bound by $\forall$ (alpha-conversion operation), and by

the introduction or suppression of quantified variables that are not free in the type part. More precisely, we quotient the set of schemes by the following two equations:

$$\begin{aligned} \forall\alpha_1 \ldots \alpha_n.\tau &= \forall\beta_1 \ldots \beta_n.[\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau) \\ \forall\alpha\alpha_1 \ldots \alpha_n.\tau &= \forall\alpha_1 \ldots \alpha_n.\tau \quad \text{if } \alpha \notin \mathcal{F}(\tau) \end{aligned}$$

The free variables in a type scheme are:

$$\mathcal{F}(\forall\alpha_1 \ldots \alpha_n.\tau) = \mathcal{F}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\}.$$

This definition is compatible with the two equations over types schemes, as can easily be checked using proposition 1.1.

We extend substitutions to type schemes as follows:

$$\varphi(\forall\alpha_1 \ldots \alpha_n.\tau) = \forall\alpha_1 \ldots \alpha_n.\varphi(\tau),$$

assuming, after renaming the α_i if necessary, that the α_i are OUT OF REACH for φ; that is, none of the α_i is in the domain of φ, and none of the α_i is free in one of the types in the range of φ. We easily check that this definition is compatible with the two equations over schemes.

Proposition 1.1 also holds with the type τ replaced by a scheme σ.

1.3.4 Typing environments

A typing environment, written E, is a finite mapping from identifiers to type schemes:

$$E \quad ::= \quad [x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n]$$

The image $\varphi(E)$ of an environment E by a substitution φ is defined in the obvious way:

$$\varphi([x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n]) = [x_1 \mapsto \varphi(\sigma_1), \ldots, x_n \mapsto \varphi(\sigma_n)].$$

The operator $\mathcal{F}$ is extended to typing environment, by taking that a type variable is free in E if and only if it is free in $E(x)$ for some x:

$$\mathcal{F}(E) = \bigcup_{x \in \mathrm{Dom}(E)} \mathcal{F}(E(x))$$

1.3.5 Typing rules

We are now going to give the typing rules for the language, using the same formalism as for the evaluation rules. The following inference rules define the TYPING JUDGEMENT $E \vdash a : \tau$, read: "in environment E, the expression a has type τ". The environment E associates a type scheme to each identifier that can appear in expression a.

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

The typing rule for identifiers states that an identifier x in an expression can be given any type that is an instance of the type scheme associated to x in the typing environment. We say that the type τ is an INSTANCE of the type scheme $\sigma = \forall\alpha_1 \ldots \alpha_n.\, \tau_0$, and we write $\tau \leq \sigma$, if and only if there exists a substitution φ whose domain is a subset of $\{\alpha_1 \ldots \alpha_n\}$ such that τ is equal to $\varphi(\tau_0)$. The relation $\tau \leq \sigma$ is obviously compatible with the two equations over schemes (alpha-conversion, and useless quantified variable elimination).

$$\frac{E + f \mapsto (\tau_1 \to \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \to \tau_2} \qquad \frac{E \vdash a_1 : \tau_2 \to \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

A function definition has type $\tau_1 \to \tau_2$ as soon as the function body has type τ_2, under the extra assumptions that its formal parameter x has type τ_1, and its internal name f has type $\tau_1 \to \tau_2$. A function application is well-typed as soon as the type of the argument corresponds to the type of the function parameter; the type of the function result is the type of the whole application node.

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let} \ x = a_1 \ \texttt{in} \ a_2 : \tau_2}$$

The `let` construct is the only one that introduces polymorphic types (that is, non-trivial type schemes) in the environment. This is due to the generalization operator `Gen`, which builds a type scheme from a type and a type environment, as follows:

$$\texttt{Gen}(\tau, E) = \forall\alpha_1 \ldots \alpha_n.\, \tau \quad \text{where} \quad \{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau) \setminus \mathcal{F}(E).$$

We shall see later that if a_1 belongs to the type τ under the assumptions E, and if α is not free in E, then a_1 also belongs to the type $[\alpha \mapsto \tau_1](\tau)$ for all types τ_1. This intuitively explains why α can be universally quantified, since it can be substituted by any type.

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \qquad \frac{\tau \leq \texttt{TypCst}(cst)}{E \vdash cst : \tau} \qquad \frac{\tau_1 \to \tau_2 \leq \texttt{TypOp}(op) \qquad E \vdash a : \tau_1}{E \vdash op(a) : \tau_2}$$

The typing of pair expressions is obvious. For constants and primitives, we assume given two functions $\texttt{TypCst} : \texttt{Cst} \to \texttt{SchTyp}$ and $\texttt{TypOp} : \texttt{Op} \to \texttt{SchTyp}$, that assign type schemes to constants and operators. For instance, we can take:

$$\begin{array}{rcll} \texttt{TypCst}(i) & = & \texttt{int} & (i = 0, 1, \ldots) \\ \texttt{TypCst}(b) & = & \texttt{bool} & (b = \texttt{true or false}) \\ \texttt{TypOp}(+) & = & \texttt{int} \times \texttt{int} \to \texttt{int} & \\ \texttt{TypOp}(\texttt{ifthenelse}) & = & \forall\alpha.\, \texttt{bool} \times \alpha \times \alpha \to \alpha & \end{array}$$

Example. The expression `let id = (f where f(x) = x) in id(1)` has type `int`. That's because the following derivation is valid:

$$\dfrac{\dfrac{\dfrac{t \le t}{[\mathtt{f} \mapsto t \to t, \mathtt{x} \mapsto t] \vdash \mathtt{x} : t}}{[\,] \vdash \mathtt{f\ where\ f(x) = x} : t \to t} \qquad \dfrac{\dfrac{\mathtt{int} \to \mathtt{int} \le \forall t.\, t \to t}{E \vdash \mathtt{id} : \mathtt{int} \to \mathtt{int}} \qquad \dfrac{\mathtt{int} \le \mathtt{int}}{E \vdash 1 : \mathtt{int}}}{E = [\mathtt{id} \mapsto \forall t.\, t \to t] \vdash \mathtt{id(1)} : \mathtt{int}}}{[\,] \vdash \mathtt{let\ id = (f\ where\ f(x) = x)\ in\ id(1)} : \mathtt{int}}$$

□

Context. The presentation given above is essentially the one of the definition of Standard ML [64], and slightly differs from Damas and Milner's presentation [22]. The Damas-Milner system assigns type schemes to expressions, not simple types, and features two separate rules for generalization and instantiation. In the presentation above, these two operations are integrated into the `let` rule (for generalization) and into the rules for identifiers and constants (for instantiation). The presentation above has one advantage: the rules are syntax-directed — that is, the shape of the expression determines the unique rule that applies; and the premises contain only strict subexpressions of the expression in the conclusion. This makes many proofs easier, especially those for the type inference algorithm. □

1.3.6 Properties of the typing rules

As we said above about the typing rule for `let`, the typing judgement is stable under substitution: if we can prove $E \vdash a : \tau$, then the judgements obtained by substituting any types for any variables in E and τ are still provable.

Proposition 1.2 (Typing is stable under substitution) *Let a be an expression, τ be a type, E be a typing environment and φ be a substitution. If $E \vdash a : \tau$, then $\varphi(E) \vdash a : \varphi(\tau)$.*

Proof: by structural induction on a. We show the two cases that do not immediately follow from the induction hypothesis.

• **Case $a = x$.** We must have $\tau \le E(x)$, with $E(x) = \forall \alpha_1 \ldots \alpha_n.\, \tau_0$. After renaming if necessary, we can assume that the α_i are out of reach for φ. Let ψ be a substitution over the α_i such that $\tau = \psi(\tau_0)$. Define a substitution θ with domain $\{\alpha_1 \ldots \alpha_n\}$ by $\theta(\alpha_i) = \varphi(\psi(\alpha_i))$. We have:

$$\begin{array}{rcccll} \theta(\varphi(\alpha_i)) & = & \theta(\alpha_i) & = & \varphi(\psi(\alpha_i)) & \text{for all } i \\ \theta(\varphi(\beta)) & = & \varphi(\beta) & = & \varphi(\psi(\beta)) & \text{for all } \beta \text{ that is not an } \alpha_i \end{array}$$

Hence $\theta(\varphi(\tau_0)) = \varphi(\psi(\tau_0)) = \varphi(\tau)$, showing that $\varphi(\tau)$ is an instance of $\varphi(E(x))$. We can therefore derive $\varphi(E) \vdash x : \varphi(\tau)$.

• **Case $a = (\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2)$.** The typing derivation ends up with:

$$\dfrac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \mathtt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau}$$

By definition, $\texttt{Gen}(\tau_1, E)$ is equal to $\forall\alpha_1 \ldots \alpha_n.\, \tau_1$, with $\{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau_1) \setminus \mathcal{F}(E)$. Let $\beta_1, \ldots, \beta_n$ be type variables out of reach for φ, and not free in E. Let ψ be the substitution $\varphi \circ [\alpha_i \mapsto \beta_i]$. We have $\psi(E) = \varphi(E)$, since the variables α_i are not free in E.

We apply the induction hypothesis twice: once to the left premise, with the substitution ψ; once to the right premise, with the substitution φ. We obtain proofs of:

$$\psi(E) \vdash a_1 : \psi(\tau_1) \qquad\qquad \varphi(E) + x \mapsto \varphi(\texttt{Gen}(\tau_1, E)) \vdash a_2 : \varphi(\tau_2)$$

We now compute $\texttt{Gen}(\psi(\tau_1), \psi(E))$, using proposition 1.1.

$$\mathcal{F}(\psi(\tau_1)) \setminus \mathcal{F}(\psi(E)) = \left(\bigcup_{\alpha \in \mathcal{F}(\tau_1)} \mathcal{F}(\psi(\alpha)) \right) \setminus \left(\bigcup_{\alpha \in \mathcal{F}(E)} \mathcal{F}(\psi(\alpha)) \right)$$

By construction of ψ, we have $\psi(\alpha_i) = \varphi(\beta_i) = \beta_i$. Moreover, for all variables α that are not an α_i, the term $\psi(\alpha)$ is equal to $\varphi(\alpha)$ and does not contain any of the β_i. Since the α_i are free in τ_1, but not free in E, we therefore have:

$$\beta_i \in \bigcup_{\alpha \in \mathcal{F}(\tau_1)} \mathcal{F}(\psi(\alpha)) \qquad\qquad \beta_i \notin \bigcup_{\alpha \in \mathcal{F}(E)} \mathcal{F}(\psi(\alpha)).$$

Hence $\{\beta_1 \ldots \beta_n\} \subseteq \mathcal{F}(\psi(\tau_1)) \setminus \mathcal{F}(\psi(E))$. We now show the converse inclusion. Let $\beta \in \mathcal{F}(\psi(\tau_1))$, such that β is not one of the β_i. Let $\alpha \in \mathcal{F}(\tau_1)$ such that $\beta \in \mathcal{F}(\psi(\alpha))$ (the existence of α follows from proposition 1.1). Necessarily, α is not one of the α_i; otherwise, β would be one of the β_i. Hence $\alpha \in \mathcal{F}(E)$, and $\beta \in \bigcup_{\alpha \in \mathcal{F}(E)} \mathcal{F}(\psi(\alpha))$ and $\beta \notin \mathcal{F}(\psi(\tau_1)) \setminus \mathcal{F}(\psi(E))$. Hence $\{\beta_1 \ldots \beta_n\} = \mathcal{F}(\psi(\tau_1)) \setminus \mathcal{F}(\psi(E))$, and

$$\texttt{Gen}(\psi(\tau_1), \psi(E)) = \forall\beta_1 \ldots \beta_n.\, \psi(\tau_1) = \varphi(\forall\alpha_1 \ldots \alpha_n.\, \tau) = \varphi(\texttt{Gen}(\tau_1, E))$$

by definition of the image of a scheme by a substitution. Since $\psi(E)$ and $\varphi(E)$ are identical, the two derivations obtained by applying the induction hypothesis therefore prove that:

$$\varphi(E) \vdash a_1 : \psi(\tau_1) \qquad \varphi(E) + x \mapsto \texttt{Gen}(\psi(\tau_1), \varphi(E)) \vdash a_2 : \varphi(\tau_2).$$

Applying the `let` typing rule to these premises, we obtain the desired result:

$$\varphi(E) \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \varphi(\tau_2).$$

□

Another property of the typing rules, relevant to the type inference algorithm: if we can prove that a has type τ under the assumptions E, then we can prove it under stronger assumptions E'. To make this notion of "strength" more precise, we take that a scheme σ' is MORE GENERAL than a scheme σ, and we write $\sigma' \geq \sigma$, if all instances of σ are also instances of σ'.

Proposition 1.3 *Let E, E' be two typing environments such that* $\mathrm{Dom}(E) = \mathrm{Dom}(E')$, *and $E'(x) \geq E(x)$ for all $x \in \mathrm{Dom}(E)$. If $E \vdash a : \tau$, then $E' \vdash a : \tau$.*

Proof: easy structural induction on a. The base case $a = x$ is straightforward by hypothesis over E and E'. For the case $a = (\texttt{let}\ x = a_1\ \texttt{in}\ a_2)$, we first show that $\mathcal{F}(E') \subseteq \mathcal{F}(E)$, implying $\texttt{Gen}(\tau_1, E') \geq \texttt{Gen}(\tau_1, E)$, where τ_1 is the type of a_1. We can therefore apply the induction hypothesis to the typing derivation for a_2; the result follows. □

1.4 Type soundness

One of the main goals of static typing is to exclude the programs that can cause a type error at run-time. We have defined the typing rules independently from the evaluation rules. It therefore remains to show that the initial goal is reached, that is, that the typing rules are *sound* with respect to the evaluation rules. The main result in this section is the following: no closed, well-typed expression can evaluate to `err`.

Proposition 1.4 (Weak soundness) *Let a be an expression, τ be a type, and r be a response. If $[\,] \vdash a : \tau$ and $[\,] \vdash a \Rightarrow r$, then r is not* `err`.

Context. This claim does not state that there always exists a value v to which a evaluates: a well-typed program may fail to terminate. The aim of a type system for a programming language is not to ensure termination, but to guarantee the structural conformity of data. □

It is difficult to prove directly the weak soundness claim, because it does not lend itself to an inductive proof: to show that $a_1(a_2)$ cannot evaluate to `err`, it is not enough to show that neither a_1 nor a_2 can evaluate to `err`; even then, the application $a_1(a_2)$ can cause a run-time type error, for instance if a_1 does not evaluate to a closure. We are therefore going to show a stronger result: the evaluation of a closed term with type τ, if it terminates, not only does not return `err`, but also returns a value that, semantically, does belong to the type τ. For instance, an expression with type `int` can only evaluate to an integer value.

1.4.1 Semantic typing

Before going any further, we must therefore define precisely what it means for a value to semantically belong to a type. We define the following three semantic typing predicates:

$\models v : \tau$	the value v belongs to the type τ
$\models v : \sigma$	the value v belongs to the scheme σ
$\models e : E$	the values contained in the evaluation evironment e belong to the corresponding type schemes in E

These predicates are defined as follows:

- $\models cst$: `int` if cst is an integer
- $\models cst$: `bool` if cst is `true` or `false`
- $\models (v_1, v_2) : \tau_1 \times \tau_2$ if $\models v_1 : \tau_1$ and $\models v_2 : \tau_2$
- $\models (f, x, a, e) : \tau_1 \to \tau_2$ if there exists a typing environment E such that

$$\models e : E \quad \text{and} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \to \tau_2$$

- $\models v : \sigma$ if for all types τ such that $\tau \leq \sigma$, we have $\models v : \tau$
- $\models e : E$ if $\mathrm{Dom}(e) = \mathrm{Dom}(E)$, and for all $x \in \mathrm{Dom}(e)$, we have $\models e(x) : E(x)$.

This definition is well-founded by structural induction over the value part v. In all cases except for schemes, the definition of $\models$ over the value v appeals to $\models$ over proper subterms of v. The case for type schemes appeals to $\models$ on the same value, but with the type scheme replaced by a simple type; hence, another case of the definition is going to be used just after, and this case considers only proper subterms of v.

Context. Appealing to the type judgement to define $\models$ over functional values is a technical trick due to Tofte [92, 93]. A more traditional definition would be "a functional value belongs to the type $\tau_1 \to \tau_2$ if it maps any value of type τ_1 to a value of type τ_2". More formally: $\models (f, x, a, e) : \tau_1 \to \tau_2$ if, for all values v_1 and results r_2 such that

$$\models v_1 : \tau_1 \quad \text{and} \quad e + f \mapsto (f, x, a, e) + x \mapsto v_1 \vdash a \Rightarrow r_2,$$

we have $r_2 \neq \texttt{err}$ and $\models r_2 : \tau_2$. That's the continuity condition that appears, for instance, in ideal models [56]. However, such a definition is problematic in the presence of recursive functions: the soundness proof cannot be proved by simple induction, because in the case of a `where` construct we cannot conclude $\models (f, x, a, e) : \tau_1 \to \tau_2$ unless we have already proved that $\models (f, x, a, e) : \tau_1 \to \tau_2$. It seems that we would have to revert to more complex proof principles, such as co-induction [62], or to richer mathematical frameworks, such as domains [33]. By appealing to the typing judgement for defining the semantic typing relation over functions, we are back to an elementary proof by induction. Moreover, the semantic typing relation defined in terms of the typing judgement turns out to be stable under substitution of type variables (proposition 1.5 below), which is not the case with the classical relation defined over functions by continuity. This stability property is important for the soundness proof. □

1.4.2 Semantic generalization

The following proposition is the key lemma that shows the soundness of the `let` typing rule.

Proposition 1.5 *Let v be a value and τ be a type such that $\models v : \tau$. Then, for all substitutions φ, we have $\models v : \varphi(\tau)$. As a consequence, for all sets of variables $\alpha_1 \ldots \alpha_n$, we have $\models v : \forall \alpha_1 \ldots \alpha_n . \tau$.*

Proof: by structural induction over v (the same induction as in the definition of the $\models$ relation).

• **Case $v = cst$ and $\tau = \texttt{int}$ ou $\tau = \texttt{bool}$.** Obvious, since $\varphi(\tau) = \tau$ in this case.

• **Case $v = (v_1, v_2)$ and $\tau = \tau_1 \times \tau_2$.** We apply the induction hypothesis to v_1 and v_2. It follows $\models v_1 : \varphi(\tau_1)$ and $\models v_2 : \varphi(\tau_2)$. Hence the result.

• **Case $v = (f, x, a, e)$ and $\tau = \tau_1 \to \tau_2$.** Let E be a typing environment such that $\models e : E$ and $E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \to \tau_2$. By proposition 1.2, we have

$$\varphi(E) \vdash (f \ \texttt{where} \ f(x) = a) : \varphi(\tau_1 \to \tau_2).$$

It remains to show that $\models e : \varphi(E)$. Let $y \in \mathrm{Dom}(e)$. Write $E(y)$ as $\forall \alpha_1 \ldots \alpha_n . \tau_y$, with the α_i chosen out of reach for φ. We then have $\varphi(E(y)) = \forall \alpha_1 \ldots \alpha_n . \varphi(\tau_y)$. We need to show that $\models e(y) : \tau'$ for all instances τ' of $\varphi(E(y))$. Let τ' be such an instance. We therefore have $\tau' = \psi(\varphi(\tau_y))$

for some substitution ψ. We also have $\models e(y) : \tau_y$, by definition of $\models$ over type schemes. We can therefore apply the induction hypothesis to the value $e(y)$, to the type τ_y and to the substitution $\psi \circ \varphi$. It follows that $\models e(y) : \tau'$. This holds for all instances τ' of $\varphi(E(y))$. Hence $\models e(y) : \varphi(E(y))$. This holds for all $y \in \mathrm{Dom}(e)$. Hence $\models e : \varphi(E)$, and the expected result. □

1.4.3 Soundness proof

We are now able to precisely state and to prove the soundness of typing:

Proposition 1.6 (Strong soundness) *Let a be an expression, τ be a type, E be a typing environment, and e be an evaluation environment such that $E \vdash a : \tau$ and $\models e : E$. If there exists a result r such that $e \vdash a \Rightarrow r$, then $r \neq \mathtt{err}$; instead, r is a value v such that $\models v : \tau$.*

Proof: the proof proceeds by induction over the size of the evaluation derivation. We argue by case analysis on a, and therefore on the last rule used in the typing derivation.

- **Constants.**

$$\frac{\tau \leq \mathtt{TypCst}(cst)}{E \vdash cst : \tau}$$

The only evaluation possibility is $e \vdash cst \Rightarrow cst$. It remains to check that $\models cst : \mathtt{TypCst}(cst)$ for all constants cst. This is true for the set of constants and the type assignment `TypCst` given as examples above.

- **Variables.**

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

The typing guarantees that x belongs to the domain of E, which is also the domain of e by hypothesis $\models e : E$. Hence, the only evaluation possibility is $e \vdash x \Rightarrow e(x)$. By hypothesis $\models e : E$, we have $\models e(x) : E(x)$. Hence $\models e(x) : \tau$ by definition of $\models$ over type schemes.

- **Functions.**

$$\frac{E + f \mapsto (\tau_1 \to \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \ \mathtt{where}\ f(x) = a) : \tau_1 \to \tau_2}$$

The only evaluation possibility is $e \vdash (f \ \mathtt{where}\ f(x) = a) \Rightarrow (f, x, a, e)$. We have $\models (f, x, a, e) : \tau_1 \to \tau_2$ by definition of $\models$, taking E for the required typing environment.

- **Applications.**

$$\frac{E \vdash a_1 : \tau_2 \to \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

We have three evaluation possibilities. The first one concludes $r = \mathtt{err}$ because $e \vdash a_1 \Rightarrow r_1$ where r_1 is not a closure; but this contradicts the induction hypothesis applied to a_1, which shows $\models r_1 : \tau_2 \to \tau_1$, hence r_1 is a fortiori a closure. The second evaluation leads to $r = \mathtt{err}$ because

$e \vdash a_2 \Rightarrow \texttt{err}$; it similarly contradicts the induction hypothesis applied to a_2. Hence the last evaluation step must be:

$$\frac{e \vdash a_1 \Rightarrow (f, x, a_0, e_0) \qquad e \vdash a_2 \Rightarrow v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \Rightarrow r}{e \vdash a_1(a_2) \Rightarrow r}$$

By induction hypothesis, we know that $\models (f, x, a_0, e_0) : \tau_2 \to \tau_1$, and $\models v_2 : \tau_2$. Hence there exists a typing environment E_0 such that $\models e_0 : E_0$ and $E_0 \vdash (f \ \texttt{where}\ f(x) = a_0) : \tau_2 \to \tau_1$. There is only one typing rule that can derive this result; its premise must therefore hold:

$$E_0 + f \mapsto (\tau_2 \to \tau_1) + x \mapsto \tau_2 \vdash a_0 : \tau_1.$$

Consider the environments:

$$e_1 = e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \quad \text{and} \quad E_1 = E_0 + f \mapsto (\tau_2 \to \tau_1) + x \mapsto \tau_2.$$

We have $\models e_1 : E_1$ and $E_1 \vdash a_0 : \tau_1$. Applying the induction hypothesis to the evaluation of a_0, we get $r \neq \texttt{err}$ and $\models r : \tau_1$, which is the expected result.

• **`let` bindings.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Two evaluations are possible. The first one is $e \vdash a_1 \Rightarrow \texttt{err}$. It contradicts the induction hypothesis applied to a_1. Hence the last step in the evaluation must be:

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e + x \mapsto v_1 \vdash a_2 \Rightarrow r}{e \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \Rightarrow r}$$

By the induction hypothesis applied to a_1, we get $\models v_1 : \tau_1$. By proposition 1.5, it follows $\models v_1 : \texttt{Gen}(\tau_1, E)$. Define

$$e_1 = e + x \mapsto v_1 \quad \text{and} \quad E_1 = E + x \mapsto \texttt{Gen}(\tau_1, E).$$

We therefore have $\models e_1 : E_1$. We apply the induction hypothesis to a_2 considered in the environments e_1 and E_1. The expected result follows: $r \neq \texttt{err}$ and $\models r : \tau_2$.

• **Pairs.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2}$$

Same reasoning as for function application, just simpler.

• **Primitive applications.**

$$\frac{\tau_1 \to \tau_2 \leq \texttt{TypOp}(op) \qquad E \vdash a : \tau_1}{E \vdash op(a) : \tau_2}$$

Applying the induction hypothesis, we get $e \vdash a \Rightarrow v_1$ with $\models v_1 : \tau_1$. The remainder of the proof depends on the types and semantics given to the operators. It is easy to check the result in the case of the integer addition or the conditional construct, the two examples given above. □

The weak soundness property (proposition 1.4) is an immediate corollary of the strong soundness property (proposition 1.6).

1.5 Type inference

We have seen that if a closed expression a has type τ, it also has type τ' for all types τ' less general than τ (that is, $\tau' = \varphi(\tau)$ for some substitution φ). We are now going to show that the set of all possible types for a is generated this way: there exists a type for a, called the PRINCIPAL TYPE for a, that is more general than all possible types for a. The computation of this type is called TYPE INFERENCE. The existence of a type inference algorithm allows ML compilers to reconstruct the types of all objects from their uses, without the programmer providing type information in the source code.

We first recall some definitions about principal unifiers. A substitution φ is said to be a UNIFIER for the two types τ_1 and τ_2 if $\varphi(\tau_1) = \varphi(\tau_2)$. Two types are UNIFIABLE if there exists a unifier for those two types. Intuitively, two types are unifiable if they can be identified by instantiating some of their variables. A unifier for two types represents the instantiations that must be performed in order to identify them. A unifier φ for τ_1 and τ_2 is said to be MOST GENERAL if all unifiers ψ for τ_1 and τ_2 can be decomposed as $\theta \circ \varphi$ for some substitution θ. The most general unifier for two types, when it exists, represents the minimal modifications required to identify the types: all other unifiers perform at least those modifications.

Proposition 1.7 *If two types τ_1 and τ_2 are unifiables, then they admit a most general unifier, unique up to a renaming, that we write* $\texttt{mgu}(\tau_1, \tau_2)$.

Proof: the set of types is a free term algebra. The most general unifier can be computed with Robinson's algorithm [85], or one of its variants. □

Among the most general unifiers for the two types τ_1 and τ_2, there exist some that "do not introduce fresh variables", that is, such that all variables not free in τ_1 nor in τ_2 are out of reach for these unifiers. (This is the case for the unifier built by Robinson's algorithm.) In the following, we assume that $\texttt{mgu}(\tau_1, \tau_2)$ is chosen so as to satisfy this property.

We now give a variant of Damas and Milner's algorithm [22], that computes the principal type for an expression, if it exists, or fails otherwise. The algorithm takes as inputs an expression a, an initial typing environment E, and a set of "fresh" variables V; it returns as results a type τ (the most general type for a), a substitution φ (representing the instantiations that had to be performed over E), and a subset V' of V (the unused fresh variables).

We write $\texttt{inst}(\sigma, V)$ for a trivial instance of the scheme σ. That is, writing $\sigma = \forall\alpha_1 \ldots \alpha_n.\,\tau$, we take n distinct variables $\beta_1 \ldots \beta_n$ in V and we define

$$\texttt{inst}(\sigma, V) = ([\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau),\ V \setminus \{\beta_1 \ldots \beta_n\}).$$

Clearly, $\texttt{inst}(\sigma, V)$ is uniquely defined up to a renaming of variables from V into variables from V.

Algorithm 1.1 $\texttt{infer}(E, a, V)$ is the triple (τ, φ, V') defined by:

If a is x and $x \in \mathrm{Dom}(E)$:
$(\tau, V') = \texttt{inst}(E(x), V)$ and $\varphi = [\,]$
If a is cst:
$(\tau, V') = \texttt{inst}(\texttt{TypCst}(cst), V)$ and $\varphi = [\,]$
If a is $(f \texttt{ where } f(x) = a_1)$:
let α and β be two variables from V
let $(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1,\ E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha,\ V \setminus \{\alpha, \beta\})$
let $\mu = \texttt{mgu}(\varphi_1(\beta), \tau_1)$
then $\tau = \mu(\varphi_1(\alpha \to \beta))$ and $\varphi = \mu \circ \varphi_1$ and $V' = V_1$
If a is $a_1(a_2)$:
let $(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E, V)$
let $(\tau_2, \varphi_2, V_2) = \texttt{infer}(a_2, \varphi_1(E), V_1)$
let α be a variable from V_2
let $\mu = \texttt{mgu}(\varphi_2(\tau_1), \tau_2 \to \alpha)$
then $\tau = \mu(\alpha)$ and $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ and $V' = V_2 \setminus \{\alpha\}$
If a is $\texttt{let } x = a_1 \texttt{ in } a_2$:
let $(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E, V)$
let $(\tau_2, \varphi_2, V_2) = \texttt{infer}(a_2,\ \varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E)),\ V_1)$
then $\tau = \tau_2$ and $\varphi = \varphi_2 \circ \varphi_1$ and $V = V_2$
If a is (a_1, a_2):
let $(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E, V)$
let $(\tau_2, \varphi_2, V_2) = \texttt{infer}(a_2, \varphi_1(E), V_1)$
then $\tau = \tau_1 \times \tau_2$ and $\varphi = \varphi_2 \circ \varphi_1$ and $V' = V_2$
If a is $op(a_1)$:
let $(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E, V)$
let $(\tau_2, V_2) = \texttt{inst}(\texttt{TypOp}(op), V_1)$
let α be a variable from V_2
let $\mu = \texttt{mgu}(\tau_1 \to \alpha, \tau_2)$
then $\tau = \mu(\alpha)$ and $\varphi = \mu \circ \varphi_1$ and $V' = V_2 \setminus \{\alpha\}$

We take that $\texttt{infer}(a, E, V)$ is not defined if at some point none of the cases match; in particular, if we try to unify two non-unifiable types.

Proposition 1.8 (Correctness of type inference) *Let a be an expression, E be a typing environment and V be a set of type variables. If $(\tau, \varphi, V') = \texttt{infer}(a, E, V)$ is defined, then we can derive $\varphi(E) \vdash a : \tau$.*

Proof: the proof is an inductive argument over the structure of a, and makes heavy use of the fact that the typing judgement is stable under substitution (proposition 1.2). We show one base step and two inductive steps; the remaining cases are similar. We use the same notations as in the algorithm.

• **Case** $a = x$**.** We have $(\tau, V') = \texttt{inst}(E(x), V)$ and $\varphi = [\,]$. By definition of `inst`, we have $\tau \leq E(x)$. We can therefore derive $E \vdash x : \tau$.

• **Case** $a = (f \texttt{ where } f(x) = a_1)$. Applying the induction hypothesis to the recursive call to `infer`, we get a derivation of

$$\varphi_1(E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha) \vdash a_1 : \tau_1.$$

By proposition 1.2, we get a derivation of:

$$\varphi(E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha) \vdash a_1 : \mu(\tau_1).$$

The substitution μ being by definition a unifier of $\varphi_1(\beta)$ and τ_1, we have $\varphi(\beta) = \psi(\tau_1)$. Hence we have shown that:

$$\varphi(E) + f \mapsto (\varphi(\alpha) \to \varphi(\beta)) + x \mapsto \varphi(\alpha) \vdash a_1 : \varphi(\beta).$$

Applying the typing rule for functions, we get:

$$\varphi(E) \vdash (f \texttt{ where } f(x) = a_1) : \varphi(\alpha) \to \varphi(\beta).$$

That's the expected result, since $\tau = \varphi(\alpha \to \beta)$.

• **Case** $a = \texttt{let } x = a_1 \texttt{ in } a_2$. We apply the induction hypothesis to the two recursive calls to `infer`. We get derivations for:

$$\varphi_1(E) \vdash a_1 : \tau_1 \qquad \varphi_2(\varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E))) \vdash a_2 : \tau_2.$$

If required, we rename the generalized variables in the left derivation so that they are out of reach for φ_2. We then show that

$$\texttt{Gen}(\varphi_2(\tau_1), \varphi_2(\varphi_1(E))) = \varphi_2(\texttt{Gen}(\tau_1, \varphi_1(E)))$$

as in the proof of proposition 1.2, `let` case. Taking $\varphi = \varphi_2 \circ \varphi_1$, we therefore have derived:

$$\varphi(E) \vdash a_1 : \varphi_2(\tau_1) \qquad \varphi(E) + x \mapsto \texttt{Gen}(\varphi_2(\tau_1), \varphi(E)) \vdash a_2 : \tau_2.$$

We conclude, by the `let` typing rule,

$$\varphi(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2.$$

That's the expected result. □

Proposition 1.9 (Completeness of type inference) *Let a be an expression, E be a typing environment, and V be an infinite set of variables such that $V \cap \mathcal{F}(E) = \emptyset$. If there exists a type τ' and a substitution φ' such that $\varphi'(E) \vdash a : \tau'$, then $(\tau, \varphi, V') = \texttt{infer}(a, E, V)$ is defined, and there exists a substitution ψ such that*

$$\tau' = \psi(\tau) \quad \textit{and} \quad \varphi' = \psi \circ \varphi \textit{ outside } V.$$

By "$\varphi' = \psi \circ \varphi$ OUTSIDE V", we mean that $\varphi'(\alpha)$ is equal to $\psi(\varphi(\alpha))$ for all variables α that do not belong to V. The condition $\varphi' = \psi \circ \varphi$ outside V captures the fact that the two substitutions φ' and $\psi \circ \varphi$ behave the same on the initial typing problem. The variables in V are variables that do not appear in the initial problem, but can be introduced at intermediate steps during inference; they do not have to be taken into account when comparing φ' (the proposed solution to the typing problem) with φ (the inferred solution).

Proof: we first remark that, under the assumptions of the proposition, if $(\tau, \varphi, V') = \texttt{infer}(a, E, V)$ is defined, then $V' \subseteq V$, and the variables in V' are not free in τ and are out of reach for φ. Hence, $V' \cap \mathcal{F}(\varphi(E)) = \emptyset$.

The proof of proposition 1.9 proceeds by structural induction over a. We show one base step and three inductive steps; the remaining cases are similar.

• **Case** $a = x$**.** Since $\varphi(E) \vdash x : \tau$, we have $x \in \mathrm{Dom}(\varphi(E))$ and $\tau \leq \varphi(E)(x)$. This implies $x \in \mathrm{Dom}(E)$. Hence $\texttt{infer}(E, x)$ is defined, and returns

$$\tau = [\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau_x) \quad \text{and} \quad \varphi = [\,] \quad \text{and} \quad V' = V \setminus \{\beta_1 \ldots \beta_n\}$$

for some variables $\beta_1 \ldots \beta_n \in V$. Write $E(x) = \forall \alpha_1 \ldots \alpha_n . \tau_x$, where the α_i are taken from V' and out of reach for φ'. We have $\varphi'(E(x)) = \forall \alpha_1 \ldots \alpha_n . \varphi'(\tau_x)$. Take ρ to be the substitution over the α_i such that $\tau' = \rho(\varphi'(\tau_x))$. Take

$$\psi = \rho \circ \varphi' \circ [\beta_1 \mapsto \alpha_1, \ldots, \beta_n \mapsto \alpha_n].$$

We have $\psi(\tau) = \rho(\varphi'(\tau_x)) = \tau'$. Moreover, all variables $\alpha \notin V$ are neither one of the α_i, nor one of the β_i, hence $\psi(\alpha) = \rho(\varphi'(\alpha)) = \varphi'(\alpha)$. This is the expected result, since $\varphi = [\,]$ here.

• **Case** $a = (f \texttt{ where } f(x) = a_1)$**.** The initial typing derivation ends up with

$$\frac{\varphi'(E) + f \mapsto (\tau'_2 \to \tau'_1) + x \mapsto \tau'_2 \vdash a_1 : \tau'_1}{\varphi'(E) \vdash (f \texttt{ where } f(x) = a) : \tau'_2 \to \tau'_1}$$

Choose α and β in V, as in the algorithm. Define the environment E_1 and the substitution φ'_1 by

$$E_1 = E + f \mapsto (\alpha \to \beta) + x \mapsto \alpha \quad \text{and} \quad \varphi'_1 = \varphi' + \alpha \mapsto \tau'_2 + \beta \mapsto \tau'_1.$$

We have $\varphi'_1(E_1) = \varphi'(E) + f \mapsto (\tau'_2 \to \tau'_1) + x \mapsto \tau'_2$. We apply the induction hypothesis to a_1, E_1, $V \setminus \{\alpha, \beta\}$, φ'_1 and τ'_2. We get

$$(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E_1, V \setminus \{\alpha, \beta\}) \quad \text{and} \quad \tau'_1 = \psi_1(\tau_1) \quad \text{and} \quad \varphi'_1 = \psi_1 \circ \varphi_1 \text{ outside } V \setminus \{\alpha, \beta\}.$$

In particular, we have $\psi_1(\varphi_1(\beta)) = \varphi'_1(\beta) = \tau'_1$, hence ψ_1 is a unifier of $\varphi_1(\beta)$ and τ_1. The most general unifier of these two types therefore exists — let us call it μ —, and $\texttt{infer}(E, a, v)$ is well-defined. Let ψ be a substitution such that $\psi_1 = \psi \circ \mu$. We now show that this substitution ψ satisfies the claim. We have:

$$\begin{array}{rcll} \psi(\tau) & = & \psi(\mu(\varphi_1(\alpha \to \beta))) & \text{by definition of } \tau \text{ in the algorithm} \\ & = & \psi_1(\varphi_1(\alpha \to \beta)) & \text{by definition of } \psi \\ & = & \varphi'_1(\alpha \to \beta) & \text{because } \alpha \text{ and } \beta \text{ are outside of } V \setminus \{\alpha, \beta\} \\ & = & \tau'_2 \to \tau'_1 & \text{by construction of } \varphi'_1 \end{array}$$

Moreover, for any variable γ outside of V,

$$\begin{array}{rcll} \psi(\varphi(\gamma)) & = & \psi(\mu(\varphi_1(\gamma))) & \text{by definition of } \varphi \text{ in the algorithm} \\ & = & \psi_1(\varphi_1(\gamma)) & \text{by definition of } \psi_1 \\ & = & \varphi'_1(\gamma) & \text{because } \gamma \notin V \\ & = & \varphi_1(\gamma) & \text{because } \gamma \notin V \text{ implies } \gamma \neq \alpha \text{ and } \gamma \neq \beta \end{array}$$

The desired result follows.

• **Case** $a = a_1(a_2)$. The initial derivation ends up with

$$\frac{\varphi'(E) \vdash a_1 : \tau'' \to \tau' \qquad \varphi'(E) \vdash a_2 : \tau''}{\varphi'(E) \vdash a_1(a_2) : \tau'}$$

We apply the induction hypothesis to a_1, E, V, $\tau' \to \tau''$ and φ', obtaining

$$(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E, V) \quad \text{and} \quad \tau'' \to \tau' = \psi_1(\tau_1) \quad \text{and} \quad \varphi' = \psi_1 \circ \varphi_1 \text{ outside } V.$$

In particular, $\varphi'(E) = \psi_1(\varphi_1(E))$. We apply the induction hypothesis to a_2, $\varphi_1(E)$, V_1, τ and ψ_1. This is legitimate, since $\mathcal{F}(\varphi_1(E)) \cap V_1 = \emptyset$ by the remark at the beginning of the proof. We get:

$$(\tau_2, \varphi_2, V_2) = \texttt{infer}(a_2, \varphi_1(E), V_1) \quad \text{and} \quad \tau'' = \psi_2(\tau_2) \quad \text{and} \quad \psi_1 = \psi_2 \circ \varphi_2 \text{ outside } V_1.$$

We have $\mathcal{F}(\tau_1) \cap V_1 = \emptyset$, hence $\psi_1(\tau_1) = \psi_2(\varphi_2(\tau_1))$. Take $\psi_3 = \psi_2 + \alpha \mapsto \tau'$. (The variable α, taken in V_2, is out of reach for ψ_2, hence ψ_3 extends ψ_2.) We have:

$$\begin{array}{lcccccl} \psi_3(\varphi_2(\tau_1)) & = & \psi_2(\varphi_2(\tau_1)) & = & \psi_1(\tau_1) & = & \tau'' \to \tau' \\ \psi_3(\tau_2 \to \alpha) & = & \psi_2(\tau_2) \to \tau'' & = & \tau'' \to \tau' & & \end{array}$$

The substitution ψ_3 is therefore a unifier of $\varphi_2(\tau_1)$ and $\tau_2 \to \alpha$. Those two types therefore admit a most general unifier, μ. Hence $\texttt{infer}(a_1(a_2), E, V)$ is well-defined. In addition, we have $\psi_3 = \psi_4 \circ \mu$ for some substitution ψ_4. We now show that $\psi = \psi_4$ meets the requirements of the proposition. With the same notations as in the algorithm, we have

$$\psi(\tau) = \psi_4(\mu(\alpha)))) = \psi_3(\alpha) = \tau',$$

on one hand, and on the other hand, for all $\beta \notin V$ (hence a fortiori $\beta \notin V_1$, $\beta \notin V_2$, $\beta \neq \alpha$):

$$\begin{array}{lcll} \psi(\varphi(\beta)) & = & \psi_4(\mu(\varphi_2(\varphi_1(\beta)))) & \text{by definition of } \varphi \\ & = & \psi_3(\varphi_2(\varphi_1(\beta))) & \text{by definition of } \psi_4 \\ & = & \psi_2(\varphi_2(\varphi_1(\beta))) & \text{because } \beta \neq \alpha \text{ and } \alpha \text{ is out of reach for } \varphi_1 \text{ and } \varphi_2 \\ & = & \psi_1(\varphi_1(\beta)) & \text{because } \varphi_1(\beta) \notin V_1 \\ & = & \varphi'(\beta) & \text{because } \beta \notin V. \end{array}$$

This is the expected result.

• **Cas** $a = (\texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2)$. The initial derivation ends up with

$$\frac{\varphi'(E) \vdash a_1 : \tau' \qquad \varphi'(E) + x \mapsto \texttt{Gen}(\tau', \varphi'(E)) \vdash a_2 : \tau''}{\varphi'(E) \vdash \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2 : \tau''}$$

We apply the induction hypothesis to a_1, E, V, τ' and φ'. We get

$$(\tau_1, \varphi_1, V_1) = \texttt{infer}(a_1, E, V) \quad \text{and} \quad \tau' = \psi_1(\tau_1) \quad \text{and} \quad \varphi' = \psi_1 \circ \varphi_1 \text{ outside } V.$$

In particular, $\varphi'(E) = \psi_1(\varphi_1(E))$. We easily check that $\psi_1(\mathtt{Gen}(\tau_1, \varphi_1(E)))$ is more general than $\mathtt{Gen}(\psi_1(\tau_1), \psi_1(\varphi_1(E)))$, that is, than $\mathtt{Gen}(\tau', \varphi'(E))$. Since we can derive

$$\varphi'(E) + x \mapsto \mathtt{Gen}(\tau', \varphi'(E)) \vdash a_2 : \tau'',$$

proposition 1.3 shows that we can a fortiori derive

$$\varphi'(E) + x \mapsto \psi_1(\mathtt{Gen}(\tau_1, \varphi_1(E))) \vdash a_2 : \tau'',$$

that is,

$$\psi_1(\varphi_1(E) + x \mapsto \mathtt{Gen}(\tau_1, \varphi_1(E))) \vdash a_2 : \tau''.$$

We now apply the induction hypothesis to a_2, in the environment $\varphi_1(E) + x \mapsto \mathtt{Gen}(\tau_1, \varphi_1(E))$, with the fresh variables V_1, the type τ'' and the substitution ψ_1. It follows that

$$(\tau_2, \varphi_2, V_2) = \mathtt{infer}(a_2,\ \varphi_1(E) + x \mapsto \mathtt{Gen}(\tau_1, \varphi_1(E)),\ V_1)$$

and $\tau'' = \psi_2(\tau_2)$ and $\psi_1 = \psi_2 \circ \varphi_2$ outside V_1. The algorithm takes $\tau = \tau_2$ and $\varphi = \varphi_2 \circ \varphi_1$ and $V' = V_2$. Let us show that $\psi = \psi_2$ meets the claim. We have $\psi(\tau) = \tau''$. And if $\alpha \notin V$, a fortiori $\alpha \notin V_1$, hence:

$$\begin{array}{rcll} \psi(\varphi(\alpha)) & = & \psi_2(\varphi_2(\varphi_1(\alpha)))) & \text{by definition of } \varphi \\ & = & \psi_1(\varphi_1(\alpha)) & \text{because } \varphi_1(\alpha) \notin V_1, \text{ since } \alpha \text{ is out of reach for } \varphi_1 \\ & = & \varphi'(\alpha) & \text{since } \alpha \notin V. \end{array}$$

Hence $\varphi' = \psi \circ \varphi$ outside V, as expected. □

Chapter 2

Three non-applicative extensions

In the present chapter, we enrich the simplified applicative language presented in the previous chapter by three important features of "real" algorithmic languages. First of all, the possibility to modify data structures in place. Second, the possibility to execute parts of the program concurrently, while communicating intermediate results. Finally, a number of non-local control structures (exceptions, coroutines, and even some forms of `goto`).

We provide these features by introducing three new kinds of "first-class" objects: references, or indirection cells, for in-place modification; communication channels, for communicating processes; and continuations, for non-local control structures. We are going to enrich the purely applicative language with primitive to create and manipulate these objects, and give the semantics of these primitives.

From the standpoint of typing, these three extensions display striking similarities. There is a natural way to type them, by introducing new unary type constructors. The resulting typing rules are simple and intuitive. The type systems thus obtained are sound in the absence of polymorphism. However, they turn out to be unsound in the presence of polymorphism. In this chapter, we just give counterexamples; a more detailed discussion of the phenomenon, and some workarounds, are postponed to the next chapters.

2.1 References

2.1.1 Presentation

References are indirection cells that can be modified in place. A reference has a current content (any language value), that can be read and written at any time. A reference itself is a value: it can be returned as the result of a function, or stored in a data structure. References are presented through the three primitive operations:

`Op`	::=	`ref`	creation of a fresh reference
	|	!	reading of the current contents
	|	:=	modification of the current contents

The expression `ref`(a) returns a fresh reference, initialized with the value of a. The expression `!`(a), which is usually written `!`a, evaluates a to a reference and returns its current contents. Finally, the expression $:= (a_1, a_2)$, which is usually written $a_1 := a_2$, evaluates a_1 to a reference and physically updates its contents by the value of a_2. The expression $a_1 := a_2$ itself has no obvious return value: it operates essentially by side-effect. We follow the convention that it returns a special constant, written `()` (read: "void").

```
Cst  ::=  ...
        |  ()   the void value
```

Example. References combined with the `let` construct provide updatable identifiers (true variables). For instance, we can write a loop for `!i` varying from 0 to `n` $-$ 1 as follows:

```
let i = ref(0) in
  while !i < n do ...; i := 1 + !i done
```

Here and elsewhere, the notation $a_1; a_2$ means " evaluate first a_1, then a_2, and return the value of a_2." This construct is actually an abbreviation for `let` $z = a_1$ `in` a_2, where z is a fresh identifier, not free in a_2. Similarly, the loop `while` a_1 `do` a_2 `done` is an abbreviation for

$$(f \texttt{ where } f(z) = \texttt{if } a_1 \texttt{ then } a_2; f(z) \texttt{ else } ())(())$$

where f and z are fresh identifiers. □

Example. References combined with data structures provide mutable data structures (data structures that can be modified in place). For instance, to implement a term algebra, we can represent variables by references to a special constant. To substitute a term for a variable, we simply store the term in the reference corresponding to the variable [11]. Substituting a variable therefore takes constant time; in contrast, with a representation of terms without references, we would have to copy the term(s) in which the variable is substituted, which is less efficient. Moreover, if the substituted variable is shared between several terms, the substitution immediately takes effect in all terms that contain the variable, with the reference-based implementation. In the implementation without references, we must explicitly substitute all terms that contain the substituted variable, requiring the programmer to keep track of all these terms. □

Example. References combined with functions provide stateful functions — a notion close to the one of object in object-oriented languages. For instance, here is a pseudo-random number generator: a function that returns the next element of a hard-to-predict sequence each time it is called. We also provide a function to reinitialize the generator.

```
let (random, set_random) =
  let seed = ref 0 in
    (λx. seed := (!seed × 25173 + 13849) mod 1000; !seed),
    (λnewseed. seed := newseed)
in ...
```

(We have used a `let` with destructuration of a pair, with the obvious semantics.) This example is delicate to program in a purely applicative language. The function `random` must then take the

state variable `seed` as argument, and return its new value as a second result. The program calling `random` is responsible for correctly propagating the value of `seed`, that is to use the value returned by the last call to `random` as argument to the next call. This clutters the program. Moreover, this is error-prone: it is easy to give by mistake the same value of `seed` to `random` each time. The reference-based solution is better programming style: safer and more modular. □

2.1.2 Semantics

We now give a relational semantics to the language enriched by references. References require the introduction of a global state (the current contents of all references) in the semantics; this global state changes during evaluation. In more practical terms, we treat references as locations of cells in a store. The semantics not only assign a value to each expression, but also describe how the store is modified by the evaluation of the expression. We therefore assume given a infinite set of *locations*, ranged over by ℓ, and we define the stores s as finite mappings from locations to values. The semantic objects are defined as:

Results:	r	::=	v/s	normal result
		|	`err`	error result
Values:	v	::=	cst	base value
		|	(v_1, v_2)	pair of values
		|	(f, x, a, e)	functional value
		|	ℓ	memory location
Environments:	e	::=	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	
Stores:	s	::=	$[\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n]$	

The evaluation predicate is now $e \vdash a/s_0 \Rightarrow r$, meaning "in evaluation environment e, the expression a taken in the initial store s_0 evaluates to the response r." A response is either `err`, or a pair v/s_1, where v is the result value and s_1 is the store at the end of evaluation.

Here are the rules defining the new evaluation predicate. We give the rules for the purely applicative constructs without comments. Constants and variables:

$$e \vdash cst/s \Rightarrow cst/s \qquad \frac{x \in \mathrm{Dom}(e)}{e \vdash x/s \Rightarrow e(x)/s} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x/s \Rightarrow \texttt{err}}$$

Function abstraction:

$$e \vdash (f \texttt{ where } f(x) = a)/s \Rightarrow (f, x, a, e)/s$$

Function application:

$$\frac{e \vdash a_1/s_0 \Rightarrow (f, x, a_0, e_0)/s_1 \qquad e \vdash a_2/s_1 \Rightarrow v_2/s_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0/s_2 \Rightarrow r_0}{e \vdash a_1(a_2)/s_0 \Rightarrow r_0}$$

$$\frac{e \vdash a_1/s_0 \Rightarrow r_1 \qquad r_1 \text{ does not match } (f, x, a_0, e_0)/s_1}{e \vdash a_1(a_2)/s_0 \Rightarrow \texttt{err}}$$

$$\frac{e \vdash a_1/s_0 \Rightarrow (f, x, a_0, e_0)/s_1 \qquad e \vdash a_2/s_1 \Rightarrow \texttt{err}}{e \vdash a_1(a_2)/s_0 \Rightarrow \texttt{err}}$$

The `let` binding:

$$\frac{e \vdash a_1/s_0 \Rightarrow v_1/s_1 \qquad e + x \mapsto v_1 \vdash a_2/s_1 \Rightarrow r_2}{e \vdash (\texttt{let } x = a_1 \texttt{ in } a_2)/s_0 \Rightarrow r_2} \qquad \frac{e \vdash a_1/s_0 \Rightarrow \texttt{err}}{e \vdash (\texttt{let } x = a_1 \texttt{ in } a_2)/s_0 \Rightarrow \texttt{err}}$$

Pairs:

$$\frac{e \vdash a_1/s_0 \Rightarrow v_1/s_1 \qquad e \vdash a_2/s_1 \Rightarrow v_2/s_2}{e \vdash (a_1, a_2)/s_0 \Rightarrow (v_1, v_2)/s_2}$$

$$\frac{e \vdash a_1/s_0 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2)/s_0 \Rightarrow \texttt{err}} \qquad \frac{e \vdash a_1/s_0 \Rightarrow v_1/s_1 \qquad e \vdash a_2/s_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2)/s_0 \Rightarrow \texttt{err}}$$

We now detail the rules dealing with references.

$$\frac{e \vdash a/s_0 \Rightarrow v/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash \texttt{ref}(a)/s_0 \Rightarrow \ell/(s_1 + \ell \mapsto v)} \qquad \frac{e \vdash a/s_0 \Rightarrow \texttt{err}}{e \vdash \texttt{ref}(a)/s_0 \Rightarrow \texttt{err}}$$

The creation of a reference first evaluates the initial value v, then picks an unused location ℓ (this is always possible, because we have assumed an infinite number of locations), and enrich the store by the binding of v to ℓ. The value part of the result is the location ℓ.

Remark. This rule might seem to destroy the determinism of evaluation: we are free to choose various locations ℓ. However, all these choices are equivalent in some sense: if $e \vdash a/s_0 \Rightarrow v/s_1$, then the response v/s_1 is unique modulo a renaming of the locations used in s_1 but not in s_0. □

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash !a/s_0 \Rightarrow s_1(\ell)/s_1}$$

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash !a/s_0 \Rightarrow \texttt{err}} \qquad \frac{e \vdash a/s_0 \Rightarrow r \qquad r \text{ does not match } \ell/s_1}{e \vdash !a/s_0 \Rightarrow \texttt{err}}$$

The dereferencing operator evaluates its argument, then reads the value stored in the location thus obtained. It is a run-time error if the location falls outside of the current store when the evaluation terminates.

$$\frac{e \vdash a/s_0 \Rightarrow (\ell, v)/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash :=(a)/s_0 \Rightarrow \texttt{()}/(s_1 + \ell \mapsto v)}$$

$$\frac{e \vdash a/s_0 \Rightarrow (\ell, v)/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash :=(a)/s_0 \Rightarrow \texttt{err}} \qquad \frac{e \vdash a/s_0 \Rightarrow r \qquad r \text{ does not match } (\ell, v)/s_1}{e \vdash :=(a)/s_0 \Rightarrow \texttt{err}}$$

The assignment operator evaluates its argument. The first component of the result must be a location ℓ already used. It then updates the store at location ℓ, substituting the old value by the value just computed.

2.1.3 Typing

The natural typing for references is as follows: at the level of types, we introduce the construction τ `ref`, which is the type of references that contain a value of type τ.

$$\begin{array}{rcll} \tau & ::= & \ldots & \\ & | & \tau\ \texttt{ref} & \text{reference type} \end{array}$$

Then, the result of a reference creation operation $\texttt{ref}(a)$ has type τ `ref`, provided that a has type τ. The dereferencing $!a$ returns a value with type τ, provided that a has type τ `ref`. Finally, the assignment $a_1 := a_2$ is well-typed if a_2 has the type expected for the contents of a_1, that is if $a_1 : \tau$ `ref` and $a_2 : \tau$ for some type τ. This is summarized by the following type assignment:

$$\begin{array}{rcl} \texttt{TypOp(ref)} & = & \forall\alpha.\,\alpha \to \alpha\ \texttt{ref} \\ \texttt{TypOp(!)} & = & \forall\alpha.\,\alpha\ \texttt{ref} \to \alpha \\ \texttt{TypOp(:=)} & = & \forall\alpha.\,\alpha\ \texttt{ref} \times \alpha \to \texttt{unit} \end{array}$$

In the type for :=, we have used a new base type, `unit`, which is the type containing only the value `()`:

$$\begin{array}{rcl} \texttt{TypBas} & = & \{\texttt{unit}; \texttt{int}; \texttt{bool}; \ldots\} \\ \texttt{TypCst(())} & = & \texttt{unit} \end{array}$$

The typings proposed above for the primitives over references look correct. It is fairly easy to show that they are sound in the absence of polymorphism. However, these typings turn out to be unsound in the presence of polymorphic types: a single reference considered with a polymorphic type suffices to break type safety.

Example. The following program is well-typed with the typings above:

```
let r = ref(λx.x) in
    r := (λn.n + 1);
    if (!r)(true) then ... else ...
```

The type of `r` in the body of the `let` is actually $\forall\alpha.\,(\alpha \to \alpha)$ `ref`. We can therefore use `r` a first time with type (`int` $\to$ `int`) `ref`, and store there the successor function; then a second time with type (`bool` $\to$ `bool`) `ref`, and apply the contents of `r` to `true`. At run-time, we end up adding 1 with `true` — a type violation, if any. □

It is now clear that a reference, once created, must always be used with the same type; otherwise, one could store an object of a given type in the reference, then read it later claiming it has a different type. This consistency between all uses of a given reference is obviously violated as soon as the reference is given a non-trivial type scheme. We must therefore modify the type system in order to prohibit such polymorphic references; this is the goal of the next two chapters.

2.2 Communication channels

2.2.1 Presentation

We have yet considered only programs that live in a closed world: their evaluation does not interfere with the state of the outside of the program. "Real" programs do not fit into this model: they interact with external entities: the user, the operating system, other programs. We would therefore like to describe the interactions with the outside world, in the language and in its formalization. The next step is to internalize this notion: we consider the program no longer as a monolithic expression that evaluates sequentially, but as several expressions that evaluate concurrently, while exchanging information. Many algorithms can then be expressed more easily — in particular, those requiring complex interleavings of computations. We also expect evaluation to be faster if several processors are available.

We are now going to introduce these notions in the simplified language from chapter 1, by means of the communication channels. A channel is a new first-class object, over which a process can ask to send a value, or to receive a value. A value is actually transmitted when two processes simultaneously ask for transmission over the same channel, one process trying to send, the other trying to receive. Channels therefore allow synchronization between processes (by the so-called "rendez-vous" mechanism), in addition to communication. Channels are presented by the following primitives:

`Op`	::=	`newchan`	creation of a new channel
	|	?	reception from a channel
	|	!	emission over a channel

We traditionally write $a?$ instead of $?(a)$ for receiving a value from a channel a, and $a_1 \,!\, a_2$ instead of $!(a_1, a_2)$ for sending the value of a_2 over the channel a_1. The expression $a?$ evaluates to the value received; the expression $a_1 \,!\, a_2$ evaluates to `()`.

To communicate by rendez-vous over a channel, we must be able to evaluate two expressions in parallel: one that sends, the other that receives. We therefore introduce two extra constructs at the level of expressions:

a	::=	...	
	|	$a_1 \parallel a_2$	parallel composition
	|	$a_1 \oplus a_2$	non-deterministic choice

The expression $a_1 \parallel a_2$ evaluates a_1 and a_2 concurrently, and returns the pair of the results. It allows a_1 and a_2 to communicate by rendez-vous. The expression $a_1 \oplus a_2$ evaluates either a_1 or a_2, the choice between the two expressions being non-deterministic. This construct serves to offer several communication possibilities to the outside world. The third way to compose evaluation, the sequence $(a_1; a_2)$, can easily be defined in terms of the `let` binding, as shown above.

Context. Various presentations of the notion of communicating processes have been proposed. In roughly chronological order: models based on shared memory plus semaphores or monitors or locks; Petri nets; models based on channels (inspired by the Multics streams and the Unix pipes): the Kahn-MacQueen process networks [43], Hoare's CSP calculus [37], Milner's CCS calculus [59] and its numerous variants; more recently, the models based on "active data" [15, 6, 9, 45].

I have followed as closely as possible the approach taken by Milner for his Calculus of Communicating Systems [59]. The main difference is that, in the calculus presented here, channels are first-class values, that are created by the `newchan` primitive, and subject to the usual lexical scoping rules. In contrast, in CCS, channels are not first-class values; they are permanently associated to global names; and to limit their scope, one must use the renaming and masking constructs, that encode a notion of dynamic scoping.

The calculus presented here is said to be higher-order, since channel values can be sent over channels. There is no general agreement on how to extend CCS to the higher order. Thomsen [91] and Berthomieu [10] propose having processes as values, but not channels. In his pi-calculus [61], Milner proposes to take channel names as first-class values, which is close, but not completely equivalent to taking channels themselves as first-class values. The "channels as values" approach seems closer to the mechanisms provided by operating systems — the Unix pipes, in particular.

Two recent proposals for extending ML with parallelism and communication take channels as first-class values, as in this section, but provide in addition a concrete type for communication events, which can be combined to define new synchronization mechanisms. These proposals are Reppy's Concurrent ML library [81], and its variant described by Berry, Milner and Turner [8]. □

Example. The `stamp` function below emits the sequence of integers over the first channel given as argument, and starts again at zero when it receives something on the second channel. Evaluated concurrently with other processes, it serves as a generator of unique stamps.

```
let stamp = λoutput. λreset.
    (f where f(n) = (output!n; f(n+1)) ⊕ (reset?; f(0)))(0) in
let st = newchan(()) in
let reset = newchan(()) in
    stamp(st)(reset) || (reset!(); ( ... st? ... st? ... ) || ( ... st? ... ))
```

□

Example. The following program, taken from [43], emits all prime numbers on the channel given as argument.

```
let sieve = λprimes.
  let integers = newchan(()) in
    ((enumerate where enumerate(n) = integers!n; enumerate(n+1))(2)) ||
    ((filter where filter(input) =
        let n = input? in
          primes!n;
          let output = newchan(()) in
            filter(output) ||
            while true do
              let m = input? in if m mod n = 0 then () else output!m
            done)
     (integers))
in ...
```

We easily recognize Eratosthene's sieve here. Such a program, that puts no a priori upper bounds on the prime numbers produced, is considerably harder to write in a purely sequential language (at least, under strict semantics).

□

2.2.2 Semantics

We now give a relational semantics to the language enriched with channels. The evaluation predicate must take into account the current state of the other processes evaluating concurrently with the expression considered, so as to determine what the current communication opportunities are. To do so, we add an argument to the evaluation predicate, that becomes $e \vdash a \stackrel{w}{\Longrightarrow} r$. Here, w is a finite sequence of EVENTS; this sequence represents the communications that must take place to reach the result r. Communication events are either $c\,!\,v$ or $c\,?\,v$, denoting the emission or the reception of the value v over the channel identified by c.

Results:	r	::=	v	normal result (a value)
		|	`err`	error result
Values:	v	::=	cst	base value
		|	(v_1, v_2)	pair of values
		|	(f, x, a, e)	functional value
		|	c	channel identifier
Environments:	e	::=	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	
Events:	evt	::=	$c\,?\,v$	emission of a value
		|	$c\,!\,v$	reception of a value
Event sequences:	w	::=	ε	the empty sequence
		|	$evt \ldots evt$	

We start by giving again the semantics of the base constructs, taking into account the sequencing of communication events. This sequencing is apparent in the decomposition of the sequence w into sub-sequences, one for each evaluation step. Constants and variables:

$$e \vdash cst \stackrel{\varepsilon}{\Longrightarrow} cst \qquad \frac{x \in \mathrm{Dom}(e)}{e \vdash x \stackrel{\varepsilon}{\Longrightarrow} e(x)} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x \stackrel{\varepsilon}{\Longrightarrow} \texttt{err}}$$

Function abstraction:

$$e \vdash (f \ \texttt{where} \ f(x) = a) \stackrel{\varepsilon}{\Longrightarrow} (f, x, a, e)$$

Function application:

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \stackrel{w_3}{\Longrightarrow} r_0}{e \vdash a_1(a_2) \stackrel{w_1 w_2 w_3}{\Longrightarrow} r_0}$$

$$\frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} r_1 \qquad r_1 \text{ does not match } (f, x, a_0, e_0)}{e \vdash a_1(a_2) \stackrel{w}{\Longrightarrow} \texttt{err}}$$

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} \texttt{err}}{e \vdash a_1(a_2) \stackrel{w_1 w_2}{\Longrightarrow} \texttt{err}}$$

The `let` binding:

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} v_1 \qquad e + x \mapsto v_1 \vdash a_2 \stackrel{w_2}{\Longrightarrow} r_2}{e \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \stackrel{w_1 w_2}{\Longrightarrow} r_2} \qquad \frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} \texttt{err}}{e \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \stackrel{w}{\Longrightarrow} \texttt{err}}$$

Pair construction:

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} v_1 \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} v_2}{e \vdash (a_1, a_2) \stackrel{w_1 w_2}{\Longrightarrow} (v_1, v_2)} \qquad \frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} \texttt{err}}{e \vdash (a_1, a_2) \stackrel{w}{\Longrightarrow} \texttt{err}} \qquad \frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} v_1 \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} \texttt{err}}{e \vdash (a_1, a_2) \stackrel{w_1 w_2}{\Longrightarrow} \texttt{err}}$$

We now describe the evaluation of the parallelism and communication constructs, starting with channel creation.

$$\frac{c \text{ is unallocated anywhere else in the derivation}}{e \vdash \texttt{newchan}(a) \stackrel{\varepsilon}{\Longrightarrow} c}$$

In the evaluation rule for `newchan`, the channel identifier c returned as result must be different from all other channels used in the same process, as well as in the other concurrent processes. To ensure this condition at the level of the rules, we would have to equip the evaluation predicate with two extra arguments: the sets of channel identifiers free before and after the evaluation. This complicates the rules to the point of unusability. To circumvent this difficulty, we add a global condition at the level of the derivations: in all evaluation derivations considered later, we require that any two instances of the `newchan` rule assign different channel identifiers to c.

The rules for sending and receiving are the only ones that add new events $c\,!\,v$ or $c\,?\,v$ to event sequences:

$$\frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} c}{e \vdash a_1? \stackrel{w.(c?v)}{\Longrightarrow} v} \qquad \frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} r \qquad r \text{ does not match } c}{e \vdash a_1? \stackrel{w}{\Longrightarrow} \texttt{err}}$$

$$\frac{e \vdash a \stackrel{w}{\Longrightarrow} (c, v)}{e \vdash !(a) \stackrel{w.(c!v)}{\Longrightarrow} ()} \qquad \frac{e \vdash a \stackrel{w}{\Longrightarrow} r \qquad r \text{ does not match } (c, v)}{e \vdash !(a) \stackrel{w}{\Longrightarrow} \texttt{err}}$$

The $\oplus$ operator is the non-deterministic choice:

$$\frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} r_1}{e \vdash a_1 \oplus a_2 \stackrel{w}{\Longrightarrow} r_1} \qquad \frac{e \vdash a_2 \stackrel{w}{\Longrightarrow} r_2}{e \vdash a_1 \oplus a_2 \stackrel{w}{\Longrightarrow} r_2}$$

Finally, here are the rules for the parallel composition of two expressions:

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} v_1 \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} v_2 \qquad \vdash w_1 \parallel w_2 \Rightarrow w}{e \vdash a_1 \parallel a_2 \stackrel{w}{\Longrightarrow} (v_1, v_2)}$$

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} \texttt{err} \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} r_2 \qquad \vdash w_1 \parallel w_2 \Rightarrow w}{e \vdash a_1 \parallel a_2 \stackrel{w}{\Longrightarrow} \texttt{err}}$$

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} r_1 \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} \texttt{err} \qquad \vdash w_1 \parallel w_2 \Rightarrow w}{e \vdash a_1 \parallel a_2 \stackrel{w}{\Longrightarrow} \texttt{err}}$$

These rules rely on the notion of SHUFFLING of two event sequences. We write $\vdash w_1 \parallel w_2 \Rightarrow w$ to express that w is one possible shuffling of w_1 and w_2. This relation is defined by the rules below.

$$\vdash \varepsilon \parallel \varepsilon \Rightarrow \varepsilon \qquad \frac{\vdash w_1 \parallel w_2 \Rightarrow w}{\vdash w_1.evt \parallel w_2 \Rightarrow w.evt} \qquad \frac{\vdash w_1 \parallel w_2 \Rightarrow w}{\vdash w_1 \parallel w_2.evt \Rightarrow w.evt}$$

$$\frac{\vdash w_1 \parallel w_2 \Rightarrow w}{\vdash w_1.(c\,?\,v) \parallel w_2.(c\,!\,v) \Rightarrow w} \qquad \frac{\vdash w_1 \parallel w_2 \Rightarrow w}{\vdash w_1.(c\,!\,v) \parallel w_2.(c\,?\,v) \Rightarrow w}$$

The first three shuffling cases are obvious. The last two rules correspond to a successful rendez-vous between two concurrent processes. Then, the value v is actually passed over c from one process to the other. Moreover, the two events $c\,!\,v$ and $c\,?\,v$ remain local to the two communicating processes; therefore, they don't appear in the event sequence w, which represents the communication activity between $a_1 \parallel a_2$ and the outside world.

Remark. The shuffling operation is not deterministic. In particular, we can choose not to perform a possible rendez-vous. □

Context. In CCS, this shuffling operation is not explicit: the CCS reduction predicate has the form $a \stackrel{\alpha}{\rightarrow} a'$, where α describes zero or one event; since the result a' is another expression, instead of a value, we can reduce again a', and thus reach a normal form by successive reductions. Throughout this work, I have opted to maintain a clear distinction between source programs and result values. (In practice, no interpreter works by successive rewritings of the source program.) Hence the use of event sequences to describe the calculus with channels. □

Example. Here is one possible evaluation for $\texttt{x}? \parallel (\texttt{x}\,!\,1 \oplus \texttt{x}\,!\,2)$, in an environment e where the identifier $\texttt{x}$ is bound to the channel c.

$$\frac{\dfrac{e \vdash \texttt{x} \stackrel{\varepsilon}{\Longrightarrow} c}{e \vdash \texttt{x}? \stackrel{c?2}{\Longrightarrow} 2} \qquad \dfrac{\dfrac{e \vdash \texttt{x} \stackrel{\varepsilon}{\Longrightarrow} c \qquad e \vdash 2 \stackrel{\varepsilon}{\Longrightarrow} 2}{e \vdash (\texttt{x}, 2) \stackrel{\varepsilon}{\Longrightarrow} (c, 2)}}{e \vdash \texttt{x}\,!\,2 \stackrel{c!2}{\Longrightarrow} ()} \qquad \vdash c\,?\,2 \parallel c\,!\,2 \Rightarrow \varepsilon}{e \vdash \texttt{x}? \parallel (\texttt{x}\,!\,1 \oplus \texttt{x}\,!\,2) \stackrel{\varepsilon}{\Longrightarrow} (2, ())}$$

□

2.2.3 Typing

Communication channels readily lend themselves to a typing similar to the one for references. We require channels to be homogeneous: all values exchanged over a given channel must have the same type. Given the fact that channels are first-class values, it is easy to see (by analogy with lists) that static typing is intractable without this restriction. Then, we give the type τ `chan` to the channels that carry values of type τ.

$$\begin{array}{rcll} \tau & ::= & \ldots & \\ & | & \tau\ \texttt{chan} & \text{channel type} \end{array}$$

For send and receive operations, the type of the value sent or received must be the same as the type of the values that can be transmitted over the channel. Hence the following type assignment:

$$\begin{array}{rcl} \texttt{TypOp(newchan)} & = & \forall\alpha.\ \texttt{unit} \to \alpha\ \texttt{chan} \\ \texttt{TypOp(?)} & = & \forall\alpha.\ \alpha\ \texttt{chan} \to \alpha \\ \texttt{TypOp(!)} & = & \forall\alpha.\ \alpha\ \texttt{chan} \times \alpha \to \texttt{unit} \end{array}$$

The $\|$ and $\oplus$ constructs have obvious typing rules:

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1 \,\|\, a_2 : \tau_1 \times \tau_2} \qquad\qquad \frac{E \vdash a_1 : \tau \qquad E \vdash a_2 : \tau}{E \vdash a_1 \oplus a_2 : \tau}$$

Example. The function `stamp` above has type $\forall\alpha, \beta.$ `int chan` $\to \alpha$ `chan` $\to \beta$. □

Context. The typing proposed above relies crucially on channels being first-class values. For instance, in Berthomieu's [10] and Nielson's [68] calculi, where processes are values but not channels, channels are permitted to carry values of different types. The type of a process consists in the names of the channels on which it offers communication opportunities, with for each channel the type of the transmitted value. Notice the analogy with the typing of extensible records [98, 69, 39, 77, 13, 78, 79]: channels correspond to labels; processes, to records; the transmitted values, to the values contained in the record fields. This typing seems not to extend to the case where channels are first-class values: my feeling is that dependent types are required. This justifies the homogeneity hypothesis over channels, which leads to the simple typing given above. □

As in the case of references, it turns out that the typing for channels given above is sound in a monomorphic type system, but is unsound as soon as polymorphism is introduced.

Example. The following program is well-typed in the type system above:

```
let c = newchan(()) in (c!true) || (1 + c?)
```

That's because `c` can be given type $\forall\alpha.\ \alpha$ `chan`, and therefore be used once with type `bool chan`, once with type `int chan`. Of course, this program produces a run-time type error, since it ends up adding 1 with `true`. □

2.3 Continuations

2.3.1 Presentation

Given a program a_0 and an occurrence of a subexpression a in a_0, we call continuation of a (in a_0) the computation that remains to perform, once a has been evaluated, to get the value of a_0. This continuation can be viewed as the function that maps the value of a to the value of a_0. For instance, in the expression $a_0 = 2 \times a + 1$, the continuation of a is the function $v_a \mapsto 2 \times v_a + 1$. I have purposefully avoided to write $\lambda v_a.\, 2 \times v_a + 1$; indeed, in a simple language such as the one in chapter 1, continuations are not objects that can be manipulated at the language level.

The third and last extension considered here consists precisely in turning continuations into first-class values, that the program can manipulate like any other value. We provide a primitive to capture the continuation of an expression, turning it into a continuation object. We also provide a primitive that discards the current continuation and installs the contents of a continuation object instead. This gives the programmer ample freedom to alter the normal progress of evaluation: continuation objects allows "skipping over" some computations, restart some computations from a saved state, interleave computations, ...

Context. Actually, almost all known control structures — including exceptions, coroutines, backtracking mechanisms, and even communicating processes as in section 2.2 — can be defined in a language with continuation objects: they need not be primitive in the language [36, 97, 80]. Continuation objects are therefore of high interest as a general tool for defining new control structures. □

Continuation objects are presented by two primitive operations:

`Op`	::=	`callcc`	capture of the current continuation
		`throw`	restarting of a captured continuation

The `callcc` primitive takes as argument a functional value. The idiom `callcc`$(\lambda k.\, a)$ binds the identifier k to the continuation of the occurrence of the `callcc`$(\ldots)$ expression in the whole program, then evaluates a and returns its value. The `throw` primitive accepts a pair of arguments, a continuation and a value, and restarts the continuation, feeding it the given value.

Example. The expression

`callcc`$(\lambda$`k.` $1 + ($`if` a `then 2 else throw(k,10`$)))$

evaluates to 3 if the condition a is true, and to 10 if a is false. In the former case, the body of the `callcc` evaluates as $1 + 2$; this value is also the value of the whole `callcc` expression. In the latter case, the continuation `k` is restarted on the value 10 (the second argument to `throw`). The current computation, $1 + ($`if` $\ldots)$, is interrupted, and we proceed as if it terminated on the value 10. □

Context. Continuation objects are usually presented as functions that, when applied to a value, restart the captured continuation over this value. This is the case in Scheme [76], for instance. The presentation above, with a separate `throw` construct for restarting a computation, is taken from Duba, Harper et MacQueen [25]. Its main motivation is to circumvent a typing difficulty in Milner's system. It is also more symmetric than the Scheme presentation, and easier to read in my opinion. □

Example. Consider the following simplified presentation of the ML exceptions: the construct `fail` aborts the current evaluation; the construct a `handle` b evaluates a and returns its value, unless the evaluation of a is aborted by a `fail`, in which case b is evaluated and its value returned. These two operations can easily be implemented using `callcc` and a stack of functions. We assume given the abstract type τ `stack` of mutable stacks. Let `exn_stack` be a global identifier with type (`unit` $\rightarrow$ `unit`) `stack`, initially bound to the empty stack. We then translate a `handle` b by:

```
callcc(λk.
    push(exn_stack, λx.throw(k, b));
    let result = a in pop(exn_stack); result)
```

and `fail` by:

```
pop(exn_stack)(());  anything
```

Here, *anything* denotes any expression that has type τ for any type τ. For instance, we can implement *anything* by the infinite loop (`f where f(x)` = `f(x)`)(`0`). The purpose of *anything* is to ensure that `fail` has type τ for all τ, as in ML. At any rate, the expression *anything* is never executed. □

Example. Most kinds of coroutines can be implemented (albeit painfully) with continuations and references.

```
callcc(λinit_k.
  let next_k = ref(init_k) in
  let communicate = λx.
    callcc(λk.let old_k = !next_k in next_k := k; throw(old_k, x)) in
  let process1 = f1 where f1(x) =
    ... communicate(y) ... f1(z) ... communicate(w) ... in
  let process2 = f2 where f2(x) =
    ... f2(z) ... communicate(y) ... communicate(t) ... in
  process1(callcc(λstart1.
    process2(callcc(λstart2. next_k := start2; throw(start1, 0)))))
```

In the example above, the two functions `process1` and `process2` interleave their computations through the `communicate` function. We first apply `process1` to the initial value 0. This function performs an arbitrary amount of computation, then calls `communicate`(v_1). The control is then transferred to `process2`: we start evaluating `process2`(v_1). When `process2` executes `communicate`(v_2), we restart the evaluation of `process1` where we left it. That is, the call `communicate`(v_1) terminates at last, returning the value v_2. And so forth. The functions `process1` and `process2` can also opt to terminate normally; the corresponding call to `communicate` then passes the return value to the other function. Everything stops when the initial call to `process1` terminates. □

2.3.2 Semantics

The addition of `callcc` and `throw` requires a major rework of the evaluation rules. It becomes necessary to maintain a semantic object that represents the current continuation at each evaluation

step. The evaluation predicate is now $e \vdash a; k \Rightarrow r$ (read: "by evaluating expression a in environment e, then passing the resulting value to the continuation k, we obtain the result r"). Here, k is a term describing what remains to be done, once the expression a is evaluated, to obtain the result r of the program. The continuation terms, as well as the other kinds of semantic objects, are defined by the grammar below. (For continuation terms k, we indicate to the right of each case which point of the computation is represented by k.)

Results:	r	::=	v	normal result (a value)
		|	`err`	error result
Values:	v	::=	cst	base value
		|	(v_1, v_2)	pair of values
		|	(f, x, a, e)	function value
		|	k	continuation
Environments:	e	::=	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	
Continuations:	k	::=	`stop`	end of the program
		|	`primc`(op, k)	after a primitive argument
		|	`app1c`(a, e, k)	after the function part of an application
		|	`app2c`(f, x, a, e, k)	after the argument part of an application
		|	`letc`(x, a, e, k)	after the left part of a `let`
		|	`pair1c`(a, e, k)	after the first argument of a pair
		|	`pair2c`(v, k)	after the second argument of a pair

The head constructor of k describes what should be done with the value of an expression: apply a primitive to it (case `primc`), call a function (case `app2c`), evaluate the other part of an application node (case `app1c`), ... The subterm of k which is itself a continuation describes similarly the next steps of the computation.

The suspended computations represented by k must be performed at some point. This execution is described by another predicate: $\vdash v \triangleright k \Rightarrow r$ (read: "the value v passed to the continuation k produces the response r"). We now give the evaluation rules defining the two predicates $e \vdash a; k \Rightarrow r$ and $\vdash v \triangleright k \Rightarrow r$. The first axiom expresses the behavior of the initial continuation.

$$\vdash v \triangleright \texttt{stop} \Rightarrow v$$

For variables, constants and functions, the value of the expression is immediately available, and we simply pass it to the current continuation.

$$\frac{x \in \mathrm{Dom}(e) \qquad \vdash e(x) \triangleright k \Rightarrow r}{e \vdash x; k \Rightarrow r} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x; k \Rightarrow \texttt{err}}$$

$$\frac{\vdash cst \triangleright k \Rightarrow r}{e \vdash cst; k \Rightarrow r} \qquad \frac{\vdash (f, x, a, e) \triangleright k \Rightarrow r}{e \vdash (f \texttt{ where } f(x) = a); k \Rightarrow r}$$

For a `let` binding, we evaluate the first subexpression after introducing the `letc` constructor on top of the current continuation. When this evaluation terminates, it passes the resulting value v

to a continuation of the form `letc`(...). This restarts the evaluation of the second subexpression of the `let`.

$$\frac{e \vdash a_1; \texttt{letc}(x, a_2, e, k) \Rightarrow r}{e \vdash (\texttt{let } x = a_1 \texttt{ in } a_2); k \Rightarrow r} \qquad \frac{e + x \mapsto v \vdash a_2; k \Rightarrow r}{\vdash v \triangleright \texttt{letc}(x, a_2, e, k) \Rightarrow r}$$

The evaluation of a pair is similar, except that two intermediate steps are required, one after each argument evaluation.

$$\frac{e \vdash a_1; \texttt{pair1c}(a_2, e, k) \Rightarrow r}{e \vdash (a_1, a_2); k \Rightarrow r} \qquad \frac{e \vdash a_2; \texttt{pair2c}(v_1, k) \Rightarrow r}{\vdash v_1 \triangleright \texttt{pair1c}(a_2, e, k) \Rightarrow r} \qquad \frac{\vdash (v_1, v_2) \triangleright k \Rightarrow r}{\vdash v_2 \triangleright \texttt{pair2c}(v_1, k) \Rightarrow r}$$

Function application is treated similarly.

$$\frac{e \vdash a_1; \texttt{app1c}(a_2, e, k) \Rightarrow r}{e \vdash a_1(a_2); k \Rightarrow r} \qquad \frac{e_2 \vdash a_2; \texttt{app2c}(f, x, a, e, k) \Rightarrow r}{\vdash (f, x, a, e) \triangleright \texttt{app1c}(a_2, e_2, k) \Rightarrow r}$$

$$\frac{v \text{ does not match } (f, x, a, e)}{\vdash v \triangleright \texttt{app1c}(a_2, e, k) \Rightarrow \texttt{err}} \qquad \frac{e + f \mapsto (f, x, a, e) + x \mapsto v_2 \vdash a; k \Rightarrow r}{\vdash v_2 \triangleright \texttt{app2c}(f, x, a, e, k) \Rightarrow r}$$

For the primitives, the first step is common to all primitives: evaluate the argument.

$$\frac{e \vdash a; \texttt{primc}(op, k) \Rightarrow r}{e \vdash op(a); k \Rightarrow r}$$

The semantics for primitives is given by the elimination rules for the `primc` continuations. For `callcc`, we duplicate the current continuation k by storing it inside the environment, then we evaluate the body of the argument function.

$$\frac{e + f \mapsto (f, x, a, e) + x \mapsto k \vdash a; k \Rightarrow r}{\vdash (f, x, a, e) \triangleright \texttt{primc}(\texttt{callcc}, k) \Rightarrow r} \qquad \frac{v \text{ does not match } (f, x, a, e)}{\vdash v \triangleright \texttt{primc}(\texttt{callcc}, k) \Rightarrow \texttt{err}}$$

Symmetrically, the rule for `throw` discards the current continuation, and uses instead the continuation k' provided by its argument. (The evaluations of `callcc` and `throw` are the only ones that do not treat the current continuation in a linear way.)

$$\frac{\vdash v \triangleright k' \Rightarrow r}{\vdash (k', v) \triangleright \texttt{primc}(\texttt{throw}, k) \Rightarrow r} \qquad \frac{v \text{ does not match } (k', v')}{\vdash v \triangleright \texttt{primc}(\texttt{throw}, k) \Rightarrow \texttt{err}}$$

2.3.3 Typing

Following a well-tried approach, we introduce the type τ `cont` of the continuation objects that expect a value of type τ as input.

$$\begin{array}{rcll} \tau & ::= & \ldots & \\ & | & \tau \texttt{ cont} & \text{continuation type} \end{array}$$

Then, $\texttt{throw}(a_1, a_2)$ is well-typed if a_1 has type τ `cont` and a_2 has type τ, for some type τ. The expression $\texttt{throw}(a_1, a_2)$ itself can be assigned any type. That's because it never terminates normally: no value is ever returned to the enclosing context. Hence the context can make any assumptions on the type of the returned value. We therefore take:

$$\texttt{TypOp(throw)} \quad = \quad \forall \alpha, \beta.\ \alpha\ \texttt{cont} \times \alpha \to \beta$$

The typing of `callcc` is more delicate. Consider the program $\Gamma[\texttt{callcc}(\lambda k.\, a)]$, where Γ is the context enclosing the `callcc` expression. Let τ be the type of $\texttt{callcc}(\lambda k.\, a)$. This expression normally returns whatever a returns; hence τ is also the type of a. But τ is also the type expected by the context Γ for the value of `callcc`. The continuation k, which is just a representation of Γ, therefore expects a value with type τ; hence its type is τ `cont`. Hence:

$$\texttt{TypOp(callcc)} \quad = \quad \forall \alpha.\ (\alpha\ \texttt{cont} \to \alpha) \to \alpha$$

Context. These types are those proposed by Duba, Harper and MacQueen for the continuations in SML-NJ [25]. Treating continuation objects as an abstract type, not as functions, circumvents one of the restrictions of the ML type system (that quantification is not allowed inside type expressions). The reader is referred to Duba, Harper and MacQueen's paper for more details, and for an excellent discussion of the alternate typings. □

The typing proposed above has convinced the ML community for about two years. Duba, Harper and MacQueen showed its soundness for a monomorphic type system [25]; they claimed that their proof easily extends to a polymorphic type system. This encouraged Felleisen and Wright to publish the soundness of the above typing for `callcc` in the ML polymorphic type system [101, first edition]. Unfortunately, the typing for `callcc` and `throw` proposed above is unsound: a continuation object with a polymorphic type compromises type safety. Here is the first known counterexample, due to Harper and Lillibridge [35].

```
let later =
    callcc(λk.
        (λx. x),
        (λf. throw(k,(f, λx. ()))))
in
    print_string(first(later)("Hello!"));
    second(later)(λx. x + 1)
```

This counterexample is more complex than the ones for references and channels; but it demonstrates basically the same phenomenon. The typing proposed for `callcc` and `throw` leads to assuming:

$$\texttt{later} \quad : \quad \forall \alpha.\, (\alpha \to \alpha) \times (\alpha \to \alpha) \to \texttt{unit}$$

The application of `first(later)` to a character string is therefore legitimate, as well as the application of `second(later)` to the successor function. However, the continuation k captured by the `callcc` is λ`later. print_string`... (the body of the `let`). Hence, the call to `second(later)` re-evaluates the body of the `let` with `first(later)` equals to λ`x. x + 1`, causing a run-time type violation when `first(later)` is applied to a character string. As in the case of references and channels, we have abused the fact that the continuation has a polymorphic type to incorrectly apply it to values that are not general enough.

Chapter 3

Dangerous variables and closure typing

This chapter presents a polymorphic type system for the imperative extensions introduced in chapter 2 (references, channels, continuations).

3.1 Informal presentation

Typing no longer ensures type safety as soon as it allows polymorphic references. (The same holds for polymorphic channels and for polymorphic continuations. To be more specific, the following discussion concentrates on references; unless otherwise mentioned, everything we say about references also applies to channels and continuations.) Hence the type system must be restricted in order to prevent references from being assigned polymorphic types — that is, non-trivial type schemes $\forall\alpha.\,\tau[\alpha]$.

3.1.1 Dangerous variables

The system I propose relies essentially on a restriction over the type generalization step (as performed by the `let` construct). The restriction consists in not generalizing type variables that are free in the type of a live reference. These variables are called variables occurring in dangerous position, or *dangerous variables* for short. The type of a reference may well contain type variables; but these variables are never generalized. Hence a given reference cannot be considered with several different types during its lifespan.

It now remains to detect, at generalization-time, the type variables that are free in the type of a live reference. To do so, we rely on the type system itself: the idea is that the type to be generalized is precise enough to keep track of those references reachable from values belonging to that type. We argue by case analysis on the type.

• **References.** All variables free in a reference type τ `ref` are dangerous, since the values belonging to that type are references to values of type τ. Example:

```
let r = ref(λx.x) in
    r := (λn.n + 1);
    if (!r)(true) then ... else ...
```

The expression bound to `r` has type $(\alpha \to \alpha)$ `ref`. The variable α occurs in dangerous position in this type, since it appears free under a `ref` type constructor. Hence it is not generalized. The body of the `let` is typed under the assumption $\texttt{r} : (\alpha \to \alpha)$ `ref`, instead of $\texttt{r} : \forall\alpha.(\alpha \to \alpha)$ `ref` as in the naive, unsound type system. The typing of the assignment can only succeed if α is instantiated to `int`. Then, the application (!r)(true) is ill-typed.

• **Data structures.** In ML, the type of a data structure (pair, list, any concrete data type) indicates not only the kind of the structure, but also the types of the components of the structure. For instance, a list type is not just `list`, but τ `list`, indicating that all components of a structure with type τ `list` have type τ. Similarly, the product type $\tau_1 \times \tau_2$ indicates that all values with that type contain one component with type τ_1, one component with type τ_2, and nothing else. References contained into data structures therefore "show through" in the type of the structure. For instance, the variables dangerous in τ `list` are those variables dangerous in τ; similarly, the variables dangerous in $\tau_1 \times \tau_2$ are the variables dangerous in τ_1 or in τ_2. Example:

```
let p = (λx.x, ref(λx.1)) in a
```

In the type $(\alpha \to \alpha) \times ((\beta \to \texttt{int})\ \texttt{ref})$, the variable β is dangerous, but not α. Hence a is typed under the assumption $\texttt{p} : \forall\alpha.(\alpha \to \alpha) \times ((\beta \to \texttt{int})\ \texttt{ref})$.

Remark. This argument extends easily to the full ML concrete types. Consider the following concrete type declaration:

$$\texttt{type}\ (\alpha_1, \ldots \alpha_n)\ T\ =\ C_1\ \texttt{of}\ \tau_1 \mid \ldots \mid C_n\ \texttt{of}\ \tau_n$$

The k^{th} parameter of T is said to be essentially dangerous if α_k occurs in dangerous position in any of the $\tau_1 \ldots \tau_n$. (For instance, if $\tau_k = \alpha_j$ `ref`. This corresponds to the case where the type T introduces a reference type by itself.) Then, a variable α is dangerous in the type expression $(\tau_1, \ldots, \tau_n)\ T$ if α is dangerous in one of the τ_j, or if α is free in τ_k and the k^{th} parameter of T is essentially dangerous. □

• **Functions without free variables.** The case of a value with type $\tau_1 \to \tau_2$ is special. A value of that type is a function that transforms values of type τ_1 into values of type τ_2; the functional value itself does not generally contain a value of type τ_1 nor a value of type τ_2. Hence a variable can be dangerous in τ_1 or τ_2, without being dangerous in $\tau_1 \to \tau_2$. A dangerous variable in τ_1 keeps track of a reference in the argument that will be passed to the function when it is applied; a dangerous variable in τ_2 reveals that a reference might appear in the function result, once it is applied; but those references are not presently contained in the functional value. Example:

```
let make_ref = λx.ref(x) in ... make_ref(1) ...
make_ref(true) ...
```

The variable α is not dangerous in $\alpha \to \alpha$ `ref`. Hence, `make_ref` is given the polymorphic type $\forall\alpha.\,\alpha \to \alpha$ `ref`, and it can be applied to objects of any type. The fact that `make_ref` has such a general type does not compromise type safety at all. That's because the only way to break the type system with `make_ref` is to apply it to a polymorphic object, then to use the resulting reference with two different types, through a `let` binding:

```
let r = make_ref(λx.x) in
    r := (λn.n + 1);
    if (!r)(true) then ... else ...
```

But this program is ill-typed: the result of `make_ref` has type $(\alpha \to \alpha)$ `ref`, and α is dangerous in this type, hence `r` remains monomorphic.

3.1.2 Closure typing

The discussion of functions above neglects a crucial point: a function can contain free identifiers, and these free identifiers can be bound to references. In this case, the functional value does contain references, in the environment part of its closure, yet this is not apparent on its type. A function type $\tau_1 \to \tau_2$ does not say anything about the types of the identifiers free in the functions belonging to that type.

Example. A reference can be presented as two functions, one for reading, the other for writing.

```
let functional_ref =
    λx. let r = ref(x) in (λ().!r), (λy.r := y) in
let (read,write) =
    functional_ref(λx.x) in
write(λn.n + 1); if read()(true) then ... else ...
```

The example above is well-typed, with the following type assignment:

$$\begin{aligned} \texttt{functional\_ref} &: \forall\alpha.\,\alpha \to (\texttt{unit} \to \alpha) \times (\alpha \to \texttt{unit}) \\ \texttt{read} &: \forall\beta.\,\texttt{unit} \to (\beta \to \beta) \\ \texttt{write} &: \forall\beta.\,(\beta \to \beta) \to \texttt{unit} \end{aligned}$$

No type variable is dangerous here: the types do not even contain the `ref` constructor. Yet, the `read`/`write` team breaks the type system just as a polymorphic reference. □

What can we do? Type functions better. Functional values are actually data structures, with components the values bound to their free identifiers. This is obvious when we think of functions as closures. Yet, in contrast with the other ML data structures, the type of a functional value does not permit to retrieve the types of its components. We are therefore going to assign more detailed types to functional values — types that also type functions viewed as data structures. This is what I call *closure typing*.

In the following examples, we shall consider function types of the form

$$\tau' \multimap\langle\sigma_1, \ldots, \sigma_n\rangle\rightarrowtail \tau''$$

where the σ_i are the types of the identifiers free in the function. (The σ_i are actually type schemes, since these identifiers can have polymorphic types.) When the function is closed, we often write $\rightarrow$ instead of $-\langle\ \rangle\rightarrow$. The set $\{\sigma_1, \ldots, \sigma_n\}$ is what I call a closure type: it types the data structure part of a functional value. The variables dangerous in the function type above are naturally the variables dangerous in one of the σ_i. In the last example above, we therefore have:

$$\texttt{functional\_ref} : \forall\alpha.\ \alpha \rightarrow (\texttt{unit} -\langle \alpha\ \texttt{ref}\rangle\rightarrow \alpha) \times (\alpha -\langle \alpha\ \texttt{ref}\rangle\rightarrow \texttt{unit})$$

That's because the two returned functions contain one free identifier, `r`, with type α `ref`. To obtain `read` and `write`, we specialize α to $\beta \rightarrow \beta$, and this leads to the types:

$$\begin{aligned} \texttt{read} &: \texttt{unit} -\langle (\beta \rightarrow \beta)\ \texttt{ref}\rangle\rightarrow (\beta \rightarrow \beta) \\ \texttt{write} &: (\beta \rightarrow \beta) -\langle (\beta \rightarrow \beta)\ \texttt{ref}\rangle\rightarrow \texttt{unit} \end{aligned}$$

The variable β is dangerous in the new types for `read` and `write`, since it is dangerous in the closure type $(\beta \rightarrow \beta)$ `ref`. Hence we cannot generalize over β, and the example is rejected as ill-typed.

Closure typing precisely reflects how a function interleaves parameter passing and internal computations. For instance, we have:

$$\begin{aligned} \lambda().\texttt{ref}(\lambda\texttt{x}.\texttt{x}) &: \texttt{unit} -\langle\ \rangle\rightarrow (\alpha \rightarrow \alpha)\ \texttt{ref} \\ \texttt{let r} = \texttt{ref}(\lambda\texttt{x}.\texttt{x})\ \texttt{in}\ \lambda().\texttt{r} &: \texttt{unit} -\langle (\alpha \rightarrow \alpha)\ \texttt{ref}\rangle\rightarrow (\alpha \rightarrow \alpha)\ \texttt{ref} \end{aligned}$$

In the former case, α is not dangerous, hence can be generalized. This is safe, since the function returns a fresh reference each time it is called. In the latter case, α is dangerous, and the function remains monomorphic. Generalizing over α would be unsafe, since the reference `r` is shared between all calls to the function.

3.1.3 Structure of closure types

I was asked several times why function types are annotated with sets of type schemes, instead of sets of dangerous variables: the variables dangerous in the types of the identifiers free in the function. This would lead to more compact type expressions. This approach raises the following issue: a function can contain free identifiers with a polymorphic type containing no dangerous variables (such as α `list`); with the approach suggested above, we will not keep track of the types of these identifiers; but later, this polymorphic type can be instantiated to a type containing dangerous variables (α `list` becomes β `ref list`, for instance). Here is an example that demonstrates this situation:

```
let K = λx.λy.x in
let f = K(ref(λz.z)) in
    f(0) := λx.x + 1;
    if f(0)(true) then ... else ...
```

The fonction `K` is given type $\forall\alpha, \beta.\ \alpha \rightarrow (\beta \rightarrow \alpha)$. The two arrow types carry no annotation: it is true that `x` is free in λ`y.x`, but the type of `x`, which is α, contains no dangerous variables. Then, `f` has type $\forall\beta, \gamma.\ \beta \rightarrow (\gamma \rightarrow \gamma)$ `ref`. The variable γ is not dangerous in this type, hence it has been

generalized. However, the remainder of the example causes a run-time type violation. The problem is that `f` contains a reference to the identity function; but we have lost track of this reference at the level of types by passing it through `K`.

One solution is to annotate function types not only by the variables that are already dangerous in the type σ of a free identifier, but also by the type variables that can become dangerous by instantiation. I call these variables the visible variables in type σ. All variables free in σ are not necessarily visible; for instance, in $\alpha \to \alpha$, the variable α can be substituted by any type without introducing dangerous variables. The visible variables are, informally, the free variables that occur somewhere else than in the left or right part of arrow types. In this approach, function types are therefore adorned by a set of type variables marked either "dangerous" (written α `dang`), or "visibles" (written α `visi`). For instance, in the example above, we obtain the following typings:

$$\begin{aligned} \texttt{functional\_ref} &: \forall\alpha.\ \alpha \to (\texttt{unit} \multimap\langle \alpha\ \texttt{dang}\rangle\mapsto \alpha) \times (\alpha \multimap\langle \alpha\ \texttt{dang}\rangle\mapsto \texttt{unit}) \\ \texttt{K} &: \forall\alpha,\beta.\ \alpha \to (\beta \multimap\langle \alpha\ \texttt{visi}\rangle\mapsto \alpha) \end{aligned}$$

The function $\lambda\texttt{y.\,x}$ possesses a free identifier, `x`, with type α; hence the variable α is marked visible in the function type.

The substitution rules over these closure types are unusual. When α becomes τ, the closure type α `dang` becomes α_1 `dang`$,\ldots,\alpha_n$ `dang`, where the α_i are the variables free in τ. The closure type α `visi` becomes β_1 `dang`$,\ldots,\beta_n$ `dang`$,\gamma_1$ `visi`$,\ldots\gamma_m$ `visi`, where the β_i are the variables dangerous in τ, and the γ_j are the variables visible but not dangerous in τ. For instance, to apply `K` to `ref`$(\lambda\texttt{z.\,z})$, we instantiate α to $(\gamma \to \gamma)$ `ref` (dangerous variables: γ; visible variables: none). The type of the result, `f`, is therefore:

$$\texttt{f} : \forall\beta.\ \beta \multimap\langle \gamma\ \texttt{dang}\rangle\mapsto (\gamma \to \gamma)\ \texttt{ref}$$

The variable γ cannot be generalized, since it is dangerous in this type.

To conclude: by annotating function types by set of variables marked "visible" or "dangerous", we correctly keep track of the dangerous objects contained into closures. This results in smaller closure types, but complicates some operations over closure types — most notably, substitution. In the following, we shall therefore stick to the initial approach, with closure types annotated by sets of complete type schemes. This makes formalization easier, and provides a simpler semantic interpretation for closure types.

3.1.4 Extensibility and polymorphism over closure types

In the informal discussion above, we have annotated function types by sets of type schemes. This simple approach to closure typing is not completely satisfactory.

First of all, we must provide a notion of extensibility over closure types: an object with type $\tau \multimap\langle\pi\rangle\mapsto \tau'$, where π is a set of type schemes, also belongs to the types $\tau \multimap\langle\pi'\rangle\mapsto \tau'$ for all supersets π' of π. This is semantically correct, since the only requirement over π is that it contains at least the types of the values contained in a closure; the set π can contain more types. Without this extension mechanism, closure typing would be restrictive to the point of being impractical.

Consider for instance two function expressions that have the same argument type, the same result type, but different closure types:

$$\begin{array}{rcl}(\texttt{f where f(x)} = \ldots) & : & \texttt{int} \multimap\langle \alpha\ \texttt{list}\rangle\rightarrow \texttt{int} \\ (\texttt{g where g(x)} = \ldots) & : & \texttt{int} \multimap\langle \beta\ \texttt{ref}\rangle\rightarrow \texttt{int}\end{array}$$

If closure types cannot be extended, then the following phrase must be rejected as ill-typed:

$$\texttt{if } \ldots \texttt{ then (f where f(x)} = \ldots\texttt{) else (g where g(x)} = \ldots\texttt{)}.$$

Clearly, this phrase is type-safe. The type system must accept it, with its natural type:

$$\texttt{int} \multimap\langle \alpha\ \texttt{list}, \beta\ \texttt{ref}\rangle\rightarrow \texttt{int}$$

If closure types can be extended, then the expected typing is obtained by extending the types of `f` and `g` to $\texttt{int} \multimap\langle \alpha\ \texttt{list}, \beta\ \texttt{ref}\rangle\rightarrow \texttt{int}$. More generally, extensibility of closure types should ensure that two function types are compatible if and only if they have identical argument types and result types — just as in the usual ML type system.

Closure typing also requires another feature: closure type variables that can be universally quantified, just like regular type variables. These are required to support higher-order functions that take as argument any function with the correct argument type and the correct result type, whatever its closure type is. For instance, the function

$$\texttt{appl where appl(f)} = 2 + \texttt{f(1)}$$

must apply to all values with type $\texttt{int} \multimap\langle \pi\rangle\rightarrow \texttt{int}$, for all closure types π. Conversely, we must not loose track of the references contained in the closures passed to higher-order functions. Otherwise, we could "launder" functions containing references, just by passing them through a higher-order function. Example:

```
let BCCI = λf.λx.f(x) in
let f = let r = ref(λz.z) in λy.r
let g = BCCI(f) in
    g(1) := (λn.n + 1); ...
```

Assume the following naive typing for `BCCI`:

$$\texttt{BCCI} : \forall\alpha, \beta.\, (\alpha \multimap\langle\ \rangle\rightarrow \beta) \rightarrow (\alpha \multimap\Big\langle \alpha \multimap\langle\ \rangle\rightarrow \beta\Big\rangle\rightarrow \beta)$$

Then, we obtain the following typings for `f` and `g`:

$$\begin{array}{rcl}\texttt{f} & : & \forall\delta.\ \delta \multimap\Big\langle(\gamma \rightarrow \gamma)\ \texttt{ref}\Big\rangle\rightarrow (\gamma \rightarrow \gamma)\ \texttt{ref} \\ \texttt{g} & : & \forall\gamma, \delta.\ \delta \multimap\langle\ \rangle\rightarrow (\gamma \rightarrow \gamma)\ \texttt{ref}\end{array}$$

The variable γ is dangerous in the type of `f`, but not in the type of `g`. That's because the closure type for `f` does reveal the presence of a reference polymorphic over γ; but this fact has been forgotten in the type of `g`. Hence, `g`(1) is a polymorphic reference. The problem lies within the type for `BCCI`.

In this type, the two arrows $-\langle\,\rangle\rightarrow$ type two closures that have the same components. Hence, when one of these two closure types is extended, the other should be extended similarly. But this is not ensured by the extension mechanism.

To solve this problem, we introduce closure type variables, with typical element u, v. These variables are used to represent unknown closure types: the closure types for functional parameters. Then, the correct type for `BCCI` is:

$$\texttt{BCCI} : \forall\alpha,\beta,u.\ (\alpha\ -\langle u\rangle\rightarrow\ \beta) \rightarrow (\alpha\ -\Big\langle\alpha\ -\langle u\rangle\rightarrow\ \beta\Big\rangle\rightarrow\ \beta)$$

Closure type variables u can be generalized just as regular type variables t. In the type scheme above, u can be instantiated by any closure type. The application `BCCI(f)` is therefore well-typed: u becomes γ `list ref`, and we get as result type:

$$\texttt{g}\ :\ \forall\delta.\ \delta\ -\Big\langle\delta\ -\langle\gamma\ \texttt{list ref}\rangle\rightarrow\ \gamma\ \texttt{list ref}\Big\rangle\rightarrow\ \gamma\ \texttt{list ref}$$

This type correctly keeps track of the reference: γ is dangerous there, hence cannot be generalized.

3.2 A first type system

In this section, we formalize a first type system for references, channels and continuations that includes the ideas of dangerous variables and closure typing. The main novelty of this system with respect to what we have described above is the use of closure type variables not only to provide polymorphism over closure types, but also to make closure types extensible. To this end, all closure types considered have the form:

$$\pi = \{\sigma_1,\ldots,\sigma_n\} \cup u$$

A closure type is therefore a set of type schemes completed by a closure type variable u. The variable u is also called an extension variable, since it suffices to instantiate u by a closure type $\pi' = \{\sigma'_1,\ldots,\sigma'_m\} \cup u'$ to get

$$\{\sigma_1,\ldots,\sigma_n,\sigma'_1,\ldots,\sigma'_m\} \cup u',$$

which is an extension of π.

Context. We are actually encoding a notion of subtyping (the extension rule) with parametricity. This approach has proved successful for the polymorphic typing of extensible records [98, 69, 39, 77, 78, 79, 32]. (What we call "extension variables" are referred to as "row variables" in the records terminology.) This approach results in type systems that are slightly less expressive than those that provide separate mechanisms for subtyping and for parametricity [13], but that are considerably simpler, and easily lend themselves to type inference. The same holds for closure types. The problems are however much simpler than in the case of extensible records: the contents of a closure type have much less structure than the contents of a record type (thanks to the set structure and the absence of labels). □

In the following, we write $\pi = \sigma_1,\ldots,\sigma_n,u$ for the closure type $\{\sigma_1,\ldots,\sigma_n\} \cup u$. The comma can be seen as a right-associative binary operator: the operator that adds a type scheme to a closure type.

3.2.1 Type expressions

The type variables are divided in two kinds: those ranging over types, written t; and those ranging over closure types, written u. A variable, written α or β, is either a type variable t or a closure type variable u.

t	$\in$	`VarTypExp`	type variable
u	$\in$	`VarTypClos`	closure type variable
α, β	$::=$	$t \mid u$	variable

The grammar below defines three sets of types: the set `TypExp` of TYPES or more precisely SIMPLE TYPES, written τ; the set `TypClos` of CLOSURE TYPES, written π; and the set `SchTyp` of TYPE SCHEMES, written σ.

τ	$::=$	ι	base type
	$\mid$	t	type variable
	$\mid$	$\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$	function type
	$\mid$	$\tau_1 \times \tau_2$	product type
	$\mid$	τ `ref`	reference type
	$\mid$	τ `chan`	channel type
	$\mid$	τ `cont`	continuation type
π	$::=$	u	extension variable
	$\mid$	σ, π	addition of scheme σ to closure type π
σ	$::=$	$\forall\alpha_1 \ldots \alpha_n.\, \tau$	type scheme

The substitutions over this type algebra are finite mappings from type variables to types, and from closure type variables to closure types.

$$\text{Substitutions:} \quad \varphi, \psi \quad ::= \quad [t \mapsto \tau, \ldots, u \mapsto \pi, \ldots]$$

As described in chapter 1, substitutions naturally extend to homomorphisms of types, closure types, and type schemes. In particular, we have:

$$\begin{aligned} \varphi(\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2) &= \varphi(\tau_1) \multimap\langle\varphi(\pi)\rangle\rightarrow \varphi(\tau_2) \\ \varphi(\sigma, \pi) &= \varphi(\sigma), \varphi(\pi) \end{aligned}$$

Type schemes are identified up to a renaming of the variables bound by $\forall$:

$$\forall\alpha_1 \ldots \alpha_n.\, \tau = \forall\beta_1 \ldots \beta_n.\, [\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n](\tau).$$

Closure types are identified modulo the following two axioms:

σ_1, σ_2, π	$=$	σ_2, σ_1, π	left commutativity
σ, σ, π	$=$	σ, π	idempotence

That is, from now on, we work in the quotient sets of the sets of type expressions defined above by those two relations. In the following, what is written τ, σ, π are representatives for elements of the quotient sets. These axioms reflect the structure of (extensible) set that we impose over closure types. They establish a one-to-one correspondence between closure types π and pairs (Σ, u) of a set Σ of schemes and of an extension variable u.

3.2.2 Free variables, dangerous variables

To each type expression τ, we associate two sets of variables: $\mathcal{F}(\tau)$, the FREE VARIABLES of τ; and $\mathcal{D}(\tau)$, the DANGEROUS VARIABLES of τ. We also define $\mathcal{F}$ and $\mathcal{D}$ over closure types π and type schemes σ. Here are their definitions, by structural induction on the type expressions:

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{F}(\iota) & = & \emptyset & \mathcal{D}(\iota) & = & \emptyset \\
\mathcal{F}(t) & = & \{t\} & \mathcal{D}(t) & = & \emptyset \\
\mathcal{F}(\tau_1 \multimap\!\langle\pi\rangle\!\rightarrow \tau_2) & = & \mathcal{F}(\tau_1) \cup \mathcal{F}(\pi) \cup \mathcal{F}(\tau_2) & \mathcal{D}(\tau_1 \multimap\!\langle\pi\rangle\!\rightarrow \tau_2) & = & \mathcal{D}(\pi) \\
\mathcal{F}(\tau_1 \times \tau_2) & = & \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) & \mathcal{D}(\tau_1 \times \tau_2) & = & \mathcal{D}(\tau_1) \cup \mathcal{D}(\tau_2) \\
\mathcal{F}(\tau\ \texttt{ref}) & = & \mathcal{F}(\tau) & \mathcal{D}(\tau\ \texttt{ref}) & = & \mathcal{F}(\tau) \\
\mathcal{F}(\tau\ \texttt{chan}) & = & \mathcal{F}(\tau) & \mathcal{D}(\tau\ \texttt{chan}) & = & \mathcal{F}(\tau) \\
\mathcal{F}(\tau\ \texttt{cont}) & = & \mathcal{F}(\tau) & \mathcal{D}(\tau\ \texttt{cont}) & = & \mathcal{F}(\tau) \\
\mathcal{F}(u) & = & \{u\} & \mathcal{D}(u) & = & \emptyset \\
\mathcal{F}(\sigma, \pi) & = & \mathcal{F}(\sigma) \cup \mathcal{F}(\pi) & \mathcal{D}(\sigma, \pi) & = & \mathcal{D}(\sigma) \cup \mathcal{D}(\pi) \\
\mathcal{F}(\forall\alpha_1 \ldots \alpha_n.\, \tau) & = & \mathcal{F}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\} & \mathcal{D}(\forall\alpha_1 \ldots \alpha_n.\, \tau) & = & \mathcal{D}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\}
\end{array}$$

It is easy to check that this definition of $\mathcal{F}$ and $\mathcal{D}$ is compatible with the axioms over type schemes and closure types.

The following proposition shows the effect of a substitution over the free variables and the dangerous variables.

Proposition 3.1 *Let φ be a substitution. For all types τ, we have:*

$$\mathcal{F}(\varphi(\tau)) = \left(\bigcup_{\alpha \in \mathcal{F}(\tau)} \mathcal{F}(\varphi(\alpha)) \right)$$

$$\left(\bigcup_{\alpha \in \mathcal{D}(\tau)} \mathcal{F}(\varphi(\alpha)) \right) \subseteq \mathcal{D}(\varphi(\tau)) \subseteq \left(\bigcup_{\alpha \in \mathcal{D}(\tau)} \mathcal{F}(\varphi(\alpha)) \right) \cup \left(\bigcup_{\alpha \in \mathcal{F}(\tau)} \mathcal{D}(\varphi(\alpha)) \right)$$

These results also hold with the type τ replaced by a closure type π or a type scheme σ.

Proof: by simultaneous structural induction over τ, π and σ. □

3.2.3 Typing rules

We now give the inference rules that define the typing judgement $E \vdash a : \tau$ ("under the assumptions E, the expression a has type τ"). The environment E is a finite mapping from identifiers to type schemes. The only rules that differ from those in chapter 1 are the rules for functions and for `let`.

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

As in chapter 1, the instantiation relation $\leq$ is defined by: $\tau \leq \sigma$ if and only if σ is $\forall\alpha_1 \ldots \alpha_n.\, \tau_0$, and there exists a substitution φ, whose domain is a subset of $\{\alpha_1 \ldots \alpha_n\}$, such that τ is $\varphi(\tau_0)$.

$$\frac{E + f \mapsto (\tau_1 \multimap\langle E(y_1), \ldots, E(y_n), \pi\rangle\rightarrow \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2 \qquad \{y_1 \ldots y_n\} = \mathcal{I}(f \texttt{ where } f(x) = a)}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle E(y_1), \ldots, E(y_n), \pi\rangle\rightarrow \tau_2}$$

The typing rule for functions therefore requires the closure type for the function to contain at least the types of the free identifiers. (Even if the rule appears to require that these types appear at the beginning of the closure type, they can actually appear anywhere in the closure type, because of the commutativity axiom.)

$$\frac{E \vdash a_1 : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1} \qquad \frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2}$$

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

The typing rule for `let` is unchanged; the difference with chapter 1 lies in the definition of the `Gen` operator. We now take:

$$\texttt{Gen}(\tau, E) = \forall\alpha_1 \ldots \forall\alpha_n.\, \tau \quad \text{with} \quad \{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau) \setminus \mathcal{D}(\tau) \setminus \mathcal{F}(E).$$

The difference with the `Gen` operator used in chapter 1 is that we do not generalize the type variables that are dangerous in τ.

$$\frac{\tau \leq \texttt{TypCst}(cst)}{E \vdash cst : \tau} \qquad \frac{\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2 \leq \texttt{TypOp}(op) \qquad E \vdash a : \tau_1}{E \vdash op(a) : \tau_2}$$

For the types of the primitives over references, channels and continuations, we keep the natural types shown in chapter 2, enriched with trivial closure types:

$$\begin{aligned}
\texttt{TypOp(ref)} &= \forall t, u.\ t \multimap\langle u\rangle\rightarrow t \texttt{ ref} \\
\texttt{TypOp(!)} &= \forall t, u.\ t \texttt{ ref} \multimap\langle u\rangle\rightarrow t \\
\texttt{TypOp(:=)} &= \forall t, u.\ t \texttt{ ref} \times t \multimap\langle u\rangle\rightarrow \texttt{unit} \\
\texttt{TypOp(newchan)} &= \forall t, u.\ \texttt{unit} \multimap\langle u\rangle\rightarrow t \texttt{ chan} \\
\texttt{TypOp(?)} &= \forall t, u.\ t \texttt{ chan} \multimap\langle u\rangle\rightarrow t \\
\texttt{TypOp(!)} &= \forall t, u.\ t \texttt{ chan} \times t \multimap\langle u\rangle\rightarrow \texttt{unit} \\
\texttt{TypOp(callcc)} &= \forall t, u, u'.\ (t \texttt{ cont} \multimap\langle u\rangle\rightarrow t) \multimap\langle u'\rangle\rightarrow t \\
\texttt{TypOp(throw)} &= \forall t, t', u.\ t \texttt{ cont} \times t \multimap\langle u\rangle\rightarrow t'
\end{aligned}$$

3.2.4 Properties of the type system

Proposition 3.2 (Typing is stable under substitution) *Let a be an expression, τ be a type, E be a typing environment and φ be a substitution. If $E \vdash a : \tau$, then $\varphi(E) \vdash a : \varphi(\tau)$.*

Proof: by structural induction over a. I give the only case that differs from the proof of proposition 1.2.

• **Case** $a = (\texttt{let}\ x = a_1\ \texttt{in}\ a_2)$. The typing derivation ends up with:

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Take $\sigma = \forall\alpha_1 \ldots \forall\alpha_n . \tau_1$ with $\{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau_1) \setminus \mathcal{D}(\tau_1) \setminus \mathcal{F}(E)$. Let $\beta_1 \ldots \beta_n$ be variables out of reach for φ, not free in E, with β_i of the same kind as α_i for all i. Define the substitution $\psi = \varphi \circ [\alpha_1 \mapsto \beta_1 \ldots \alpha_n \mapsto \beta_n]$.

We apply the induction hypothesis twice: once to the left premise, with the substitution ψ; and once to the right premise, with the substitution φ. We get proofs of:

$$\psi(E) \vdash a_1 : \psi(\tau_1) \qquad \varphi(E) + x \mapsto \varphi(\texttt{Gen}(\tau_1, E)) \vdash a_2 : \varphi(\tau_2)$$

Since the α_i are not free in E, we have $\psi(E) = \varphi(E)$. It now remains to show that $\texttt{Gen}(\psi(\tau_1), \psi(E))$ equals $\varphi(\texttt{Gen}(\tau_1, E))$. Write

$$V = \mathcal{F}(\psi(\tau_1)) \setminus \mathcal{D}(\psi(\tau_1)) \setminus \mathcal{F}(\psi(E)).$$

By construction of ψ and of the β_i, we have $\psi(\alpha_i) = \varphi(\beta_i) = \beta_i$. Moreover, for any variable α that is not one of the α_i, none of the β_i are free in the type $\psi(\alpha) = \varphi(\alpha)$.

We now fix i. Since α_i is free in τ_1, β_i is free in $\psi(\tau_1)$ (proposition 3.1, first result). Since α_i is not free in E, β_i is not free in $\psi(E)$. Otherwise, we would have $\beta_i \in \mathcal{F}(\psi(\alpha))$ for some α that belongs to $\mathcal{F}(E)$, by proposition 3.1; but only α_i meets the first requirement, and it is not free in E. Finally, β_i is not dangerous in $\psi(\tau_1)$. Otherwise, we would have (by proposition 3.1, second result) either $\beta_i \in \mathcal{F}(\psi(\alpha))$ for some $\alpha \in \mathcal{D}(\tau_1)$, or $\beta_i \in \mathcal{D}(\psi(\alpha))$ for some $\alpha \in \mathcal{F}(\tau_1)$. In both cases, only $\alpha = \alpha_i$ fits the bill. Yet α_i is not in $\mathcal{D}(\tau_1)$, which excludes the first alternative. And $\mathcal{D}(\psi(\alpha_i)) = \mathcal{D}(\beta_i) = \emptyset$, which excludes the other alternative. We therefore conclude that

$$\{\beta_1 \ldots \beta_n\} \subseteq V.$$

We now show the converse inclusion. Let β be a variable free in $\psi(\tau_1)$, and which is not one of the β_i. Take $\alpha \in \mathcal{F}(\tau_1)$ such that $\beta \in \mathcal{F}(\psi(\alpha))$. The variable α cannot be one of the α_i, because otherwise β would be one of the β_i. Hence either α is free in E, or α is dangerous in τ_1. If α is free in E, then β is free in $\psi(E)$. If α is dangerous in τ_1, then β is dangerous in $\psi(\tau_1)$. In both cases, $\beta \notin V$. Hence the converse inclusion.

It follows that

$$\texttt{Gen}(\psi(\tau_1), \psi(E)) = \forall\beta_1 \ldots \beta_n . \psi(\tau_1) = \varphi(\forall\alpha_1 \ldots \alpha_n . \tau_1) = \varphi(\texttt{Gen}(\tau_1, E)),$$

by definition of substitution over type schemes. And we recall that $\psi(E) = \varphi(E)$, since the α_i are not free in E. The two derivations obtained by induction therefore allow to conclude, by the `let` rule:

$$\varphi(E) \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \varphi(\tau_2).$$

That's the expected result. □

Proposition 3.3 *Let a be an expression, τ be a type, and E, E' be two typing environments such that* $\mathrm{Dom}(E) = \mathrm{Dom}(E')$, *and $E'(x) \geq E(x)$ for all x free in a — that is, all instances of $E(x)$ are also instances of $E'(x)$ for all x free in a. If $E \vdash a : \tau$, then $E' \vdash a : \tau$.*

Proof: same proof as for proposition 1.3. □

3.3 Type soundness

In this section, we show that the type system presented above is sound with respect to the three semantics given in chapter 2: the one with references, the one with channels, and the one with continuations. The three proofs follow the same approach as the soundness proof in chapter 1. However, for each proof, we have to adapt the semantic typing relations to the new language objects; then show that it is semantically correct to generalize over a non-dangerous variable; finally, prove by induction on the evaluation derivation a soundness property that is not quite the same as the one in chapter 1.

3.3.1 References

To take into account the value sharing (aliasing) introduced by the references, we need a new semantic tool: the STORE TYPINGS. A store typing, written S, associates a type to each active memory location.

$$\text{Store typing:} \quad S \quad ::= \quad [\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n]$$

The goal of store typings is to ensure that all references to a memory location ℓ have the same monomorphic type $S(\ell)$ `ref`, thus preventing any inconsistent use of the address ℓ [21, 92, 93]. The store typing appears as an extra parameter to the semantic typing relation, which become:

$S \models v : \tau$	the value v, considered in a store of type S, belongs to the type τ
$S \models v : \sigma$	the value v, considered in a store of type S, belongs to the type scheme σ
$S \models e : E$	the values contained in the evaluation environment e, considered in a store of type S, belong to the corresponding type schemes in E
$\models s : S$	the store s has type S.

These relations are defined as follows:

- $S \models cst$: `unit` if cst is `()`
- $S \models cst$: `int` if cst is an integer
- $S \models cst$: `bool` if cst is `true` or `false`
- $S \models (v_1, v_2) : \tau_1 \times \tau_2$ if $S \models v_1 : \tau_1$ and $S \models v_2 : \tau_2$
- $S \models \ell : \tau$ `ref` if $\ell \in \mathrm{Dom}(S)$ and $\tau = S(\ell)$
- $S \models (f, x, a, e) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ if there exists a typing environment E such that

$$S \models e : E \quad \text{and} \quad E \vdash (f \ \texttt{where}\ f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$$

- $S \models v : \forall\alpha_1 \ldots \alpha_n.\,\tau$ if none of the α_i belongs to $\mathcal{D}(\tau)$, and if $S \models v : \varphi(\tau)$ for all substitutions φ whose domain is a subset of $\{\alpha_1 \ldots \alpha_n\}$
- $S \models e : E$ if $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, and for all $x \in \mathrm{Dom}(E)$, we have $S \models e(x) : E(x)$
- $\models s : S$ if $\mathrm{Dom}(s) = \mathrm{Dom}(S)$, and for all $\ell \in \mathrm{Dom}(s)$, we have $S \models s(\ell) : S(\ell)$.

Remark. In the case of functional values, we can further assume that $\mathrm{Dom}(E)$ is identical to $\mathcal{I}(f \texttt{ where } f(x) = a)$. That's because $E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \prec\langle\pi\rangle\rightarrow \tau_2$ implies $\mathrm{Dom}(E) \supseteq \mathcal{I}(f \texttt{ where } f(x) = a)$, given the structure of the typing rules. By proposition 3.3, we can therefore replace E by the restriction of E to the identifiers free in $(f \texttt{ where } f(x) = a)$. □

Context. The introduction of a store typing to parameterize $\models$ follows Tofte's approach [92, 93]. The main difference with Tofte's approach is the replacement of Tofte's quaternary relation $s : S \models v : \tau$ by the conjunction of two simpler relations, $S \models v : \tau$ and $\models s : S$. In other terms, to check that v belongs to the type τ in the store s under the store typing S, I first show that v belongs to τ assuming that the values contained in the locations $\ell_1, \ldots, \ell_n$ reachable from v belong to the corresponding types $S(\ell_1), \ldots, S(\ell_n)$. Then, I show that this assumption holds, by checking that $S \models s(\ell) : S(\ell)$ for all locations ℓ.

Tofte's quaternary relation is more synthetic than my conjunction of relations, but considerably harder to handle. It is defined exactly like my relation $S \models v : \tau$, except for the case where v is a location ℓ. In this case, Tofte defines $s : S \models \ell : \tau$ `ref` if $S(\ell) = \tau$ and $s : S \models s(\ell) : \tau$. Here lies the difficulty: the value $s(\ell)$ can be arbitrarily large, hence the definition of the quaternary predicate is not well-founded by induction over v. Actually, there are cases where the store can contain cycles, as in the following example:

```
let r =
  ref(λn.n + 1) in
let fact =
  λn. if n = 0 then 1 else n × !r(n − 1) in
r := fact;  a
```

At the time a is evaluated, the location ℓ to which `r` is bound contains the closure for `fact`, and the environment part of this closure contains the same location ℓ.

This leads Tofte to treat the pseudo-inductive definition of the relation $s : S \models v : \tau$ as a fixpoint equation, and take for $\models$ the greatest solution. Tofte shows that the smallest fixpoint does not work as expected: in the example above, the value of `fact` does not semantically belong to the type `int` $\rightarrow$ `int` if $\models$ is taken to be the smallest fixpoint. The greatest fixpoint turns out to possess all the required properties. Unfortunately, the usual proof techniques by induction do not apply to relations defined by greatest fixpoints; all proofs must be carried by co-induction [62, 92, 93].

My soundness proof does not require all these complications: it works perfectly by replacing the predicate $s : S \models v : \tau$ by the conjunction of $S \models v : \tau$ and $\models s : S$. (This conjunction is stronger than the quaternary predicate: it requires all memory location to contain values of the expected types, while the condition $s : S \models v : \tau$ constraints only those locations reachable from v.) The definition of $S \models v : \tau$ by induction over v is well-founded, since the case where v is a memory location is a base case. □

We say that a store typing S' EXTENDS another store typing S if $\mathrm{Dom}(S) \subseteq \mathrm{Dom}(S')$, and $S(\ell) = S'(\ell)$ for all $\ell \in \mathrm{Dom}(S)$. This notion captures one aspect of program execution: more and more memory locations are allocated, but a given location is always considered with the same type all along. We can therefore build an increasing sequence of store typings (for the extension ordering) that parallels the evaluation steps. Then, a semantic typing relation such as $S \models v : \tau$ that holds at some point in the evaluation remains true afterwards.

Proposition 3.4 *If S' extends S, then $S \models v : \tau$ implies $S' \models v : \tau$. Similarly, $S \models e : E$ implies $S' \models e : E$.*

Proof: easy induction over v. □

The following proposition is the key lemma for the soundness proof. It shows that the notion of dangerous variable correctly plays its role: it is semantically correct to generalize variables that are not dangerous.

Proposition 3.5 *Let v be a value, τ be a type and S be a store typing such that $S \models v : \tau$. Let $\alpha_1, \ldots, \alpha_n$ be type variables such that $\alpha_i \notin \mathcal{D}(\tau)$ for all i. For all substitutions φ whose domain is included in $\{\alpha_1 \ldots \alpha_n\}$, we have $S \models v : \varphi(\tau)$. As a consequence, $S \models v : \forall \alpha_1 \ldots \alpha_n . \tau$.*

Proof: by structural induction over v.

• **Case $v = cst$.** Straightforward, since τ is a closed type.

• **Case $v = (v_1, v_2)$ and $\tau = \tau_1 \times \tau_2$.** Since $\mathcal{D}(\tau_1 \times \tau_2) = \mathcal{D}(\tau_1) \cup \mathcal{D}(\tau_2)$, we have $\alpha_i \notin \mathcal{D}(\tau_1)$ and $\alpha_i \notin \mathcal{D}(\tau_2)$ for all i. By induction hypothesis, it follows that $S \models v_1 : \varphi(\tau_1)$ and $S \models v_2 : \varphi(\tau_2)$. Hence the result.

• **Case $v = \ell$ and $\tau = \tau_1$ `ref`.** Then, $\mathcal{D}(\tau) = \mathcal{F}(\tau_1)$. Since none of the α_i is dangerous in τ, it follows that none of the α_i is free in τ_1. Hence $\varphi(\tau_1) = \tau_1 = S(\ell)$, and the expected result.

• **Case $v = (f, x, a, e)$ and $\tau = \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$.** Let E be a typing environment such that:

$$S \models e : E \quad \text{and} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2.$$

We also assume that $\mathrm{Dom}(E) = \mathcal{I}(f \texttt{ where } f(x) = a)$, as justified by the remark above. We are going to show that:

$$S \models e : \varphi(E) \quad \text{and} \quad \varphi(E) \vdash (f \texttt{ where } f(x) = a) : \varphi(\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2).$$

The rightmost property follows from the fact that typing is stable under substitution (proposition 3.2). Let us show the leftmost property. Let y be an identifier from the domain of E. We must show $S \models e(y) : \varphi(E(y))$. Write $E(y)$ as $\forall \beta_1 \ldots \beta_k . \tau'$, with the β_i taken out of reach for φ, and distinct from the α_i. We therefore have $\varphi(E)(y) = \forall \beta_1 \ldots \beta_k . \varphi(\tau')$. We must show $S \models e(y) : \psi(\varphi(\tau'))$ for all substitutions ψ whose domain is a subset of $\{\beta_1, \ldots, \beta_k\}$. Let ψ be such a substitution. By definition of $\models$ over type schemes, we have $\models e(y) : \tau'$, and none of the β_i are dangerous in τ'.

Consider the substitution $\psi \circ \varphi$. We have $\mathrm{Dom}(\psi \circ \varphi) \subseteq \{\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_k\}$. None of these variables are dangerous in τ'. The β_i, by definition of $\models$ over type schemes. The α_i, because $\mathcal{D}(\tau) \setminus \{\beta_1 \ldots \beta_k\} = \mathcal{D}(E(y))$ and the α_i are not dangerous in $E(y)$. That's because y is free in $(f \texttt{ where } f(x) = a)$, hence, according to the typing rules for function, $E(y)$ must appear in the closure type π. Therefore, $\mathcal{D}(E(y)) \subseteq \mathcal{D}(\pi) = \mathcal{D}(\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2)$. It follows that none of the α_i are dangerous in $E(y)$.

We can therefore apply the induction hypothesis to the value $e(y)$, the type τ', the variables $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_k$, and the substitution $\psi \circ \varphi$. We get $\models e(y) : \psi(\varphi(\tau'))$. This holds for all substitutions ψ over the β_i. Moreover, the β_i are out of reach for φ; hence none of the β_i are dangerous in $\varphi(\tau')$, since they are not dangerous in τ'. We conclude that $\models e(y) : \forall\beta_1 \ldots \beta_k.\, \varphi(\tau')$, that is, $\models e(y) : \varphi(E)(y)$. This holds for all y. Hence $S \models e : \varphi(E)$, and finally $S \models (f, x, a, e) : \varphi(\tau)$. □

We are now going to show a strong soundness property for the calculus with references, similar to proposition 1.6 for the purely applicative calculus.

Proposition 3.6 (Strong soundness for references) *Let a be an expression, τ be a type, E be a typing environment, e be an evaluation environment, s be a store, S be a store typing such that:*

$$E \vdash a : \tau \quad \textit{and} \quad S \models e : E \quad \textit{and} \quad \models s : S.$$

If there exists a result r such that $e \vdash a/s \Rightarrow r$, then $r \neq \texttt{err}$; instead, r is equal to v/s' for some v and some s', and there exists a store typing S' such that:

$$S' \textit{ extends } S \quad \textit{and} \quad S' \models v : \tau \quad \textit{and} \quad \models s' : S'.$$

Proof: the proof is an inductive argument on the size of the evaluation derivation. We argue by case analysis on a, and therefore on the last rule used in the typing derivation. I show all cases for the sake of honesty; the only new case is the one for `let`, but it is straightforward once proposition 3.5 has been established.

- **Constants.**

$$\frac{\tau \leq \texttt{TypCst}(cst)}{E \vdash cst : \tau}$$

The only possible evaluation is $e \vdash cst/s \Rightarrow cst/s$. We check $S \models cst : \texttt{TypCst}(cst)$. We conclude with $S' = S$.

- **Variables.**

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

From hypothesis $S \models e : E$ it follows that $x \in \mathrm{Dom}(e)$ and $S \models e(x) : E(x)$. The only possible evaluation is $e \vdash x/s \Rightarrow e(x)/s$. By definition of $\models$ over type schemes, $S \models e(x) : E(x)$ implies $S \models e(x) : \tau$. This is the expected result, taking $S' = S$.

• **Functions.**

$$\frac{E + f \mapsto (\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2}$$

The only possible evaluation is $e \vdash (f \texttt{ where } f(x) = a)/s \Rightarrow (f, x, a, e)/s$. We have $S \models (f, x, a, e) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ be definition of $\models$, taking E for the required typing environment. We conclude with $S' = S$.

• **Function application.**

$$\frac{E \vdash a_1 : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

There are three evaluation possibilities. The first one leads to $r = \texttt{err}$ because $e \vdash a_1 \Rightarrow r_1$ and r_1 is not $(f, x, a_0, e_0)/s$; but this contradicts the induction hypothesis applied to a_1, which says $r_1 = v_1/s_1$ and $\models v_1 : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1$, hence v_1 is a closure. The second evaluation possibility concludes $r = \texttt{err}$ from $e \vdash a_2 \Rightarrow \texttt{err}$; it similarly contradicts the induction hypothesis applied to a_2. Hence the evaluation derivation must end up with:

$$\frac{e \vdash a_1/s \Rightarrow (f, x, a_0, e_0)/s_1 \qquad e \vdash a_2/s_1 \Rightarrow v_2/s_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0/s_2 \Rightarrow r}{e \vdash a_1(a_2)/s \Rightarrow r}$$

By induction hypothesis applied to a_1, we get a store typing S_1 such that:

$$S_1 \models (f, x, a, e) : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1 \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S.$$

Hence there exists E_0 such that $S_1 \models e_0 : E_0$, and $E_0 \vdash (f \texttt{ where } f(x) = a_0) : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1$. There is only one typing rule that concludes the latter result; its premise must therefore hold:

$$E_0 + f \mapsto (\tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1) + x \mapsto \tau_2 \vdash a_0 : \tau_1.$$

Applying the induction hypothesis to a_2, we get S_2 such that

$$S_2 \models v_2 : \tau_2 \quad \text{and} \quad \models s_2 : S_2 \quad \text{and} \quad S_2 \text{ extends } S_1.$$

Consider the environments:

$$e_2 = e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \qquad\qquad E_2 = E_0 + f \mapsto (\tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1) + x \mapsto \tau_1$$

We have shown that $S_2 \models e_2 : E_2$. Hence we can apply the induction hypothesis to the expression a_0, in the environments e_2 and E_2, and the store $s_2 : S_2$. It follows that r is equal to v/s', with, for some S',

$$S' \models v : \tau_1 \quad \text{and} \quad \models s' : S' \quad \text{and} \quad S' \text{ extends } S_2.$$

That's the expected result, since a fortiori S' extends S.

• **The `let` binding.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

There are two possible evaluations. The first one corresponds to $e \vdash a_1 \Rightarrow \texttt{err}$. It contradicts the induction hypothesis applied to a_1. Hence the last step in the evaluation is:

$$\frac{e \vdash a_1/s \Rightarrow v_1/s_1 \qquad e + x \mapsto v_1 \vdash a_2/s_1 \Rightarrow r}{e \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2)/s \Rightarrow r}$$

By induction hypothesis applied to a_1, we get S_1 such that:

$$S_1 \models v_1 : \tau_1 \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S.$$

By proposition 3.5, we have $S_1 \models v_1 : \texttt{Gen}(\tau_1, E)$. That's because the `Gen` operator does not generalize any dangerous variable in τ_1. Writing

$$e_1 = e + x \mapsto v_1 \qquad\qquad E_1 = E + x \mapsto \texttt{Gen}(\tau_1, E),$$

we therefore have $S_1 \models e_1 : E_1$. Applying the induction hypothesis to a_2, e_1, E_1, s_1, S_1, it follows that r is equal to v_2/s_2, and there exists S_2 such that

$$S_2 \models v_2 : \tau_2 \quad \text{and} \quad \models s_2 : S_2 \quad \text{and} \quad S_2 \text{ extends } S_1.$$

That's the expected result.

- **Pair construction.** Same argument as for application.

- **Reference creation.**

$$\frac{\tau \multimap\langle\pi\rangle\mapsto \tau\ \texttt{ref} \leq \forall\alpha, u.\, \alpha \multimap\langle u\rangle\mapsto \alpha\ \texttt{ref} \qquad E \vdash a : \tau}{E \vdash \texttt{ref}(a) : \tau\ \texttt{ref}}$$

The evaluation must end up with:

$$\frac{e \vdash a/s \Rightarrow v/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash \texttt{ref}(a)/s \Rightarrow \ell/(s_1 + \ell \mapsto v)}$$

(The evaluation that conclude $r = \texttt{err}$ because a evaluates to `err` contradicts the induction hypothesis.) By induction hypothesis applied to a, we get S_1 such that

$$S_1 \models v : \tau \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S.$$

Take $S' = S_1 + \ell \mapsto \tau$. Since $\mathrm{Dom}(s_1) = \mathrm{Dom}(S_1)$, we have $\ell \notin \mathrm{Dom}(S_1)$, hence S' extends S_1, and also S. We therefore have $S' \models v : \tau$, hence $\models s_1 + \ell \mapsto v : S'$ and $S' \models \ell : \tau\ \texttt{ref}$, which is the expected result.

- **Dereferencing.**

$$\frac{\tau\ \texttt{ref} \multimap\langle\pi\rangle\mapsto \tau \leq \forall\alpha, u.\, \alpha\ \texttt{ref} \multimap\langle u\rangle\mapsto \alpha \qquad E \vdash a : \tau\ \texttt{ref}}{E \vdash\ !a : \tau}$$

There are three evaluation possibilities. The first one leads to `err` because a evaluates to an answer that is not ℓ/s'. It contradicts the induction hypothesis applied to a. The second possibility ends up with:

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash\, !a/s_0 \Rightarrow \mathtt{err}}$$

By induction hypothesis applied to a, we get S_1 such that $S_1 \models \ell : \tau$ `ref`, implying $l \in \mathrm{Dom}(S_1)$, and such that $\models s_1 : S_1$, implying $\mathrm{Dom}(s_1) = \mathrm{Dom}(S_1)$. Hence $\ell \in \mathrm{Dom}(s_1)$, and a contradiction. Therefore, only the third possibility remains:

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash\, !a/s_0 \Rightarrow s_1(\ell)/s_1}$$

By induction hypothesis applied to a, we get S_1 such that

$$S_1 \models \ell : \tau\ \mathtt{ref} \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S.$$

As a consequence, $S_1(\ell) = \tau$, hence $S_1 \models s_1(\ell) : \tau$, and the expected result follows with $S' = S_1$.

- **Assignment.**

$$\frac{\tau\ \mathtt{ref} \times \tau \multimap\langle\pi\rangle\rightarrow \mathtt{unit} \leq \forall\alpha, u.\ \alpha\ \mathtt{ref} \times \alpha \multimap\langle u\rangle\rightarrow \mathtt{unit} \qquad E \vdash a : \tau\ \mathtt{ref} \times \tau}{E \vdash :=(a) : \mathtt{unit}}$$

As in the previous case, the evaluation must end up with:

$$\frac{e \vdash a/s_0 \Rightarrow (\ell, v)/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash :=(a)/s_0 \Rightarrow \mathtt{()}/(s_1 + \ell \mapsto v)}$$

By induction hypothesis applied to a, we get S_1 such that:

$$S_1 \models (\ell, v) : \tau\ \mathtt{ref} \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S.$$

This implies $S_1 \models v : \tau$ and $S_1(\ell) = \tau$. Hence $\models s_1 + \ell \mapsto v : S_1$, and obviously $S_1 \models \mathtt{()} : \mathtt{unit}$. The result follows with $S' = S_1$. □

3.3.2 Communication channels

The soundness proof for channels is very close to the one for references.

We introduce the notion of channel typing: a CHANNEL TYPING, written Γ, assigns types to channel identifiers c.

$$\text{Channel typing:} \quad \Gamma \ ::= \ [c_1 \mapsto \tau_1, \ldots, c_n \mapsto \tau_n]$$

We make use of the following semantic typing relations:

$\Gamma \models v : \tau$	the value v belongs to the type τ
$\Gamma \models v : \sigma$	the value v belongs to the type scheme σ
$\Gamma \models e : E$	the values contained in the evaluation environment e belong to the corresponding type schemes in E
$\models w :? \Gamma$	the reception events $(c ? v)$ contained in the event sequent w respect the channel typing Γ
$\models w :! \Gamma$	the emission events $(c ! v)$ contained in the event sequent w respect the channel typing Γ

These relations are defined by:

- $\Gamma \models cst : \texttt{unit}$ if cst is `()`
- $\Gamma \models cst : \texttt{int}$ if cst is an integer
- $\Gamma \models cst : \texttt{bool}$ if cst is `true` or `false`
- $\Gamma \models (v_1, v_2) : \tau_1 \times \tau_2$ if $\Gamma \models v_1 : \tau_1$ and $\Gamma \models v_2 : \tau_2$
- $\Gamma \models c : \tau\ \texttt{chan}$ if $c \in \mathrm{Dom}(\Gamma)$ and $\tau = \Gamma(c)$
- $\Gamma \models (f, x, a, e) : \tau_1 \multimap\!\langle\pi\rangle\!\rightarrow \tau_2$ if there exists a typing environment E such that:

$$\Gamma \models e : E \quad \text{and} \quad E \vdash (f\ \ \texttt{where}\ \ f(x) = a) : \tau_1 \multimap\!\langle\pi\rangle\!\rightarrow \tau_2$$

- $\Gamma \models v : \forall\alpha_1 \ldots \alpha_n.\, \tau$ if none of the α_i belongs to $\mathcal{D}(\tau)$, and if $\Gamma \models v : \varphi(\tau)$ for all substitutions φ over $\{\alpha_1, \ldots, \alpha_n\}$
- $\Gamma \models e : E$ si $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, and for all $x \in \mathrm{Dom}(E)$, we have $\Gamma \models e(x) : E(x)$
- $\models w :? \Gamma$ if $\Gamma \models v : \Gamma(c)$ for all reception event $c ? v$ belonging to the sequence w
- $\models w :! \Gamma$ if $\Gamma \models v : \Gamma(c)$ for all emission event $c ! v$ belonging to the sequence w.

Notice that if w is the concatenation $w_1 \ldots w_n$, we have $\models w :! \Gamma$ if and only if $\models w_i :! \Gamma$ for all i. The same holds if :! is replaced by :? in this property.

As in the case of references, the dangerous variables in a type τ have a simple semantic interpretation: the dangerous variables are those variables that can be free in the type of a channel identifier reachable from a value of type τ. It follows that it is semantically correct to generalize over non-dangerous variables.

Proposition 3.7 *Let v be a value, τ be a type and Γ be a channel typing such that $\Gamma \models v : \tau$. Let $\alpha_1 \ldots \alpha_n$ be type variables such that $\alpha_i \notin \mathcal{D}(\tau)$ for all i. For all substitutions φ over $\{\alpha_1 \ldots \alpha_n\}$, we have $\Gamma \models v : \varphi(\tau)$. As a consequence, $\Gamma \models v : \forall\alpha_1 \ldots \alpha_n.\, \tau$.*

Proof: same proof as for proposition 3.5. □

Proposition 3.8 (Weak soundness for channels) *Let a_0 be an expression and τ_0 be a type such that $[\,] \vdash a_0 : \tau_0$. Let r_0 be a result such that $[\,] \vdash a_0 \stackrel{\varepsilon}{\Rightarrow} r_0$. Then, $r_0 \neq \texttt{err}$.*

We have considered the evaluation of a complete program here, in order to define a channel typing Γ global to the whole program evaluation. (Incrementally constructing Γ at each step of the soundness proof, as we did for S in the proof for reference, does not work well in the presence of parallel evaluations.) Informally, we are going to construct Γ as follows: for each occurrence of the `newchan` rule in the evaluation of a_0,

$$\frac{c \text{ is unallocated elsewhere in the derivation}}{e \vdash \texttt{newchan}(a) \stackrel{\varepsilon}{\Rightarrow} c}$$

we take $\Gamma(c)$ equal to the type τ such that τ `chan` is the type assigned to the expression $\texttt{newchan}(a)$ in the typing of a_0. This construction is not precise enough, since the same expression $\texttt{newchan}(a)$ can occur several times in the program a_0, with different types.

To make this construction precise enough, we shall take advantage of the fact that the argument a of $\texttt{newchan}(a)$ is any term with type `unit`, which is not evaluated. We assume given a countable family of constants O_i, for all integers i, with type `unit`:

$$\begin{array}{rcl} \texttt{Cst} & ::= & \ldots \mid O_1 \mid O_2 \mid \ldots \\ \texttt{TypCst}(O_i) & = & \texttt{unit} \end{array}$$

Let a_0 be the closed expression in the claim 3.8. We construct an expression a_0' by replacing in a_0 all subterms $\texttt{newchan}(a)$ by $\texttt{newchan}(O_i)$, where i is chosen so that O_i appears only once in a_0. It is easy to check that $[\,] \vdash a_0 : \tau_0$ implies $[\,] \vdash a_0' : \tau_0$. (That's because the constants O_i do belong to the type `unit` required for the arguments of `newchan`.) Similarly, $[\,] \vdash a_0 \stackrel{\varepsilon}{\Rightarrow} r_0$ implies $[\,] \vdash a_0 \stackrel{\varepsilon}{\Rightarrow} r_0'$ for some result r_0' equal to r_0 modulo the replacement inside closures of subterms $\texttt{newchan}(a)$ by $\texttt{newchan}(a')$. (That's because the argument a in $\texttt{newchan}(a)$ is never evaluated, and can therefore be replaced by O_i without changing the structure of the evaluation derivation.) In particular, if we show that r_0' cannot be `err`, then it follows that r_0 cannot be `err`.

We have therefore reduced proposition 3.8 to the case where all channel creations appearing in a_0 are of the form $\texttt{newchan}(O_i)$, with, for any i, O_i appearing at most once in a_0. In the remainder of this section, we fix a derivation $\mathcal{E}$ of the evaluation $[\,] \vdash a_0 \stackrel{\varepsilon}{\Rightarrow} r_0$, and a derivation $\mathcal{T}$ of the typing $[\,] \vdash a_0 : \tau_0$.

Given the typing rules, any occurrence of a subterm a of a_0 is given one and exactly one type in the typing derivation $\mathcal{T}$. As a consequence, for all subterms $\texttt{newchan}(O_i)$ of a, the derivation $\mathcal{T}$ contains one and exactly one sub-derivation that concludes $E_i \vdash \texttt{newchan}(O_i) : \tau_i$ `chan`, for some type τ_i and some environment E_i. We then define Γ as the least defined channel typing satisfying the following condition. Consider all occurrences of the evaluation rule for `newchan` in the derivation $\mathcal{E}$:

$$\frac{c \text{ is unallocated elsewhere in } \mathcal{E}}{e \vdash \texttt{newchan}(O_i) \stackrel{\varepsilon}{\Rightarrow} c}$$

For this channel c, we take $\Gamma(c)$ equals to the type τ_i such that τ_i `chan` is the type assigned to $\texttt{newchan}(O_i)$ in the typing derivation $\mathcal{T}$. As shown above, this type τ_i is unique. Moreover, two occurrences of the evaluation rule for `newchan` cannot share the same channel identifier c. Hence

the condition above defines a mapping Γ from channel identifiers to term types. Moreover, this mapping Γ is such that if

$$E \vdash \texttt{newchan}(O_i) : \tau \ \texttt{chan} \quad \text{and} \quad e \vdash \texttt{newchan}(O_i) \stackrel{\varepsilon}{\Longrightarrow} c$$

are conclusions of sub-derivations of $\mathcal{T}$ and $\mathcal{E}$ respectively, then $\tau = \Gamma(c)$.

The strong soundness claim that we are now going to prove by induction is more complex than the one for references: the conclusions describe not only the value to which an expression evaluates, but also the values emitted over channels during evaluation; symmetrically, the assumptions revolve not only about the evaluation environment, but also about the values received on channels during evaluation.

Proposition 3.9 (Strong soundness for channels) *Let $e \vdash a \stackrel{w}{\Longrightarrow} r$ be the conclusion of a sub-derivation of $\mathcal{E}$, and $E \vdash a : \tau$ be the conclusion of a sub-derivation of $\mathcal{T}$, for the same expression a. Assume $\Gamma \models e : E$.*

1. *If $\models w :? \Gamma$, then $r \neq \texttt{err}$. Instead, r is a value v, such that $\Gamma \models v : \tau$. Moreover, $\models w :! \Gamma$.*
2. *If $w = w'.c\,!\,v.w''$ and $\models w' :? \Gamma$, then $\Gamma \models v : \Gamma(c)$.*

The property (2) expresses the fact that if the evaluation of a reaches the point where it emits a value over a channel, then this value correctly belongs to the type associated with the channel — even if the evaluation of a causes a run-time violation later, in which case the final result is `err` and (1) does not hold. The property (2) is crucial to show the soundness of the parallel composition.

Proof: the proof is by induction over the size of the evaluation sub-derivation. We argue by case analysis on a, hence on the last rule used in the typing sub-derivation. For the sequential constructs, (1) is proved as in the proof of proposition 3.6; hence I only show the proof of (2). I give the full proofs for the parallelism and communication constructs.

• **Constants, variables, or functions.** (1) is omitted. (2) is obviously true, since w can only be ε.

• **Function application.** (1) is omitted. For (2), we consider all evaluation possibilities for $a_1(a_2)$, and all decompositions of the event sequence $w'.c\,!\,v.w''$ into one, two or three sequences. We always write w'_1, w'_2, w'_3 for event sequences that are well-typed for reception events: that is, we know that $\models w'_i :? \Gamma$ as a consequence of the hypothesis $\models w' :? \Gamma$. We write w''_1, w''_2, w''_3 for event sequences for which we do not know anything.

$$\frac{e \vdash a_1 \stackrel{w'_1.c!v.w''_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w''_2}{\Longrightarrow} v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \stackrel{w''_3}{\Longrightarrow} r_0}{e \vdash a_1(a_2) \stackrel{w'_1.c!v.w''_1.w''_2.w''_3}{\Longrightarrow} r_0}$$

$$\frac{e \vdash a_1 \stackrel{w'_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w'_2.c!v.w''_2}{\Longrightarrow} v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \stackrel{w''_3}{\Longrightarrow} r_0}{e \vdash a_1(a_2) \stackrel{w'_1.w'_2.c!v.w''_2.w''_3}{\Longrightarrow} r_0}$$

$$\frac{e \vdash a_1 \stackrel{w'_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w'_2}{\Longrightarrow} v_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0 \stackrel{w'_3.c!v.w''_3}{\Longrightarrow} r_0}{e \vdash a_1(a_2) \stackrel{w'_1.w'_2.w'_3.c!v.w''_3}{\Longrightarrow} r_0}$$

$$\frac{e \vdash a_1 \stackrel{w'.c!v.w''}{\Longrightarrow} r_1 \qquad r_1 \text{ does not match } (f, x, a_0, e_0)}{e \vdash a_1(a_2) \stackrel{w'.c!v.w''}{\Longrightarrow} \texttt{err}}$$

$$\frac{e \vdash a_1 \stackrel{w'_1.c!v.w''_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w''_2}{\Longrightarrow} \texttt{err}}{e \vdash a_1(a_2) \stackrel{w'_1.c!v.w''_1.w''_2}{\Longrightarrow} \texttt{err}}$$

$$\frac{e \vdash a_1 \stackrel{w'_1}{\Longrightarrow} (f, x, a_0, e_0) \qquad e \vdash a_2 \stackrel{w'_2.c!v.w''_2}{\Longrightarrow} \texttt{err}}{e \vdash a_1(a_2) \stackrel{w'_1.w'_2.c!v.w''_2}{\Longrightarrow} \texttt{err}}$$

In all cases, we have a sub-evaluation $e \vdash a_i \stackrel{w'_k.c!v.w''_k}{\Longrightarrow} r_i$, for some i and k, to which we can apply the induction hypothesis (2). It follows that $\Gamma \models v : \Gamma(c)$, which is the expected result.

• **The `let` binding.** (1) follows from proposition 3.7. (2) is obvious once we have enumerated all evaluation possibilities, as in the case of function application.

• **Pair construction.** (1) is omitted. (2) follows from the enumeration of all evaluation possibilities.

• **Channel creation.** We first show (1).

$$\frac{\texttt{unit} \multimap\!\langle\pi\rangle\!\rightarrow \tau\ \texttt{chan} \leq \forall\alpha, u.\, \texttt{unit} \multimap\!\langle u\rangle\!\rightarrow \alpha\ \texttt{chan} \qquad E \vdash O_i : \texttt{unit}}{E \vdash \texttt{newchan}(O_i) : \tau\ \texttt{chan}}$$

The only possible evaluation is $e \vdash \texttt{newchan}(O_i) \stackrel{\varepsilon}{\Longrightarrow} c$, for some channel c. By construction of Γ from the derivations $\mathcal{D}$ et $\mathcal{T}$, we have $\Gamma(c) = \tau$. Hence (1).

(2) is obvious, since $w = \varepsilon$.

• **Reception over a channel.**

$$\frac{\tau\ \texttt{chan} \multimap\!\langle\pi\rangle\!\rightarrow \tau \leq \forall\alpha, u.\, \alpha\ \texttt{chan} \multimap\!\langle u\rangle\!\rightarrow \alpha \qquad E \vdash a : \tau\ \texttt{chan}}{E \vdash a? : \tau}$$

We first show (1). There are two possible evaluations. The first one ends up with:

$$\frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} r \qquad r \text{ does not match } c}{e \vdash a_1? \stackrel{w}{\Longrightarrow} \texttt{err}}$$

It contradicts induction hypothesis (1) applied to a, since if $e \vdash a_1 \stackrel{w}{\Longrightarrow} r$, then $\Gamma \models r : \tau\ \mathtt{chan}$. Hence the evaluation derivation can only end up with:

$$\frac{e \vdash a_1 \stackrel{w}{\Longrightarrow} c}{e \vdash a_1? \stackrel{w.(c?v)}{\Longrightarrow} v}$$

Since $\models w.(c\,?\,v) :? \Gamma$ by hypothesis, we have $\Gamma \models v : \Gamma(c)$. And since $\Gamma \models c : \tau\ \mathtt{chan}$, we have $\tau = \Gamma(c)$. Hence $\Gamma \models v : \tau$. The second result, $\models w.(c\,?\,v) :! \Gamma$, immediately follows from $\models w :! \Gamma$, as obtained by induction hypothesis (1) applied to a_1.

Property (2) comes by examination of the two evaluation possibilities.

• **Emission over a channel.**

$$\frac{\tau\ \mathtt{chan} \times \tau \multimap\langle\pi\rangle\mapsto \mathtt{unit} \leq \forall\alpha, u.\, \alpha\ \mathtt{chan} \times \alpha \multimap\langle u\rangle\mapsto \mathtt{unit} \qquad E \vdash a : \tau\ \mathtt{chan} \times \tau}{E \vdash !(a) : \mathtt{unit}}$$

We first show (1). The first evaluation possibility ends up with:

$$\frac{e \vdash a \stackrel{w}{\Longrightarrow} r \qquad r \text{ does not match } (c, v)}{e \vdash !(a) \stackrel{w}{\Longrightarrow} \mathtt{err}}$$

It contradicts induction hypothesis (1) applied to a, which implies$\Gamma \models r : \tau\ \mathtt{chan} \times \tau$. Hence the evaluation derivation ends up with:

$$\frac{e \vdash a \stackrel{w'}{\Longrightarrow} (c, v)}{e \vdash !(a) \stackrel{w'.(c!v)}{\Longrightarrow} \mathtt{unit}}$$

We have $\Gamma \models \mathtt{()} : \mathtt{unit}$. By (1) applied to a, we have $\Gamma \models (c, v) : \tau\ \mathtt{chan} \times \tau$ and $\models w' :! \Gamma$. Hence $\Gamma(c) = \tau$ and $\Gamma \models v : \tau$. It follows that $\models w'.(c\,!\,v) :! \Gamma$. Hence (1) for $!a$.

We now show (2). If the evaluation ends up with

$$\frac{e \vdash a \stackrel{w}{\Longrightarrow} r \qquad r \text{ does not match } (c, v)}{e \vdash !(a) \stackrel{w}{\Longrightarrow} \mathtt{err}}$$

then property (2) follows from induction hypothesis (2) applied to a. If the evaluation ends up with:

$$\frac{e \vdash a \stackrel{w}{\Longrightarrow} (c, v)}{e \vdash !(a) \stackrel{w.(c!v)}{\Longrightarrow} \mathtt{unit}}$$

we have to consider two cases, depending on how $w.(c\,!\,v)$ is decomposed into $w'.(c'\,!\,v').w''$. Either $w'' \neq \varepsilon$, and then $\Gamma \models v' : \Gamma(c')$ follows by induction hypothesis (2) applied to the evaluation of a. Or, $w'' = \varepsilon$ and $w' = w$ and $c' = c$ and $v' = v$, in which case we do have $\Gamma \models v : \Gamma(c)$, as shown in the proof of (1).

• **Non-deterministic choice.** (1) and (2) obviously follow from the induction hypothesis.

• **Parallel composition.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1 \parallel a_2 : \tau_1 \times \tau_2}$$

We first show (1). The three possible evaluations have the form:

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} r_1 \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} r_2 \qquad \vdash w_1 \parallel w_2 \Rightarrow w}{e \vdash a_1 \parallel a_2 \stackrel{w}{\Longrightarrow} r}$$

We need to check that $\models w_1 :? \Gamma$ and $\models w_2 :? \Gamma$ to apply the induction hypothesis to the evaluations of a_1 and a_2. This is not obvious, since w_1 and w_2 can contain internal events, that do not appear in w. For instance, we can have:

$$w = \varepsilon \qquad w_1 = c\,!\,\texttt{true}.\,\varepsilon \qquad w_2 = c\,?\,\texttt{true}.\,\varepsilon$$

and it is not apparent that $\Gamma(c)$ can only be equal to `bool`. Actually, this is true, but we need to invoke property (2): a well-typed program would never send `true` over a channel that is not a `bool chan`. We formalize this argument in the sub-proposition 3.10 below.

Sub-proposition 3.10 *Let w', w'_1, w'_2 be left prefixes of w, w_1, w_2 respectively. If $\vdash w'_1 \parallel w'_2 \Rightarrow w'$, then $\models w'_1 :? \Gamma$ and $\models w'_2 :? \Gamma$.*

Proof: for sub-proposition 3.10. The proof is by induction over the derivation of $\vdash w'_1 \parallel w'_2 \Rightarrow w'$. The base case $\vdash \varepsilon \parallel \varepsilon \Rightarrow \varepsilon$ is obvious. For the next two cases:

$$\frac{\vdash w'_1 \parallel w'_2 \Rightarrow w'}{\vdash w'_1.evt \parallel w'_2 \Rightarrow w'.evt} \qquad \frac{\vdash w'_1 \parallel w'_2 \Rightarrow w'}{\vdash w'_1 \parallel w'_2.evt \Rightarrow w'.evt}$$

we know by hypothesis that $\models w'.evt :? \Gamma$, hence if evt is a reception event, it is well-typed. Since $\vdash w'_1 :? \Gamma$ by induction hypothesis, we conclude that $\vdash w'_1.evt :? \Gamma$, and similarly for w'_2.

It remains the following two cases:

$$\frac{\vdash w'_1 \parallel w'_2 \Rightarrow w'}{\vdash w'_1.(c\,?\,v) \parallel w'_2.(c\,!\,v) \Rightarrow w'}$$

and the symmetrical case obtained by exchanging ! and ? above. The sequences w'_1 et w'_2 are left prefixes of w_1 and w_2, respectively. Applying the induction hypothesis for sub-proposition 3.10, we get $\models w'_1 :? \Gamma$ and $\models w'_2 :? \Gamma$. We apply the induction hypothesis (2) for proposition 3.9 to the evaluation $e \vdash a_2 \stackrel{w_2}{\Longrightarrow} r_2$ with the decomposition $w_2 = w'_2.(c\,!\,v).w''_2$. We get $\Gamma \models v : \Gamma(c)$. Hence $\models w'_1.(c\,?\,v) :? \Gamma$. The other result, $\models w'_2.(c\,!\,v) :? \Gamma$ obviously follows from $\models w'_2 :? \Gamma$. □

We now conclude the proof of proposition 3.9. By sub-proposition 3.10, we have $\models w_1 :? \Gamma$ and $\models w_2 :? \Gamma$. We can therefore apply the induction hypothesis (1) to a_1 and a_2. We get

$$r_1 \neq \texttt{err} \quad \text{and} \quad \Gamma \models r_1 : \tau_1 \quad \text{and} \quad \models w_1 :? \Gamma$$

$$r_2 \neq \texttt{err} \quad \text{and} \quad \Gamma \models r_2 : \tau_2 \quad \text{and} \quad \models w_2 :? \Gamma$$

The last step in the evaluation is therefore:

$$\frac{e \vdash a_1 \stackrel{w_1}{\Longrightarrow} v_1 \qquad e \vdash a_2 \stackrel{w_2}{\Longrightarrow} v_2 \qquad \vdash w_1 \parallel w_2 \Rightarrow w}{e \vdash a_1 \parallel a_2 \stackrel{w}{\Longrightarrow} (v_1, v_2)}$$

We immediately have $\Gamma \models (v_1, v_2) : \tau_1 \times \tau_2$. Similarly, $\models w :! \Gamma$, since all emission events appearing in w appear in w_1 or in w_2. Hence property (1) for a.

Concerning property (2), if w is decomposed as $w'.c\,!\,v.w''$ with $\models w' :? \Gamma$, then the event $c\,!\,v$ must appear in w_1 or in w_2. Assume that it appears in w_1. Then, w_1 is equal to $w_1'.c\,!\,v.w_1''$, with $\models w_1' :? \Gamma$ by sub-proposition 3.10. Hence, $\Gamma \models v : \Gamma(c)$ by induction hypothesis (2) applied to a_1. □

3.3.3 Continuations

In the case of continuations, the semantic typing relations are very close to those in section 1.4. In particular, no extra argument is required to account for sharing, as in the case of references and channels. We just have to add a relation, $\models k :: \tau$, for the semantic typing of continuation objects.

$\models v : \tau$	the value v belongs to the type τ
$\models v : \sigma$	the value v belongs to the type scheme σ
$\models e : E$	the values contained in the evaluation environment e belong to the corresponding type schemes in E
$\models k :: \tau$	the continuation k accepts all values belonging to type τ

These relations are defined by structural induction on the value part, as follows:

- $\models cst : \texttt{unit}$ if cst is `()`
- $\models cst : \texttt{int}$ if cst is an integer
- $\models cst : \texttt{bool}$ if cst is `true` or `false`
- $\models (v_1, v_2) : \tau_1 \times \tau_2$ if $\models v_1 : \tau_1$ and $\models v_2 : \tau_2$
- $\models k : \tau$ `cont` if $\models k :: \tau$
- $\models (f, x, a, e) : \tau_1 \multimap\!\langle\pi\rangle\!\rightarrow \tau_2$ if there exists a typing environment E such that:

$$\models e : E \quad \text{and} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\!\langle\pi\rangle\!\rightarrow \tau_2$$

- $\models v : \forall\alpha_1 \ldots \alpha_n.\,\tau$ if none of the variables α_i belong to $\mathcal{D}(\tau)$, and if $\models v : \varphi(\tau)$ for all substitutions φ over $\{\alpha_1, \ldots, \alpha_n\}$

- $\models e : E$ if $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, and for all $x \in \mathrm{Dom}(E)$, we have $\models e(x) : E(x)$
- $\models \mathtt{stop} :: \tau$ for all types τ
- $\models \mathtt{app1c}(a, e, k) :: \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ if there exists a typing environment E such that

$$E \vdash a : \tau_1 \quad \text{and} \quad \models e : E \quad \text{and} \quad \models k :: \tau_2$$

- $\models \mathtt{app2c}(f, x, a, e, k) :: \tau$ if there exists a typing environment E, a type τ' and a closure type π such that

$$E \vdash (f \ \mathtt{where}\ f(x) = a) : \tau \multimap\langle\pi\rangle\rightarrow \tau' \quad \text{and} \quad \models e : E \quad \text{and} \quad \models k :: \tau'$$

- $\models \mathtt{letc}(x, a, e, k) :: \tau$ if there exists a typing environment E and a type τ' such that

$$E + x \mapsto \mathtt{Gen}(\tau, E) \vdash a : \tau' \quad \text{and} \quad \models e : E \quad \text{and} \quad \models k :: \tau'$$

- $\models \mathtt{pair1c}(a, e, k) :: \tau$ if there exists a typing environment E and a type τ' such that

$$E \vdash a : \tau' \quad \text{and} \quad \models e : E \quad \text{and} \quad \models k :: \tau \times \tau'$$

- $\models \mathtt{pair2c}(v, k) :: \tau$ if there exists a type τ' such that

$$\models v : \tau' \quad \text{and} \quad \models k :: \tau' \times \tau$$

- $\models \mathtt{primc}(\mathtt{callcc}, k) :: \tau \ \mathtt{cont} \multimap\langle\pi\rangle\rightarrow \tau$ if $\models k :: \tau$, for all π
- $\models \mathtt{primc}(\mathtt{throw}, k) :: \tau \ \mathtt{cont} \times \tau$ for all τ.

Context. A simpler definition for $\models k :: \tau$ would be "k has type τ if, for all values v belonging to the type τ, the continuation k applied to v does not evaluate to `err`". More formally, we would take $\models k :: \tau$ if for all values v such that $\models v : \tau$ and for all results r such that $\vdash v \triangleright k \Rightarrow r$, we have $r \neq \mathtt{err}$. That's the approach taken by Duba, Harper and MacQueen [25]. This definition, although nicer than the definition given above, is unfortunately not well-founded by induction over v, since it quantifies over an arbitrary complex value with type τ. This is not a problem in the setting of Duba, Harper and MacQueen, since they define $\models$ over functional values with the usual continuity condition (see section 1.4.1, first context), hence their definition of $\models$ is well-founded by induction on the type component. But I have to use Tofte's condition ("there exists E such that $E \vdash (f \ \mathtt{where}\ f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ and ..."), and this condition leads to a definition of $\models$ that is not well-founded by induction over the type, since E can generally be more complex than the function type $\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$.[1] Hence the definition of $\models k :: \tau$ by structural induction over k and frequent appeal to the typing predicate given above. □

[1] In the type system of the present chapter, it turns out that the definition of $\models$ using Tofte's condition is also well-founded by induction over the type, thanks to closure typing: the relevant parts of E, that is $E(y)$ for all y free in $(f \ \mathtt{where}\ f(x) = a)$, necessarily appear in the closure type π, hence are subterms of the function type $\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$. However, this property does not hold anymore in the systems in chapters 4 and 6; that's why I don't rely on it here.

Remark. On this definition of the semantic typing predicate, the difficulty with polymorphic continuations is clearly apparent. The predicate $\models k :: \tau$ is not stable by substitutions of type variables inside τ: in the case for `letc`, from the hypothesis $E + x \mapsto \mathtt{Gen}(\tau, E) \vdash a : \tau'$, we cannot deduce in general $\varphi(E) + x \mapsto \mathtt{Gen}(\varphi(\tau), \varphi(E)) \vdash a : \varphi(\tau')$.

Example. We have

$$\models \mathtt{letc}(\mathtt{x}, \mathtt{x}(\mathtt{x}), [\,], \mathtt{stop}) : t \dashv\langle u\rangle\rightarrow t$$

since $\mathtt{x}(\mathtt{x})$ is well-typed under the assumption $\mathtt{x} : \forall t, u.\, t \dashv\langle u\rangle\rightarrow t$, but we don't have

$$\models \mathtt{letc}(\mathtt{x}, \mathtt{x}(\mathtt{x}), [\,], \mathtt{stop}) : \mathtt{int} \dashv\langle u\rangle\rightarrow \mathtt{int},$$

since the self-application is ill-typed under the assumption $\mathtt{x} : \forall u.\, \mathtt{int} \dashv\langle u\rangle\rightarrow \mathtt{int}$. □

That's why it is not semantically correct to generalize over any type variables, unlike in the purely applicative calculus (proposition 1.5). □

The semantic interpretation of the dangerous variables is as follows: the variables dangerous in the type τ of a value v are those variables that can be free in the type of a continuation object reachable from v. As a consequence, it is semantically correct to generalize over non-dangerous variables.

Proposition 3.11 *Let v be a value and τ be a type such that $\models v : \tau$. Let $\alpha_1 \ldots \alpha_n$ be type variables such that $\alpha_i \notin \mathcal{D}(\tau)$ for all i. For all substitutions φ over $\{\alpha_1 \ldots \alpha_n\}$, we have $\models v : \varphi(\tau)$. As a consequence, $\models v : \forall \alpha_1 \ldots \alpha_n.\, \tau$.*

Proof: the proof is a structural induction over v, essentially identical to the proof of proposition 3.5. The only difference is the base case where v is a continuation k.

• **Case** $v = k$. Then, τ is equal to τ_1 `cont`. Since $\mathcal{D}(\tau) = \mathcal{F}(\tau_1)$, it follows that none of the α_i is free in τ. Hence $\varphi(\tau) = \tau$, and the expected result. □

Proposition 3.12 (Weak soundness for continuations)

1. *Let a be an expression, τ be a type, e be an evaluation environment, E be a typing environment, k be a continuation and r be a result such that*

$$E \vdash a : \tau \quad \textit{and} \quad \models e : E \quad \textit{and} \quad \models k :: \tau \quad \textit{and} \quad e \vdash a; k \Rightarrow r.$$

   Then $r \neq \mathtt{err}$.

2. *Let v be a value, k be a continuation, τ be a type and r be a result such that*

$$\models v : \tau \quad \textit{and} \quad \models k :: \tau \quad \textit{and} \quad \vdash v \triangleright k \Rightarrow r.$$

   Then $r \neq \mathtt{err}$.

Context. I will not give a strong soundness result ("a program with type `int` evaluates to an integer") for the calculus with continuations. It is still an open issue to prove the strong soundness of a type system with respect to a continuation semantics [25]. (See [101] for a proof of strong soundness with respect to a rewriting semantics.) Don't worry: strong soundness is not required to prove weak soundness for the calculus with continuations. Now that we have made explicit the current continuation at each evaluation step, we can directly prove the weak soundness result by induction over the evaluation.

Proving just the weak soundness of Milner's type system with respect to a continuation semantics is a premiere by itself. As Milner himself writes [60]:

> When I was working on the original soundness proof of ML typing, wrt a denotational semantics (using ideals), I tried to get the proof to work using a continuation semantics, having worked it out for a direct semantics. The amusing thing was that the proof didn't work. The annoying part is that I can't find the notes. But the memory I have of it is that it was a real crunch point, and that anyone who cares to try to adapt the original proof to a continuation semantics will run into the same difficulty.

I ascribe the difficulty mentioned by Milner to the semantic typing of functional values. Since he used denotational semantics, he was certainly using the classical continuity condition ($f : \tau_1 \to \tau_2$ if for all values $v : \tau_1$ and for all continuations $k :: \tau_2$, we have $f(v)(k) \neq \texttt{wrong}$), which has no reasons to be stable under type instantiation. In contrast, since I have used an operational framework, I can use Tofte's condition (section 1.4.1, first context), that is stable under instantiation. □

Proof: we prove (1) and (2) at the same time by induction over the size of the evaluation derivations (of $e \vdash a; k \Rightarrow r$ and of $\vdash v \triangleright k \Rightarrow r$ respectively). We argue by case analysis over a for (1), and over k for (2).

• **(1), constants.**

$$\frac{\tau \leq \texttt{TypCst}(cst)}{E \vdash cst : \tau}$$

The only valid evaluation is:

$$\frac{\vdash cst \triangleright k \Rightarrow r}{e \vdash cst; k \Rightarrow r}$$

We have $\models cst : \tau$, given the definition of `TypCst`. Since $\models k :: \tau$, applying induction hypothesis (2) to the evaluation $\vdash cst \triangleright k \Rightarrow r$, we get $r \neq \texttt{err}$.

• **(1), variables.**

$$\frac{\tau \leq E(x)}{E \vdash x : \tau}$$

From hypothesis $\models e : E$ it follows that $x \in \mathrm{Dom}(e)$ and $\models e(x) : E(x)$. The only valid evaluation is therefore:

$$\frac{x \in \mathrm{Dom}(e) \qquad \vdash e(x) \triangleright k \Rightarrow r}{e \vdash cst; k \Rightarrow r}$$

Since $\models e(x) : E(x)$, we have $\models e(x) : \tau$, hence $r \neq \texttt{err}$ by (2) and hypothesis $\models k :: \tau$.

• **(1), functions.**

$$\frac{E + f \mapsto (\tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2}$$

There is only one valid evaluation:

$$\frac{\vdash (f, x, a, e) \triangleright k \Rightarrow r}{e \vdash (f \texttt{ where } f(x) = a); k \Rightarrow r}$$

We have $\models (f, x, a, e) : \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$ by definition of $\models$ (taking E for the required typing environment). Since $\models k :: \tau_1 \multimap\langle\pi\rangle\rightarrow \tau_2$, it follows $r \neq \texttt{err}$ by (2).

• **(1), function applications.**

$$\frac{E \vdash a_1 : \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1 \qquad E \vdash a_2 : \tau_2}{E \vdash a_1(a_2) : \tau_1}$$

The last evaluation step can only be:

$$\frac{e \vdash a_1; \texttt{app1c}(a_2, e, k) \Rightarrow r}{e \vdash a_1(a_2); k \Rightarrow r}$$

We have $\models \texttt{app1c}(a_2, e, k) :: \tau_2 \multimap\langle\pi\rangle\rightarrow \tau_1$ by definition of $\models$ over `app1c` continuations, taking E for the required typing environment. Applying the induction hypothesis (1) to the premise of the last evaluation step, it follows $r \neq \texttt{err}$.

• **(1), `let` bindings.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

The last evaluation step can only be:

$$\frac{e \vdash a_1; \texttt{letc}(x, a_2, e, k) \Rightarrow r}{e \vdash (\texttt{let } x = a_1 \texttt{ in } a_2); k \Rightarrow r}$$

We have $\models \texttt{letc}(x, a_2, e, k) :: \tau_1$ by definition of $\models$ over `letc` continuations, taking E for the required typing environment, and τ_2 for the required type. The result $r \neq \texttt{err}$ follows from induction hypothesis (1).

• **(1), pair construction.** Same reasoning as for applications.

• **(1), `callcc` primitive.**

$$\frac{(\tau\ \texttt{cont} \dashv\langle\pi'\rangle\rightarrowtail \tau) \dashv\langle\pi\rangle\rightarrowtail \tau \leq \forall t, u, v.\, (t\ \texttt{cont} \dashv\langle u\rangle\rightarrowtail t) \dashv\langle v\rangle\rightarrowtail t \qquad E \vdash a : \tau\ \texttt{cont} \dashv\langle\pi'\rangle\rightarrowtail \tau}{E \vdash \texttt{callcc}(a) : \tau}$$

The last evaluation step is:

$$\frac{e \vdash a; \texttt{primc}(\texttt{callcc}, k) \Rightarrow r}{e \vdash \texttt{callcc}(a); k \Rightarrow r}$$

We have $\models \texttt{primc}(\texttt{callcc}, k) : \tau\ \texttt{cont} \dashv\langle\pi'\rangle\rightarrowtail \tau$, since $\models k :: \tau$. Hence $r \neq \texttt{err}$ by (1).

• **(1), `throw` primitive.** Same reasoning as for `callcc`.

• **(2), `stop` continuations.** The only valid evaluation is $\vdash v \triangleright \texttt{stop} \Rightarrow v$, hence r equals v, hence r is not `err`.

• **(2), `app1c` continuations.** We have $k = \texttt{app1c}(a_1, e_1, k_1)$. By hypothesis $\models k :: \tau$, the type τ is equal to $\tau_1 \dashv\langle\pi\rangle\rightarrowtail \tau_2$, with

$$E_1 \vdash a_1 : \tau_1 \quad \text{and} \quad \models e_1 : E_1 \quad \text{and} \quad \models k_1 :: \tau_2$$

for some environment E. By hypothesis $\models v : \tau$, the value v is a closure (f, x, a, e), and there exists a typing environment E such that

$$\models e : E \quad (3) \quad \text{and} \quad E \vdash (f\ \texttt{where}\ f(x) = a) : \tau_1 \dashv\langle\pi\rangle\rightarrowtail \tau_2 \quad (4).$$

This excludes the first evaluation possibility: the one that concludes $r = \texttt{err}$ because v is not a closure. Hence the last evaluation step can only be:

$$\frac{e_1 \vdash a_1; \texttt{app2c}(f, x, a, e, k) \Rightarrow r}{\vdash (f, x, a, e) \triangleright \texttt{app1c}(a_1, e_1, k_1) \Rightarrow r}$$

From (3) and (4), we get $\models \texttt{app2c}(f, x, a, e, k) :: \tau_1$ by definition of $\models$ over `appc2` continuations. Applying the induction hypothesis (1) to the evaluation of $e_1 \vdash a_1; \texttt{app2c}(f, x, a, e, k) \Rightarrow r$, it follows that $r \neq \texttt{err}$, as expected.

• **(2), `app2c` continuations.** We have $k = \texttt{app2c}(f, x, a, e, k)$. By hypothesis $\models k :: \tau$, we have

$$E \vdash (f\ \texttt{where}\ f(x) = a) : \tau \dashv\langle\pi\rangle\rightarrowtail \tau' \quad (3) \quad \text{and} \quad \models e : E \quad (4) \quad \text{and} \quad \models k :: \tau' \quad (5)$$

for some environment E and some types τ' and π. The last step in the evaluation can only be

$$\frac{e + f \mapsto (f, x, a, e) + x \mapsto v_2 \vdash a; k \Rightarrow r}{\vdash v_2 \triangleright \texttt{app2c}(f, x, a, e, k) \Rightarrow r}$$

Consider the environments

$$e_1 = e + f \mapsto (f, x, a, e) + x \mapsto v \quad \text{and} \quad E_1 = e + f \mapsto (\tau \dashv\langle\pi\rangle\rightarrowtail \tau') + x \mapsto \tau.$$

By (3) and (4), we have $\models (f, x, a, e) : \tau \dashv\langle\pi\rangle\mapsto \tau'$. Combined with hypothesis $\models v : \tau$ and with (4), this fact implies $\models e_1 : E_1$. Moreover, there is only one typing rule that concludes (3); hence its premise must hold: $E_1 \vdash a : \tau'$. We can therefore apply induction hypothesis (1) to the evaluation $e_1 \vdash a; k \Rightarrow r$. We get the expected result: $r \neq \texttt{err}$.

• **(2), `letc` continuations.** The continuation k is equal to $\texttt{letc}(x, a, e, k')$. By hypothesis $\models k :: \tau$, we have

$$E + x \mapsto \texttt{Gen}(\tau, E) \vdash a : \tau' \quad \text{and} \quad \models e : E \quad \text{and} \quad \models k' :: \tau'$$

for some typing environment E and some type τ'. The last evaluation step must be:

$$\frac{e + x \mapsto v \vdash a; k' \Rightarrow r}{\vdash v \triangleright \texttt{letc}(x, a, e, k') \Rightarrow r}$$

By hypothesis, we have $\models v : \tau$. Since `Gen` does not generalize variables that are dangerous in τ, it follows that $\models v : \texttt{Gen}(\tau, E)$ by proposition 3.11. Hence:

$$\models (e + x \mapsto v) : (E + x \mapsto \texttt{Gen}(\tau, E)).$$

We can therefore apply the induction hypothesis (1) to the evaluation $e + x \mapsto v \vdash a; k' \Rightarrow r$. We get $r \neq \texttt{err}$, which is the expected result

• **(2), `pair1c` continuations.** Similar to the case for `app1c`.

• **(2), `pair2c` continuations.** Straightforward.

• **(2), $\texttt{primc}(\texttt{callcc}, k')$ continuations.** By hypothesis $\models k :: \tau$, we have τ equal to $\tau'\ \texttt{cont} \dashv\langle\pi\rangle\mapsto \tau'$, and $\models k' :: \tau'$. By hypothesis $\models v : \tau$, we therefore have, for some E,

$$v = (f, x, a, e) \quad \text{and} \quad \models e : E \quad (3) \quad \text{and} \quad E \vdash (f\ \texttt{where}\ f(x) = a) : \tau'\ \texttt{cont} \dashv\langle\pi\rangle\mapsto \tau' \quad (4)$$

There are two evaluation possibilities. The first one leads to $r = \texttt{err}$ because v is not a closure; it contradicts the hypothesis $\models v : \tau$. The other one ends up with:

$$\frac{e + f \mapsto (f, x, a, e) + x \mapsto k \vdash a; k \Rightarrow r}{\vdash (f, x, a, e) \triangleright \texttt{primc}(\texttt{callcc}, k) \Rightarrow r}$$

From (3) and from the hypothesis over v and k, it follows that

$$\models (e + f \mapsto (f, x, a, e) + x \mapsto k) : (E + f \mapsto (\tau'\ \texttt{cont} \dashv\langle\pi\rangle\mapsto \tau') + x \mapsto \tau'\ \texttt{cont}).$$

From (4) and from the typing rule for functions, we get

$$E + f \mapsto (\tau'\ \texttt{cont} \dashv\langle\pi\rangle\mapsto \tau') + x \mapsto \tau' \vdash a : \tau'.$$

Hence we can apply the induction hypothesis (1) to the evaluation$e + f \mapsto (f, x, a, e) + x \mapsto k \vdash a; k \Rightarrow r$. We get, as expected, $r \neq \texttt{err}$.

• **(2), $\texttt{primc}(\texttt{throw}, k_1)$ continuations.** Since $\models k :: \tau$, we have $\tau = \tau'\ \texttt{cont} \times \tau'$. The hypothesis $\models v : \tau$ therefore implies

$$v = (k', v') \quad \text{and} \quad \models k' :: \tau' \quad \text{and} \quad \models v' : \tau'.$$

This rules out the first evaluation possibility: the one that concludes $r = \texttt{err}$ because v does not match (k', v'). It remains the second possibility:

$$\frac{\vdash v' \triangleright k' \Rightarrow r}{\vdash (k', v') \triangleright \texttt{primc}(\texttt{throw}, k_1) \Rightarrow r}$$

We apply the induction hypothesis (2) to the evaluation $\vdash v' \triangleright k' \Rightarrow r$. We get $r \neq \texttt{err}$. This concludes the proof. □

3.4 Type inference

In this section, we show that any well-typed expression possesses a principal type, and we give a type inference algorithm — an adaptation of the Damas-Milner algorithm — that computes this principal type.

3.4.1 Unification issues

The type system of the present chapter does not naturally lend itself to type inference. That's because the type algebra does not enjoy the principal unifier property, as a consequence of the commutativity and idempotence axioms over closure types.

Example. Consider the types:

$$\tau_1 = t \multimap\langle \texttt{int}, \texttt{bool}, u\rangle\rightarrow t \qquad \tau_2 = t \multimap\langle \texttt{int}, \texttt{char}, v\rangle\rightarrow t$$

Here are two unifiers of τ_1 and τ_2:

$$\begin{aligned} \varphi_1 &= [u \mapsto \texttt{char}, w;\ v \mapsto \texttt{bool}, w] \\ \varphi_2 &= [u \mapsto \texttt{char}, \texttt{int}, w;\ v \mapsto \texttt{bool}, w] \end{aligned}$$

We do have, by idempotence and left commutativity:

$$\varphi_2(\texttt{int}, \texttt{bool}, u) = \texttt{int}, \texttt{bool}, \texttt{char}, \texttt{int}, w = \texttt{int}, \texttt{bool}, \texttt{char}, w = \varphi_2(\texttt{int}, \texttt{char}, v).$$

Yet φ_1 and φ_2 are incompatible: there are no substitutions θ such that $\varphi_1 = \theta \circ \varphi_2$ or $\varphi_2 = \theta \circ \varphi_1$. □

Example. Consider the types:

$$\tau_1 = t \multimap\langle t\ \texttt{ref}, u\rangle\rightarrow t \qquad \tau_2 = t \multimap\langle \texttt{int ref}, v\rangle\rightarrow t.$$

The two substitutions below are unifiers of τ_1 and τ_2:

$$\varphi_1 = [u \mapsto \texttt{int ref}, w;\ v \mapsto t\ \texttt{ref}, w] \qquad \varphi_2 = [t \mapsto \texttt{int};\ u \mapsto w;\ v \mapsto w].$$

We have

$$\varphi_1(t\ \texttt{ref}, u) = t\ \texttt{ref}, \texttt{int ref}, t\ \texttt{ref}, v = t\ \texttt{ref}, \texttt{int ref}, v = \varphi_1(\texttt{int ref}, u)$$

by idempotence and commutativity. Yet the two substitutions φ_1 and φ_2 are incompatible. □

In the first example above, the two unifiers both map τ_1 and τ_2 to $t \multimap\!\langle \texttt{int}, \texttt{bool}, \texttt{char}, w \rangle\!\rightarrow t$, which is, intuitively, the most general common instance of the two types. The only way to distinguish φ_1 from φ_2 is to apply these substitutions to closure types ending with u; in addition, these closure types must not contain `int`. In particular, φ_1 and φ_2 are indistinguishable if, among the closure types considered, the only ones that end with u are always equal to $\texttt{int}, \texttt{bool}, u$.

It turns out that this hypothesis always holds in a principal typing: two closure types ending in the same expansion variable are always equal. Here is an intuitive explanation for this phenomenon. Closure types are always created with fresh, different expansion variables. The sole operation that leads to share an expansion variable between two closure types is the identification of two function types $\tau_1 \multimap\!\langle \pi \rangle\!\rightarrow \tau_2$ and $\tau_1' \multimap\!\langle \pi' \rangle\!\rightarrow \tau_2'$. But this operation completely identifies π and π': they will now share the same expansion variable, but this variable will follow the same set of type schemes in both closure types.

I shall use the adjective "homogeneous" to refer to this situation where two different closure types never share the same expansion variable. (In the remainder of this section, we shall define more precisely this homogeneity property.) We are now going to consider only unification problems between two homogeneous types, whose solutions will be only applied to types that are homogeneous with the two initial types.

3.4.2 Homogeneous types

A CLASSIFICATION, written K, is a finite mapping from extension variables (the closure type variables) to sets of type schemes. We now define what it means for a type, a type scheme or a closure type to be HOMOGENEOUS with K, or K-homogeneous. This homogeneity relation is written $:: K$. First of all, a closure type $\sigma_1, \ldots, \sigma_n, u$ is homogeneous with K if K assigns the set $\{\sigma_1, \ldots, \sigma_n\}$ to the extension variable u. Moreover, the σ_i themselves must be K-homogeneous.

$$\frac{\{\sigma_1 \ldots \sigma_n\} = K(u) \qquad \sigma_1 :: K \quad \ldots \quad \sigma_n :: K}{\sigma_1, \ldots, \sigma_n, u :: K}$$

A type is K-homogeneous if all closure types contained in it are K-homogeneous.

$$\iota :: K \qquad t :: K \qquad \frac{\tau_1 :: K \qquad \tau_2 :: K}{\tau_1 \times \tau_2 :: K} \qquad \frac{\tau_1 :: K \qquad \pi :: K \qquad \tau_2 :: K}{\tau_1 \multimap\!\langle \pi \rangle\!\rightarrow \tau_2 :: K}$$

$$\frac{\tau :: K}{\tau\ \texttt{ref} :: K} \qquad \frac{\tau :: K}{\tau\ \texttt{chan} :: K} \qquad \frac{\tau :: K}{\tau\ \texttt{cont} :: K}$$

Finally, a type scheme is K-homogeneous if there exists an extension K' of K to the expansion variables universally quantified in the scheme such that the type inside the scheme is K'-homogeneous.

$$\frac{\{u_1 \dots u_m\} = \{\alpha_1, \dots, \alpha_n\} \cap \texttt{VarTypClos} \qquad \tau :: K + u_1 \mapsto \Sigma_1 + \cdots + u_m \mapsto \Sigma_m}{(\forall \alpha_1 \dots \alpha_n . \tau) :: K}$$

We immediately extend the relation $:: K$ to typing environments by taking $E :: K$ if and only if $E(x) :: K$ for all $x \in \mathrm{Dom}(E)$.

Remark. If $\tau :: K$, then the domain of K contains all the expansion variables free in τ. This property also holds if the type τ is replaced by a type scheme or a closure type. □

Let φ be a substitution. We say that φ is HOMOGENEOUS FROM K TO K', and we write $\varphi :: K \Rightarrow K'$, if for all types τ such that $\tau :: K$, we have $\varphi(\tau) :: K'$.

Remark. If $\varphi :: K \Rightarrow K'$ and $\varphi' :: K' \Rightarrow K''$, we immediately have $\varphi' \circ \varphi :: K \Rightarrow K''$. □

3.4.3 Unification

In this section, we give a unification algorithm between K-homogeneous types, and we show that it computes a principal unifier of these types. In the following, we write Q for a set of equations between simple types $(\tau_1 = \tau_2)$ and between closure types $(\pi_1 = \pi_2)$. Hence Q never contains meaningless equations such as $\tau = \pi$. (We therefore work in a two-sorted algebra: one sort is the simple types, the other is the closure types.)

Algorithm 3.1 Let K be a classification. Let Q be a set of equations between simple types and closure types, such that all types appearing in Q are K-homogeneous. We define a substitution $\texttt{mgu}(Q)$ by:

If $Q = \emptyset$:
  $\texttt{mgu}(Q) = [\,]$
If $Q = \{\pi = \pi'\} \cup Q'$:
  write $\pi = \sigma_1, \dots, \sigma_n, u$ and $\pi' = \sigma'_1, \dots, \sigma'_m, u'$
  if $u = u'$, then $\texttt{mgu}(Q) = \texttt{mgu}(Q')$
  if $u \in \mathcal{F}(\pi')$ ou $u' \in \mathcal{F}(\pi)$, then $\texttt{mgu}(Q)$ is not defined
  else take $\texttt{mgu}(Q) = \texttt{mgu}(\varphi(Q')) \circ \varphi$
  with $\varphi = [u \mapsto (\sigma'_1, \dots, \sigma'_m, u),\ u' \mapsto (\sigma_1, \dots, \sigma_n, u)]$
If $Q = \{t_1 = t_2\} \cup Q'$ and $t_1 = t_2$:
  $\texttt{mgu}(Q) = \texttt{mgu}(Q')$
If $Q = \{t = \tau\} \cup Q'$ or $Q = \{\tau = t\} \cup Q'$:
  if $t \in \mathcal{F}(\tau)$ then $\texttt{mgu}(Q)$ is undefined
  else $\texttt{mgu}(Q) = \texttt{mgu}(\varphi(Q')) \circ \varphi$ with $\varphi = [t \mapsto \tau]$
If $Q = \{\iota_1 = \iota_2\} \cup Q'$ and $\iota_1 = \iota_2$:
  $\texttt{mgu}(Q) = \texttt{mgu}(Q')$
If $Q = \{\tau_1 \multimap\!\langle\pi\rangle\!\rightarrowtail \tau_2 = \tau'_1 \multimap\!\langle\pi'\rangle\!\rightarrowtail \tau'_2\} \cup Q'$:
  $\texttt{mgu}(Q) = \texttt{mgu}(\{\tau_1 = \tau'_1,\ \pi = \pi',\ \tau_2 = \tau'_2\} \cup Q')$

If $Q = \{\tau_1 \times \tau_2 = \tau_1' \times \tau_2'\} \cup Q'$:
$\quad \mathtt{mgu}(Q) = \mathtt{mgu}(\{\tau_1 = \tau_1',\ \tau_2 = \tau_2'\} \cup Q')$
If $Q = \{\tau\ \mathtt{ref} = \tau'\ \mathtt{ref}\} \cup Q'$
or $Q = \{\tau\ \mathtt{chan} = \tau'\ \mathtt{chan}\} \cup Q'$
or $Q = \{\tau\ \mathtt{cont} = \tau'\ \mathtt{cont}\} \cup Q'$:
$\quad \mathtt{mgu}(Q) = \mathtt{mgu}(\{\tau = \tau'\} \cup Q')$

In all other cases, the substitution $\mathtt{mgu}(Q)$ is undefined.

Remark. The algorithm always terminates, since at each step the sum of the height of the types in Q strictly decreases (the height being defined as $h(t) = 1$, $h(\tau_1 \times \tau_2) = 1 + \max(h(\tau_1), h(\tau_2))$, and so on). □

Remark. The substitution $\mathtt{mgu}(Q)$ does not introduce new variables with respect to Q: any variable that is not free in Q is out of reach for $\mathtt{mgu}(Q)$. □

Proposition 3.13 *Let Q be a set of K-homogeneous equations. If $\mu = \mathtt{mgu}(Q)$ is defined, then μ is a unifier of Q, moreover there exists a classification K' such that $\mu :: K \Rightarrow K'$.*

Proof: by step induction on the algorithm. Except the case $\pi = \pi'$, all cases are similar to those in the proof of Robinson's algorithm [85]. Classification handling is trivial in these cases: since no extension variables are instantiated, `mgu` is recursively applied to sets of equations that are also K-homogeneous, and we can take for K' the K' obtained by the induction hypothesis. Therefore, I detail only the new case.

• **Case** $Q = \{\pi = \pi'\} \cup Q'$. Write $\pi = \sigma_1, \ldots, \sigma_n, u$ et $\pi' = \sigma_1', \ldots, \sigma_m', u'$. By K-homogeneity hypothesis, we have $\{\sigma_1, \ldots, \sigma_n\} = K(u)$ and $\{\sigma_1', \ldots, \sigma_m'\} = K(u')$.

If $u = u'$, we therefore have $\{\sigma_1, \ldots, \sigma_n\} = \{\sigma_1', \ldots, \sigma_m'\}$, hence $\pi = \pi'$ by application of the commutativity and idempotence axioms. Since $\mathtt{mgu}(Q')$ is a unifier of Q', as the induction hypothesis shows, $\mathtt{mgu}(Q')$ is also a unifier of Q.

If $u \neq u'$, the algorithm guarantees that $u \notin \mathcal{F}(\pi')$ and $u' \notin \mathcal{F}(\pi)$. The substitution φ defined in the algorithm is therefore such that:

$$\begin{aligned} \varphi(\pi) &= \varphi(\sigma_1), \ldots, \varphi(\sigma_n), \varphi(u) \\ &= \varphi(\sigma_1), \ldots, \varphi(\sigma_n), \sigma_1', \ldots, \sigma_m', u \\ &= \sigma_1, \ldots, \sigma_n, \sigma_1', \ldots, \sigma_m', u \end{aligned}$$

That's because $\varphi(\sigma_i) = \sigma_i$ for all i, since neither u nor u' are free in σ_i. If u' was free in σ_i, this would contradict the hypothesis $u' \notin \mathcal{F}(\pi)$. If u was free in σ_i, this would contradict the K-homogeneity hypothesis: u can only appear in σ_i following the same schemes $\sigma_1, \ldots, \sigma_n$, which is impossible since π has finite size. By symmetry, we also have

$$\varphi(\pi') = \sigma_1', \ldots, \sigma_m', \sigma_1, \ldots, \sigma_n, u.$$

The substitution φ is therefore a unifier of π and π'. Define

$$K_1 = \varphi(K) + u \mapsto \{\sigma_1, \ldots, \sigma_n, \sigma_1', \ldots, \sigma_m'\}.$$

We have $\varphi :: K \Rightarrow K_1$. That's because, for all closure types π, one of the following three cases hold. Either π ends with u, and then $\pi = \sigma_1, \ldots, \sigma_n, u$ by K-homogeneity, hence $\varphi(\pi) = \sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_m, u$ is K_1-homogeneous. Or π ends with u', and then $\pi = \sigma'_1, \ldots, \sigma'_m, u'$, and we conclude as in the previous case. Or π ends with an expansion variable v that is neither u nor u'. Hence $K_1(v) = \varphi(K(v))$, and $\varphi(\pi) :: \varphi(K)$ implies $\varphi(\pi) :: K_1$.

It follows that $\varphi(Q')$ is K_1-homogeneous. The induction hypothesis proves that $\mathtt{mgu}(\varphi(Q'))$ is a unifier of $\varphi(Q')$, and that there exists K' tel que $\mathtt{mgu}(\varphi(Q')) :: K_1 \Rightarrow K'$. It follows that $\mathtt{mgu}(\varphi(Q')) \circ \varphi$ is a unifier of Q, and $\mathtt{mgu}(\varphi(Q')) \circ \varphi :: K \Rightarrow K'$. This is the expected result. □

We are now going to show that $\mathtt{mgu}(Q)$ is a principal unifier of Q when we consider only K-homogeneous types. To formalize this idea, we say that two substitutions φ_1 and φ_2 are K-EQUAL, and we write $\varphi_1 \stackrel{K}{=} \varphi_2$, if $\varphi_1(\tau) = \varphi_2(\tau)$ for all types τ that are K-homogeneous, and similarly for all type schemes σ and all closure types π instead of τ.

Proposition 3.14 *Let Q be a set of K-homogeneous equations. If there exists a substitution ψ that is a unifier of Q, then $\mu = \mathtt{mgu}(Q)$ is defined, and there exists a substitution θ such that $\psi \stackrel{K}{=} \theta \circ \mu$.*

Proof: we proceed by induction over the sum of the sizes of the types in Q. Except the case $\pi = \pi'$, all cases are proved as for Robinson's algorithm.

• **Case** $Q = \{\pi = \pi'\} \cup Q'$. Write $\pi = \sigma_1, \ldots, \sigma_n, u$ et $\pi' = \sigma'_1, \ldots, \sigma'_m, u'$. If $u = u'$, we have $\pi = \pi'$ by K-homogeneity, and the result immediately follows from the induction hypothesis. Hence we now assume $u \neq u'$. First of all, we have $u \notin \mathcal{F}(\pi')$. Otherwise, by K-homogeneity, the whole closure type π would appear as a strict subterm of π'; then, $\psi(\pi)$ is a strict subterm of $\psi(\pi')$, which contradicts $\psi(\pi) = \psi(\pi')$. Symmetrically, we also have $u' \notin \mathcal{F}(\pi)$. Consider the substitution φ built by the algorithm:

$$\varphi = [u \mapsto (\sigma'_1, \ldots, \sigma'_m, u),\ u' \mapsto (\sigma_1, \ldots, \sigma_n, u)]$$

We now show that $\psi \stackrel{K}{=} \psi \circ \varphi$. Let $\pi_1 :: K$ be a closure type. We show $\psi(\pi_1) = \psi(\varphi(\pi_1))$ by induction over π_1. If π_1 ends with u, then $\pi_1 = \pi$. But then,

$$\psi(\varphi(\pi)) = \psi(\sigma'_1, \ldots, \sigma'_m, \sigma_1, \ldots, \sigma_n, u) = \psi(\sigma'_1), \ldots, \psi(\sigma'_m), \psi(\pi) = \psi(\pi).$$

That's because the closure type $\psi(\pi)$ contains at least $\psi(\sigma'_1), \ldots, \psi(\sigma'_m)$, since $\psi(\pi) = \psi(\pi')$. If π_1 ends up with u', then $\pi_1 = \pi'$, and we similarly have $\psi(\varphi(\pi')) = \psi(\pi')$. In all other cases, π_1 ends with an expansion variable v that is neither u nor u'. Hence, $\varphi(v) = v$. Moreover, by induction hypothesis, the schemes that appear before v, being K-homogeneous, are mapped to the same scheme by ψ and by $\psi \circ \varphi$. Hence $\psi(\pi_1) = \psi(\varphi(\pi_1))$ for all $\pi_1 :: K$.

Moreover, since $\psi \stackrel{K}{=} \psi \circ \varphi$, the substitution ψ is a unifier of $\varphi(Q')$, and $\varphi :: K \to K_1$, where the classification K_1 is defined as in the proof of proposition 3.13. Hence, $\varphi(Q')$ is K_1-homogeneous. Applying the induction hypothesis, it follows that $\mathtt{mgu}(\varphi(Q'))$ is defined, and that $\psi \stackrel{K_1}{=} \theta \circ \mathtt{mgu}(\varphi(Q'))$ for some substitution θ. We conclude that $\mathtt{mgu}(Q) = \mathtt{mgu}(\varphi(Q')) \circ \varphi$ is well-defined, and

$$\psi \stackrel{K}{=} \psi \circ \varphi \stackrel{K}{=} \psi \circ \mathtt{mgu}(\varphi(Q')) \circ \varphi = \theta \circ \mathtt{mgu}(Q).$$

This is the expected result. □

3.4.4 The type inference algorithm

Now that we have defined a satisfactory notion of principal unifier, it is easy to adapt the Damas-Milner algorithm to type inference in the presence of closure types. The algorithm takes as input an expression a, a typing environment E and an infinite set of "fresh" variables V. It returns a type τ (the most general type for a), a substitution φ (representing the instantiations that have been performed in E), and a subset V' of V (the "fresh" variables that have not been used).

We write $\texttt{Inst}(\sigma, V)$ for a trivial instance of the type scheme σ. That is, writing $\sigma = \forall\alpha_1 \ldots \alpha_n.\, \tau$, we choose n variables $\beta_1 \ldots \beta_n$ in V, with β_i of the same sort as α_i for all i, and we take

$$\texttt{Inst}(\sigma, V) = ([\alpha_1 \mapsto \beta_1, \ldots, \alpha_1 \mapsto \beta_1](\tau),\ V \setminus \{\beta_1 \ldots \beta_n\}).$$

$\texttt{Inst}(\sigma, V)$ is defined up to a renaming of the variables in V into variables in V.

Algorithm 3.2 $\texttt{Infer}(E, a, V)$ is the triple (τ, φ, V') defined by:

If a est x and $x \in \mathrm{Dom}(E)$:
  $(\tau, V') = \texttt{Inst}(E(x), V)$ and $\varphi = [\,]$
If a est cst:
  $(\tau, V') = \texttt{Inst}(\texttt{TypCst}(cst), V)$ and $\varphi = [\,]$
If a est $(f \texttt{ where } f(x) = a_1)$:
  let t and t'be two type variables and u an extension variable, taken from V
  let $\{x_1, \ldots, x_n\}$ be the free identifiers of $(f \texttt{ where } f(x) = a_1)$
  let $\pi = E(x_1), \ldots, E(x_n), u$
  let $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1,\ E + f \mapsto (t \dashv\langle\pi\rangle\mapsto t') + x \mapsto t,\ V \setminus \{t, t', u\})$
  let $\mu = \texttt{mgu}(\varphi_1(t'), \tau_1)$
  then $\tau = \mu(\varphi_1(t \dashv\langle\pi\rangle\mapsto t'))$ and $\varphi = \mu \circ \varphi_1$ and $V' = V_1$
If a est $a_1(a_2)$:
  let $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
  let $(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), V_1)$
  let $t \in V_2$ be a type variable and $u \in V_2$ be an extension variable
  let $\mu = \texttt{mgu}(\varphi_2(\tau_1), \tau_2 \dashv\langle u\rangle\mapsto t)$
  then $\tau = \mu(t)$ and $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ and $V' = V_2 \setminus \{t, u\}$
If a est $\texttt{let } x = a_1 \texttt{ in } a_2$:
  let $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
  let $(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E) + x \mapsto \texttt{Gen}(\tau_1, \varphi_1(E)), V_1)$
  then $\tau = \tau_2$ and $\varphi = \varphi_2 \circ \varphi_1$ and $V = V_2$
If a est (a_1, a_2):
  let $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
  let $(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), V_1)$
  then $\tau = \tau_1 \times \tau_2$ and $\varphi = \varphi_2 \circ \varphi_1$ and $V' = V_2$
If a est $op(a_1)$:
  let $(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V)$
  let $(\tau_2, V_2) = \texttt{Inst}(\texttt{TypOp}(op), V_1)$
  let $t \in V_2$ be a type variable and $u \in V_2$ be an extension variable
  let $\mu = \texttt{mgu}(\tau_1 \dashv\langle u\rangle\mapsto t, \tau_2)$

then $\tau = \mu(t)$ and $\varphi = \mu \circ \varphi_1$ and $V' = V_2 \setminus \{t, u\}$

We take that $\texttt{Infer}(a, E, V)$ is undefined if, at some point, no case applies; in particular, if we try to unify two types that are not unifiable. $\texttt{Infer}(a, E, V)$ is defined up to a renaming of variables from V into variables from V.

Proposition 3.15 (Correctness of type inference) *Let a be an expression, E be a typing environment and V be an infinite set of type variables. If $(\tau, \varphi, V') = \texttt{Infer}(a, E, V)$ is defined, then we can derive $\varphi(E) \vdash a : \tau$.*

Proof: the proof follows exactly that for proposition 1.8, and relies essentially on the stability of the typing judgement under substitution (proposition 3.2). □

Proposition 3.16 (Completeness of type inference) *Let K be a classification, a be an expression, $E :: K$ be a typing environment, and V a set of variables containing infinitely many type variables and infinitely many closure type variables, and such that $V \cap \mathcal{F}(E) = \emptyset$. If there exists a type τ' and a substitution φ' such that $\varphi'(E) \vdash a : \tau'$, then $(\tau, \varphi, V') = \texttt{Infer}(a, E, V)$ is defined, and there exists a substitution ψ such that*

$$\tau' = \psi(\tau) \quad \textit{and} \quad \varphi' \overset{K}{=} \psi \circ \varphi \textit{ outside } V.$$

(That is, $\varphi'(\tau) = \psi(\varphi(\tau))$ for all types $\tau :: K$ such that $\mathcal{F}(\tau) \cap V = \emptyset$.)

Proof: first, notice that if $(\tau, \varphi, V') = \texttt{Infer}(a, E, V)$ is defined, then there exists K' such that $\tau :: K'$ and $\varphi :: K \Rightarrow K'$. Moreover, $V' \subseteq V$, and the variables in V' are not free in τ and are out of reach for φ. This can easily be shown by step induction over the algorithm, using proposition 3.13 and the fact that the unifier $\texttt{mgu}(\tau_1, \tau_2)$ does not introduce new variables. As a consequence of these remarks, $\varphi(E) :: K'$ and $V' \cap \mathcal{F}(\varphi(E)) = \emptyset$.

The proof of the proposition is an inductive argument on the derivation of $\varphi'(E) \vdash a : \tau'$, and by case analysis over a. The proof proceeds exactly as that of proposition 1.9, with some additional classification handling. I detail one case, to illustrate the use of the K-homogeneity hypothesis.

• **Case $a = a_1(a_2)$.** The initial derivation ends up with

$$\frac{\varphi'(E) \vdash a_1 : \tau'' \multimap\langle\pi'\rangle\rightarrow \tau' \qquad \varphi'(E) \vdash a_2 : \tau''}{\varphi'(E) \vdash a_1(a_2) : \tau'}$$

We apply the induction hypothesis to a_1, $E :: K$, V, $\tau'' \multimap\langle u'\rangle\rightarrow \tau'$ and φ'. It follows that

$$(\tau_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, V) \quad \text{and} \quad \tau'' \multimap\langle\pi'\rangle\rightarrow \tau' = \psi_1(\tau_1) \quad \text{and} \quad \varphi' \overset{K}{=} \psi_1 \circ \varphi_1 \text{ outside } V \quad \text{and}$$

$$\tau_1 :: K_1 \quad \text{and} \quad \varphi_1 :: K \Rightarrow K_1.$$

In particular, $\varphi'(E) = \psi_1(\varphi_1(E))$ and $\varphi_1(E) :: K_1$. We apply the induction hypothesis to a_2, $\varphi_1(E) :: K_1$, V_1, τ and ψ_1. We have $\mathcal{F}(\varphi_1(E)) \cap V_1 = \emptyset$ as required, by the remark at the beginning of the proof. It follows that:

$$(\tau_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), V_1) \quad \text{and} \quad \tau'' = \psi_2(\tau_2) \quad \text{and} \quad \psi_1 \overset{K_1}{=} \psi_2 \circ \varphi_2 \text{ outside } V_1 \quad \text{and}$$

$$\tau_2 :: K_2 \quad \text{and} \quad \varphi_2 :: K_1 \Rightarrow K_2.$$

We have $\mathcal{F}(\tau_1) \cap V_1 = \emptyset$, hence $\psi_1(\tau_1) = \psi_2(\varphi_2(\tau_1))$. Take

$$\psi_3 = \psi_2 + t \mapsto \tau' + u \mapsto \pi' \qquad K_3 = K_2 + u \mapsto \Sigma' \text{ avec } (\Sigma', u') = \pi'.$$

The variables t and u, taken from V_2, are out of reach for ψ_2, hence ψ_3 extends ψ_2. Similarly, we can assume $u \notin \mathrm{Dom}(K_2)$, hence K_3 extends K_2. It follows that:

$$\begin{array}{rclcrcl} \psi_3(\varphi_2(\tau_1)) & = & \psi_2(\varphi_2(\tau_1)) & = & \psi_1(\tau_1) & = & \tau'' \to \tau' \\ \psi_3(\tau_2 \to \alpha) & = & \psi_2(\tau_2) \to \tau'' & = & \tau'' \to \tau' & & \end{array}$$

The substitution φ_3 is therefore a unifier of $\varphi_2(\tau_1)$ and $\tau_2 \dashv\langle u \rangle\rightarrow t$. Moreover, these two types are K_3-homogeneous. The principal unifier of these two types, μ, therefore exists, and $\texttt{Infer}(a_1(a_2), E, V)$ is well-defined. Moreover, we have $\mu :: K_3 \Rightarrow K_4$ and $\psi_3 = \psi_4 \circ \mu$ for some substitution ψ_4 and some classification K_4. We now show that $\psi = \psi_4$ and $K' = K_4$ give the expected result. With the same notations as in the algorithm, we have:

$$\psi(\tau) = \psi_4(\mu(\alpha)))) = \psi_3(\alpha) = \tau'.$$

Moreover, for all $\tau :: K$ such that $\mathcal{F}(\tau) \cap V = \emptyset$ (hence a fortiori $\mathcal{F}(\tau) \cap V_1 = \emptyset$, $\beta \notin V_2$, $\beta \neq \alpha$):

$$\begin{array}{rcll} \psi(\varphi(\tau)) & = & \psi_4(\mu(\varphi_2(\varphi_1(\tau)))) & \text{by definition of } \varphi \\ & = & \psi_3(\varphi_2(\varphi_1(\tau))) & \text{by definition of } \psi_4 \\ & = & \psi_2(\varphi_2(\varphi_1(\tau))) & \text{because } t \notin \mathcal{F}(\tau) \text{ and } t \text{ is out of reach for } \varphi_1 \text{ and for } \varphi_2 \\ & = & \psi_1(\varphi_1(\tau)) & \text{because } \varphi_1(\tau) :: K_1 \text{ and } \mathcal{F}(\tau) \cap V_1 = \emptyset \\ & = & \varphi'(\tau) & \text{because } \tau :: K \text{ and } \mathcal{F}(\tau) \cap V = \emptyset. \end{array}$$

Finally, we immediately have $\varphi :: K \Rightarrow K_4$. This concludes the proof of the expected result. □

Chapter 4

Refined closure typing

In the present chapter, we introduce and study a variant of the type system presented in chapter 3, which relies on the same concepts of dangerous variables and closure typing, but which performs closure typing more finely. Closure typing as implemented by the system in chapter 3, turns out to be too weak in some cases, as we shall now illustrate; the goal of this second type system is to palliate these weaknesses.

4.1 Non-conservativity of the first type system

The type system proposed in chapter 3, though ensuring type safety at run-time, is not entierly satisfactory. The problem is that this system rejects as ill-typed some purely applicative programs (that is, programs that do not use any reference, channel, or continuation) that are well-typed in ML. I name "conservativity" this property that a type system for some algorithmic extensions of ML accepts all programs that are well-typed in the purely applicative core ML language. The conservativity property is strong evidence that the proposed type system is a proper extension of the ML one.

The non-conservativity of the type system in chapter 3 does not come from the restriction of generalization to non-dangerous variables: no variable is dangerous in a purely applicative program. The problem lies within closure typing: even in a purely applicative program, closure typing leads to function types that are richer, hence possibly more selective, on the one hand, and on the other hand that may contain more free variables, which can prevent the generalization of some variables in other types.

4.1.1 Recursive closure types

Closure typing leads to assign different types to functions that have the same type in ML. For instance, two functions from integers to integers can have the two different types $\texttt{int} \multimap\langle \sigma_1, u\rangle\rightarrow \texttt{int}$ and $\texttt{int} \multimap\langle \sigma_2, v\rangle\rightarrow \texttt{int}$, while in ML they have the same type $\texttt{int} \rightarrow \texttt{int}$. Most often, this causes no trouble: we can identify most closure types by proper instantiation of their extension variable. In the previous example, if u and v do not occur in σ_1 nor in σ_2, we can identify the two types

by replacing u by σ_2, w and v by σ_1, w. Unfortunately, identification is not always possible when the extension variables of the closure types also occur inside the closure types, in one of the type schemes. For instance, there is no common instance for the two types

$$\tau_1 \multimap\langle u\rangle\rightarrowtail \tau_2 \quad \text{and} \quad \tau_1 \multimap\langle \tau_1 \multimap\langle u\rangle\rightarrowtail \tau_2, v\rangle\rightarrowtail \tau_2.$$

However, if `f` has type the leftmost type, then the rightmost type is the type of the eta-expansed form of `f`, that is λ`x.f(x)`. Hence, since these types are incompatible, this means that the following phrase is ill-typed:

$$\lambda\texttt{f.\,if ... then f else }\lambda\texttt{x.f(x)}.$$

It is perfectly correct, however — and well-typed in ML.

We can circumvent this problem by allowing recursive closure types: closure types of the form $\mu u.\,\pi$, standing for the possibly infinite type solution of $\pi = u$. In the eta-expansion example, the two previously incompatible types now have the common instance

$$\tau_1 \multimap\langle \mu u.\,\tau_1 \multimap\langle u\rangle\rightarrowtail \tau_2, v\rangle\rightarrowtail \tau_2,$$

obtained by substituting u by $\mu u.\,\tau_1 \multimap\langle u\rangle\rightarrowtail \tau_2, v$ in the rightmost type, and by the equivalent form $\tau_1 \multimap\langle \mu u.\,\tau_1 \multimap\langle u\rangle\rightarrowtail \tau_2, v\rangle\rightarrowtail \tau_2, v$ in the leftmost type. This common instance represents the following infinite type:

$$\tau_1 \multimap\langle \tau_1 \multimap\langle \tau_1 \multimap\langle \tau_1 \multimap\langle \ldots, v\rangle\rightarrowtail \tau_2,\ v\rangle\rightarrowtail \tau_2,\ v\rangle\rightarrowtail \tau_2,\ v\rangle\rightarrowtail \tau_2.$$

More generally, we can show that, with recursive closure types, any two closure types always admit a common instance. Hence, two functions have compatible types if and only if the types of their arguments are compatible, as well as the types of their results, just as in the case of the ML type system. (In the remainder of this chapter, we shall give a different presentation of closure types, that does not make use of infinite type such as $\mu u.\,\pi$. This alternate presentation also enjoys the property that any two closure types always possess a common instance.)

4.1.2 Variable capture in closure types

Even if we arrange that any two closure types are always compatible, there remains "pure" ML programs that are no longer well-typed when performing closure typing. There is indeed another reason why closure typing can lead to typings less general than in ML: some type variables can occur free in a function type with closure type, while they are not free in the corresponding ML type. For instance, t is free in `int` $\multimap\langle t$ `list`$, u\rangle\rightarrowtail$ `int`, while it is not free in `int` $\rightarrow$ `int`. If the type `int` $\multimap\langle t$ `list`$, u\rangle\rightarrowtail$ `int` belongs to the current typing environment, this means that the variable t is not generalizable, while it would be generalizable in the ML type system. Here is an example that demonstrates this phenomenon:

$$\lambda\texttt{f. let id} = \lambda\texttt{y. either(f)(}\lambda\texttt{z.y;z);y in id(id)}$$

where the `either` function is defined as:

$$\texttt{let either} = \lambda\texttt{x.}\,\lambda\texttt{y. if ... then x else y}$$

and is intended to force its two arguments to have the same type. Let us try to typecheck this phrase, under the hypothesis $\mathtt{y} : t_1$. The function $\lambda\mathtt{z}.\mathtt{y};\mathtt{z}$ has type $t_2 \multimap\langle t_1, u\rangle\rightarrowtail t_2$. This type is also the type of `f`, because of the constraint imposed by `either`. The left-hand part of the `let`, $\lambda\mathtt{y}\ldots$, has a type of the form $t_1 \multimap\langle \pi'\rangle\rightarrowtail t_1$. At the time we generalize this latter type, the typing environment is

$$\mathtt{f} : t_2 \multimap\langle t_1, u\rangle\rightarrowtail t_2.$$

The variable t_1 is free in this environment, and therefore not generalizable in $t_1 \multimap\langle \pi'\rangle\rightarrowtail t_1$. As a consequence, `id` remains monomorphic, and the self-application `id(id)` is ill-typed. The same phrase is well-typed in ML, since, without closure typing, t_1 does not occur in the type of `f`.

This phenomenon of variable capture through the closure types is considerably harder to eliminate than the incompatibility phenomenon between closure types. A first approach for avoiding it is to ignore, when computing the free variables of a function type $\tau_1 \multimap\langle \pi\rangle\rightarrowtail \tau_2$, all variables that are free in π, but not in τ_1 nor in τ_2. An alternate approach is, when assigning the type $\tau_1 \multimap\langle \pi\rangle\rightarrowtail \tau_2$ to a function (f `where` $f(x) = a$), to avoid recording in π the type of an identifier free in (f `where` $f(x) = a$) if this type does not have any free variable in common with the argument type τ_1 or the result type τ_2.

It might seem that we are loosing track of some polymorphic references hidden inside closures by doing so. For instance, in a functional value of type $\alpha \multimap\langle \beta\ \mathtt{ref}, u\rangle\rightarrowtail \alpha$, there is a reference with type β `ref`. With the two approaches for avoiding variable capture proposed above, we are going to ignore the fact that β is dangerous in that type, and hence allow the generalization of β, thus assigning a polymorphic type to a reference. Is that unsafe? No, it is not, because we cannot access this reference. Given its type, the function cannot store (parts of) its argument inside this reference, since the argument has type α while the reference contents have type β. The function cannot either return the reference in its result since the result has type α while the reference has type β `ref`. Finally, the function cannot store the reference in another reference, this one reachable, because this reference should have type β `ref ref` and be reachable from the environment, hence β would have been considered dangerous at generalization-time.

The idea behind this reasoning is as follows: if we can type an expression under the hypothesis $x : \alpha$ and $y : \beta$ (two distinct type variables), then, during the evaluation of the expression, no communication can take place between the values of x and y. This intuition is the basis of some adaptations of the Damas-Milner algorithm to the static inference of sharing properties [5]. I don't know any precise formulation of this result (how can we characterize communication?), let alone any proof.

4.1.3 Importance of conservativity

The conservativity property is highly desirable. The ML type system is generally considered as satisfactory for a purely applicative language. It is therefore preferable not to be more restrictive than the ML type system. Also, it is intellectually comforting to see that the typing mechanisms added to control the imperative extensions do not interfere when we do not use these extensions. From this standpoint, the type system presented in chapter 3 is not satisfactory.

From a more pragmatic standpoint, it should be noticed that the non-conservativity of this system shows up only on artificial, complicated examples. In particular, I haven't yet encountered

a realistic program that demonstrate a variable capture through closure types. To support this claim, I have equipped Caml Light [49], my ML compiler, with the type system described in chapter 3, and I have fed it about ten thousand lines of ML source code; the variable capture phenomenon did not show up. I therefore claim that the system in chapter 3 is conservative for most, if not all, practical purposes.

This kind of claim does not, however, possess the mathematical rigor that is expected from a doctorate dissertation in Fundamental Computer Science from university Paris 7. My tests are obviously not exhaustive; I have considered only relatively simple programs, leaving aside a number of idioms using higher-order functions intensively, such as those that appear in programs extracted from proofs [24], or in clever encodings of weird data structures [23].

I therefore spent a lot of time looking for a type system featuring dangerous variables and closure types that enjoys the conservativity property. The remainder of this chapter presents the result of this research. To achieve conservativity and avoid the capture phenomenon, a fine control over generalization is required — much finer than in the system in chapter 3. Therefore I had to abandon the fairly classical presentation of the ML type system followed in chapters 1 and 3, and to adopt a rather unusual graph-based presentation.

4.2 The indirect type system

4.2.1 Presentation

The type system in the present chapter differs from the system in chapter 3 in two points: the representation of closure types and the representation of polymorphic types.

4.2.1.1 Labels and constraints instead of closure types

The starting point for this system is to add an extra indirection level in the association function type/closure type. Instead of directly annotating function types by the extensible sets of type schemes that represent the closure types, we simply annotate function types $\tau_1 \to \tau_2$ by a single variable, called a LABEL, and written u. Function types therefore has the form $\tau_1 \multimap\langle u\rangle\to \tau_2$. Outside the type expressions, in a separate environment, we associate to each label u a set of type schemes: the types of the values possibly contained in the closures labeled u. This separate environment is presented as a set of CONSTRAINTS $\sigma_1 \lhd u_1, \ldots, \sigma_n \lhd u_n$. The constraint $\sigma \lhd u$ reads "any closure labeled u can contain a value of type σ".

We are therefore going to manipulate pairs (τ, C) of a type expression τ, whose function types are annotated by labels, and of a set of constraints C defining the contents of the labels. We shall write $\tau \,/\, C$ for these pairs, and call them CONSTRAINED TYPES. A constrained type plays roughly the same role as a type expression in the system in chapter 3: the constrained type

$$\tau_1 \multimap\langle u\rangle\to \tau_2 \,/\, C$$

corresponds, in the type algebra of chapter 3, to the direct type

$$\tau_1 \multimap\langle \sigma_1, \ldots, \sigma_n, u\rangle\to \tau_2$$

where $\sigma_1 \ldots \sigma_n$ are the type schemes associated with u in C (that is, those type schemes σ such that the constraint $\sigma \triangleleft u$ appears in C).

This indirect representation for types turns out to be more powerful than the direct representation in several ways. First of all, closure types can naturally be recursive: in a constraint $\sigma \triangleleft u$, the label u can appear again in σ. For instance, the constrained type

$$\tau_1 \multimap\langle v\rangle\rightarrow \tau_2 \;/\; \tau_1 \multimap\langle v\rangle\rightarrow \tau_2 \triangleleft v$$

represents the infinite type that was required in the eta-expansion example,

$$\tau_1 \multimap\langle \tau_1 \multimap\langle \tau_1 \multimap\langle \ldots, v\rangle\rightarrow \tau_2,\ v\rangle\rightarrow \tau_2,\ v\rangle\rightarrow \tau_2.$$

Also, this representation syntactically ensures that the closure types are always homogeneous, in the sense of section 3.4.2. This simplifies the proofs of the type inference algorithms: there is no need to manipulate sorts anymore.

Context. This representation was suggested to me by Didier Rémy, at the beginning of 1990. It seems to be inspired by the treatment of subtyping hypothesis in type inference systems with subtypes [65, 28]. Since its publication in [52], this indirect representation of type has been applied to the type inference problem for effect systems [90, 100]. □

4.2.1.2 Generic types instead of type schemes

Second difference between the system presented below and the one previously seen: type schemes are now represented by marking specially the variables that are to be generalized, instead of universally quantifying them. We therefore partition the variables in two classes, the generic variables (written with a g subscript), and the non-generic variables (written with a n subscript). Type schemes are represented by generic types, still written σ: types that can contain generic variables. Simple types are represented by non-generic types, still written τ: types whose variables are all non-generic. The instantiation operation consists in substitution the generic variables of a generic type by non-generic types, resulting in a non-generic type. The generalization operation consists in renaming non-generic variables into generic variables.

Example. The simple type $\alpha \times \texttt{int}$, in the old notation, becomes the non-generic type $\tau = \alpha_n \times \texttt{int}$. The schema $\forall\beta.\, \alpha \times \beta$ becomes the generic type $\sigma = \alpha_n \times \beta_g$. It is true that τ is an instance of σ, by substitution of β_g by $\texttt{int}$. □

Context. This representation is used (with minor differences) in some implementations of the Damas-Milner algorithm [11]. It makes it possible to have a common representation for types and for schemes, and to determine locally whether a variable is generic or not. In his thesis [78, chapter 3], Didier Rémy reformulates the ML typing in terms of generic/non-generic types, and show the equivalence of this formulation with the classical formulation in terms of simple types/type schemes. □

The strength of this representation for schemes is that the generic variables have global, unlimited scope, while universally quantified variables are generic only in the scope of the $\forall$. This

limited scope of quantification is problematic when closure types are represented indirectly. On the one hand, it matters that the constraints describing the contents of the labels be shared between all types. This requires a typing judgement of the form $(E \vdash a : \tau) \,/\, C$, where the constraints in C apply both to the type τ and to the type schemes contained in E. On the other hand, some constraints must be considered as being parts of one of the schemes in E, and therefore must contain generalized variables. But these constraint fall out of the scope of the universal quantifiers that could appear in E.

The use of generic variables solves this difficulty: a given generic variable can appear both in a generic type in E and in the current constraint set C; when we take an instance of this generic type, we substitution the generic variable by a non-generic type both in the generic type and in the constraint set C.

Context. It is possible to combine universal quantification with indirect types, by considering type schemes of the form $\forall \alpha_1 \ldots \alpha_n . (\tau \,/\, C)$, where C is a local set of constraints over the universally quantified variables, that is added to the global constraint set when taking an instance of the schema. That's the approach taken in a first version of the work presented here [52]. However, these local constraints raise technical difficulties when we try to avoid the variable capture phenomenon. □

4.2.2 The type algebra

4.2.2.1 Type variables

We assume given four infinite sets of type variables:

t_n	$\in$	`VarTypExpNongen`	non-generic type variable
t_g	$\in$	`VarTypExpGen`	generic type variable
u_n	$\in$	`EtiqNongen`	non-generic labels
u_g	$\in$	`EtiqGen`	generic labels

We name the pairwise unions of these sets of variables as follows:

t	$\in$	`VarTypExp` $=$ `VarTypExpNongen` $\cup$ `VarTypExpGen` type variables, generic or not
u	$\in$	`Etiq` $=$ `EtiqNongen` $\cup$ `EtiqGen` labels, generic or not
α_n	$\in$	`VarTypNongen` $=$ `VarTypExpNongen` $\cup$ `EtiqNongen` non-generic variables (type variables or labels)
α_g	$\in$	`VarTypGen` $=$ `VarTypExpGen` $\cup$ `EtiqGen` generic variables (type variables or labels)

4.2.2.2 Type expressions

The set `TypGen` of GENERIC TYPES (written σ) is defined by the following grammar.

σ	::=	ι	base type
	$\mid$	t	type variable (generic or not)
	$\mid$	$\sigma_1 \multimap\!\langle u \rangle\!\rightarrowtail \sigma_2$	labeled function type
	$\mid$	$\sigma_1 \times \sigma_2$	product type
	$\mid$	σ `ref`	reference type
	$\mid$	σ `chan`	channel type
	$\mid$	σ `cont`	continuation type

The set `TypNongen` of NON-GENERIC TYPES (written τ) is the subset of the generic types that do not contain any generic type variable. This subset is described by the following grammar.

τ	::=	ι	base type
	$\mid$	t_n	non-generic variable
	$\mid$	$\tau_1 \multimap\!\langle u_n \rangle\!\rightarrowtail \tau_2$	labeled function type
	$\mid$	$\tau_1 \times \tau_2$	product type
	$\mid$	τ `ref`	reference type
	$\mid$	τ `chan`	channel type
	$\mid$	τ `cont`	continuation type

The CONSTRAINTS are pairs of a generic type and a label (generic or not). They are written $\sigma \triangleleft u$ (read: "σ is in u"). Sets of constraint are written C.

$$C \quad ::= \quad \{\sigma_1 \triangleleft u_1, \ldots, \sigma_n \triangleleft u_n\} \quad \text{constraint sets}$$

4.2.2.3 Substitutions

The substitutions over this type algebra are finite mappings of type variables to types, and of labels to labels:

$$\text{Substitutions:} \quad \varphi, \psi \quad ::= \quad [t \mapsto \sigma, \ldots, u \mapsto u', \ldots]$$

We will often need to precise the domain or the range of a substitution. To this end, we write $\varphi : V \Rightarrow T$ to express that φ is a substitution whose domain is included in the set of variables V, and whose codomain is included in the set of types and labels T. Similarly, we write $\varphi : V \Leftrightarrow V'$ to express that φ is a renaming of all variables in V into variables in V'. A RENAMING is an injective substitution whose codomain contains only variables. Any renaming $\varphi : V \Leftrightarrow V'$ has an inverse, written φ^{-1}.

A substitution φ naturally extends to a morphism $\overline{\varphi}$ of types and labels, in the following way:

$$\begin{array}{rcl}
\overline{\varphi}(\alpha) & = & \varphi(\alpha) \text{ if } \alpha \in \mathrm{Dom}(\varphi) \\
\overline{\varphi}(\alpha) & = & \alpha \text{ if } \alpha \notin \mathrm{Dom}(\varphi) \\
\overline{\varphi}(\iota) & = & \iota \\
\overline{\varphi}(\sigma_1 \multimap\langle u\rangle\rightarrowtail \sigma_2) & = & \overline{\varphi}(\sigma_1) \multimap\langle \overline{\varphi}(u)\rangle\rightarrowtail \overline{\varphi}(\sigma_2) \\
\overline{\varphi}(\sigma_1 \times \sigma_2) & = & \overline{\varphi}(\sigma_1) \times \overline{\varphi}(\sigma_2) \\
\overline{\varphi}(\sigma\ \texttt{ref}) & = & \overline{\varphi}(\sigma)\ \texttt{ref} \\
\overline{\varphi}(\sigma\ \texttt{chan}) & = & \overline{\varphi}(\sigma)\ \texttt{chan} \\
\overline{\varphi}(\sigma\ \texttt{cont}) & = & \overline{\varphi}(\sigma)\ \texttt{cont}
\end{array}$$

For constraint sets, we take:

$$\overline{\varphi}(C) = \{(\overline{\varphi}(\sigma) \triangleleft \overline{\varphi}(u)) \mid (\sigma \triangleleft u) \in C\}.$$

From now on, we do not distinguish anymore φ from its extension $\overline{\varphi}$, and we write φ for both.

4.2.2.4 Free variables, dangerous variables

Let σ be a generic type. We define the set $\mathcal{F}(\sigma)$ of variables directly free in the type σ, and the set $\mathcal{D}(\sigma)$ of variables directly dangerous in the type σ as follows.

$$\begin{array}{rcl}
\mathcal{F}(\iota) & = & \emptyset \\
\mathcal{F}(t) & = & \{t\} \\
\mathcal{F}(\sigma_1 \multimap\langle u\rangle\rightarrowtail \sigma_2) & = & \mathcal{F}(\sigma_1) \cup \mathcal{F}(u) \cup \mathcal{F}(\sigma_2) \\
\mathcal{F}(\sigma_1 \times \sigma_2) & = & \mathcal{F}(\sigma_1) \cup \mathcal{F}(\sigma_2) \\
\mathcal{F}(\sigma\ \texttt{ref}) & = & \mathcal{F}(\sigma) \\
\mathcal{F}(\sigma\ \texttt{chan}) & = & \mathcal{F}(\sigma) \\
\mathcal{F}(\sigma\ \texttt{cont}) & = & \mathcal{F}(\sigma) \\
\mathcal{F}(u) & = & \{u\}
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{D}(\iota) & = & \emptyset \\
\mathcal{D}(t) & = & \emptyset \\
\mathcal{D}(\sigma_1 \multimap\langle u\rangle\rightarrowtail \sigma_2) & = & \mathcal{D}(u) \\
\mathcal{D}(\sigma_1 \times \sigma_2) & = & \mathcal{D}(\sigma_1) \cup \mathcal{D}(\sigma_2) \\
\mathcal{D}(\sigma\ \texttt{ref}) & = & \mathcal{F}(\sigma) \\
\mathcal{D}(\sigma\ \texttt{chan}) & = & \mathcal{F}(\sigma) \\
\mathcal{D}(\sigma\ \texttt{cont}) & = & \mathcal{F}(\sigma) \\
\mathcal{D}(u) & = & \emptyset
\end{array}$$

Here is how the directly free and directly dangerous variables evolve when a substitution is applied.

Proposition 4.1 *Let σ be a generic type, and φ be a substitution. We have:*

$$\begin{array}{rcl}
\mathcal{F}(\varphi(\sigma)) & = & \displaystyle\bigcup_{\alpha\in\mathcal{F}(\sigma)} \mathcal{F}(\varphi(\alpha)) \\
\mathcal{D}(\varphi(\sigma)) & \supseteq & \displaystyle\bigcup_{\alpha\in\mathcal{D}(\sigma)} \mathcal{F}(\varphi(\alpha)) \\
\mathcal{D}(\varphi(\sigma)) & \subseteq & \left(\displaystyle\bigcup_{\alpha\in\mathcal{D}(\sigma)} \mathcal{F}(\varphi(\alpha))\right) \cup \left(\displaystyle\bigcup_{\alpha\in\mathcal{F}(\sigma)} \mathcal{D}(\varphi(\alpha))\right)
\end{array}$$

The same results obviously hold with the type σ replaced by a label u.

Proof: easy induction over σ. □

We write $\mathcal{F}_g$ for the free generic variables, $\mathcal{F}_n$ for the free non-generic variables, $\mathcal{D}_g$ for the dangerous generic variables, and $\mathcal{D}_n$ for the dangerous non-generic variables:

$$\begin{array}{rclcrcl}
\mathcal{F}_g(\sigma) & = & \mathcal{F}(\sigma) \cap \texttt{VarTypGen} & \qquad & \mathcal{D}_g(\sigma) & = & \mathcal{D}(\sigma) \cap \texttt{VarTypGen} \\
\mathcal{F}_n(\sigma) & = & \mathcal{F}(\sigma) \cap \texttt{VarTypNongen} & & \mathcal{D}_n(\sigma) & = & \mathcal{D}(\sigma) \cap \texttt{VarTypNongen}
\end{array}$$

The definitions of $\mathcal{F}$ and $\mathcal{D}$ given above do not take into account the constraints over the labels appearing in the type τ. In the following, we also need a notion of free or dangerous variable in a constrained type $\sigma \,/\, C$, either directly, either indirectly through a constraint. We will call these variables recursively free or recursively dangerous in $\sigma \,/\, C$, by contrast with the directly free or directly dangerous variables in σ defined above.

The intuition is that a label u should be considered as a node whose sons are the generic types attached to u in the constraint set C. In particular, to make closure typing work as intended, we must express that a variable is recursively free (or dangerous) in $\sigma_1 \multimap\!\langle u \rangle\!\rightarrow \sigma_2 \,/\, C$ if it is recursively free (or dangerous) in $\sigma \,/\, C$, for some σ such that the constraint $\sigma \triangleleft u$ appears in C. Translating directly this intuition, we define:

$$\begin{array}{rcl}
\mathcal{F}(\iota \,/\, C) & = & \emptyset \\
\mathcal{F}(t \,/\, C) & = & \{t\} \\
\mathcal{F}(\sigma_1 \multimap\!\langle u \rangle\!\rightarrow \sigma_2 \,/\, C) & = & \mathcal{F}(\sigma_1 \,/\, C) \cup \mathcal{F}(u \,/\, C) \cup \mathcal{F}(\sigma_2 \,/\, C) \\
\mathcal{F}(\sigma_1 \times \sigma_2 \,/\, C) & = & \mathcal{F}(\sigma_1 \,/\, C) \cup \mathcal{F}(\sigma_2 \,/\, C) \\
\mathcal{F}(\sigma\ \texttt{ref} \,/\, C) & = & \mathcal{F}(\sigma \,/\, C) \\
\mathcal{F}(\sigma\ \texttt{chan} \,/\, C) & = & \mathcal{F}(\sigma \,/\, C) \\
\mathcal{F}(\sigma\ \texttt{cont} \,/\, C) & = & \mathcal{F}(\sigma \,/\, C) \\
\mathcal{F}(u \,/\, C) & = & \{u\} \cup \bigcup_{(\sigma \triangleleft u) \in C} \mathcal{F}(\sigma \,/\, C)
\end{array}$$

Similarly for dangerous variables:

$$\begin{array}{rcl}
\mathcal{D}(\iota \,/\, C) & = & \emptyset \\
\mathcal{D}(t \,/\, C) & = & \emptyset \\
\mathcal{D}(\sigma_1 \multimap\!\langle u \rangle\!\rightarrow \sigma_2 \,/\, C) & = & \mathcal{D}(u \,/\, C) \\
\mathcal{D}(\sigma_1 \times \sigma_2 \,/\, C) & = & \mathcal{D}(\sigma_1 \,/\, C) \cup \mathcal{D}(\sigma_2 \,/\, C) \\
\mathcal{D}(\sigma\ \texttt{ref} \,/\, C) & = & \mathcal{F}(\sigma \,/\, C) \\
\mathcal{D}(\sigma\ \texttt{chan} \,/\, C) & = & \mathcal{F}(\sigma \,/\, C) \\
\mathcal{D}(\sigma\ \texttt{cont} \,/\, C) & = & \mathcal{F}(\sigma \,/\, C) \\
\mathcal{D}(u \,/\, C) & = & \bigcup_{(\sigma \triangleleft u) \in C} \mathcal{D}(\sigma \,/\, C)
\end{array}$$

The equalities above do not constitute a well-founded definition. In the case for $\mathcal{F}(u \,/\, C)$ or $\mathcal{D}(u \,/\, C)$, one of the types σ over which we take the union can contain as a sub-term a function type labeled by u. In other terms, nothing prevents the constraint set from being cyclic. Hence, the "definition" above must be understood as a set of equations, for which we must find a smallest solution. It is easy to check that the solutions are exactly the fixpoints of an increasing, bounded operator; hence there exists a smallest solution defining $\mathcal{F}(\sigma \,/\, C)$ and $\mathcal{D}(\sigma \,/\, C)$.

We are now going to give an alternate characterization of $\mathcal{F}$ and $\mathcal{D}$, that turns out to be more convenient for proofs. For all integers n, we define:

$$\begin{array}{rcl}
\mathcal{F}^0(\sigma) & = & \emptyset \\
\mathcal{F}^{n+1}(\iota \,/\, C) & = & \emptyset \\
\mathcal{F}^{n+1}(t \,/\, C) & = & \{t\} \\
\mathcal{F}^{n+1}(\sigma_1 \multimap\!\langle u\rangle\!\rightarrow \sigma_2 \,/\, C) & = & \mathcal{F}^n(\sigma_1 \,/\, C) \cup \mathcal{F}^n(u \,/\, C) \cup \mathcal{F}^n(\sigma_2 \,/\, C) \\
\mathcal{F}^{n+1}(\sigma_1 \times \sigma_2 \,/\, C) & = & \mathcal{F}^n(\sigma_1 \,/\, C) \cup \mathcal{F}^n(\sigma_2 \,/\, C) \\
\mathcal{F}^{n+1}(\sigma \texttt{ ref} \,/\, C) & = & \mathcal{F}^n(\sigma \,/\, C) \\
\mathcal{F}^{n+1}(\sigma \texttt{ chan} \,/\, C) & = & \mathcal{F}^n(\sigma \,/\, C) \\
\mathcal{F}^{n+1}(\sigma \texttt{ cont} \,/\, C) & = & \mathcal{F}^n(\sigma \,/\, C) \\
\mathcal{F}^n(u \,/\, C) & = & \{u\} \cup \bigcup_{(\sigma \lhd u) \in C} \mathcal{F}^n(\sigma \,/\, C)
\end{array}$$

For dangerous variables, we take:

$$\begin{array}{rcl}
\mathcal{D}^0(\sigma) & = & \emptyset \\
\mathcal{D}^{n+1}(\iota \,/\, C) & = & \emptyset \\
\mathcal{D}^{n+1}(t \,/\, C) & = & \emptyset \\
\mathcal{D}^{n+1}(\sigma_1 \multimap\!\langle u\rangle\!\rightarrow \sigma_2 \,/\, C) & = & \mathcal{D}^n(u \,/\, C) \\
\mathcal{D}^{n+1}(\sigma_1 \times \sigma_2 \,/\, C) & = & \mathcal{D}^n(\sigma_1 \,/\, C) \cup \mathcal{D}^n(\sigma_2 \,/\, C) \\
\mathcal{D}^{n+1}(\sigma \texttt{ ref} \,/\, C) & = & \mathcal{F}(\sigma \,/\, C) \\
\mathcal{D}^{n+1}(\sigma \texttt{ chan} \,/\, C) & = & \mathcal{F}(\sigma \,/\, C) \\
\mathcal{D}^{n+1}(\sigma \texttt{ cont} \,/\, C) & = & \mathcal{F}(\sigma \,/\, C) \\
\mathcal{D}^n(u \,/\, C) & = & \bigcup_{(\sigma \lhd u) \in C} \mathcal{D}^n(\sigma \,/\, C)
\end{array}$$

Proposition 4.2 *For all constrained types $\sigma \,/\, C$, we have:*

$$\mathcal{F}(\sigma \,/\, C) = \bigcup_{n \geq 0} \mathcal{F}^n(\sigma \,/\, C) \qquad \mathcal{D}(\sigma \,/\, C) = \bigcup_{n \geq 0} \mathcal{D}^n(\sigma \,/\, C).$$

Proof: we first show that $\bigcup_{n\geq 0} \mathcal{F}^n(\sigma \,/\, C)$ and $\bigcup_{n\geq 0} \mathcal{D}^n(\sigma \,/\, C)$ satisfy the equations whose smallest solutions are $\mathcal{F}$ and $\mathcal{D}$. This establishes the inclusion $\subseteq$. For the converse inclusion, let $\mathcal{F}'$ and $\mathcal{D}'$ be any solutions of these equations. We show that $\mathcal{F}^n(\sigma/C) \subseteq \mathcal{F}'(\sigma/C)$ and that $\mathcal{D}^n(\sigma/C) \subseteq \mathcal{D}'(\sigma/C)$ for all n and for all σ, by induction over n. This establishes the converse inclusions, and the equalities claimed above. □

Here is the usual technical lemma that describes the effect of a substitution over the recursively free and dangerous variables in a type.

Proposition 4.3 *Let σ be a generic type, C a set of constraint, and φ a substitution. We have:*

$$\begin{array}{rcl}
\mathcal{F}(\varphi(\sigma) \,/\, C) & \supseteq & \bigcup_{\alpha \in \mathcal{F}(\sigma/C)} \mathcal{F}(\varphi(\alpha)) \\
\mathcal{F}(\varphi(\sigma) \,/\, C) & = & \bigcup_{\alpha \in \mathcal{F}(\sigma)} \mathcal{F}(\varphi(\alpha) \,/\, C)
\end{array}$$

$$\mathcal{D}(\varphi(\sigma) \,/\, C) \supseteq \bigcup_{\alpha \in \mathcal{D}(\sigma)} \mathcal{F}(\varphi(\alpha) \,/\, C)$$

$$\mathcal{D}(\varphi(\sigma) \,/\, C) \subseteq \left(\bigcup_{\alpha \in \mathcal{D}(\sigma)} \mathcal{F}(\varphi(\alpha) \,/\, C) \right) \cup \left(\bigcup_{\alpha \in \mathcal{F}(\sigma)} \mathcal{D}(\varphi(\alpha) \,/\, C) \right)$$

Proof: for the first line, we show the corresponding inclusion for $\mathcal{F}^n$, by induction over n, and we conclude by proposition 4.2. For the last three lines, we proceed by structural induction over σ, and application of the equalities defining $\mathcal{D}$ and $\mathcal{F}$. □

4.2.3 Equivalence between constrained types

The presentation of types in two parts (a type expression plus a set of constraints) has one drawback: since the constraint set is shared with other types (e.g. the types in the typing environment), it can contain some constraints that do not concern any of the labels reachable fron the type expression. These extra constraints do not change the semantic properties of the type; but they can change its syntactic properties. For instance, if we substitute in $\tau \,/\, C$ a type variable α that is not free in $\tau \,/\, C$, we get a constrained type $\tau \,/\, C'$, where C' may differ from C: the variable α can indeed occur in C, inside constraints over labels that are not reachable from τ.

Example. After performing $t \mapsto \texttt{int}$ in the type $\texttt{int} \multimap\langle u \rangle\rightarrow \texttt{int} \,/\, t \triangleleft v$, we get $\texttt{int} \multimap\langle u \rangle\rightarrow \texttt{int} \,/\, \texttt{int} \triangleleft v$, which is syntactically different from the original type. □

We are now going to introduce an equivalence between constrained types that captures the fact that two constrained types, while syntactically different, describe the same graph. Let σ be a type and C be a constraint set. We define the CONNECTED COMPONENT of σ in C, written $C \restriction \sigma$, as the set of those constraints in C that concern a label which is free in $\sigma \,/\, C$:

$$C \restriction \sigma = \{(\sigma' \triangleleft u) \in C \mid u \in \mathcal{F}(\sigma \,/\, C)\}.$$

We define similarly $C \restriction E$, the connected component of a typing environment E, and $C \restriction \{\sigma_1, \ldots, \sigma_n\}$, the connected component of a set of types.

Let $\sigma_1 \,/\, C_1$ and $\sigma_2 \,/\, C_2$ be two constrained types. We say that $\sigma_1 \,/\, C_1$ is EQUIVALENT to $\sigma_2 \,/\, C_2$, and we write $\sigma_1 \,/\, C_1 \equiv \sigma_2 \,/\, C_2$, if $\sigma_1 = \sigma_2$ and $C_1 \restriction \sigma_1 = C_2 \restriction \sigma_2$. The relation $\equiv$ is obviously an equivalence relation.

The remainder of this section is devoted to some properties of the connected components and of the relation $\equiv$.

Proposition 4.4 *For all constrained type $\sigma \,/\, C$, we have*

$$\mathcal{F}(\sigma \,/\, C) = \mathcal{F}(\sigma \,/\, (C \restriction \sigma)) \quad \textit{and} \quad \mathcal{D}(\sigma \,/\, C) = \mathcal{D}(\sigma \,/\, (C \restriction \sigma)).$$

As a consequence, if $\sigma_1 \,/\, C_1 \equiv \sigma_2 \,/\, C_2$, then $\mathcal{F}(\sigma_1 \,/\, C_1) = \mathcal{F}(\sigma_2 \,/\, C_2)$ and $\mathcal{D}(\sigma_1 \,/\, C_1) = \mathcal{D}(\sigma_2 \,/\, C_2)$.

Proof: we show the corresponding equalities for $\mathcal{F}^n$ and $\mathcal{D}^n$, for all n, by induction over n. We conclude by proposition 4.2. □

Proposition 4.5 *Let $\sigma \,/\, C$ be a constrained type, and φ a substitution. We have*

$$\varphi(C \restriction \sigma) \subseteq \varphi(C) \restriction \varphi(\sigma).$$

Proof: if a label u is free in $\sigma \,/\, C$, then the label $\varphi(u)$ is free in $\varphi(\sigma) \,/\, \varphi(C)$ (proposition 4.3). The result follows. □

The inclusion can be proper, because the substitution can identify two labels, and therefore introduce additional constraints in the connected component of a type.

Example. The connected component of $\texttt{int} \dashv\langle u\rangle\rightarrow \texttt{int}$ in $\texttt{bool} \triangleleft u, \texttt{string} \triangleleft v$ is $\texttt{bool} \triangleleft u$. But if we apply the substitution $[v \mapsto u]$, the connected component becomes $\texttt{bool} \triangleleft u, \texttt{string} \triangleleft u$. □

As a consequence, the equivalence relation between constrained types is not stable under substitution.

Example. We have

$$\texttt{int} \dashv\langle u\rangle\rightarrow \texttt{int} \,/\, \texttt{bool} \triangleleft u, \texttt{string} \triangleleft v \quad\equiv\quad \texttt{int} \dashv\langle u\rangle\rightarrow \texttt{int} \,/\, \texttt{bool} \triangleleft u, \texttt{char} \triangleleft v.$$

But if we apply the substitution $[v \mapsto u]$ on both sides, we obtain two constrained types that are no longer equivalent. □

There is however an important class of substitutions that commute with the "connected component" operation, and therefore preserve the equivalence between constrained types: the renamings.

Proposition 4.6 *For all renamings θ, we have*

$$\theta(C \restriction \sigma) = \theta(C) \restriction \theta(\sigma).$$

As a consequence, $\sigma_1 \,/\, C_1 \equiv \sigma_2 \,/\, C_2$ implies $\theta(\sigma_1) \,/\, \theta(C_1) \equiv \theta(\sigma_2) \,/\, \theta(C_2)$.

Proof: by proposition 4.5 applied to θ and to θ^{-1}, we have

$$C \restriction \sigma = \theta^{-1}(\theta(C \restriction \sigma)) \subseteq \theta^{-1}(\theta(C) \restriction \theta(\sigma)) \subseteq \theta^{-1}(\theta(C)) \restriction \theta^{-1}(\theta(\sigma)) = C \restriction \sigma.$$

Hence $\theta^{-1}(\theta(C) \restriction \theta(\sigma)) = C \restriction \sigma$, and the desired equality by applying θ on both sides. □

In chapters 1 and 3, we made heavy use of the following property, which holds in all usual type algebras: if $\mathcal{F}(\tau)$ and $\mathrm{Dom}(\varphi)$ are disjoint, then $\varphi(\tau)$ is identical to τ. This property does not hold for the type algebra in this chapter: we can have $\mathcal{F}(\sigma \,/\, C)$ and $\mathrm{Dom}(\varphi)$ disjoint, yet $\varphi(\sigma) \,/\, \varphi(C)$ is not equivalent to $\sigma \,/\, C$. That's because the substitution φ can "graft" additional constraints over labels that are free in $\sigma \,/\, C$.

Example. . Consider the type $\sigma = \texttt{int} \dashv\langle u\rangle\rightarrow \texttt{int}$ under the constraints $C = \texttt{bool} \triangleleft v$. The label v is not free in $\sigma \,/\, C$. Nonetheless, after substituting v by u, we get the constrained type $\texttt{int} \dashv\langle u\rangle\rightarrow \texttt{int} \,/\, \texttt{bool} \triangleleft u$, which is not equivalent to $\sigma \,/\, C$. □

However, the expected property holds if the substitution is a renaming.

Proposition 4.7 *Let θ be a renaming and $\sigma \,/\, C$ be a constrained type. If* $\mathrm{Dom}(\theta) \cap \mathcal{F}(\sigma \,/\, C) = \emptyset$, *then* $\theta(\sigma) \,/\, \theta(C) \equiv \sigma \,/\, C$.

Proof: we have $\theta(\sigma) = \sigma$, since, a fortiori, $\mathrm{Dom}(\theta) \cap \mathcal{F}(\sigma) = \emptyset$. Moreover, by proposition 4.6, $\theta(C) \restriction \theta(\sigma) = \theta(C \restriction \sigma)$. Let $\sigma' \triangleleft u$ be a constraint of $C \restriction \sigma$. All variables free in $\sigma' \triangleleft u$ are free in $\sigma \,/\, C$, and therefore invariant by θ. Hence $\theta(\sigma' \triangleleft u) = \sigma' \triangleleft u$. It follows that $\theta(C) \restriction \theta(\sigma) = C \restriction \sigma$. Hence the expected result. □

4.2.4 Typing rules

In the indirect system, the typing predicate has the form $E \vdash a : \tau / C$, read: "in the environment E, the expression a has the non-generic type τ, relative to the constraints over labels appearing in C". The environment E is a finite mapping from identifiers to generic types. The generic variables in $E(x)$ are considered universally quantified.

The typing rules are essentially a reformulation of the rules in section 3.2.3, with direct types replaced by indirect types plus a constraint set, and the type schemes replaced by generic types. The main differences are: first, the generalization step in the `let` rule, and second, the introduction of a simplification rule over constraint sets.

$$\frac{\tau \leq E(x) \,/\, C}{E \vdash x : \tau \,/\, C}$$

The instantiation relation between a generic type and a non-generic type is defined as follows: $\tau \leq \sigma \,/\, C$ if and only if there exists a substitution $\varphi : \mathcal{F}_g(E(x)) \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$ such that $\varphi(\sigma) = \tau$ and $\varphi(C) \subseteq C$. Instantiating a generic type therefore corresponds to a substitution of the generic variables of this type by non-generic types and labels. This substitution is also applied to the constraints: C can contain constraints over generic labels that appear in σ. The condition $\varphi(C) \subseteq C$ expresses the fact that C correctly keeps track of those constraints after the instantiation.

$$\frac{\begin{array}{c} E + f \mapsto (\tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2 \,/\, C \\ (E(y) \triangleleft u_n) \in C \text{ for all } y \in \mathcal{I}(f \texttt{ where } f(x) = a) \end{array}}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2 \,/\, C}$$

The typing rule for functions requires that the label u_n be associated in C to the types of the identifiers free in the function.

$$\frac{E \vdash a_1 : \tau_2 \multimap\langle u_n\rangle\rightarrow \tau_1 \,/\, C \qquad E \vdash a_2 : \tau_2 \,/\, C}{E \vdash a_1(a_2) : \tau_1 \,/\, C} \qquad \frac{E \vdash a_1 : \tau_1 \,/\, C \qquad E \vdash a_2 : \tau_2 \,/\, C}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2 \,/\, C}$$

$$\frac{E \vdash a_1 : \tau_1 \,/\, C_1 \qquad (\sigma, C) = \texttt{Gen}(\tau_1, C_1, E) \qquad E + x \mapsto \sigma \vdash a_2 : \tau_2 \,/\, C}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2 \,/\, C}$$

The generalization relation is defined as follows: $(\sigma, C) = \mathtt{Gen}(\tau_1, C_1, E)$ if and only if there exists a renaming θ of $\mathcal{F}(\tau_1 \,/\, C_1) \setminus \mathcal{D}(\tau_1 \,/\, C_1) \setminus \mathcal{F}_n(E) \setminus \mathcal{D}(E \,/\, C_1)$ to generic variables that are not free in E nor in C_1 such that $\sigma = \theta(\tau_1)$ and $C = \theta(C_1)$.

In other terms, the domain of θ is composed of those non-generic variables in the type τ_1 that we want to generalize. To do so, we rename these variables to fresh generic variables. The renaming is also applied to the current constraint set.

In contrast with the system in chapter 3, we now allow the generalization of variables that are recursively free in $E \,/\, C_1$, provided they are neither directly free in E, nor dangerous in $E \,/\, C_1$. In other terms, we ignore the non-dangerous variables that are captured by the closure types in E. (See the example below.)

The rules for constants and operators are classical:

$$\frac{\tau \leq \mathtt{TypCst}(cst) \,/\, C}{E \vdash cst : \tau \,/\, C}$$

$$\frac{\tau_1 \multimap\langle u_n \rangle\rightarrow \tau_2 \leq \mathtt{TypOp}(op) \,/\, C \qquad E \vdash a : \tau_1 \,/\, C}{E \vdash op(a) : \tau_2 \,/\, C}$$

The last typing rule is a simplification rule, that allows removing constraints that are useless in the typing derivation, and replacing these constraints by similar constraints:

$$\frac{E' \vdash a : \tau' \,/\, C' \qquad \tau \,/\, C \equiv \tau' \,/\, C' \qquad E\,|_{\mathcal{I}(a)} \,/\, C \equiv E'\,|_{\mathcal{I}(a)} \,/\, C'}{E \vdash a : \tau \,/\, C}$$

The intuition behind this rule is that only those constraints that belong to the connected component of τ and of the restriction of E to the identifiers occurring free in a are relevant in the typing of the expression. The remaining constraints can freely be erased, or replaced by other unconnected constraints.

The types of the primitive operators over references, channels and continuations are those of section 3.2.3, rewritten with labels instead of expansion variables, and generic variables instead of quantified variables.

$$\begin{aligned}
\mathtt{TypOp(ref)} &= t_g \multimap\langle u_g \rangle\rightarrow t_g\ \mathtt{ref} \\
\mathtt{TypOp(!)} &= t_g\ \mathtt{ref} \multimap\langle u_g \rangle\rightarrow t_g \\
\mathtt{TypOp(:=)} &= t_g\ \mathtt{ref} \times t_g \multimap\langle u_g \rangle\rightarrow \mathtt{unit} \\
\mathtt{TypOp(newchan)} &= \mathtt{unit} \multimap\langle u_g \rangle\rightarrow t_g\ \mathtt{chan} \\
\mathtt{TypOp(?)} &= t_g\ \mathtt{chan} \multimap\langle u_g \rangle\rightarrow t_g \\
\mathtt{TypOp(!)} &= t_g\ \mathtt{chan} \times t_g \multimap\langle u_g \rangle\rightarrow \mathtt{unit} \\
\mathtt{TypOp(callcc)} &= (t_g\ \mathtt{cont} \multimap\langle u_g \rangle\rightarrow t_g) \multimap\langle u'_g \rangle\rightarrow t_g \\
\mathtt{TypOp(throw)} &= t_g\ \mathtt{cont} \times t_g \multimap\langle u_g \rangle\rightarrow t'_g
\end{aligned}$$

Example. Consider again the non-conservativity example from section 4.1.2.

$$\lambda \mathtt{f.\ let\ id} = \lambda \mathtt{y.\ either(f)}(\lambda \mathtt{z.y;z});\ \mathtt{y\ in\ id(id)}$$

The principal typing for the left part of the `let` results in:

$$[\mathtt{f} : t_n \multimap\!\langle u_n \rangle\!\rightarrowtail t_n] \vdash (\lambda \mathtt{y.\ either(f)}(\lambda \mathtt{z.y;z});\mathtt{y}) : t'_n \multimap\!\langle u'_n \rangle\!\rightarrowtail t'_n \ /\ t'_n \triangleleft u_n,\ (t_n \multimap\!\langle u_n \rangle\!\rightarrowtail t_n) \triangleleft u'_n.$$

The variables that can be generalized are t'_n and u'_n. That's because t'_n is not directly free in the typing environment, though it is free in a constraint reachable from that environment. Hence we can generalize t'_n to t_g and u'_n to u_g, and the self-application `id(id)` is well-typed. □

4.2.5 Properties of the typing rules

In this section, we show that the typing predicate is, under certain conditions, stable by substitution of type variables and by addition of constraints.

Proposition 4.8 (Typing is stable under substitution) *Let a be an expression, τ be a non-generic type, E be a typing environment, C be a constraint set. For all substitutions $\varphi : \mathtt{VarNongen} \Rightarrow \mathtt{TypNongen} \cup \mathtt{EtiqNongen}$ of non-generic types for non-generic variables, if we have $E \vdash a : \tau \ / \ C$, then we have $\varphi(E) \vdash a : \varphi(\tau) \ / \ \varphi(C)$.*

Proposition 4.9 (Typing is stable under addition of constraints) *Let a be an expression, τ be a non-generic type, E be a typing environment, C be a constraint set such that $E \vdash a : \tau \ / \ C$. Let C' be a constraint set that does not contain any of the generic variables free in $E(y)$ for all y free in a. That is:*

$$(\{u\} \cup \mathcal{F}(\sigma)) \cap \mathcal{F}_g(E(y)) = \emptyset \text{ for all } (\sigma \triangleleft u) \in C',\ y \in \mathcal{I}(a).$$

Then, $E \vdash a : \tau \ / \ C \cup C'$.

The two propositions are proved simultaneously, by induction on the height of the derivation of $E \vdash a : \tau \ / \ C$, and by case analysis on the last rule employed. Since the proof is lengthy, I prefer to present it as follows: I prove each proposition separately, assuming that the other holds. I also show that the derivations obtained (of $\varphi(E) \vdash a : \varphi(\tau) \ / \ \varphi(C)$ for proposition 4.8, of $E \vdash a : \tau \ / \ C \cup C'$ for proposition 4.9) have the same height as the initial derivation (of $E \vdash a : \tau \ / \ C$). This ensures that the simultaneous induction is well-founded, because the other proposition is always applied to derivations strictly smaller than the original derivation.

Proof: for proposition 4.8. We easily check on each case that the derivation of $\varphi(E) \vdash a : \varphi(\tau) \ / \ \varphi(C)$ built has the same height as the initial derivation of $E \vdash a : \tau \ / \ C$.

• **Instantiation rule.** We have $a = x$ and $\tau \leq E(x) \ / \ C$. Let $\psi : \mathcal{F}_g(E(x)) \Rightarrow \mathtt{TypNongen} \cup \mathtt{EtiqNongen}$ be a substitution such that $\psi(E(x)) = \tau$ and $\psi(C) \subseteq C$. We define a substitution θ by $\theta(\alpha_g) = \varphi(\psi(\alpha_g))$ for all generic variables α_g. We have:

$$\begin{array}{lllll} \theta(\varphi(\alpha_g)) & = & \theta(\alpha_g) & = & \varphi(\psi(\alpha_g)) \quad \text{for all generic variables } \alpha_g \\ \theta(\varphi(\alpha_n)) & = & \varphi(\alpha_n) & = & \varphi(\psi(\alpha_n)) \quad \text{for all non-generic variables } \alpha_n \end{array}$$

Hence $\theta \circ \varphi = \varphi \circ \psi$. As a consequence, $\theta(\varphi(E(x))) = \varphi(\psi(E(x))) = \varphi(\tau)$, and similarly $\theta(\varphi(C)) = \varphi(\psi(C)) \subseteq \varphi(C)$. Finally, since φ operates only on non-generic variables and does not introduce fresh generic variables, we have $\mathcal{F}_g(\varphi(E(x))) = \mathcal{F}_g(E(x))$, and therefore $\theta : \mathcal{F}_g(\varphi(E(x))) \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$. This establishes $\varphi(\tau) \leq \varphi(E(x)) \,/\, \varphi(C)$. Hence we can derive $\varphi(E) \vdash x : \varphi(\tau) \,/\, \varphi(C)$.

• **The `where` rule.** The derivation ends up with:

$$\frac{E + f \mapsto (\tau_1 \dashv\langle u_n \rangle\rightarrow \tau_2) + x \mapsto \tau_1 \vdash a : \tau_2 \,/\, C \qquad (E(y) \triangleleft u_n) \in C \text{ for all } y \in \mathcal{I}(f \texttt{ where } f(x) = a)}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \dashv\langle u_n \rangle\rightarrow \tau_2 \,/\, C}$$

By induction hypothesis applied to a, we get a proof of:

$$\varphi(E + f \mapsto (\tau_1 \dashv\langle u_n \rangle\rightarrow \tau_2) + x \mapsto \tau_1) \vdash a : \varphi(\tau_2) \,/\, \varphi(C).$$

By definition of substitution over constraints, we have $(\varphi(\sigma) \triangleleft \varphi(u_n)) \in \varphi(C)$ as soon as $(\sigma \triangleleft u_n) \in C$. This holds in particular if σ is $E(y)$ for some y free in a. Hence we can derive the expected result:

$$\varphi(E) \vdash (f \texttt{ where } f(x) = a) : \varphi(\tau_1) \dashv\langle \varphi(u_n) \rangle\rightarrow \varphi(\tau_2) \,/\, \varphi(C).$$

• **The `let` rule.** The derivation ends up with:

$$\frac{E \vdash a_1 : \tau_1 \,/\, C_1 \qquad (\sigma, C) = \texttt{Gen}(\tau_1, C_1, E) \qquad E + x \mapsto \sigma \vdash a_2 : \tau_2 \,/\, C}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2 \,/\, C}$$

Define $\{\alpha_1 \ldots \alpha_k\} = \mathcal{F}(\tau \,/\, C_1) \setminus \mathcal{D}(\tau \,/\, C_1) \setminus \mathcal{F}_n(E) \setminus \mathcal{D}(E \,/\, C_1)$. (The α_i are non-generic. The n subscript is omitted for the sake of readability.) Let $\theta : \{\alpha_1 \ldots \alpha_k\} \Leftrightarrow \texttt{VarTypGen}$ be the renaming such that $\sigma = \theta(\tau_1)$ and $C = \theta(C_1)$. Let $\beta_1 \ldots \beta_k$ be non-generic variables, pairwise distinct, not free in E, not free in C_1, and out of reach for φ, with β_i of the same kind as α_i for all i. Define the substitution

$$\psi = \varphi \circ [\alpha_1 \mapsto \beta_1 \ldots \alpha_k \mapsto \beta_k].$$

We have $\psi(E) = \varphi(E)$, since the α_i are not free in E. We apply twice the induction hypothesis, once to the left premise, with the substitution φ, and once to the right premise, with the substitution φ. We obtain proofs of:

$$\psi(E) \vdash a_1 : \psi(\tau_1) \,/\, \psi(C_1) \qquad\qquad \varphi(E) + x \mapsto \varphi(\sigma) \vdash a_2 : \varphi(\tau_2) \,/\, \varphi(C).$$

Take $V = \mathcal{F}(\psi(\tau_1) \,/\, \psi(C_1)) \setminus \mathcal{D}(\psi(\tau_1) \,/\, \psi(C_1)) \setminus \mathcal{F}_n(\psi(E)) \setminus \mathcal{D}(\psi(E) \,/\, \psi(C_1))$. We are now going to show that $V = \{\beta_1 \ldots \beta_k\}$. By construction of ψ and of the β_i, we have $\psi(\alpha_i) = \beta_i$, and $\beta_i \in \mathcal{F}(\psi(\alpha))$ for some variable α implies $\alpha = \alpha_i$.

We now fix i. Since α_i is free in $\tau_1 \,/\, C_1$, we have β_i free in $\psi(\tau_1) \,/\, \psi(C_1)$ (prop. 4.3, case 1). Since α_i is not free in E, we have $\beta_i \notin \mathcal{F}_n(\psi(E))$. Otherwise, we would have $\beta_i \in \mathcal{F}(\psi(\alpha))$ for some $\alpha \in \mathcal{F}(E)$ (prop 4.1, case 1); but by definition of ψ, this α can only be α_i, which is not free in E.

Moreover, $\beta_i \notin \mathcal{D}(\psi(\tau_1) \,/\, \psi(C_1))$. Otherwise, we would have either $\beta_i \in \mathcal{F}(\psi(\alpha) \,/\, \psi(C_1))$ for some α in $\mathcal{D}(\tau_1)$, or $\beta_i \in \mathcal{D}(\psi(\alpha) \,/\, \psi(C_1))$ for some α in $\mathcal{F}(\tau_1)$ (prop 4.3, case 4). The first possibility is excluded, since by construction of the β_i, it implies $\alpha = \alpha_i$; but α_i is not dangerous in $\tau_1 \,/\, C_1$, hence a fortiori it is not directly dangerous in τ_1. The other possibility implies similarly $\alpha = \alpha_i$, but

$$\mathcal{D}(\psi(\alpha_i) \,/\, \psi(C_1)) = \mathcal{D}(\beta_i \,/\, \psi(C_1)) = \emptyset$$

which is a contradiction. In the same way, we show $\beta_i \notin \mathcal{D}(\psi(E) \,/\, \psi(C_1))$. We therefore conclude that

$$\{\beta_1 \ldots \beta_k\} \subseteq V.$$

We now show the converse inclusion. Let β be a non-generic variable that is free in $\psi(\tau_1) \,/\, \psi(C_1)$, and that is not one of the β_i. Let $\alpha \in \mathcal{F}(\tau_1 \,/\, C_1)$ be a non-generic variable such that $\beta \in \mathcal{F}(\psi(\alpha))$. The variable α cannot be one of the α_i, since otherwise β would be one of the β_i. Hence either α is directly free in E, or α is dangerous in $\tau_1 \,/\, C_1$, or α is dangerous in $E \,/\, C_1$. If α is free in E, then β is free in $\psi(E)$ (prop 4.1, case 1). If α is dangerous in $\tau_1 \,/\, C$, then β is dangerous in $\psi(\tau_1) \,/\, \psi(C_1)$ (prop 4.3, case 3). Finally, if α is dangerous in $E \,/\, C$, then β is dangerous in $\psi(E) \,/\, \psi(C_1)$ (prop 4.3, case 3). In all three cases, $\beta \notin V$. Hence the converse inclusion.

Define the renaming $\xi = [\beta_1 \mapsto \theta(\alpha_1), \ldots, \beta_k \mapsto \theta(\alpha_k)]$. We have $\xi : \{\beta_1, \ldots, \beta_k\} \Leftrightarrow \texttt{VarTypGen}$. Moreover, for all i:

$$\begin{array}{rcll} \xi(\psi(\alpha_i)) & = & \xi(\beta_i) & \text{by definition of } \psi \\ & = & \theta(\alpha_i) & \text{by definition of } \xi \\ & = & \varphi(\theta(\alpha_i)) & \text{since } \varphi \text{ does not change generic variables.} \end{array}$$

In addition, for all variables α that are neither one of the α_i, nor one of the β_i:

$$\begin{array}{rcll} \xi(\psi(\alpha)) & = & \xi(\varphi(\alpha)) & \text{by definition of } \psi \\ & = & \varphi(\alpha) & \text{because } \alpha \notin \{\beta_1 \ldots \beta_k\} \text{ and the } \beta_i \text{ are out of reach for } \varphi \\ & = & \varphi(\theta(\alpha)) & \text{because } \theta(\alpha) = \alpha \text{ since } \alpha \notin \{\alpha_1 \ldots \alpha_k\}. \end{array}$$

Since the β_i are not free in τ_1 nor in C_1, it follows that $\xi(\psi(\tau_1)) = \varphi(\theta(\tau_1)) = \varphi(\tau)$ and $\xi(\psi(C_1)) = \varphi(\theta(C_1)) = \varphi(C)$. We have therefore established that:

$$(\varphi(\sigma), \varphi(C)) = \texttt{Gen}(\psi(\tau_1), \psi(C_1), \varphi(E)).$$

Combining this fact with the two derivations obtained by applying the induction hypothesis:

$$\varphi(E) \vdash a_1 : \psi(\tau_1) \,/\, \psi(C_1) \qquad \varphi(E) + x \mapsto \varphi(\sigma) \vdash a_2 : \varphi(\tau_2) \,/\, \varphi(C),$$

we deduce, by the `let` rule, the expected result:

$$\varphi(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \varphi(\tau_2) \,/\, \varphi(C).$$

• **Simplification rule.** The initial derivation ends up with:

$$\frac{E \vdash a : \tau \,/\, C \qquad \tau \,/\, C \equiv \tau \,/\, C' \qquad E\,|_{\,\mathcal{I}(a)} \,/\, C \equiv E'\,|_{\,\mathcal{I}(a)} \,/\, C'}{E' \vdash a : \tau \,/\, C'}$$

The proof of this case is delicate, because we do not necessarily have $\varphi(\tau) \,/\, \varphi(C) \equiv \varphi(\tau) \,/\, \varphi(C')$ and $\varphi(E \mid_{\mathcal{I}(a)}) \,/\, \varphi(C) \equiv \varphi(E' \mid_{\mathcal{I}(a)}) \,/\, \varphi(C')$. For instance, we have

$$\texttt{int} \multimap\!\langle u \rangle\!\rightarrowtail \texttt{int} \,/\, \texttt{bool} \triangleleft v \quad \equiv \quad \texttt{int} \multimap\!\langle u \rangle\!\rightarrowtail \texttt{int} \,/\, \emptyset,$$

but this equivalence does not hold anymore if v is substituted by u. We'll solve this difficulty by massive renaming. Let U be the set of the labels free in $\tau \,/\, C$ or in $E \mid_{\mathcal{I}(a)} \,/\, C$. We decompose C as $C_0 \cup C_1$ and C' as $C_0 \cup C_1'$, where C_0 constraints only labels $u_1 \ldots u_k$ belonging to U, while the labels $v_1 \ldots v_m$ constrained in C_1 and the labels $v_1' \ldots v_p'$ constrained in C_1' do not belong to U. Let $w_1 \ldots w_m$ and $w_1' \ldots w_p'$ be fresh distinct labels, out of reach for φ, and not belonging to U. Define the two substitutions θ and ψ as follows:

$$\begin{array}{rcl} \theta & = & [v_1' \mapsto w_1', \ldots, v_p' \mapsto w_p'] \\ \psi & = & \varphi + v_1 \mapsto w_1 + \cdots + v_m \mapsto w_m + w_1' \mapsto \varphi(v_1') + \cdots + w_p' \mapsto \varphi(v_p') \end{array}$$

By construction, none of the variables free in $E \mid_{\mathcal{I}(a)}$ appears in the constraints in $\theta(C_1')$. Applying proposition 4.9, we obtain a derivation of

$$E \vdash a : \tau \,/\, C \cup \theta(C_1')$$

with the same height as the derivation of $E \vdash a : \tau \,/\, C$. We can therefore apply the induction hypothesis to the derivation obtained. We get a derivation of

$$\psi(E) \vdash a : \psi(\tau) \,/\, \psi(C \cup \theta(C_1'))$$

which also has the same height as the derivation of $E \vdash a : \tau \,/\, C$. Since the v_i and the w_j' are not free in τ, we have $\psi(\tau) = \varphi(\tau)$. Similarly, $\psi(E \mid_{\mathcal{I}(a)}) = \varphi(E' \mid_{\mathcal{I}(a)})$. Finally, $\psi(C_0) = \varphi(C_0)$ since all labels free in one of the constraints of C_0 belong to U. Moreover, $\psi(\theta(C_1')) = \varphi(C_1')$ by construction. Hence:

$$\psi(C \cup \theta(C_1')) = \psi(C_0 \cup C_1 \cup \theta(C_1')) = \varphi(C_0) \cup \psi(C_1) \cup \varphi(C_1') = \varphi(C') \cup \psi(C_1).$$

Notice also that the constraints in $\psi(C_1)$ only concern the labels $w_1 \ldots w_m$, that are not free in $\varphi(\tau) \,/\, \varphi(C') \cup \psi(C_1)$, nor in $\varphi(E' \mid_{\mathcal{I}(a)}) \,/\, \varphi(C') \cup \psi(C_1)$, since the antecedents of these labels, $v_1 \ldots v_m$, are not free in $E' \mid_{\mathcal{I}(a)} \,/\, C'$, nor in $\tau \,/\, C'$. We have therefore established that:

$$\begin{array}{rcl} \psi(E) \vdash a : \psi(\tau) \,/\, \varphi(C') \cup \psi(C_1) & & \\ \psi(\tau) \,/\, \varphi(C') \cup \psi(C_1) & \equiv & \varphi(\tau) \,/\, \varphi(C') \\ \psi(E \mid_{\mathcal{I}(a)}) \,/\, \varphi(C') \cup \psi(C_1) & \equiv & \varphi(E' \mid_{\mathcal{I}(a)}) \,/\, \varphi(C') \end{array}$$

From these premises, we can conclude, by the simplification rule:

$$\varphi(E') \vdash a : \varphi(\tau) \,/\, \varphi(C')$$

and the derivation thus obtained has the same height as the initial derivation of $E' \vdash a : \tau \,/\, C$. □

Proof: of proposition 4.9. We give the cases that do not follow immediately from the induction hypothesis. Once more, we easily check on each case that the derivation of $E \vdash a : \tau \,/\, C'$ we obtain has the same height as the initial derivation of $E' \vdash a : \tau \,/\, C'$.

• **Instantiation rule.** The derivation is:

$$\frac{\tau \leq E(x) \,/\, C}{E \vdash x : \tau \,/\, C}$$

Let $\varphi : \mathcal{F}_g(E(x)) \Rightarrow \texttt{TypNongen} \cup \texttt{EtiqNongen}$ be the substitution such that $\varphi(\sigma) = \tau$ and $\varphi(C) \subseteq C$. Since φ modifies only the generic variables free in E, we have $\varphi(C') = C'$ by hypothesis over C', and therefore

$$\varphi(C \cup C') = \varphi(C) \cup \varphi(C') = \varphi(C) \cup C' \subseteq C \cup C'.$$

Hence $\tau \leq E(x) \,/\, C \cup C'$, and the instantiation rule concludes $E \vdash x : \tau \,/\, C \cup C'$.

• **The `let` rule.** The initial derivation ends up with:

$$\frac{E \vdash a_1 : \tau_1 \,/\, C_1 \qquad (\sigma, C) = \texttt{Gen}(\tau_1, C_1, E) \qquad E + x \mapsto \sigma \vdash a_2 : \tau_2 \,/\, C}{E \vdash \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2 : \tau_2 \,/\, C}$$

Take $\{\alpha_1 \ldots \alpha_k\} = \mathcal{F}(\tau_1 \,/\, C_1) \setminus \mathcal{D}(\tau_1 \,/\, C_1) \setminus \mathcal{F}_n(E)$. We can assume that the α_i do not occur in C', by renaming α_i to fresh variables β_i in the first premise. The derivation remains valid, as shown in the proof for the `let` case in proposition 4.8.

We apply the induction hypothesis to the two premises. We get proofs for:

$$E \vdash a_1 : \tau_1 \,/\, C_1 \cup C' \qquad\qquad E + x \mapsto \sigma \vdash a_2 : \tau_2 \,/\, C \cup C'$$

To conclude, it remains to show that

$$\texttt{Gen}(\tau_1, C_1 \cup C', E) = (\sigma, C \cup C').$$

First of all, we have

$$\mathcal{F}(\tau_1 \,/\, C_1) \setminus \mathcal{D}(\tau_1 \,/\, C_1 \cup C') \setminus \mathcal{F}_n(E) \setminus \mathcal{D}(E \,/\, C_1 \cup C') = \{\alpha_1 \ldots \alpha_k\}.$$

That's because $\mathcal{D}(\tau_1 \,/\, C_1 \cup C')$ contains at least $\mathcal{D}(\tau_1 \,/\, C_1)$. And also α_i does not belong to $\mathcal{D}(\tau_1 \,/\, C_1 \cup C')$, since α_i does not belong to $\mathcal{D}(\tau_1 \,/\, C_1)$, and α_i does not belong to C'. Same reasoning for $\mathcal{D}(E \,/\, C_1 \cup C')$.

Let $\theta : \{\alpha_1 \ldots \alpha_k\} \Leftrightarrow \texttt{VarTypGen}$ be the renaming such that $\sigma = \theta(\tau_1)$ and $C = \theta(C_1)$. Since the α_i do not appear in C', we have $\theta(C') = C'$, hence $\theta(C_1 \cup C') = \theta(C_1) \cup C' = C \cup C'$. It is therefore true that

$$\texttt{Gen}(\tau_1, C_1 \cup C', E) = (\sigma, C \cup C').$$

Combining this fact with the two derivations obtained by applying the induction hypothesis, we conclude, by the `let` rule:

$$E \vdash \texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2 : \tau_2 \,/\, C \cup C'.$$

• **Simplification rule.** The initial derivation ends up with:

$$\frac{E_1 \vdash a : \tau \,/\, C_1 \qquad \tau \,/\, C_1 \equiv \tau \,/\, C \qquad E_1 \,|_{\,\mathcal{I}(a)} \,/\, C_1 \equiv E \,|_{\,\mathcal{I}(a)} \,/\, C}{E \vdash a : \tau \,/\, C}$$

We apply the induction hypothesis to the premise $E_1 \vdash a : \tau \,/\, C_1$. We get a proof of $E_1 \vdash a : \tau \,/\, C_1 \cup C'$. Obviously, $\tau \,/\, C_1 \equiv \tau \,/\, C$ implies $\tau \,/\, C_1 \cup C' \equiv \tau \,/\, C \cup C'$, and similarly for $E \mid_{\mathcal{I}(a)}$ and $E_1 \mid_{\mathcal{I}(a)}$. Applying the simplification rule to these premises, it follows that $E \vdash a : \tau \,/\, C \cup C'$. □

Proposition 4.10 (Typing is stable under substitution, 2) *Let a be an expression, τ be a non-generic type, E be a typing environment, C be a constraint set. For all substitutions φ : $\mathtt{VarNongen} \Rightarrow \mathtt{TypNongen} \cup \mathtt{EtiqNongen}$ of non-generic types for non-generic variables, if we have $E \vdash a : \tau \,/\, C$ and $\varphi(C) \subseteq C$, then we have $\varphi(E) \vdash a : \varphi(\tau) \,/\, C$.*

Remark. Proposition 4.10 is unfortunately not a consequence of propositions 4.8 and 4.9: even if $\varphi(C) \subseteq C$, the set $C' = C \setminus \varphi(C)$ does not necessarily satisfy the hypothesis of proposition 4.9. □

Proof: the proof is very similar to the proof of proposition 4.8. The only difference is for the instantiation rule.

• **Instantiation rule.** We have $a = x$ and $\tau \leq E(x) \,/\, C$. Let $\psi : \mathcal{F}_g(E(x)) \Rightarrow \mathtt{TypNongen} \cup \mathtt{EtiqNongen}$ be the substitution such that $\psi(E(x)) = \tau$ and $\psi(C) \subseteq C$. Define the substitution θ by $\theta(\alpha_g) = \varphi(\psi(\alpha_g))$ for all generic α_g. We have:

$$\begin{array}{rcccl} \theta(\varphi(\alpha_g)) & = & \theta(\alpha_g) & = & \varphi(\psi(\alpha_g)) \\ \theta(\varphi(\alpha_n)) & = & \varphi(\alpha_n) & = & \varphi(\psi(\alpha_n)) \end{array}$$

Hence $\theta \circ \varphi = \varphi \circ \psi$. It follows that $\theta(\varphi(E(x))) = \varphi(\psi(E(x))) = \varphi(\tau)$. Moreover, $\theta(C) = \varphi(\psi(C)) \subseteq \varphi(C) \subseteq C$. Finally, since φ modifies only non-generic variables and does not introduce fresh generic variables, we have $\mathcal{F}_g(\varphi(E(x))) = \mathcal{F}_g(E(x))$, hence $\theta : \mathcal{F}_g(\varphi(E(x))) \Rightarrow \mathtt{TypNongen} \cup \mathtt{EtiqNongen}$. This shows that $\varphi(\tau) \leq \varphi(E(x)) \,/\, C$. We can therefore derive $\varphi(E) \vdash x : \varphi(\tau) \,/\, C$. □

4.3 Soundness proofs

In this section, we show that the indirect type system proposed in the present chapter is sound with respect to the three semantics given in chapter 2: for references, for channels, and for continuations.

4.3.1 References

The semantic typing predicates we shall use are essentially identical to those in section 3.3.1, expressed with indirect types and generic types instead of direct types and type schemes. The STORE TYPINGS, ranged over by S, now map memory locations to constrained types.

$$\text{Store typings:} \quad S \;::=\; [\ell_1 \mapsto \tau_1 \,/\, C_1, \ldots, \ell_n \mapsto \tau_n \,/\, C_n]$$

As in section 3.3.1, the purpose of the store typings is to ensure that all references to a given memory location ℓ has the same monomorphic type τ `ref` $/\, C$, where $\tau \,/\, C$ is identical to $S(\ell)$. This prevents all inconsistent uses of the location ℓ. Actually, we slightly weaken the condition above, by only requiring that $\tau \,/\, C$ be equivalent to $S(\ell)$, in the sense of section 4.2.3. Here are the relations we'll use:

$S \models v : \tau \,/\, C$	the value v, considered in a store of type S, belongs to the non-generic type τ in the context C
$S \models v : \sigma \,/\, C$	the value v, considered in a store of typee S, belongs to the generic type σ in the context C
$S \models e : E \,/\, C$	the values contained in the evaluation environment e, considered in a store of type S, belong to the corresponding types in the typing environment E
$\models s : S$	the values contained in the store s belong to the corresponding types in S.

These relations are defined as follows:

- $S \models cst : \texttt{unit} \,/\, C$ if cst is `()`
- $S \models cst : \texttt{int} \,/\, C$ if cst is an integer
- $S \models cst : \texttt{bool} \,/\, C$ if cst is `true` or `false`
- $S \models (v_1, v_2) : \tau_1 \times \tau_2 \,/\, C$ if $S \models v_1 : \tau_1 \,/\, C$ and $S \models v_2 : \tau_2 \,/\, C$
- $S \models \ell : \tau_1 \ \texttt{ref} \,/\, C$ if $\ell \in \mathrm{Dom}(S)$ and $S(\ell) \equiv \tau_1 \,/\, C$.
- $S \models (f, x, a, e) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrowtail \tau_2 \,/\, C$ if there exists a typing environment E such that

$$S \models e : E \,/\, C \quad \text{and} \quad E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \multimap\!\langle u \rangle\!\rightarrowtail \tau_2.$$

- $S \models v : \sigma \,/\, C$ if $\sigma \,/\, C$ contains no dangerous generic variables, and if $S \models v : \rho(\sigma) \,/\, \rho(C)$ for all renamings ρ of the generic variables of σ into non-generic variables.
- $S \models e : E \,/\, C$ if $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, and for all x in $\mathrm{Dom}(E)$, we have $S \models e(x) : E(x) \,/\, C$.
- $\models s : S$ if $\mathrm{Dom}(s) = \mathrm{Dom}(S)$, and for all $\ell \in \mathrm{Dom}(s)$, we have $S \models s(\ell) : S(\ell)$.

Remark. As in section 3.3, in the case of functional values, we can always assume $\mathrm{Dom}(E) = \mathcal{I}(f \ \texttt{where} \ f(x) = a)$. If this is not the case, it suffices to apply the simplification rule to restrict E to the identifiers free in $(f \ \texttt{where} \ f(x) = a)$. □

We easily check that the semantic typing predicates defined above are stable under store extension, that is, by replacing the store typing S by an extension S' of S.

Proposition 4.11 *Let v be a value, S be a store typing, and $\tau \,/\, C$ and $\tau' \,/\, C'$ be two non-generic constrained types. If $\tau \,/\, C \equiv \tau' \,/\, C'$, then $S \models v : \tau \,/\, C$ is equivalent to $S \models v : \tau' \,/\, C'$.*

Proof: given the symmetry of $\equiv$, and the fact that $\tau = \tau'$, it suffices to show that $S \models v : \tau \,/\, C$ implies $S \models v : \tau \,/\, C'$. We proceed by structural induction over v.

• **Case** $v = cst$. Obviously true.

• **Case** $v = (v_1, v_2)$. We then have $\tau = \tau_1 \times \tau_2$, with $S \models v_1 : \tau_1 \,/\, C$ and $S \models v_2 : \tau_2 \,/\, C$. Since τ_1 and τ_2 are subterms of τ, we have $\tau_1 \,/\, C \equiv \tau_1 \,/\, C'$; the same for τ_2. By induction hypothesis applied to v_1 and v_2, it follows $S \models v_1 : \tau_1 \,/\, C'$ and $S \models v_2 : \tau_2 \,/\, C'$; hence the expected result.

• **Case** $v = \ell$. We then have $\tau = \tau_1$ `ref`, with $\tau_1 \,/\, C \equiv S(\ell)$. Since τ_1 is subterm of τ, we have $\tau_1 \,/\, C \equiv \tau_1 \,/\, C'$. By transitivity of $\equiv$, it follows that $\tau_1 \,/\, C' \equiv S(\ell)$, and therefore $S \models \ell : \tau \,/\, C'$.

• **Case** $v = (f, x, a, e)$. Then, $\tau = \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2$, and there exists a typing environment E such that

$$S \models e : E \,/\, C \quad \text{and} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2 \,/\, C.$$

Let E be such an environment. Let $y \in \mathrm{Dom}(E)$. By the remark above, we can assume that y is free in $(f \texttt{ where } f(x) = a)$. Given the typing rules for functions, the constraint $E(y) \triangleleft u_n$ appears in C. It follows that $\mathcal{F}(E(y) \,/\, C) \subseteq \mathcal{F}(u_n \,/\, C) \subseteq \mathcal{F}(\tau \,/\, C)$. Hence, $E(y) \,/\, C \equiv E(y) \,/\, C'$ for all y in the domain of E. This has two consequences. First of all, for all $y \in \mathrm{Dom}(E)$, we can apply the induction hypothesis to $e(y)$ and $E(y)$; it follows that $S \models e(y) : E(y) \,/\, C'$. Hence $S \models e : E \,/\, C'$. Second consequence, $E \,/\, C \equiv E \,/\, C'$. Applying the simplification rules with the following premises:

$$E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2 \,/\, C \qquad \tau_1 \multimap\langle u\rangle\rightarrow \tau_2 \,/\, C \equiv \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2 \,/\, C' \qquad E \,/\, C \equiv E \,/\, C',$$

we deduce that

$$E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2 \,/\, C'.$$

This concludes the proof of the expected result: $S \models (f, x, a, e) : \tau_1 \multimap\langle u\rangle\rightarrow \tau_2 \,/\, C'$. □

We now give two results of stability under substitution for the semantic typing relations.

Proposition 4.12 *Let v be a value, τ be a non-generic type, C be a constraint set and S be a store typing such that $S \models v : \tau \,/\, C$. Let θ be a renaming of non-generic variables into non-generic variables such that* $\mathrm{Dom}(\theta) \cap \mathcal{D}(\tau \,/\, C) = \emptyset$. *Then, $S \models v : \theta(\tau) \,/\, \theta(C)$.*

Proof: by structural induction over v.

• **Case** $v = cst$. Obvious, since τ is then a base type, hence $\theta(\tau) = \tau$.

• **Case** $v = (v_1, v_2)$ **and** $\tau = \tau_1 \times \tau_2$. Since $\mathcal{D}(\tau \,/\, C) = \mathcal{D}(\tau_1 \,/\, C) \cup \mathcal{D}(\tau_2 \,/\, C)$, we have $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau_1 \,/\, C) = \emptyset$ and $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau_2 \,/\, C) = \emptyset$. By induction hypothesis, it follows that $S \models v_1 : \theta(\tau_1) \,/\, \theta(C)$ and $S \models v_2 : \theta(\tau_2) \,/\, \theta(C)$. Hence $S \models (v_1, v_2) : \theta(\tau_1 \times \tau_2) \,/\, \theta(C)$.

• **Case** $v = \ell$ **and** $\tau = \tau_1$ `ref`. By definition of $\models$, we have $S(\ell) \equiv \tau_1 \,/\, C$. Since $\mathcal{D}(\tau_1 \texttt{ ref} \,/\, C) = \mathcal{F}(\tau_1 \,/\, C)$, we have $\mathrm{Dom}(\theta) \cap \mathcal{F}(\tau_1 \,/\, C) = \emptyset$. Hence $\theta(\tau_1) \,/\, \theta(C) \equiv \tau_1 \,/\, C$ by proposition 4.6. By transitivity of $\equiv$, it follows that $S(\ell) \equiv \theta(\tau_1) \,/\, \theta(C)$, hence $S \models \ell : \theta(\tau_1 \texttt{ ref}) \,/\, \theta(C)$, as expected.

• **Case** $v = (f, x, a, e)$ **and** $\tau = \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2$. Let E be a typing environment such that

$$S \models e : E \,/\, C \quad \text{and} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2 \,/\, C.$$

By proposition 4.8, we get:

$$\theta(E) \vdash (f \texttt{ where } f(x) = a) : \theta(\tau_1 \multimap\langle u_n\rangle\rightarrow \tau_2) \,/\, \theta(C).$$

It remains to show that

$$S \models e : \theta(E) \,/\, \theta(C). \tag{1}$$

Take y in the domain of E. We can assume y free in f `where` $f(x) = a$. Let us show $S \models e(y) : \theta(E(y)) \,/\, \theta(C)$. Let ρ be a renaming of the generic variables of $\theta(E(y))$ into non-generic variables. We can decompose $\rho \circ \theta$ as $\theta' \circ \rho$, where ρ' is a renaming of the generic variables of $E(y)$ into non-generic variables, and θ' is a renaming of non-generic variables into non-generic variables, with $\mathrm{Dom}(\rho') = \mathrm{Dom}(\rho)$ and $\mathrm{Dom}(\theta') \subseteq \mathrm{Dom}(\theta) \cup \mathrm{Codom}(\rho')$. From hypothesis $S \models e(y) : E(y) \,/\, C$ and from the definition of $\models$ over type schemes, it follows that $S \models e(y) : \rho'(E(y)) \,/\, \rho'(C)$. Since $E(y) \,/\, C$ does not contain dangerous generic variables, we have $\mathrm{Dom}(\theta') \cap \mathcal{D}(\rho'(E(y)) \,/\, \rho'(C)) = \emptyset$. Applying the induction hypothesis to $e(y)$, il follows that $S \models e(y) : \theta'(\rho'(E(y))) \,/\, \theta'(\rho'(C))$, that is, $S \models e(y) : \rho(\theta(E(y))) \,/\, \rho(\theta(C))$. This holds for all ρ, hence $S \models e(y) : \theta(E(y)) \,/\, \theta(C)$. This establishes (1), and the expected result. □

Proposition 4.13 *Let v be a value, $\tau \,/\, C$ be a non-generic constrained type, and S be a store typing such that $S \models v : \tau \,/\, C$. Let φ : `VarNongen` $\Rightarrow$ `TypNongen` $\cup$ `EtiqNongen` be a substitution such that* $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau \,/\, C) = \emptyset$ *and* $\varphi(C) \subseteq C$. *Then,* $S \models v : \varphi(\tau) \,/\, C$.

Proof: by structural induction over v.

• **Case** $v = cst$. Obvious, since τ is then a base type, hence $\varphi(\tau) = \tau$.

• **Case** $v = (v_1, v_2)$ **and** $\tau = \tau_1 \times \tau_2$. Since $\mathcal{D}(\tau \,/\, C) = \mathcal{D}(\tau_1 \,/\, C) \cup \mathcal{D}(\tau_2 \,/\, C)$, we have $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau_1 \,/\, C) = \emptyset$ and $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau_2 \,/\, C) = \emptyset$. By induction hypothesis, we have $S \models v_1 : \varphi(\tau_1) \,/\, C$ and $S \models v_2 : \varphi(\tau_2) \,/\, C$. Hence $S \models (v_1, v_2) : \varphi(\tau_1 \times \tau_2) \,/\, C$.

• **Case** $v = \ell$ **and** $\tau = \tau_1$ `ref`. By definition of $\models$, we have $S(\ell) \equiv \tau_1 \,/\, C$. Since $\mathcal{D}(\tau_1$ `ref` $\,/\, C) = \mathcal{F}(\tau_1 \,/\, C)$, we have $\varphi(\tau_1) = \tau_1$. Hence $S(\ell) \equiv \varphi(\tau_1) \,/\, C$, and therefore $S \models \ell : \varphi(\tau_1$ `ref`$) \,/\, C$, as expected.

• **Case** $v = (f, x, a, e)$ **and** $\tau = \tau_1 \multimap\!\langle u \rangle\!\rightarrow \tau_2$. Let E be a typing environment such that

$$S \models e : E \,/\, C \quad \text{and} \quad E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\!\langle u \rangle\!\rightarrow \tau_2 \,/\, C.$$

By proposition 4.10, we get:

$$\varphi(E) \vdash (f \texttt{ where } f(x) = a) : \varphi(\tau_1 \multimap\!\langle u \rangle\!\rightarrow \tau_2) \,/\, C.$$

It remains to show that

$$S \models e : \varphi(E) \,/\, C. \tag{1}$$

Take y in $\mathrm{Dom}(E)$. Let ρ be a renaming of the generic variables of $\varphi(E(y))$ into non-generic variables. We must show that $S \models e(y) : \rho(\varphi(E(y))) \,/\, \rho(C)$. Let ξ be a renaming of the generic variables of $E(y)$ into non-generic variables out of reach for φ. Since $S \models e(y) : E(y) \,/\, C$, we already know $S \models e(y) : \xi(E(y)) \,/\, \xi(C)$. Since $\mathrm{Codom}(\xi)$ is out of reach for φ, the two substitutions φ and ξ commute. Hence, $\varphi(\xi(C)) = \xi(\varphi(C)) \subseteq \xi(C)$. Moreover,

$$\mathrm{Dom}(\varphi) \cap \mathcal{D}(\xi(E(y)) \,/\, \xi(C)) = \emptyset.$$

Indeed, if α were dangerous in $\xi(E(y)) / \xi(C)$ and in the domain of $\mathrm{Dom}(\varphi)$, we would have $\xi(\alpha) = \alpha$ by hypothesis $\mathrm{Codom}(\xi)$ out of reach for φ, and therefore α would be both dangerous in $E(y) \,/\, C$

and inside the domain of φ, contradicting the hypothesis $\mathrm{Dom}(\varphi) \cap \mathcal{D}(\tau \,/\, C) = \emptyset$. We can therefore apply the induction hypothesis to $e(y)$ with type $\xi(E(y)) \,/\, \xi(C)$, and to the substitution φ. It follows:

$$S \models e(y) : \varphi(\xi(E(y))) \,/\, \xi(C).$$

Let $\theta = \rho \circ \xi^{-1}$. Since the generic variables free in $E(y)$ are the same as the generic variables free in $\varphi(E(y))$, the substitution θ is a renaming of non-generic variables into non-generic variables. Moreover,

$$\mathrm{Dom}(\theta) \cap \mathcal{D}(\varphi(\xi(E(y))) \,/\, \xi(C)) = \emptyset,$$

because $\mathrm{Dom}(\theta)$, which is equal to $\mathrm{Codom}(\xi)$ since $\mathrm{Dom}(\rho) = \mathrm{Dom}(\xi)$, is out of reach for φ, and because $\mathrm{Dom}(\theta) \cap \mathcal{D}(\xi(E(y)) \,/\, \xi(C)) = \emptyset$. Applying proposition 4.12, we get:

$$S \models e(y) : \theta(\varphi(\xi(E(y)))) \,/\, \theta(\xi(C)).$$

By construction, $\theta \circ \xi = \rho$. Hence $\theta \circ \varphi \circ \xi = \theta \circ \xi \circ \varphi = \rho \circ \varphi$. The result provided by proposition 4.12 therefore reads:

$$S \models e(y) : \rho(\varphi(E(y))) \,/\, \rho(C).$$

This holds for all renamings ρ. Hence $S \models e(y) : \varphi(E(y)) \,/\, C$. This holds for all y. Hence (1), and the expected result. □

As corollaries of those two stability properties, it follows that the generalization and instantiation operations are semantically sound.

Proposition 4.14 (Semantic generalization) *Let S be a store typing, v_1 be a value and $\tau_1 \,/\, C_1$ be a non-generic constrained type such that $S \models v : \tau_1 \,/\, C_1$. For all renamings ξ of non-generic variables not dangerous in $\tau_1 \,/\, C_1$ into generic variables, we have $S \models v : \xi(\tau_1) \,/\, \xi(C_1)$. As a corollary, for all E, if $(\sigma, C) = \mathtt{Gen}(\tau_1, C_1, E)$, then $S \models v : \sigma \,/\, C$.*

Proof: let ρ be a renaming of the generic variables of $\xi(\tau_1)$ into non-generic variables. We must show that

$$S \models v : \rho(\xi(\tau_1)) \,/\, \rho(\xi(C_1)). \tag{1}$$

Define $\theta = \rho \circ \xi$. Since $\mathrm{Codom}(\xi) = \mathrm{Dom}(\rho)$, the substitution θ is a renaming of non-generic variables into non-generic variables. Moreover, $\mathrm{Dom}(\theta) \cap \mathcal{D}(\tau_1 \,/\, C_1) = \emptyset$, since $\mathrm{Dom}(\theta) = \mathrm{Dom}(\xi)$. Applying proposition 4.12, it follows $S \models v : \theta(\tau_1) \,/\, \theta(C_1)$, that is, (1). This holds for all renamings ρ, hence $S \models v : \xi(\tau_1) \,/\, \xi(C_1)$.

The corollary immediately follows by definition of `Gen`. □

Proposition 4.15 (Semantic instantiation) *Let v be a value, $\sigma \,/\, C$ be a generic constrained type and S be a store typing such that $S \models v : \sigma \,/\, C$. Then, $S \models v : \tau \,/\, C$ for all instances $\tau \leq \sigma \,/\, C$.*

Proof: let $\tau \leq \sigma \,/\, C$. Let $\theta : \mathcal{F}_g(\sigma) \Rightarrow \mathtt{TypNongen} \cup \mathtt{EtiqNongen}$ be a substitution such that $\tau = \theta(\sigma)$ and $\theta(C) \subseteq C$. Let ρ be a renaming of the generic variables of σ into non-generic variables out of reach for θ. Define $\varphi = \theta \circ \rho^{-1}$. We have $\theta = \varphi \circ \rho$, and φ is a substitution of non-generic types for non-generic variables. By hypothesis $S \models v : \sigma \,/\, C$, it follows that

$$S \models v : \rho(\sigma) \,/\, \rho(C).$$

By construction of ρ, we have $\rho^{-1} \circ \varphi \circ \rho = \varphi \circ \rho$. Hence, the hypothesis $\theta(C) \subseteq C$, that is, $\varphi(\rho(C)) \subseteq C$, implies $\varphi(\rho(C)) \subseteq \rho(C)$. Moreover, the generic variables of σ are not dangerous in $\sigma \,/\, C$. Since ρ is a renaming, this implies that the variables in $\mathrm{Codom}(\rho)$ are not dangerous in $\rho(\sigma) \,/\, \rho(C)$. Hence, taking into account $\mathrm{Dom}(\varphi) = \mathrm{Codom}(\rho)$,

$$\mathrm{Dom}(\varphi) \cap \mathcal{F}(\rho(\sigma) \,/\, \rho(C)) = \emptyset.$$

Applying proposition 4.13 to the substitution φ, it follows that

$$S \models v : \varphi(\rho(\sigma)) \,/\, \rho(C).$$

Consider the renaming ρ^{-1}. This is a renaming of non-generic variables into generic variables. Moreover, the variables in $\mathrm{Dom}(\rho^{-1})$, that is in $\mathrm{Codom}(\rho)$, are not dangerous in $\varphi(\rho(\sigma)) \,/\, \rho(\sigma)$. That's because if $\alpha \in \mathrm{Codom}(\rho)$ was dangerous in $\varphi(\rho(\sigma)) \,/\, \rho(C)$, it would also be dangerous in $\rho(\sigma) \,/\, \rho(C)$, since α is out of reach for φ; but this is impossible, as shown above. Applying proposition 4.14 to the renaming ρ^{-1}, it follows that

$$S \models v : \rho^{-1}(\varphi(\rho(\sigma))) \,/\, \rho^{-1}(\rho(C)),$$

that is, $S \models v : \theta(\sigma) \,/\, C$, taking into account $\rho^{-1} \circ \varphi \circ \rho = \varphi \circ \rho = \theta$. That's the expected result. □

Once these preliminary results are established, we can now show the strong soundness property.

Proposition 4.16 (Strong soundness for references) *Let a be an expression, τ be a non-generic type, E be a typing environment, C be a constraint set, e be an evaluation environment, s be a store and S be a store typing such that*

$$E \vdash a : \tau \,/\, C \quad \textit{and} \quad S \models e : E \mid_{\mathcal{I}(a)} \,/\, C \quad \textit{and} \quad \models s : S.$$

If there exists a response r such that $e \vdash a/s \Rightarrow r$, then $r \neq \texttt{err}$; instead, r is equal to v/s' for some v and s', and there exists a store typing S' such that:

$$S' \textit{ extends } S \quad \textit{and} \quad S' \models v : \tau \,/\, C \quad \textit{and} \quad \models s' : S'.$$

Proof: the proof is by induction on the size of the evaluation derivation and, in case of ties, on the size of the typing derivation. We argue by case analysis on the last rule used in the typing derivation. The proof closely follows the proof of proposition 3.6. I show the cases that are not completely similar.

• **Instantiation rule.**

$$\frac{\tau \leq E(x) \,/\, C}{E \vdash x : \tau \,/\, C}$$

By hypothesis, $S \models e : E \mid_{\mathcal{I}(x)} \,/\, C$, hence $x \in \mathrm{Dom}(e)$ and $S \models e(x) : E(x) \,/\, C$. The only possible evaluation is therefore $e \vdash x/s \Rightarrow e(x)/s$. By proposition 4.15, we have $S \models e(x) : \tau \,/\, C$ as expected.

• **Function rule.**

$$\frac{\begin{array}{c} E + f \mapsto \tau_1 \dashv\langle u_n \rangle\rightarrow \tau_2 + x \mapsto \tau_1 \vdash a : \tau_2 \,/\, C \\ (E(y) \triangleleft u_n) \in C \text{ for all } y \text{ free in } f \ \texttt{where}\ f(x) = a \end{array}}{E \vdash (f \ \texttt{where}\ f(x) = a) : \tau_1 \dashv\langle u_n \rangle\rightarrow \tau_2 \,/\, C}$$

The only possible evaluation is $e \vdash (f \texttt{ where } f(x) = a)/s \Rightarrow (f, x, a, e)/s$. We have $S \models (f, x, a, e) : \tau_1 \multimap\langle u_n\rangle\rightarrowtail \tau_2 \;/\; C$ by definition of $\models$, taking $E \mid_{\mathcal{I}(a)}$ for the required typing environment.

• **Application rule.**

$$\frac{E \vdash a_1 : \tau_2 \multimap\langle u_n\rangle\rightarrowtail \tau_1 \;/\; C \qquad E \vdash a_2 : \tau_2 \;/\; C}{E \vdash a_1(a_2) : \tau_1 \;/\; C}$$

We have three evaluation possibilities. The first one concludes $r = \texttt{err}$ because $e \vdash a_1 \Rightarrow r_1$ and r_1 does not match $(f, x, a_0, e_0)/s'$; but it contradicts the induction hypothesis applied to a_1, which says that $r_1 = v_1/s_1$ and $\models v_1 : \tau_2 \rightarrow \tau_1 \;/\; C$, hence a fortiori v_1 is a closure. The second evaluation possibility concludes $r = \texttt{err}$ because $e \vdash a_2 \Rightarrow \texttt{err}$; it similarly contradicts the induction hypothesis applied to a_2. Hence we are in the third case:

$$\frac{e \vdash a_1/s \Rightarrow (f, x, a_0, e_0)/s_1 \qquad e \vdash a_2/s_1 \Rightarrow v_2/s_2 \qquad e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \vdash a_0/s_2 \Rightarrow r}{e \vdash a_1(a_2) \Rightarrow r}$$

By induction hypothesis applied to a_1, we get a store typing S_1 such that:

$$S_1 \models v_1 : \tau_2 \multimap\langle u_n\rangle\rightarrowtail \tau_1 \;/\; C \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S.$$

By definition of $\models$, there exists E_0 such that

$$S_1 \models e_0 : E_0 \quad \text{and} \quad E_0 \vdash (f \texttt{ where } f(x) = a_0) : \tau_2 \multimap\langle u_n\rangle\rightarrowtail \tau_1 \;/\; C.$$

Only one typing rule allows to derive the rightmost judgement; hence the first premise of this rule holds:

$$E_0 + f \mapsto (\tau_2 \multimap\langle u_n\rangle\rightarrowtail \tau_1) + x \mapsto \tau_2 \vdash a_0 : \tau_1 \;/\; C.$$

Applying the induction hypothesis to a_2, we get S_2 such that

$$S_2 \models v_2 : \tau_2 \;/\; C \quad \text{and} \quad \models s_2 : S_2 \quad \text{and} \quad S_2 \text{ extends } S_1.$$

Consider the environments:

$$e_2 = e_0 + f \mapsto (f, x, a_0, e_0) + x \mapsto v_2 \qquad\qquad E_2 = E_0 + f \mapsto (\tau_2 \multimap\langle u_n\rangle\rightarrowtail \tau_1) + x \mapsto \tau_2$$

The identifiers free in a_0 are x, f, and the identifiers free in $(f \texttt{ where } f(x) = a_0)$. Hence $S_2 \models e_2 : E_2 \mid_{\mathcal{I}(a_0)} / C$. We can therefore apply the induction hypothesis to a_0, in the environments e_2 and E_2, and the store $s_2 : S_2$. It follows that $r = v/s'$. Moreover, there exists S' such that:

$$S' \models v : \tau_1 \quad \text{and} \quad \models s' : S' \quad \text{and} \quad S' \text{ extends } S_2.$$

This is the expected result, since S' extends S a fortiori.

• **The `let` rule.**

$$\frac{E \vdash a_1 : \tau_1 \;/\; C_1 \qquad (\sigma, C) = \texttt{Gen}(\tau_1, C_1, E) \qquad E + x \mapsto \sigma \vdash a_2 : \tau_2 \;/\; C}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2 \;/\; C}$$

For starters, let us show that $S \models e : E \mid_{\mathcal{I}(a_1)} / C_1$. By definition of `Gen`, we have $C = \theta(C_1)$ and $E = \theta(E)$ for some renaming θ of variables not dangerous in E/C_1 into generic variables. Let y be an identifier free in a_1. We need to show that $S \models e(y) : E(y)/C_1$. Let ρ be a substitution of the generic variables of $E(y)$ into non-generic variables. We need to show that $S \models e(y) : \rho(E(y)) \,/\, \rho(C_1)$. The substitution $\rho \circ \theta^{-1}$ is a renaming of the generic variables of $E(y)$ into non-generic variables. From hypothesis $S \models e : E \,/\, C$ it therefore follows that $S \models e(y) : \rho(\theta^{-1}(E(y))) \,/\, \rho(\theta^{-1}(C))$, that is, $S \models e(y) : \rho(E(y)) \,/\, \rho(C_1)$. This holds for all ρ and for all y, hence $S \models e : E \mid_{\mathcal{I}(a_1)} / C_1$ as expected.

We have two possible evaluations for the `let` expression. The first one corresponds to $e \vdash a_1 \Rightarrow$ `err`. It contradicts the induction hypothesis applied to a_1 and to e with type $E \,/\, C_1$. Hence the last step in the evaluation is:

$$\frac{e \vdash a_1/s \Rightarrow v_1/s_1 \qquad e + x \mapsto v_1 \vdash a_2/s_1 \Rightarrow r}{e \vdash (\texttt{let}\ \ x = a_1\ \ \texttt{in}\ \ a_2)/s \Rightarrow r}$$

By induction hypothesis applied to a_1, we get S_1 such that:

$$S_1 \models v_1 : \tau_1 \,/\, C_1 \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S.$$

By proposition 4.14, it follows that $S_1 \models v_1 : \sigma \,/\, C$. Take

$$e_1 = e + x \mapsto v_1 \qquad\qquad E_1 = E + x \mapsto \sigma.$$

For all y free in a_2, either $y = x$, or y is free in a; in both cases, we have $S_1 \models e_1(y) : E_1(y) \,/\, C$. Hence $S_1 \models e_1 : E_1 \mid_{\mathcal{I}(a_2)} / C$. Applying the induction hypothesis to a_2 considered in environments e_1 and E_1, and in the store $s_1 : S_1$, we obtain that r is v/s', and there exists S' such that

$$S' \models v : \tau_2 \,/\, C \quad \text{and} \quad \models s' : S' \quad \text{and} \quad S' \text{ extends } S_1.$$

That's the expected result.

• **Primitives rule, case `ref`.**

$$\frac{\tau \multimap\langle u_n\rangle\rightarrowtail \tau\ \ \texttt{ref} \leq t_g \multimap\langle u_g\rangle\rightarrowtail t_g\ \ \texttt{ref} \,/\, C \qquad E \vdash a : \tau \,/\, C}{E \vdash \texttt{ref}(a) : \tau\ \ \texttt{ref} \,/\, C}$$

There are two evaluation possibilities. One leads to $r =$ `err` because a evaluates to `err`; it contradicts the induction hypothesis applied to a. Hence we are in the second evaluation case for $\texttt{ref}(a)$, ending up with:

$$\frac{e \vdash a/s_0 \Rightarrow v/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash \texttt{ref}(a)/s_0 \Rightarrow \ell/(s_1 + \ell \mapsto v)}$$

By induction hypothesis applied to a, we get S_1 such that

$$S_1 \models v : \tau \,/\, C \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S.$$

Define:

$$s' = s_1 + \ell \mapsto v \quad \text{and} \quad S' = S_1 + \ell \mapsto (\tau \,/\, C).$$

Since $\mathrm{Dom}(s_1) = \mathrm{Dom}(S_1)$, we have $\ell \notin \mathrm{Dom}(S_1)$. Hence S' extends S_1, and S too. We therefore have $S' \models v : \tau \,/\, C$, that is, $S' \models s'(\ell) : S'(\ell)$. It follows $\models s' : S'$ and $S' \models \ell : \tau\ \texttt{ref} \,/\, C$, as expected.

• **Primitives rule, case !.**

$$\frac{\tau\ \texttt{ref} \multimap\langle u_n\rangle\rightarrowtail \tau \leq t_g\ \ \texttt{ref} \multimap\langle u_g\rangle\rightarrowtail t_g \,/\, C \qquad E \vdash a : \tau\ \texttt{ref} \,/\, C}{E \vdash\ !(a) : \tau \,/\, C}$$

We have three evaluation possibilities. One leads to `err` because a evaluates to a response that does not match ℓ/s'. It contradicts the induction hypothesis applied to a. The second possibility ends up with:

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \notin \mathrm{Dom}(s_1)}{e \vdash\ !a/s_0 \Rightarrow \texttt{err}}$$

By induction hypothesis applied to a, we get S_1 such that $S_1 \models \ell : \tau\ \texttt{ref}$ and $\models s_1 : S_1$. This implies $\ell \in \mathrm{Dom}(s_1)$; contradiction. There remains the third possibility:

$$\frac{e \vdash a/s_0 \Rightarrow \ell/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash\ !a/s_0 \Rightarrow s_1(\ell)/s_1}$$

By induction hypothesis applied to a, we get S_1 such that

$$S_1 \models \ell : \tau\ \texttt{ref} \,/\, C \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S.$$

Hence $S_1(\ell) \equiv \tau \,/\, C$ and $S_1 \models s_1(\ell) : S_1(\ell)$. By proposition 4.11, it follows that $S_1 \models s_1(\ell) : \tau \,/\, C$; taking $S' = S_1$, that's the expected result.

• **Primitives rule, case :=.**

$$\frac{\tau\ \texttt{ref} \times \tau \multimap\langle u_n\rangle\rightarrowtail \texttt{unit} \leq t_g\ \ \texttt{ref} \times t_g \multimap\langle u_g\rangle\rightarrowtail \texttt{unit} \,/\, C \qquad E \vdash a : \tau\ \texttt{ref} \times \tau \,/\, C}{E \vdash\ :=(a) : \texttt{unit}}$$

As in the case of the ! primitive, evaluation must end up with:

$$\frac{e \vdash a/s_0 \Rightarrow (\ell, v)/s_1 \qquad \ell \in \mathrm{Dom}(s_1)}{e \vdash :=(a)/s_0 \Rightarrow \texttt{()}/(s_1 + \ell \mapsto v)}$$

By induction hypothesis applied to a, we get S_1 such that

$$S_1 \models (\ell, v) : \tau\ \texttt{ref} \times \tau \,/\, C \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S.$$

Take $s' = s_1 + \ell \mapsto v$ and $S' = S_1$. Since $S_1(\ell) \equiv \tau/C$, we have $S' \models s'(\ell) : S'(\ell)$ by proposition 4.11. And, obviously, $S' \models \texttt{()} : \texttt{unit} \,/\, C$. Hence the expected result.

• **Simplification rule.**

$$\frac{E \vdash a : \tau \,/\, C \qquad \tau \,/\, C \equiv \tau' \,/\, C' \qquad E\,|_{\mathcal{I}(a)} \,/\, C \equiv E'\,|_{\mathcal{I}(a)} \,/\, C'}{E' \vdash a : \tau' \,/\, C'}$$

By proposition 4.11, $S \models e : E'\,|_{\mathcal{I}(a)} \,/\, C'$ implies $S \models e : E\,|_{\mathcal{I}(a)} \,/\, C$. We apply the induction hypothesis to the same evaluation derivation, and to the derivation of $E \vdash a : \tau \,/\, C$ as typing derivation. If follos that $r = v/s'$ and there exists a store typing S' such that

$$S' \text{ extends } S \quad \text{and} \quad S' \models v : \tau \,/\, C \quad \text{and} \quad \models s' : S'.$$

Since $\tau \,/\, C \equiv \tau' \,/\, C'$, we also have $S' \models v : \tau' \,/\, C'$ by proposition 4.11. That's the expected result. □

4.3.2 Communication channels

The soundness proof for the calculus with channels is essentially identical to the one in section 3.3.2, with technical lemmas similar to those in section 4.3.1. I just define the semantic typing predicates, and sketch the steps of the proof. We assume given a channel typing Γ that associates a non-generic constrainted type to each channel identifier c.

$$\text{Channle typing:} \quad \Gamma \quad ::= \quad [c_1 \mapsto \tau_1 \,/\, C_1, \ldots, c_n \mapsto \tau_n \,/\, C_n]$$

Here are the semantic typing predicates used:

$\Gamma \models v : \tau \,/\, C$	the value v belongs to the non-generic type τ in the context C
$\Gamma \models v : \sigma \,/\, C$	value v belongs to the generic type σ in the context C
$\Gamma \models e : E \,/\, C$	the values contained in the evaluation environment e belong to the corresponding type schemes in $E \,/\, C$
$\models w :? \Gamma$	the reception events $(c \,?\, v)$ contained in the event sequence w respect the channel types given by Γ
$\models w :! \Gamma$	the emission events $(c \,!\, v)$ contained in the event sequence w respect the channel types given by Γ

Here is their precise definition:

- $\Gamma \models cst : \texttt{unit} \,/\, C$ if cst is `()`
- $\Gamma \models cst : \texttt{int} \,/\, C$ if cst is an integer
- $\Gamma \models cst : \texttt{bool} \,/\, C$ if cst is `true` or `false`
- $\Gamma \models (v_1, v_2) : \tau_1 \times \tau_2 \,/\, C$ if $\Gamma \models v_1 : \tau_1 \,/\, C$, and $\Gamma \models v_2 : \tau_2 \,/\, C$
- $\Gamma \models c : \tau_1 \ \texttt{chan} \,/\, C$ if $\Gamma(c) \equiv \tau_1 \,/\, C$
- $\Gamma \models (f, x, a, e) : \tau_1 \multimap\!\langle u_n \rangle\!\mapsto \tau_2 \,/\, C$ if there exists a typing environment E such that

$$\Gamma \models e : E \,/\, C \quad \text{and} \quad E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \multimap\!\langle u_n \rangle\!\mapsto \tau_2.$$

- $\Gamma \models v : \sigma \,/\, C$ if $\sigma \,/\, C$ contains no dangerous generic variables, and if $\Gamma \models v : \rho(\sigma) \,/\, \rho(C)$ for all renamings ρ of the generic variables of σ into non-generic variables.
- $\Gamma \models e : E \,/\, C$ if $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, and for all x in $\mathrm{Dom}(E)$, we have $\Gamma \models e(x) : E(x) \,/\, C$.
- $\models w :? \Gamma$ if $\Gamma \models v : \Gamma(c)$ for all reception events $c\,?\,v$ belonging to the sequence w
- $\models w :! \Gamma$ if $\Gamma \models v : \Gamma(c)$ for all emission events $c\,!\,v$ belonging to the sequence w.

The following propositions are the main lemmas for the soundness proof.

Proposition 4.17 (Stability under equivalence) *Let v be a value, Γ be a channel typing, and $\tau \,/\, C$ and $\tau' \,/\, C'$ be two non-generic constrained types. If $\tau \,/\, C \equiv \tau' \,/\, C'$, then $\Gamma \models v : \tau \,/\, C$ is equivalent to $\Gamma \models v : \tau' \,/\, C'$.*

Proof: same proof as for proposition 4.11. □

Proposition 4.18 (Semantic generalization) *Let Γ be a channel typing, v_1 be a value and $\tau_1 \,/\, C_1$ be a non-generic constrained type such that $\Gamma \models v : \tau_1 \,/\, C_1$. If $(\sigma, C) = \mathtt{Gen}(\tau_1, C_1, E)$, then $\Gamma \models v : \sigma \,/\, C$.*

Proof: same proof as for proposition 4.14. We use a lemma showing that $\models$ is stable under renaming of non-dangerous variables similar to proposition 4.12. □

Proposition 4.19 (Semantic instantiation) *Let v be a value, $\sigma \,/\, C$ be a generic constrained type and Γ be a channel typing such that $\Gamma \models v : \sigma \,/\, C$. Then, $\Gamma \models v : \tau \,/\, C$ for all instances $\tau \leq \sigma \,/\, C$.*

Proof: same proof as for proposition 4.15. We use a lemma showing that $\models$ is stable under substitution of non-dangerous variables similar to proposition 4.13. □

We now proceed to show the strong soundness property. As in section 3.3.2, we assume given a closed, well-typed term a_0, where all $\mathtt{newchan}(a)$ subexpressions are distinct. We take a typing derivation $\mathcal{T}$ for a_0, and an evaluation derivation $\mathcal{E}$ for a_0. We construct a channel typing Γ adapted to $\mathcal{T}$ and $\mathcal{E}$ as described in section 3.3.2.

Proposition 4.20 (Strong soundness for channels) *Let $e \vdash a \stackrel{w}{\Longrightarrow} r$ be the conclusion of a sub-derivation in $\mathcal{E}$, and $E \vdash a : \tau \,/\, C$ be the conclusion of a sub-derivation in $\mathcal{T}$, for the same expression a. Assume $\Gamma \models e : E \,|_{\mathcal{I}(a)} \,/\, C$.*

1. *If $\models w :? \Gamma$, then $r \neq \mathtt{err}$; instead, r is a value v such that $\Gamma \models v : \tau \,/\, C$, and in addition $\models w :! \Gamma$.*
2. *If $w = w'.c\,!\,v.w''$ and $\models w' :? \Gamma$, then $\Gamma \models v : \Gamma(c)$.*

Proof: the proof is almost identical to the one for proposition 3.9. We use lemma 4.19 for the case of identifiers, lemma 4.18 for the case of `let` bindings, and lemma 4.17 for the case of the primitives ! and ?, and for the case of the simplification rule. The other cases are exactly the same as in the proof of proposition 3.9. □

4.3.3 Continuations

The proof of soundness for the calculus with continuation combines most of sections 3.3.3 (for the soundness proof) and 4.3.1 (for the technical lemmas). We use the following semantic typing relations:

$\models v : \tau \,/\, C$	the value v belongs to the non-generic type τ in context C
$\models v : \sigma \,/\, C$	the value v belongs to the generic type σ in context C
$\models e : E \,/\, C$	the values contained in the evaluation environment e belong to the corresponding types in $E \,/\, C$
$\models k :: \tau \,/\, C$	the continuation k accepts all values belonging to the type $\tau \,/\, C$

Here are their precise definitions:

- $\models cst : \texttt{unit} \,/\, C$ if cst is `()`
- $\models cst : \texttt{int} \,/\, C$ if cst is an integer
- $\models cst : \texttt{bool} \,/\, C$ if cst is `true` or `false`
- $\models (v_1, v_2) : \tau_1 \times \tau_2 \,/\, C$ if $\models v_1 : \tau_1 \,/\, C$ and $\models v_2 : \tau_2 \,/\, C$
- $\models k : \tau_1 \ \texttt{cont} \,/\, C$ if $\models k :: \tau_1 \,/\, C$
- $\models (f, x, a, e) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrowtail \tau_2 \,/\, C$ if there exists a typing environment E such that
$$\models e : E \,/\, C \quad \text{and} \quad E \vdash (f \ \texttt{where} \ f(x) = a) : \tau_1 \multimap\!\langle u_n \rangle\!\rightarrowtail \tau_2.$$
- $\models v : \sigma \,/\, C$ if $\sigma \,/\, C$ contains no dangerous generic variables, and if $\models v : \rho(\sigma) \,/\, \rho(C)$ for all renamings ρ of the generic variables of σ into non-generic variables.
- $\models e : E \,/\, C$ if $\mathrm{Dom}(E) \subseteq \mathrm{Dom}(e)$, and for all x in $\mathrm{Dom}(E)$, we have $\models e(x) : E(x) \,/\, C$.
- $\models \texttt{stop} :: \tau \,/\, C$ for all types $\tau \,/\, C$
- $\models \texttt{app1c}(a, e, k) :: \tau_1 \multimap\!\langle u_n \rangle\!\rightarrowtail \tau_2 \,/\, C$ if there exists a typing environment E such that
$$E \vdash a : \tau_1 \,/\, C \quad \text{and} \quad \models e : E \,/\, C \quad \text{and} \quad \models k :: \tau_2 \,/\, C$$
- $\models \texttt{app2c}(f, x, a, e, k) :: \tau \,/\, C$ if there exists a typing environment E, a type τ' and a label u_n such that
$$E \vdash (f \ \texttt{where} \ f(x) = a) : \tau \multimap\!\langle u_n \rangle\!\rightarrowtail \tau' \,/\, C \quad \text{and} \quad \models e : E \,/\, C \quad \text{and} \quad \models k :: \tau' \,/\, C$$
- $\models \texttt{letc}(x, a, e, k) :: \tau \,/\, C$ if there exists a typing environment E and a type τ' such that
$$E + x \mapsto \texttt{Gen}(\tau, E) \vdash a : \tau' \,/\, C \quad \text{and} \quad \models e : E \,/\, C \quad \text{and} \quad \models k :: \tau' \,/\, C$$
- $\models \texttt{pair1c}(a, e, k) :: \tau \,/\, C$ if there exists a typing environment E and a type τ' such that
$$E \vdash a : \tau' \,/\, C \quad \text{and} \quad \models e : E \,/\, C \quad \text{and} \quad \models k :: \tau \times \tau' \,/\, C$$

- $\models \texttt{pair2c}(v,k) :: \tau \,/\, C$ if there exists a type τ' such that

$$\models v : \tau' \,/\, C \quad \text{and} \quad \models k :: \tau' \times \tau \,/\, C$$

- $\models \texttt{primc}(\texttt{callcc},k) :: \tau\ \texttt{cont} \multimap\langle u_n\rangle\rightarrow \tau \,/\, C$ if $\models k :: \tau \,/\, C$
- $\models \texttt{primc}(\texttt{throw},k) :: \tau\ \texttt{cont} \times \tau \,/\, C$ for all τ.

The following propositions are the main lemmas for the soundness proof.

Proposition 4.21 (Stability under equivalence) *Let $\tau \,/\, C$ and $\tau' \,/\, C'$ be two non-generic constrained types such that $\tau \,/\, C \equiv \tau' \,/\, C'$. For all values v, $\models v : \tau \,/\, C$ is equivalent to $\models v : \tau' \,/\, C'$. For all continuations k, $\models k :: \tau \,/\, C$ is equivalent to $\models k :: \tau' \,/\, C'$.*

Proof: the proof is by structural induction over v and k. The cases are similar to those for proposition 4.11. □

Proposition 4.22 (Semantic generalization) *Let v_1 be a value and $\tau_1 \,/\, C_1$ be a non-generic constrained type such that $\models v : \tau_1 \,/\, C_1$. If $(\sigma, C) = \texttt{Gen}(\tau_1, C_1, E)$, then $\models v : \sigma \,/\, C$.*

Proof: same proof as for proposition 4.14. We use a lemma similar to proposition 4.12: if $\models v : \tau/C$ and if θ is a renaming of variables not dangerous in $\tau \,/\, C$, then $\models v : \theta(\tau) \,/\, \theta(C)$. This lemma is proved as lemma 4.12. The only case that differs is $v = k$. In this case, $\tau = \tau_1\ \texttt{cont}$ and $\mathcal{D}(\tau \,/\, C) = \mathcal{F}(\tau_1 \,/\, C)$. Hence $\tau_1 \,/\, C \equiv \theta(\tau_1) \,/\, \theta(C)$ by proposition 4.6. Since $\models k :: \tau_1 \,/\, C$, it follows from proposition 4.21 that $\models k :: \theta(\tau_1) \,/\, \theta(C)$. Hence $\models k : \theta(\tau)\ \texttt{cont} \,/\, \theta(C)$, as expected. □

Proposition 4.23 (Semantic instantiation) *Let v be a value and $\sigma \,/\, C$ be a generic constrained type such that $\Gamma \models v : \sigma \,/\, C$. Then $\models v : \tau \,/\, C$ for all instances $\tau \leq \sigma \,/\, C$.*

Proof: same proof as for proposition 4.15. We use a lemma similar to proposition 4.13: if $\models v : \tau/C$ and if φ is a substitution of non-dangerous variables by non-generic types such that $\varphi(C) \subseteq C$, then $\models v : \varphi(\tau) \,/\, C$. This result is proved exactly as lemma 4.13. □

Proposition 4.24 (Weak soundness for continuations)

1. *Let a be an expression, τ be a non-generic type, C be a constraint set, E be a typing environment, k be a continuation and r be a response such that*

$$E \vdash a : \tau \,/\, C \quad \textit{and} \quad \models e : E \mid_{\mathcal{I}(a)} \,/\, C \quad \textit{and} \quad \models k :: \tau \,/\, C \quad \textit{and} \quad e \vdash a; k \Rightarrow r.$$

   Then $r \neq \texttt{err}$.

2. *Let v be a value, k be a continuation, τ be a non-generic type, C be a constraint set and r be a response such that*

$$\models v : \tau \,/\, C \quad \textit{and} \quad \models k :: \tau \,/\, C \quad \textit{and} \quad \vdash v \triangleright k \Rightarrow r.$$

   Then $r \neq \texttt{err}$.

Proof: the proof mimics exactly the one for proposition 3.12. The case where a is an identifier is settled by lemma 4.23; the case where k is a continuation `letc`, by lemma 4.22. The other cases are identical to those in proposition 3.12. □

4.4 Type inference

In this section, we show that all well-typed programs in the indirect type system admit a principal type, and we give an algorithm that computes this principal type.

4.4.1 Unification

The Damas-Milner algorithm easily adapts to the indirect type system. That's because the type algebra enjoys the principal unifier property.

Proposition 4.25 *If two types τ_1 and τ_2 are unifiable, then they admit a principal unifier. We write $\mathtt{mgu}(\tau_1, \tau_2)$ for a principal unifier of τ_1 and τ_2, if it exists.*

Proof: the claim follows from the fact that type expressions belong to a two-sorted free algebra (the two sorts being the types and the labels). Moreover, $\mathtt{mgu}(\tau_1, \tau_2)$ can be obtained by one of the well-known unification algorithm between terms of a free algebra, such as Robinson's algorithm [85]. □

As in section 1.5, we impose the following condition over $\mathtt{mgu}(\tau_1, \tau_2)$ in the remainder of this section: all variables not free in τ_1 nor in τ_2 are out of reach for $\mathtt{mgu}(\tau_1, \tau_2)$. This condition has two useful consequences. First of all, if τ_1 and τ_2 are non-generic types, then the substitution $\mathtt{mgu}(\tau_1, \tau_2)$ is a substitution of non-generic types for non-generic type variables. Furthermore, the principal unifier does not "graft" extra constraints on the types being unified. This property can be stated more formally as follows:

Proposition 4.26 *Let τ_1, τ_2 be two types, T be a set of types, and C be a constraint set. If μ is the unifier $\mathtt{mgu}(\tau_1, \tau_2)$ defined above, then*

$$\mu(C \restriction (\{\tau_1, \tau_2\} \cup T)) \;=\; \mu(C) \restriction (\{\mu(\tau_1)\} \cup \mu(T)).$$

(As in section 4.2.3, we write $C \restriction T$, where T is a set of types, for the connected component of the types from T in C, that is, the restriction of the constraint set C to those labels that are free in $\tau \,/\, C$ for some $\tau \in T$.)

Proof: the $\subseteq$ inclusion is a consequence of proposition 4.5. Assume that this inclusion is proper. Then, C contains a constraint over a label u that is not free in any of the $\sigma \,/\, C$ (σ being either τ_1, or τ_2, or one of the types in T), but such that $\mu(u)$ is free in one of the $\mu(\sigma) \,/\, \mu(C)$. This requires $\mu(u) \neq u$. Hence u is directly free in τ_1 or in τ_2, since all variables not free in τ_1 or in τ_2 are out of reach for μ. But then u is free in $\tau_1 \,/\, C$ or in $\tau_2 \,/\, C$, and this is contradictory. □

4.4.2 The type inference algorithm

The type inference algorithm takes as inputs an expression a, a typing environments E, an initial constraint set C, and an infinite set of "fresh" non-generic variables V. It returns a non-generic type τ (the most general type for a), a substitution φ (representing the instantiations that had to be performed over E), a constraint set C', and a subset V' of V.

We write $\texttt{Inst}(\sigma, C, V)$ for a trivial instance of the generic type $\sigma \,/\, C$: defining θ as a renaming of the generic variables of σ into non-generic variables from V, we take:

$$\texttt{Inst}(\sigma, C, V) = (\theta(\sigma),\ \theta(C) \cup C,\ V \setminus \mathrm{Codom}(\theta)).$$

Algorithm 4.1 We take $\texttt{Infer}(a, E, C, V)$ to be the 4-tuple (τ, C', φ, V') defined by:

If a is x:
$(\tau, C', V') = \texttt{Inst}(E(x), C, V)$ and $\varphi = [\,]$

If a is cst:
$(\tau, C', V') = \texttt{Inst}(\texttt{TypCst}(cst), C, V)$ and $\varphi = [\,]$

If a is f `where` $f(x) = a_1$:
let $t_1 \in V$ and $t_2 \in V$ be two non-generic variables
and $u \in V$ be a non-generic label
let $C_0 = C \cup \{E(y) \triangleleft u \mid y \in \mathcal{I}(a)\}$
let $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E + f \mapsto (t_2 \dashv\langle u\rangle\rightarrow t_1) + x \mapsto t_2,\ C_0,\ V \setminus \{t_1, t_2, u\})$
let $\mu = \texttt{mgu}(\varphi_1(t_1), \tau_1)$
then $\tau = \mu(\varphi_1(t_2 \dashv\langle u\rangle\rightarrow t_1))$ and $\varphi = \mu \circ \varphi_1$ and $C' = \mu(C_1)$ and $V' = V_1$

If a is $a_1(a_2)$:
let $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, C, V)$
let $(\tau_2, C_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), C_1, V_1)$
let $t \in V$ be a non-generic type variable and $u \in V$ be a non-generic label
let $\mu = \texttt{mgu}(\varphi_2(\tau_1), \tau_2 \dashv\langle u\rangle\rightarrow t)$
then $\tau = \mu(t)$ and $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ and $C' = \mu(C_2)$ and $V' = V_2 \setminus \{t, u\}$

If a is `let` $x = a_1$ `in` a_2:
let $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, C, V)$
let $(\sigma, C_0) = \texttt{Gen}(\tau_1, C_1, \varphi_1(E))$
let $(\tau_2, C_2, \varphi_2, V_2) = \texttt{Infer}(a_2,\ \varphi_1(E) + x \mapsto \sigma,\ C_0,\ V_1)$
then $\tau = \tau_2$ and $\varphi = \varphi_2 \circ \varphi_1$ and $C' = C_2$ and $V' = V_2$

If a is (a_1, a_2):
let $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, C, V)$
let $(\tau_2, C_2, \varphi_2, V_2) = \texttt{Infer}(a_2, \varphi_1(E), C_1, V_1)$
then $\tau = \tau_1 \times \tau_2$ and $\varphi = \varphi_2 \circ \varphi_1$ and $C' = C_2$ and $V' = V_2$

If a is $op(a_1)$:
let $(\tau_0, C_0, V_0) = \texttt{Inst}(\texttt{TypOp}(\texttt{op}), C, V)$
let $(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, C_0, V_0)$
let $t \in V_1$ be a non-generic variable and $u \in V_1$ be a non-generic label
let $\mu = \texttt{mgu}(\tau_1 \dashv\langle u\rangle\rightarrow t, \tau_0)$
then $\tau = \mu(t)$ and $\varphi = \mu \circ \varphi_1$ and $C' = \mu(C_1)$ and $V' = V_1 \setminus \{t, u\}$

We take that $\texttt{Infer}(a, E, C, V)$ is undefined if, at some point during the computation, none of the cases match; in particular, if `mgu` is applied to non-unifiable types.

A few remarks on the output of the algorithm. If $(\tau, C', \varphi, V') = \texttt{Infer}(a, E, C, V)$, and if $\mathcal{F}(E) \cap V = \emptyset$, and if $\mathcal{F}(C) \cap V = \emptyset$, then:

1. φ is a substitution of non-generic types for non-generic variables

2. $V' \subseteq V$
3. the variables in V' are not free in τ', nor in C'
4. the variables in V' are out of reach for φ, hence not free in $\varphi(E)$
5. $C' \supseteq \varphi(C)$
6. no generic variable free in $\varphi(E)$ is free in $C' \setminus \varphi(C)$.

Proposition 4.27 (Correctness of type inference) *Let a be an expression, E be a typing environment, C be a constraint set, and V be a set of non-generic type variables such that $\mathcal{F}(E) \cap V = \emptyset$ and $\mathcal{F}(C) \cap V = \emptyset$. If $(\tau, \varphi, C', V_1) = \texttt{Infer}(a, E, C, V)$ is defined, then we can derive $\varphi(E) \vdash a : \tau \,/\, C_1$.*

Proof: the proof is by structural induction over a, and closely resembles the proof of proposition 1.8, with additional handling of constraint sets. The proof essentially relies over the fact that typing is stable under substitution (proposition 4.8) and under addition of constraints (proposition 4.9). I show one base case and two inductive steps; the other cases are similar.

• **Case** $a = x$. We have $(\tau, C', V') = \texttt{Inst}(E(x), C, V)$. By definition of `Inst`, we have $\tau \leq E(x) \,/\, C'$. Hence we can derive $E \vdash x : \tau \,/\, C'$.

• **Case** $a = (f \texttt{ where } f(x) = a_1)$. With the same notations as in the algorithm, the induction hypothesis leads to a proof of:

$$\varphi_1(E + f \mapsto (t_1 \multimap\langle u \rangle\rightarrow t_2) + x \mapsto t_1) \vdash a_1 : \tau_1 \,/\, C_1.$$

By proposition 4.8, writing $\varphi = \mu \circ \varphi_1$, we get a proof of:

$$\varphi(E) + f \mapsto (\varphi(t_1) \multimap\langle \varphi(u) \rangle\rightarrow \varphi(t_2)) + x \mapsto \varphi(t_1) \vdash a_1 : \mu(\tau_1) \,/\, \mu(C_1).$$

The substitution μ is a unifier of $\varphi_1(t_2)$ and τ_1. Hence $\varphi(t_2) = \mu(\varphi_1(t_2)) = \mu(\tau_1)$. We therefore have shown that:

$$\varphi(E) + f \mapsto (\varphi(t_1) \multimap\langle \varphi(u) \rangle\rightarrow \varphi(t_2)) + x \mapsto \varphi(t_1) \vdash a_1 : \varphi(t_2) \,/\, \mu(C_1).$$

Moreover, C_1 contains $\varphi_1(C_0)$ by remark (5), hence a fortiori the constraints $\varphi_1(E(y)) \triangleleft \varphi_1(u)$ for all y free in a. Hence, $\mu(C_1)$ contains the constraints $\varphi(E(y)) \triangleleft \varphi(u)$ for all y free in a, and we can apply the typing rule for functions, obtaining:

$$\varphi(E) \vdash (f \texttt{ where } f(x) = a_1) : \varphi(t_1 \multimap\langle u \rangle\rightarrow t_2) \,/\, \mu(C_1).$$

This is the expected result.

• **Case** $a = a_1(a_2)$. We apply the induction hypothesis to the two recursive calls to `Infer`. We get proofs of:

$$\varphi_1(E) \vdash a_1 : \tau_1 \,/\, C_1 \qquad \varphi_2(\varphi_1(E)) \vdash a_2 : \tau_2 \,/\, C_2.$$

We apply the substitutions $\mu \circ \varphi_2$ to the leftmost judgement, and μ to the rightmost judgement. By proposition 4.8, we get proofs of:

$$\mu(\varphi_2(\varphi_1(E))) \vdash a_1 : \mu(\varphi_2(\tau_1)) \,/\, \mu(\varphi_2(C_1)) \qquad \mu(\varphi_2(\varphi_1(E))) \vdash a_2 : \mu(\tau_2) \,/\, \mu(C_2).$$

We then extend $\mu(\varphi_2(C_1))$ to $\mu(C_2)$. By remark (5), we have $\mu(\varphi_2(C_1)) \subseteq \mu(C_2)$. By remark (6), none of the generic variables free in $\mu(\varphi_2(\varphi_1(E)))$ is free in $\mu(C_2) \setminus \mu(\varphi_2(C_1))$. (The substitution μ does not introduce new generic variables.) Hence, by proposition 4.9:

$$\mu(\varphi_2(\varphi_1(E))) \vdash a_1 : \mu(\varphi_2(\tau_1)) \,/\, \mu(C_2).$$

Moreover, μ is a unifier of $\varphi_2(\tau_1)$ and $\tau_2 \dashv\langle u\rangle\!\rightarrow t$. Taking $\tau = \mu(t)$ and $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $C' = \mu(C_2)$, as in the algorithm, we have therefore established that:

$$\varphi(E) \vdash a_1 : \mu(\tau_2) \dashv\langle\mu(u)\rangle\!\rightarrow \tau \,/\, C' \qquad \varphi(E) \vdash a_2 : \mu(\tau_2) \,/\, C'.$$

The expected result follows by the typing rule for function applications:

$$\varphi(E) \vdash a_1(a_2) : \tau \,/\, C'.$$

□

Proposition 4.28 (Completeness of type inference) *Let a be an expression, E be a typing environment, C be a constraint set, and V be a set of non-generic variables containing infinitely many type variables and infinitely many labels, and such that $\mathcal{F}(E) \cap V = \emptyset$ and $\mathcal{F}(C) \cap V = \emptyset$. If there exists a type $\overline{\tau}$, a substitution $\overline{\varphi}$ of non-generic types for non-generic type variables, and a constraint set $\overline{C}$ such that*

$$\overline{\varphi}(E) \vdash a : \overline{\tau} \,/\, \overline{C} \quad \textit{and} \quad \overline{\varphi}(C) \subseteq \overline{C} \quad \textit{and} \quad \mathcal{F}_g(\overline{\varphi}(E)) \cap \mathcal{F}_g(\overline{C} \setminus \overline{\varphi}(C)) = \emptyset,$$

then $(\tau, C', \varphi, V') = \texttt{Infer}(a, E, C, V)$ is defined, and there exists a substitution ψ such that

$$\overline{\tau} = \psi(\tau) \quad \textit{and} \quad \overline{C} \supseteq \psi(C' \restriction \{\tau, \varphi(E)\}) \quad \textit{and} \quad \overline{\varphi} = \psi \circ \varphi \textit{ outside } V.$$

(That is, $\overline{\varphi}(\alpha_n) = \psi(\varphi(\alpha_n))$ for all non-generic variables $\alpha_n \notin V$.)

Proof: the proof is by induction on the derivation of $\overline{\varphi}(E) \vdash a : \overline{\tau} \,/\, \overline{C}$, and by case analysis on the last rule used. The proof closely follows the proof of proposition 1.9, with some additional constraint handling. I show four representative cases.

• **Instantiation rule.**

$$\frac{\tau \leq \overline{\varphi}(E(x)) \,/\, \overline{C}}{\overline{\varphi}(E) \vdash x : \overline{\tau} \,/\, \overline{C}}$$

$\texttt{Infer}(x, E, C, V)$ is defined, since $x \in \mathrm{Dom}(e)$, and returns $\tau = \theta(E(x))$ and $C' = \theta(C) \cup C$ and $\varphi = [\,]$ and $V' = V \setminus \mathrm{Codom}(\varphi)$, for some renaming $\theta : \mathcal{F}_g(E(x)) \Leftrightarrow V$. By definition of the instantiation relation, we have $\tau' = \rho(\overline{\varphi}(E(x)))$ for some substitution $\rho : \mathcal{F}_g(\varphi(E(x))) \Rightarrow \texttt{TypNongen}$, and moreover $\rho(\overline{C}) \subset \overline{C}$. Define $\psi = \rho \circ \overline{\varphi} \circ \theta^{-1}$. First of all, we have:

$$\psi(\tau) = \rho(\overline{\varphi}(\theta^{-1}(\tau))) = \rho(\overline{\varphi}(E(x))) = \tau'.$$

Then, for all variables $\alpha_n \notin V$, we have $\alpha \notin \mathrm{Dom}(\theta^{-1})$, hence $\psi(\alpha_n) = \rho(\overline{\varphi}(\alpha_n)) = \overline{\varphi}(\alpha_n)$, since $\overline{\varphi}(\alpha_n)$ is a non-generic type. Finally, let us show that $\psi(C') \subseteq \overline{C}$. We have:

$$\psi(C') = \psi(C) \cup \psi(\theta(C)) = \rho(\overline{\varphi}(\theta^{-1}(C))) \cup \rho(\overline{\varphi}(C)) = \rho(\overline{\varphi}(C)) \cup \rho(\overline{\varphi}(C)),$$

since $V \cap \mathcal{F}(C) = \emptyset$ and $\mathrm{Dom}(\theta^{-1}) \subseteq V$. On the other hand, we know that:

$$\rho(\overline{\varphi}(C)) \subseteq \rho(\overline{C}) \subseteq \overline{C}.$$

The expected result follows.

• **Function abstraction rule.**

$$\frac{\overline{\varphi}(E) + f \mapsto \overline{\tau}_2 \multimap\langle\overline{u}\rangle\rightarrow \overline{\tau}_1 + x \mapsto \overline{\tau}_2 \vdash a_1 : \overline{\tau}_1 \,/\, \overline{C} \qquad (\overline{\varphi}(E(y)) \triangleleft \overline{u}) \in \overline{C} \text{ for all } y \in \mathcal{I}(f \texttt{ where } f(x) = a)}{\overline{\varphi}(E) \vdash (f \texttt{ where } f(x) = a_1) : \overline{\tau}_2 \multimap\langle\overline{u}\rangle\rightarrow \overline{\tau}_1 \,/\, \overline{C}}$$

Let t_1 and t_2 be two non-generic type variables taken from V, and u be a non-generic label taken from V. Consider the environment

$$E_1 = E + f \mapsto (t_2 \multimap\langle u\rangle\rightarrow t_1) + x \mapsto t_2$$

and the substitution

$$\overline{\varphi}_1 = \overline{\varphi} + t_1 \mapsto \overline{\tau}_1 + t_2 \mapsto \overline{\tau}_2 + u \mapsto \overline{u}.$$

The first premise of the typing rule above also reads $\overline{\varphi}_1(E_1) \vdash a : \overline{\tau}_1 \,/\, \overline{C}$. We apply the induction hypothesis to this premise. This shows that

$$(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E_1, C, V \setminus \{t_1, t_2, u\})$$

is defined. Moreover, we get a substitution ψ_1 such that:

$$\overline{\tau}_1 = \psi_1(\tau_1) \quad \text{and} \quad \overline{C} \supseteq \psi_1(C_1 \restriction \{\tau_1, \varphi_1(E_1)\}) \quad \text{and} \quad \overline{\varphi}_1 = \psi_1 \circ \varphi_1 \text{ outside } V \setminus \{t_1, t_2, u\}.$$

We have $\psi_1(\varphi_1(t_1)) = \overline{\varphi}_1(t_1) = \overline{\tau}_1 = \psi_1(\tau_1)$. The types $\varphi_1(t_1)$ and τ_1 therefore admit ψ_1 as a unifier. Hence $\mu = \texttt{mgu}(\varphi_1(t_1), \tau_1)$ exists. It follows that $\texttt{Infer}(E, a, C, V)$ is defined. Moreover, we have $\psi_1 = \psi \circ \mu$ for some substitution ψ. The algorithm takes $\varphi = \mu \circ \varphi_1$ and $\tau = \mu(\varphi_1(t_2 \multimap\langle u\rangle\rightarrow t_1))$. We have:

$$\psi(\tau) = \psi(\mu(\varphi_1(t_2 \multimap\langle u\rangle\rightarrow t_1))) = \psi_1(\varphi_1(t_2 \multimap\langle u\rangle\rightarrow t_1)) = \overline{\varphi}_1(t_2 \multimap\langle u\rangle\rightarrow t_1) = \overline{\tau}_2 \multimap\langle\overline{u}\rangle\rightarrow \overline{\tau}_1 = \overline{\tau}.$$

(That's because t_1, t_2 and u do not belong to $V \setminus \{t_1, t_2, u\}$, hence these variables have the same image under $\overline{\varphi}_1$ and under $\psi_1 \circ \varphi_1$.) Similarly, for all $\alpha_n \notin V'$, we have a fortiori $\alpha_n \notin V$ and $\alpha \notin \{t_1, t_2, u\}$, hence

$$\psi(\varphi(\alpha_n)) = \psi(\mu(\varphi_1(\alpha_n))) = \psi_1(\varphi_1(\alpha_n)) = \overline{\varphi}_1(\alpha_n) = \overline{\varphi}(\alpha_n).$$

Finally, let us show that $\overline{C} \supseteq \psi(C' \restriction \{\tau, \varphi(E)\})$. By proposition 4.26, we get

$$\mu(C_1 \restriction \{\tau_1, \varphi_1(E_1)\}) = \mu(C_1) \restriction \{\mu(\tau_1), \mu(\varphi_1(E_1))\}.$$

Moreover, by definition of E_1,

$$\mu(C_1) \restriction \{\mu(\tau_1), \mu(\varphi_1(E_1))\} = \mu(C_1) \restriction \{\mu(\tau_1), \mu(\varphi_1(E))\} = \mu(C_1) \restriction \{\tau, \varphi(E)\}.$$

That's because the type variables free in $\mu(\varphi_1(E_1)) \,/\, \mu(C_1)$ but not free in $\mu(\varphi_1(E)) \,/\, \mu(C_1)$ are a subset of the variables free in $\mu(\varphi_1(t_2 \dashv\langle u \rangle\!\rightarrow t_1)) \,/\, \mu(C_1)$, that is, free in $\tau \,/\, \mu(C_1)$. And by induction hypothesis we know that

$$\overline{C} \supseteq \psi(\mu(C_1 \restriction \{\tau_1, \varphi_1(E_1)\})).$$

Hence

$$\overline{C} \supseteq \psi(\mu(C_1) \restriction \{\tau, \varphi(E)\}).$$

Moreover, given the typing rule for `where`, we know that

$$(\overline{\varphi}(E(y)) \triangleleft \overline{u}) \in \overline{C} \text{ for all } y \in \mathcal{I}(f \texttt{ where } f(x) = a).$$

We have $\overline{\varphi}(E) = \psi(\varphi(E))$, since the free variables of E are outside V. Hence

$$(\psi(\varphi(E(y))) \triangleleft \overline{u}) \in \overline{C} \text{ for all } y \in \mathcal{I}(f \texttt{ where } f(x) = a).$$

Since, by definition, $C' = \mu(C_1) \cup \{\varphi(E(y)) \triangleleft \varphi(u) \mid y \in \mathcal{I}(a)\}$, we have shown that

$$\overline{C} \supseteq \psi(C' \restriction \{\tau, \varphi(E)\}).$$

This completes the proof in the case of function abstraction.

• **The `let` rule.**

$$\frac{\overline{\varphi}(E) \vdash a_1 : \overline{\tau}_1 \,/\, \overline{C}_1 \qquad (\overline{\sigma}, \overline{C}) = \texttt{Gen}(\overline{\tau}_1, \overline{C}_1, \overline{\varphi}(E)) \qquad \overline{\varphi}(E) + x \mapsto \overline{\sigma} \vdash a_2 : \overline{\tau}_2 \,/\, \overline{C}}{\overline{\varphi}(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \overline{\tau}_2 \,/\, \overline{C}}$$

By the induction hypothesis applied to the left premise, it follows that

$$(\tau_1, C_1, \varphi_1, V_1) = \texttt{Infer}(a_1, E, C, V)$$

is defined, and there exists a substitution ψ_1 such that

$$\overline{\tau}_1 = \psi_1(\tau_1) \quad \text{and} \quad \overline{C}_1 \supseteq \psi_1(C_1 \restriction \{\tau_1, \varphi_1(E)\}) \quad \text{and} \quad \overline{\varphi} = \psi_1 \circ \varphi_1 \text{ outside } V.$$

In particular, we have $\overline{\varphi}(E) = \psi_1(\varphi_1(E))$.

Let $(\sigma, C_0) = \texttt{Gen}(\tau_1, C_1, \varphi_1(E))$. It is easy to check that all instances of $\overline{\sigma} \,/\, \overline{C}$ are also instances of $\psi_1(\sigma) \,/\, \overline{C}$. From the right premise of the `let` typing rule, we therefore obtain a proof of

$$\psi_1(\varphi_1(E) + x \mapsto \sigma) \vdash a_2 : \overline{\tau}_2 \,/\, \overline{C}.$$

We apply the induction hypothesis to this judgement. It follows that

$$(\tau_2, C_2, \varphi_2, V_2) = \texttt{Infer}(a_2,\ \varphi_1(E) + x \mapsto \sigma,\ C_0,\ V_1)$$

is defined, and there exists a substitution ψ_2 such that

$$\overline{\tau}_2 = \psi_2(\tau_2) \quad \text{and} \quad \overline{C} \supseteq \psi_2(C_2 \restriction \{\tau_2, \varphi_2(\varphi_1(E))\}) \quad \text{and} \quad \psi_1 = \psi_2 \circ \varphi_2 \text{ outside } V_1.$$

We then show that $\psi = \psi_2$ meets the conditions of the claim, as in the case of functions.

• **Simplification rule.**

$$\frac{\overline{\varphi}(E) \vdash a : \overline{\tau} \,/\, \overline{C}' \qquad \overline{\tau} \,/\, \overline{C} \equiv \overline{\tau} \,/\, \overline{C}' \qquad \overline{\varphi}(E) \,/\, \overline{C} \equiv \overline{\varphi}(E) \,/\, \overline{C}'}{\overline{\varphi}(E) \vdash a : \overline{\tau} \,/\, \overline{C}}$$

We apply the induction hypothesis to the derivation of $\overline{\varphi}(E) \vdash a : \overline{\tau} \,/\, \overline{C}'$. It follows that $(\tau, \varphi, C', V') = \texttt{Infer}(a, E, C, V)$ is defined, and, for some substitution ψ, we have

$$\overline{\tau} = \psi(\tau) \quad \text{and} \quad \overline{C}' \supseteq \psi(C \restriction \{\tau, \varphi(E)\}) \quad \text{and} \quad \overline{\varphi} = \psi \circ \varphi \text{ outside } V.$$

To obtain the expected result, it suffices to show $\overline{C} \supseteq \psi(C \restriction \{\tau, \varphi(E)\})$. The following inclusions hold:

$$\psi(C \restriction \{\tau, \varphi(E)\}) \subseteq \psi(C) \restriction \{\psi(\tau), \psi(\varphi(E))\} = \psi(C) \restriction \{\overline{\tau}, \overline{\varphi}(E)\} \subseteq \overline{C}' \restriction \{\overline{\tau}, \overline{\varphi}(E)\}.$$

According to the second and third premises of the simplification rule, the rightmost set is equal to $\overline{C} \restriction \{\overline{\tau}, \overline{\varphi}(E)\}$, which is a subset of $\overline{C}$. Hence the expected result. □

4.5 Conservativity

In this section, we show that the indirect type system is a proper extension of Milner's type system (chapter 1): any program that does not use references, channels, or continuations, and is well-typed in Milner's type system, is also well-typed in the indirect type system.

In this section, what we call a "pure expression" is an expression a that does not use any of the operators over references, channels, and continuations introduced in chapter 2. A pure expression can be typed in two ways: either in Milner's type system (the type system for the purely applicative calculus in chapter 1); or in the indirect type system with dangerous variables and closure typing (the indirect system, for short) introduced in the present chapter. The goal of this section is to show that all pure expressions that are well-typed in Milner's system are also well-typed in the indirect system; in other terms, that the indirect system is a conservative extension of Milner's system.

Proposition 4.29 *Let a be a pure expression and ι be a base type. If we can prove that $[\,] \vdash a : \iota$ in Milner's type system (section 1.3.5), then, there exists a constraint set C such that we can prove $[\,] \vdash a : \iota \,/\, C$ in the indirect system (section 4.2.4).*

To prove this claim, the natural approach is to take a typing derivation in Milner's system and to transform it into a typing derivation in the indirect system. The problem with this approach is that we have to annotate the function types and synthesize a constraint set in a globally consistent way. This can be done, but duplicates large parts of the type inference algorithm for the indirect system and of its correctness proof.

Remark. On the other hand, the technique outlined above works well for showing the converse result: all pure programs that are well-typed in the indirect system are also well-typed in Milner's type system. That's because it suffices to erase the labels and the constraints from a typing derivation in the indirect system, and transform generic types into type schemes, to obtain a valid typing derivation in Milner's system. □

The approach followed in this section requires less work. It consists in comparing the type inference algorithms for the two systems, and showing that if the inference algorithm for Milner's system terminates successfully, then the inference algorithm for the indirect system terminates successfully. The idea is that these two algorithms perform nearly identical operations over nearly identical data, and that unification between two indirect types cannot fail because of the labels (which are mere variables, hence always unifiable).

To precisely relate the two type inference algorithms, we define a *label erasing* operation, written $\Downarrow$, that maps type expressions from the indirect system back to type expression from Milner's type system. The type $\sigma\Downarrow$ is called the SKELETON of σ.

$$\begin{aligned} \iota\Downarrow &= \iota \\ t\Downarrow &= t \\ (\sigma_1 \multimap\!\langle u\rangle\!\rightarrow \sigma_2)\Downarrow &= \sigma_1\Downarrow \rightarrow \sigma_2\Downarrow \\ (\sigma_1 \times \sigma_2)\Downarrow &= \sigma_1\Downarrow \times \sigma_2\Downarrow \end{aligned}$$

Similarly, if V is a set of labels and type variables, we write $V\Downarrow$ for the set V without the labels. We define a variant of $\Downarrow$, written $\Downarrow_s$, that turns an indirect generic type into a type scheme from Milner's type system:

$$\sigma\Downarrow_s = \forall t_1, \ldots, t_n.\, \sigma\Downarrow \quad \text{with} \quad \{t_1 \ldots t_n\} = \mathcal{F}_g(\sigma)\Downarrow.$$

We extend $\Downarrow$ and $\Downarrow_s$ to substitutions and to typing environments, pointwise. It is easy to check that $(\varphi \circ \psi)\Downarrow = \varphi\Downarrow \circ \psi\Downarrow$ for all substitutions φ and ψ.

The $\Downarrow$ operator is not defined over types of the form σ `ref`, σ `chan` ou σ `cont`. That's because all type expressions considered below do not contain any of the constructors `ref`, `chan` ou `cont`. It is easy to convince oneself that the type inference algorithm for the indirect system does not produce any type containing `ref`, `chan` ou `cont` when it is applied to a pure expression.

The following property of the indirect type algebra is crucial to the conservativity proof:

Proposition 4.30 *Two types τ_1 et τ_2 are unifiable if and only if their skeletons $\tau_1\Downarrow$ and $\tau_2\Downarrow$ are unifiable. In this case,* $\texttt{mgu}(\tau_1, \tau_2)\Downarrow$ *is equal, up to a renaming, to* $\texttt{mgu}(\tau_1\Downarrow, \tau_2\Downarrow)$.

Proof: if φ is a unifier of τ_1 and τ_2, we obviously have

$$\varphi\Downarrow(\tau_1\Downarrow) = (\varphi(\tau_1))\Downarrow = (\varphi(\tau_2))\Downarrow = \varphi\Downarrow(\tau_2\Downarrow)$$

hence $\tau_1\Downarrow$ and $\tau_2\Downarrow$ are unifiable, and $\varphi\Downarrow$ is one of their unifiers. Taking φ to be the principal unifier of τ_1 and τ_2, we obtain that $\texttt{mgu}(\tau_1, \tau_2)\Downarrow$ is less general than $\texttt{mgu}(\tau_1\Downarrow, \tau_2\Downarrow)$.

Conversely, let ψ be a unifier of $\tau_1\Downarrow$ and $\tau_2\Downarrow$. Let u be some arbitrary label. We define a lifting operator $\Uparrow$ that maps Milner types to indirect types as follows:

$$\begin{array}{rcl} \iota\Uparrow & = & \iota \\ t\Uparrow & = & t \\ (\tau_1 \to \tau_2)\Uparrow & = & \tau_1\Uparrow \multimap\langle u\rangle\rightarrowtail \tau_2\Uparrow \\ (\tau_1 \times \tau_2)\Uparrow & = & \tau_1\Uparrow \times \tau_2\Uparrow \end{array}$$

We obviously have $\tau\Uparrow\Downarrow = \tau$ for all Milner type τ. Consider the substitution φ defined by

$$\varphi = \psi\Uparrow + u_1 \mapsto u + \cdots + u_n \mapsto u,$$

where $u_1, \ldots, u_n$ are the labels free in τ_1 or in τ_2. It is easy to check that φ is a unifier of τ_1 and τ_2. Moreover, $\varphi\Downarrow = \psi\Uparrow\Downarrow = \psi$. Now, take φ to be the principal unifier of $\tau_1\Downarrow$ and $\tau_2\Downarrow$. The associated substitution φ is necessarily equal to $\xi \circ \mathtt{mgu}(\tau_1, \tau_2)$ for some ξ. Hence

$$\mathtt{mgu}(\tau_1\Downarrow, \tau_2\Downarrow) = \varphi\Downarrow = \xi\Downarrow \circ \mathtt{mgu}(\tau_1, \tau_2)\Downarrow.$$

Hence $\mathtt{mgu}(\tau_1\Downarrow, \tau_2\Downarrow)$ is less general than $\mathtt{mgu}(\tau_1, \tau_2)\Downarrow$. It follows that these two substitutions are equal up to a renaming. □

We are now going to parallel the two type inference algorithms: the Damas-Milner algorithm and the algorithm for the indirect system. To avoid ambiguities, we write $W(a, E, V)$ for the result of the Damas-Milner algorithm (algorithm 1.1) and $I(a, E, C, V)$ for the result of the algorithm for the indirect system (algorithm 4.1)

Proposition 4.31 *Let a be a pure expression, E be a typing environment from the indirect system, V be an infinite set of variables and labels, and C be a constraint set. If $(\tau_w, \varphi_w, V_w) = W(a, E\Downarrow_s, V\Downarrow)$ is defined, then $(\tau_i, C_i, \varphi_i, V_i) = I(a, E, C, V)$ is also defined, and the following equalities hold:*

$$\tau_i\Downarrow = \tau_w \quad \textit{and} \quad \varphi_i\Downarrow = \varphi_w \quad \textit{and} \quad V_i\Downarrow = V_w$$

up to a renaming of the variables in V_i to variables in V_i.

Proof: the proof is by structural induction over a. We show three representative cases; the remaining cases are similar.

• **Case** $a = x$. Since $W(E\Downarrow_s, x)$ is defined, we have $x \in \mathrm{Dom}(E\Downarrow_s) = \mathrm{Dom}(E)$, hence $I(x, E, C, V)$ is defined. Moreover, we have $(\tau_w, V_w) = \mathtt{Inst}(x, E(x)\Downarrow_s, V\Downarrow)$ and $(\tau_i, C_i, V_i) = \mathtt{Inst}(x, E(x)\Downarrow, V\Downarrow)$. Taking $\{t_1, \ldots, t_n, u_1, \ldots, u_m\} = \mathcal{F}_g(E(x))$, we have $E(x)\Downarrow_s = \forall t_1 \ldots t_n.\, E(x)\Downarrow$. Hence τ_w is equal to $[t_1 \mapsto t'_1, \ldots, t_n \mapsto t'_n](E(x)\Downarrow)$ for some variables t'_i taken from V. Taking $\theta = [t_1 \mapsto t'_1, \ldots, t_n \mapsto t'_n, u_1 \mapsto u'_1, \ldots, u_m \mapsto u'_m]$ as the instantiation substitution for $E(x)$, we have as expected:

$$\tau_i\Downarrow = \theta(E(x))\Downarrow = \theta\Downarrow(E(x)\Downarrow) = \tau_w.$$

Similarly,

$$V_i\Downarrow = (V \setminus \{t'_1, \ldots, t'_n, u'_1, \ldots, u'_m\})\Downarrow = V\Downarrow \setminus \{t'_1, \ldots, t'_n\} = V_w.$$

Finally, $\varphi_i = \varphi_w = [\,]$. The expected result follows.

• **Case** $a = (f \texttt{ where } f(x) = a_1)$. Let t_1 and t_2 be the fresh variables chosen by the algorithm W. After renaming if necessary, we can assume that the algorithm I chooses the same variables. We know that

$$(\tau_{w1}, \varphi_{w1}, V_{w1}) = W(a_1, E{\Downarrow_s} + f \mapsto t_1 \to t_2 + x \mapsto t_1, V{\Downarrow} \setminus \{t_1, t_2\})$$

is defined. Applying the induction hypothesis to the expression a_1, it follows that

$$(\tau_{i1}, C_{i1}, \varphi_{i1}, V_{i1}) = I(a_1, E + f \mapsto t_1 \multimap\langle u\rangle\!\!\to t_2 + x \mapsto t_1, C, V \setminus \{t_1, t_2, u\})$$

is well-defined. Moreover, after renaming some variables from V to variables from V if necessary, we have $\tau_{i1}{\Downarrow} = \tau_{w1}$, and $\varphi_{i1}{\Downarrow} = \varphi_{w1}$, and $V_{i1}{\Downarrow} = V_{w1}$. Moreover, we know that $\varphi_{w1}(t_2)$ and τ_{w1} are unifiable. But $\varphi_{w1}(t_2) = (\varphi_{i1}(t_2)){\Downarrow}$ and $\tau_{w1} = \tau_{i1}{\Downarrow}$. Hence, by proposition 4.30, $\varphi_{i1}(t_2)$ and τ_{i1} are unifiable. This shows that $I(a, E, C, V)$ is well-defined. Moreover, defining $\mu_w = \texttt{mgu}(\varphi_{1w}(t_2), \tau_{w1})$ and $\mu_i = \texttt{mgu}(\varphi_{i1}(t_2), \tau_{i1})$, we have $\mu_w = \mu_i{\Downarrow}$. Hence, taking into account the definition of τ_w and τ_i:

$$\tau_i{\Downarrow} = \mu_i(\varphi_{i1}(t_1 \multimap\langle u\rangle\!\!\to t_2)){\Downarrow} = \mu_i{\Downarrow}(\varphi_{i1}{\Downarrow}((t_1 \multimap\langle u\rangle\!\!\to t_2){\Downarrow})) = \mu_w(\varphi_{w1}(t_1 \to t_2)) = \tau_w.$$

Similarly,

$$\varphi_i{\Downarrow} = (\mu_i \circ \varphi_{i1}){\Downarrow} = \mu_i{\Downarrow} \circ \varphi_{i1}{\Downarrow} = \mu_w \circ \varphi_{w1} = \varphi_w$$

$$V_i{\Downarrow} = (V_{i1} \cup \{t_1, t_2, u\}){\Downarrow} = V_{w1} \cup \{t_1, t_2\} = V_w.$$

• **Case** $a = (\texttt{let } x = a_1 \texttt{ in } a_2)$. We know that $(\tau_{w1}, \varphi_{w1}, V_{w1}) = W(a_1, E{\Downarrow_s}, V{\Downarrow})$ is defined. Hence, by induction hypothesis, $(\tau_{i1}, \varphi_{i1}, C_1, V_{i1}) = I(a_1, E, C, V)$ is defined, and $\tau_{i1}{\Downarrow} = \tau_{w1}$, and $\varphi_{i1}{\Downarrow} = \varphi_{w1}$. Define $(\sigma_i, C_i) = \texttt{Gen}(\tau_{i1}, \varphi_{i1}(E), C_{i1})$, and $\sigma_w = \texttt{Gen}(\tau_{w1}, \varphi_{w1}(E{\Downarrow_s}))$. We now show that $\sigma' = \sigma{\Downarrow_s}$. First of all, $\mathcal{F}(\tau_{w1}) = \mathcal{F}(\tau_{i1}{\Downarrow}) = \mathcal{F}(\tau_{i1}){\Downarrow}$, and similarly $\mathcal{F}(\varphi_{i1}(E{\Downarrow_s})) = \mathcal{F}_n(\varphi_{i1}(E)){\Downarrow}$. Then, $\mathcal{D}(\tau_{i1} / C_1) = \emptyset$. That's because a_1 is a pure expression, hence the constrained type τ_{i1}/C_1 returned by I contains no sub-term of the form τ `ref`, ou τ `chan`, ou τ `cont`. Similarly, $\mathcal{D}(\varphi_{i1}(E) \,/\, C_1) = \emptyset$. Hence, the variables generalized by I are 1- the type variables generalized by W, plus 2- some labels, plus 3- some type variables indirectly free in $\tau_{i1} \,/\, C_{i1}$, but not directly free in τ_{w1}. Hence, given the axioms over type schemes from section 1.3.3, $\sigma_w = \sigma_i{\Downarrow_s}$. Since $(\tau_{w2}, \varphi_{w2}, V_{w2}) = W(a_2, E{\Downarrow_s} + x \mapsto \sigma_w, V_1{\Downarrow})$ is defined by hypothesis, the induction hypothesis shows that $(\tau_{i2}, \varphi_{i2}, C_2, V_{i2}) = I(a_2, E + x \mapsto \sigma_i, C_i, V_{i1})$ is defined. Moreover, $\tau_{i2}{\Downarrow} = \tau_{w2}$ and $\varphi_{i2}{\Downarrow} = \varphi_{i2}$ and $V_{i2}{\Downarrow} = V_{w2}$. Hence we conclude that $I(a, E, C, V)$ is defined, and the following equalities hold:

$$\tau_i{\Downarrow} = \tau_{i2}{\Downarrow} = \tau_{w2} = \tau_w$$

$$\varphi_i{\Downarrow} = (\varphi_{i2} \circ \varphi_{i1}){\Downarrow} = \varphi_{i2}{\Downarrow} \circ \varphi_{i1}{\Downarrow} = \varphi_{w2} \circ \varphi_{w1} = \varphi_w$$

$$V_i{\Downarrow} = V_{i2}{\Downarrow} = V_{w2} = V_w.$$

This is the expected result. □

Proposition 4.29 immediately follows from the result above:

Proof: of proposition 4.29. We recall the hypotheses. Let a be a pure expression and ι be a base type. We assume that we can derive $[\,] \vdash a : \iota$ in Milner's system. From the completeness of algorithm W (proposition 1.9), it follows that $W([\,], a)$ is defined, and returns (ι, φ') for some φ' as result. By proposition 4.31, we deduce that $(\tau, \varphi, C) = I([\,], a)$ is defined. Moreover, $\tau\Downarrow = \iota$. The only possibility is $\tau = \iota$. And, since algorithm I is correct (proposition 4.27), we can derive $[\,] \vdash a : \iota \,/\, C$ in the indirect system. This is the expected result. □

Remark. Using the same techniques, one can show that the indirect type system in the present chapter is an extension of the direct system in chapter 3: all programs (pure or not) that are well-typed in the system in chapter 3 are also well-typed in the indirect system. □

Chapter 5

Comparisons

In this chapter, we compare the type systems studied in chapters 3 and 4 with other proposed polymorphic type systems for ML enriched with references. (Some of these systems have been applied to other extensions of ML, besides references. The comparison is carried only for references, the common feature of all these systems.) The comparison criterias are expressiveness (how many correct programs are recognized as well-typed?), on the one hand, and on the other hand, the practicality of these systems, in particular with respect to modular programming.

Concerning expressiveness, it turns out that there are generally no proper inclusions between these systems: in most cases, there exists one example that is well-typed in one system, but ill-typed in another. Hence the comparison can only be carried on examples that are representative of actual programming situations. I therefore start by presenting the test programs used, before commenting the results obtained with the various type systems.

5.1 Presentation of the test programs

The first test measures the capacity of the systems to polymorphically type functions that operate on mutable data structures. The first test is the typing of the function

$$\texttt{let make\_ref} = \lambda \texttt{x}.\,\texttt{ref(x)}$$

The `make_ref` function is the paradigm of most generic functions over arrays, matrices, hash tables, B-trees, in-place balanced trees, and so on. A good system must absolutely give a generic type to the `make_ref` function. Failure, here, means that we cannot implement these useful but complex data structures once and for all, in a library.

The second test determines whether it is possible to write generic functions in a non-purely applicative style, for instance by using local references to accumulate intermediate results. The test consists in typing the function

$$\begin{array}{l}\texttt{let imp\_map} = \lambda \texttt{f}.\,\lambda \texttt{l}.\\ \quad \texttt{let arg} = \texttt{ref(l) and res} = \texttt{ref([])}\ \texttt{in}\end{array}$$

```
while not null(!arg) do
    res := f(head(!arg)) :: !res;
    arg := tail(!arg)
done;
reverse(!res)
```

then, its application to the identity function and to the empty list, `imp_map id []`. We also compare the type of `imp_map` with the type of the equivalent, but purely applicative function:

```
let rec appl_map = λf. λarg.
    if null(arg) then [] else
        f(head(arg)) :: appl_map f (tail(arg))
```

Many functions over graphs, for instance (depth-first or breadth-first search, computation of the connex components, computation of a shortest path, topological sort, etc [88, chapters 29–34]) typecheck in the same way as the `imp_map` function. A good type system must give the same type to the two functions `imp_map` et `appl_map`; the application `imp_map id []` must have the polymorphic type $\forall\alpha.\,\alpha$ `list`. If a type system does not pass this test, this means that polymorphism does not interact satisfactorily with the non-purely applicative features in this system.

The third test measures the compatibility between functions over mutable structures and higher-order functions. It consists in applying the identity function to the `make_ref` function defined above:

```
id (make_ref)
```

A more realistic variant of this test consists in partially applying the `appl_map` functional defined above to the `make_ref` function. Finally, we also test the partial application of the `imp_map` functional to the identity function. A good system must assign the same type to `id make_ref` and to `make_ref`, and must give a generic type to `map make_ref` and to `imp_map id`. Otherwise, we can conclude that the type system does not correctly handle full functionality.

The fourth tests is more artificial and more specific to the approach taken in this dissertation. It detects the problems mentioned at the beginning of chapter 4: infinite type expressions (section 4.1.1) and variable capture through the environment (section 4.1.2). For each of the two problems, we give two variants of the test, one without references, the other with references. For the problem of infinite types, we consider:

```
let eta = λf. either f (λx.f(x))
```

where the `either` function is defined as:

```
let either = λa. λb. if cond then a else b
```

where *cond* is any boolean expression. The `either` function is intended to constraint its two arguments to have the same type. Here is the variant with references of this test:

```
let eta_ref = λf.
    let r = ref(f) in
        either f (λx.!(either r (ref(f)))(x))
```

The program that exercises variable capture is the counterexample from section 4.1.2:

```
let capt_id = λf.
    let id = λy. (either f (λz.y;z)); y
    in id(id)
```

Here is the variant with a few references added:

```
let capt_id_ref = λf.
    let id = λy.
        let r = ref(y) in
            either f (λz.either r (ref(y));z);
        y
    in id(id)
```

A good system must be a proper extension of the ML type system, and therefore it must accept the "pure" versions of these two programs. The variants with references are semantically correct, and therefore should be accepted, though failure to detect this fact is not too serious in practice.

The final test is highly artificial, too. It mimics the creation of a polymorphic reference as far as types are concerned, without actually creating a reference:

```
let fake_ref = (raise An_exception : α ref)
```

We have used an exception and a type constrait for the sake of clarity, but these two features are not essential. Here is a similar example that does not use them:

```
let fake_ref = (λx.!x;x)(loop(0) where loop(x) = loop(x))
```

This test illustrates the difference between detecting the creation of references, by an extra analysis superimposed over typing, and detecting the presence of references, by mere examination of the types: the former approach correctly concludes that nothing dangerous is happening; the latter approach notices that the result has type α `ref`, and incorrectly concludes that we might be creating a polymorphic reference here.

5.2 Comparison with other type systems

The results of the tests are summarized in figure 5.1. We follow the convention that all examples are toplevel phrases, and therefore their types must be closed. A dash — means that the example is rejected because some type variables are not generalizable in its type, and thus its type cannot be closed. A smiley ☺ means that the type can be closed, and therefore the example is accepted.

5.2.1 Weak variables

A first approach to the control of polymorphic references consists in typing specially the primitives that create references, in order to mark the type variables that are free in the type of the reference created. These marked variables are called *weak variables*, or *imperative variables*, and are subject to generalization conditions more restrictive than those for the regular variables. This approach is followed by the first three systems presented below, with various restrictions over generalization, from the strongest to the weakest.

— Criterion	CAML [5.2.1.1]	SML [5.2.1.2]	Damas [5.2.2.3]	SML-NJ [5.2.1.3]	Simple effects [5.2.2.1]	Alloc. effects [5.2.2.2]	Effects + regions [5.2.2.4]	Chap. 3	Chap. 4
`make_ref`	—	☺	☺	☺	☺	☺	☺	☺	☺
`imp_map`	—	☺	☺	☺	☺	☺	☺	☺	☺
`imp_map id []`	—	—	—	—	—	—	☺	☺	☺
Same type for `imp_map` and `appl_map id(make_ref)`	—	—	—	—	—	—	☺	☺	☺
`appl_map(make_ref)`	—	—	—	—	—	—	☺	☺	☺
`imp_map(id)`	—	—	—	☺	☺	☺	☺	☺	☺
`eta`	☺	☺	☺	☺	☺	☺	☺	—	☺
`eta_ref`	—	☺	?	☺	☺	—	—	—	☺
`capt_id`	☺	☺	☺	☺	☺	☺	☺	—	☺
`capt_id_ref`	—	☺	?	☺	☺	—	—	—	—
`fake_ref`	☺	☺	☺	☺	☺	☺	☺	—	—

Figure 5.1: Comparison between the type systems

5.2.1.1 Caml

The simplest restriction consists in never generalizing weak variables. Then, a reference can never be assigned a polymorphic type, because 1- its type at creation-time contains only weak variables, 2- the type of any object that gets in touch with the reference is similarly weakened (by unification), and 3- weak variables are never generalized later.

History. This idea seems to have occurred independently to several persons when references subsumed the mutable variables of the original LCF-ML [31, page 52]. No presentation of this idea has been published, nor formal typing rules, nor a soundness proof. This approach is still being employed in the Caml system, for the typing of mutable objects [19, page 41] [99, page 79].

Results. This approach turns out to be extremely restrictive in practice. The classical data structures (hash tables, etc) cannot be implemented as libraries of generic functions: the type system tolerates only monomorphic versions of the functions that create these structures. A fortiori, most generic functions cannot allocate mutable structures holding intermediate results. Hence, genericity and imperative programming are completely incompatible in this system.

5.2.1.2 Standard ML

The next step consists in allowing the generalization of weak variables in the type of non-expansive expressions: those expressions whose evaluation does not create fresh references. Tofte proposed the following syntactic approximation of non-expansiveness: an expression is non-expansive if it is a variable, a constant, or a function $\lambda x.\, a$. All other kinds of expressions are taken to be expansive. Consider the expression $\lambda\mathtt{x.ref(x)}$, which has the weak type $\alpha^* \to \alpha^*$ `list`. (Weak variables are written with a star superscript.) The variable α is marked weak, because this function creates a reference with type α. However, the expression $\lambda\mathtt{x.ref(x)}$ is non-expansive, according to Tofte's classification. Hence, α^* can be generalized in the remainder of the program. Thus, the following example is well-typed:

```
let make_ref = λx.ref(x) in
    ... make_ref(1) ... make_ref(true) ...
```

The body of the `let` is indeed typed under the assumption `make_ref` $: \forall\alpha^*.\, \alpha^* \to \alpha^*$ `list`. In this type scheme, the weak generic variable α^* can only be instantiated by weak types, that is, types that contain only weak variables. This ensures type safety:

```
let make_ref = λx.ref(x) in
let r = make_ref(λx.x) in ...
```

Here, the application `make_ref`$(\lambda\mathtt{x.x})$ has type $(\beta^* \to \beta^*)$ `ref`. The variable β is necessarily weak, since the type by which α^* is instantiated must be weak. Moreover, this expression is taken to be expansive, since it is an application, not a constant, variable or function. Hence β^* is not generalized, thus avoiding the inconsistent use of `r`.

History. This system was introduced by Tofte in 1987, then incorporated into Standard ML. It is therefore employed by all ML implementations that follow the Standard (Edinburgh ML, Poly ML, Poplog ML). It is described in Tofte's thesis [92, 93] and in the definition of Standard ML

[64, 63]. Tofte [92, 93] gives a proof of the soundness of this system for ML with references, using relational semantics and semantic typing predicates. Wright and Felleisen [101] give soundness proofs for this system applied to ML with references, ML with exceptions, and ML with `callcc`, using reduction semantics and the fact that reductions preserve typing.

Results. This system is a considerable improvement over the one of Caml: we can at last define generic functions that create mutable structures; this allows providing implementations for many data structures in libraries (see [7] for an example of such a library).

However, this system has two major weaknesses. First of all, the non-expansiveness criterion is naive, and fails to recognize as non-expansive many expressions that, semantically, are non-expansive; these expressions remain monomorphic, while they should be given a polymorphic type. This phenomenon occurs frequently when we combine higher-order functions with functions that create references. In the example

$$\texttt{let make\_ref2} = (\lambda \texttt{x.x})(\texttt{make\_ref}) \texttt{ in } \ldots$$

the type of `make_ref2` remains monomorphic in the body of the `let`: the variable α^* is not generalizable in its type, $\alpha^* \to \alpha^*$ `ref`, since the expression $(\lambda \texttt{x.x})(\texttt{make\_ref})$ is expansive, according to Tofte's criterion. Obviously, `make_ref2` should be given the same weakly polymorphic type as `make_ref`, since both identifiers have the same value. Similarly, the partial application `map make_ref` has the type α^* `list` $\to \alpha^*$ `ref list`, in which α^* cannot be generalized, since the expression `map make_ref` is expansive according to Tofte's criterion. Nonetheless, the eta-expanded form λ`l. map make_ref l` can have a weakly polymorphic type, since this expression is non-expansive.

Weakness number two: the fact that some references have a limited scope is not taken into account. If an object a gets in touch with an object b whose type contains weak type variables, then the corresponding type variables in the type of a are automatically weakened; weak variables thus propagate through the types, even if the reference in which type they appear has long disappeared. For instance, `imp_map` is given the weakly polymorphic type

$$\forall \alpha^*, \beta^*. (\alpha^* \to \beta^*) \to \alpha^* \texttt{ list} \to \beta^* \texttt{ list},$$

while `appl_map`, which computes the same function, is given the fully polymorphic type

$$\forall \alpha, \beta. (\alpha \to \beta) \to \alpha \texttt{ list} \to \beta \texttt{ list}.$$

This difference is apparent when we apply those two functions to polymorphic arguments, such as the identity function and the empty list: `appl_map id` [] has type β `list` with β generalizable, while `imp_map id` [] has type β^* `list` with β^* not generalizable, since the expression is expansive. Similarly, most generic function built on top of `imp_map` possesses a less general type than if `appl_map` has been employed instead — even though those two functions have the same semantics. The reason for this discrepancy is that `imp_map` internally creates two references with types α^* `list ref` and β^* `list ref`, thus requiring the variables α^* and β^* to be weak in the type of `imp_map`. The fact that those two references are local to each call to the function is not taken into account: the result of `imp_map id` [] has type β^* `list`, where β^* is still marked weak; yet, the reference with type β^* `list` created by the function `imp_map` has became unreachable, and therefore there is no need to restrict the generalization of β^* any longer.

5.2.1.3 Standard ML of New Jersey

In order to detect more precisely the time when references are created inside curried functions, MacQueen proposed an extension of Tofte's system where variables, instead of being partitioned into weak and non-weak variables, are associated to an integer, their "weakness degree" or more exactly their "strength". The degree of a variable measures the number of function applications that must be performed before a reference whose type contains this variable is allocated. Variables that do not appear in the type of a `ref` have degree $+\infty$. Variables that do appear in the type of a `ref` have a degree equal to the number of function abstractions that separate the introduction of the variable from the occurrence of the `ref`. Variables with degree 0 are not generalizable; variables with degree $n > 0$ are generalizable; each function application decrements the degree of variables in the result type. For instance, in MacQueen's system, the function:

$$\texttt{let f} = \lambda\texttt{x}.\,\lambda\texttt{y}.\,\texttt{ref(x,y)}$$

is given type $\forall\alpha^2, \beta^2.\ \alpha^2 \to \beta^2 \to (\alpha^2 \times \beta^2)$ `ref`. The partial application `f(1)` has type $\beta^1 \to (\texttt{int} \times \beta^1)$ `ref`, in which β is still generalizable, since it has degree 1. As a consequence, the following example is well-typed, while it is rejected by Tofte's system:

```
let f = λx. λy.ref(x,y) in
let g = f(1) in
    ... g(true) ... g(2) ...
```

History. This system is used in the Standard ML of New Jersey implementation. It is briefly described in the reference manual for this implementation [3]. The typing rules have never been published; neither have soundness proofs. The system seems to have slightly changed in a recent release.

Results. In practice, this system turns out to be not much more expressive than Tofte's. Only one of my examples (the partial application of `imp_map` to the identity function) is recognized correct by MacQueen's system, but not by Tofte's. Other examples of partial applications (`appl_map make_ref`) are still wrongly rejected as ill-typed, indicating that full functionality is still not handled correctly. Also, the fact that some references have local scope is still not taken into account (`imp_map`).

5.2.2 Effect systems

The three systems I shall now describe rely, for controlling polymorphic references, over the addition of an *effect system* to the type system. In the same way as the type of an expression approximates the value of the expression, the effect of an expression approximates the side-effects performed during its evaluation. The typing judgements now have the form:

$$E \vdash a : \tau, F$$

where F ranges over an effect algebra to be specified later. The computation of types and the computation of effects interfere in two points. First of all, the type of a function is annotated by its latent effect, that is, by a description of the side-effects it performs when applied. Function types therefore have the format $\tau_1 \to\!\langle F \rangle\!\to \tau_2$, where F is the latent effect. (In spite of the similarities in

notations, there is a major difference between closure typing and effect systems: the closure type is a static information: it describes what the functional value already contains; while the latent effect is a dynamic information: it describes what the function will do once applied.) The typing rule for functions illustrates this interference:

$$\frac{E, x : \tau_1 \vdash a : \tau_2, F}{E \vdash \lambda x.\, a : (\tau_1 \multimap\langle F\rangle\rightarrow \tau_2), \emptyset}$$

(We write $\emptyset$ for the empty effect: the effect given to expressions that have no side-effects.)

Interference number two: the effect of an expression is taken into account when generalizing its type, so as to avoid the creation of polymorphic references. As we shall now see, various generalization criteria, more or less precise, can be applied, depending on the effect algebra used.

Context. The presentation of effect systems given above is incomplete, and considers only the aspect "polymorphic reference control" in a type inference framework. The effect systems were introduced by Lucassen and Gifford in their FX language [29, 54], mainly to aid in automatically parallelizing imperative programs: two expressions without side-effects, or whose side-effects are performed on disjoint sets of references, can be evaluated concurrently without changing the behavior of the program. Moreover, the FX language is originally a language with type and effects explicit in the source program; the problem of inferring types and effects have not been considered until much later [40, 90], and have imposed additional restrictions over the algebra of types and effects. In the following discussion, I consider only those effect systems for which a type inference algorithm exists. □

5.2.2.1 Simple effects

In the simplest case, an effect is a set of constants and effect variables:

$$\begin{array}{rcl} F & ::= & \{f_1, \ldots, f_n\} \cup \varsigma_1 \cup \ldots \cup \varsigma_k \\ f & ::= & \texttt{alloc} \mid \texttt{read} \mid \texttt{write} \end{array}$$

Here, the `alloc` constant means that the expression creates references; `read`, that it reads references; `write`, that it modifies references. We write ς for effect variables, which introdce a notion of polymorphism over effects similar to the polymorphism over types found in ML. With the effect algebra above, the computation of effects is a slightly refined variant of the purity analysis ("does this expression perform side-effects?"). The generalization of the type of an expression is permitted, then, only if the expression is pure, that is, if it has effect $\emptyset$:

$$\frac{E \vdash a_1 : \tau_1, \emptyset \qquad \{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau_1) \setminus \mathcal{F}(E) \qquad E + x \mapsto \forall \alpha_1 \ldots \alpha_n.\, \tau_1 \vdash a_2 : \tau_2, F}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2, F}$$

$$\frac{E \vdash a_1 : \tau_1, F_1 \qquad F_1 \neq \emptyset \qquad E + x \mapsto \tau_1 \vdash a_2 : \tau_2, F_2}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2, (F_1 \cup F_2)}$$

The simple typing of references is sound, then: with the restriction above over generalization, the value whose type is generalized cannot contain newly created references; hence, we never generalize over a type variable which is free in the type of a reference.

History. The system described above is close to the one described by Gifford and Lucassen in [29]. Jouvelot and Gifford [40] study type inference for a similar system. They claim that type inference in the presence of the two `let` rules above requires backtracking, and therefore opt for an even simpler type system, where only non-expansive expressions (in the sense of Tofte) can be generalized. Clearly, "non-expansive" implies "has effect $\emptyset$", but the converse does not hold. Hence, this restriction makes the effect system behave just as poorly as the SML type system, as far as polymorphic type references are considered. I do not apply this restriction in the comparisons.

Results. The system described above gives better results than the systems based on weak variables on the examples involving higher-order functions. For instance, the application `id make_ref` is correctly recognized as pure, and therefore its type can be generalized. (The function `id` has no latent effect, and its argument is a pure expression.) Annotating function types by their latent effect allows to detect more precisely when references are created. For instance, the type given to `appl_map` is:

$$\texttt{appl\_map} : \forall \alpha, \beta, \varsigma.\ (\alpha \multimap\langle\varsigma\rangle\rightarrow \beta) \multimap\langle\emptyset\rangle\rightarrow \alpha\ \texttt{list} \multimap\langle\varsigma\rangle\rightarrow \beta\ \texttt{list}.$$

(Effect variables are generalized exactly in the same way as type variables.) The partial application `appl_map make_ref` therefore has effect $\emptyset$, and its type can be generalized.

On the other hand, the generic functions that allocate local references are handled just as poorly as in SML. The allocation, though purely local, is witnessed in the latent effect. Hence, the type of these functions differs from the type of equivalent purely applicative functions. For instance:

$$\begin{array}{rcl} \texttt{appl\_map} & : & \forall \alpha, \beta, \varsigma.\ (\alpha \multimap\langle\varsigma\rangle\rightarrow \beta) \multimap\langle\emptyset\rangle\rightarrow \alpha\ \texttt{list} \multimap\langle\varsigma\rangle\rightarrow \beta\ \texttt{list} \\ \texttt{imp\_map} & : & \forall \alpha, \beta, \varsigma.\ (\alpha \multimap\langle\varsigma\rangle\rightarrow \beta) \multimap\langle\emptyset\rangle\rightarrow \alpha\ \texttt{list} \multimap\langle\{\texttt{alloc}\} \cup \varsigma\rangle\rightarrow \beta\ \texttt{list} \end{array}$$

Hence, `imp_map id []` is given effect $\{\texttt{alloc}\}$, preventing the generalization of its type.

Although this does not appear in figure 5.1, this first effect system is sometimes less powerful than the systems based on weak variables. For instance,

$$\texttt{let id} = (\texttt{ref}\ [\,]);(\lambda \texttt{x.x})\ \texttt{in id(id)}$$

is accepted in SML, but rejected in this effect system: the expression $(\texttt{ref}\ [\,]);(\lambda \texttt{x.x})$ has effect $\{\texttt{alloc}\}$, hence its type cannot be generalized.

5.2.2.2 Typed allocation effects

To correct the weakness above, we can enrich the effect algebra so as to record not only the fact that an expression creates a reference, but also the type of the reference created.

$$\begin{array}{rcl} F & ::= & \{f_1 \ldots f_n\} \cup \varsigma_1 \cup \ldots \cup \varsigma_k \\ f & ::= & \texttt{alloc}(\tau) \mid \texttt{read} \mid \texttt{write} \end{array}$$

The modified generalization rule is: the variables that are free in an allocation effect of the expression cannot be generalized.

$$\frac{E \vdash a_1 : \tau_1, F_1 \qquad \{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau_1) \setminus \mathcal{F}(F_1) \setminus \mathcal{F}(E) \qquad E + x \mapsto \forall \alpha_1 \ldots \alpha_n . \tau_1 \vdash a_2 : \tau_2, F_2}{E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2, (F_1 \cup F_2)}$$

History. A recent paper by Wright [100] studies a similar system. Wright does not keep track of the `read` and `write` effects (that are useless for the control of references), and replaces the effect $\texttt{alloc}(\tau)$ by the set of the type variables free in τ (that are all we need to implement the `let` rule above). In his system, an effect is thus a set of effect variables and type variables: the variables free in the types of the references created. His system gives the same results as the one described in this section.

Results. On my tests, this system behaves exactly like the first effect system: good handling of full functionality, poor handling of references with local scope. It accepts in addition a few anecdotic examples, such as

$$\texttt{let id} = (\texttt{ref}\ [\,])\texttt{;}\ (\lambda \texttt{x.x})\ \texttt{in id(id)}$$

With typed allocation effects, the expression $(\texttt{ref}\ [\,])\texttt{;}\ (\lambda \texttt{x.x})$ is given type $\alpha \multimap\!\langle\emptyset\rangle\!\rightarrow \alpha$ and effect $\{\texttt{alloc}(\beta\ \texttt{list ref})\}$. The variable α can therefore be generalized, since it is not free in the effect.

5.2.2.3 Damas' system

One of the very first type systems for references in ML, proposed by Damas in 1985, can be viewed as a simplified allocation effect system. In Damas' system, function type schemes are annotated by a set of type variables, the variables possibly free in the types of the references allocated by the function. In contrast with Wright's system, the function types located inside type expressions are not annotated. More precisely, Damas' type algebra is as follows:

Types	τ	::=	α	type variable
		|	$\tau_1 \rightarrow \tau_2$	simple function type
		|	$\ldots$	
Schemes	σ	::=	$\forall \alpha_1 \ldots \alpha_n . \tau$	regular scheme
		|	$\forall \alpha_1 \ldots \alpha_n . \tau_1 \multimap\!\langle \beta_1, \ldots \beta_k \rangle\!\rightarrow \tau_2$	function scheme, annotated by its allocation effect

The motivation for this restriction seems to be that having no effects inside type expressions makes type inference considerably easier.

History. This system is studied in Damas' thesis [21]. Damas gives a soundness proof for this system, with respect to a denotational semantics, and proposes a type inference algorithm. This work is fairly hard to read. In particular, the soundness proof requires complicated constructions of domains and ideals. Tofte [92] claims that this proof is not complete. Damas' system has never been implemented in an ML compiler, to the best of my knowledge.

Results. Damas' system gives basically the same results as the SML system. Since effects are omitted inside the types, the examples involving higher-order functions are not handled correctly (`appl_map make_ref`). The scope of references is not taken into account (`imp_map`).

5.2.2.4 Typed allocation effects plus regions

To take into account the fact that references can disappear (more exactly, become unreachable) when leaving the scope of the identifiers to which they were bound, we can introduce the notion of *region* inside types and inside effects. A region stands for a set of references. The reference types now indicate not only the type of the referenced object, but also the region to which the reference belongs. Similarly, effects now keep track of the region involved.

τ	::=	τ `ref`$_\rho$	type of a reference belonging to region ρ
	$\mid$	$\tau_1 \multimap\langle F\rangle\rightarrow \tau_2$	function type
	$\mid$	$\ldots$	
F	::=	$\{f_1, \ldots, f_n\} \cup \varsigma_1 \cup \ldots \cup \varsigma_k$	effect
f	::=	`alloc`$_\rho(\tau)$	allocation of a τ `ref` in region ρ
	$\mid$	`read`$_\rho$	dereferencing of a reference from region ρ
	$\mid$	`write`$_\rho$	modification of a reference from region ρ

From the standpoint of typing, regions are treated like type variable. In particular, identifying two reference types τ `ref`$_\rho$ et τ `ref`$_{\rho'}$ causes the identification of ρ and ρ', that is, the merging of the two regions. Also, if two references have types τ `ref`$_\rho$ and τ' `ref`$_{\rho'}$ with $\rho \neq \rho'$, this means that the two references cannot be aliases (they cannot point to the same memory location). This is valuable information for efficient compilation and automatic parallelization. Regions similarly indicate when a reference becomes unreachable. If, at some point during evaluation, the current evaluation context contains no references belonging to region ρ, this means that all references previously allocated in region ρ are now unreachable. Hence, we can ignore the effects concerning region ρ: these effects have no influence on the remainder of the computation. This is expressed by the following effect simplification rule:

$$\frac{E \vdash a : \tau, F}{E \vdash a : \tau,\ \texttt{Observe}(F, E, \tau)}$$

The `Observe` operator returns the subset of F composed of those effects over regions free in E or τ. These effects can possibly be "observed" from the outside world. The other effects can be erased, since they concern unreachable references, and therefore cannot be observed anymore.

History. The notion of region and the effect masking rule above are described in [54], for an explicitly typed language where regions are explicitly declared and associated to the references created in the program. Recently, Talpin and Jouvelot [90] have shown how to infer regions and effects for an ML-like source language.

Results. Thanks to effect masking, generic functions can now have the same type, whether they internally use references or not. For instance, the principal typing for `imp_map` leads to a type of the form:

$$\texttt{imp\_map} : (\alpha \multimap\langle\varsigma\rangle\rightarrow \beta) \multimap\langle\emptyset\rangle\rightarrow \alpha\ \texttt{list} \multimap\langle\varsigma \cup \{\texttt{alloc}_\rho(\alpha), \texttt{alloc}_{\rho'}(\beta)\}\rangle\rightarrow \beta\ \texttt{list}$$

where, provided that the typing is principal, ρ and ρ' are two "fresh" regions, that do not appear anywhere else; this is due to the fact that the two references created by `imp_map` are never passed

as argument to another function. Then, the masking rule allows erasing the two allocation effects, leading to the type

$$\texttt{imp\_map} : (\alpha \multimap\langle\varsigma\rangle\rightarrow \beta) \multimap\langle\emptyset\rangle\rightarrow \alpha\ \texttt{list} \multimap\langle\varsigma\rangle\rightarrow \beta\ \texttt{list},$$

which is exactly the principal type for `appl_map`.

The properties of this type system are close to the ones of the system in chapter 3. It is therefore no surprise that the problems mentioned in section 4.1 appear also for this system: the `eta_ref` example illustrates the need for infinite (recursive) effects; the `capt_id_ref` example shows a phenomenon of variable capture through the effects. Let us detail those two examples.

```
let eta_ref = λf.
    let r = ref(f) in
        either f (λx.!(either r (ref(f)))(x))
```

When typing the outer `either`, we have:

$$\begin{aligned} \texttt{f} &: \alpha \multimap\langle\varsigma\rangle\rightarrow \alpha \\ \texttt{r} &: (\alpha \multimap\langle\varsigma\rangle\rightarrow \alpha)\ \texttt{ref}_\rho \end{aligned}$$

The second argument to `either`, $(\lambda\texttt{x}.!(\texttt{either r (ref(f))})(\texttt{x}))$, has type:

$$\alpha \multimap\langle\varsigma \cup \{\texttt{alloc}_\rho(\alpha \multimap\langle\varsigma\rangle\rightarrow \alpha)\}\rangle\rightarrow \alpha.$$

Indeed, this function allocates a reference to `f`, and this reference must belong to the same region ρ as `r`, because of the typing of the inner `either`. Since ρ is reachable from the environment, the allocation effect is not maskable. But there is no (finite) effect that is a common instance of ς and of $\varsigma \cup \{\texttt{alloc}_\rho(\alpha \multimap\langle\varsigma\rangle\rightarrow \alpha)\}$.

The second kind of failures comes from a phenomenon of variable capture through the effects, that prevents the generalization of the captured variables, as explained in section 4.1.2.

```
let capt_id_ref = λf.
    let id = λy.
        let r = ref(y) in
            either f (λz.either r (ref(y)); z);
        y
    in id(id)
```

Before typing the outer `either`, we have the following types:

$$\begin{aligned} \texttt{f} &: \alpha \multimap\langle\varsigma\rangle\rightarrow \alpha \\ \texttt{y} &: \beta \\ \texttt{r} &: \beta\ \texttt{ref}_\rho \\ (\lambda\texttt{z.either r (ref(y)); z}) &: \gamma \multimap\langle\texttt{alloc}_\rho(\beta)\rangle\rightarrow \gamma \end{aligned}$$

The function λz... allocates a reference to y, in the same region ρ as r, because of the inner either. The allocation effect is not maskable, since ρ is reachable through r. Identifying the types of f and λz... leads to the following types:

$$\begin{aligned} \texttt{f} &: \alpha \to\!\!\!\!\!\!\langle \varsigma \cup \{\texttt{alloc}_\rho(\beta)\}\rangle\!\!\!\!\!\!\to \alpha \\ \texttt{y} &: \beta \\ \texttt{r} &: \beta\ \texttt{ref}_\rho \end{aligned}$$

The expression bound to id therefore has type $\beta \to\!\!\langle F\rangle\!\!\to \beta$, for some effect F, in the typing environment $\texttt{f} : \alpha \to\!\!\!\!\!\!\langle \varsigma \cup \{\texttt{alloc}_\rho(\beta)\}\rangle\!\!\!\!\!\!\to \alpha$. The variable β is free in this environment, and therefore cannot be generalized. Hence id remains monomorphic, and the self-application id(id) fails.

Remark. At first sight, it might seem that effect masking suffices to eliminate the variable capture via latent effect problem. This is not the case, as demonstrated above. The effect masking rule is a "garbage collection" rule: we can ignore everything that happens in a region if this region is mentioned nowhere in the current typing context (environment and type). It does not reflect the abstraction properties of functions: a function can reference a given region internally, without making this region accessible from outside. The variable capture phenomenon comes from the fact that this abstraction property is not reflected in the typing rules. In contrast, the simplification operation over closure types described in section 4.1.2 does take this abstraction property into account, and therefore essentially differs from a "garbage collection" operation like effect masking. □

5.2.3 The systems proposed in this Thesis

I now comment the results obtained with the approach I propose: dangerous variables plus closure typing.

Results. In contrast with the previous approaches, this approach does not try to gather information on the dynamic behavior of evaluation ("this function does this and that"); just like the regular ML typing, it only gather purely static information ("this value contains this and that"). Because of this fact, it remains much closer to a conventional type system.

As a consequence, it is no surprise that full functionality causes absolutely no difficulties: id make_ref naturally has the same type as make_ref; about appl_map make_ref, the typing says that this is a functional value whose closure contains no references, hence it is given a fully polymorphic type.

Similarly, the fact that some references have only local lifespan can easily be seen on the types: if a function creates a reference, but this reference does not appear in the return value, nor in other reachable references — this can easily be checked on the types —, then these references are clearly local. Hence, imp_map has the very same type as appl_map, and can be employed in place of appl_map in any program.

The technical examples (eta, eta_ref, capt_id, capt_id_ref) reveal some weaknesses of the type system described in chapter 3. The system in chapter 4, while keeping the good behavior of the system in chapter 3 over the practical examples, fixes most of these weaknesses: recursive

closure types do not cause type errors anymore (`eta` and `eta_ref`); and the capture of variables through closure types is avoided in the `capt_id` example. As the `capt_id_ref` example shows, this capture phenomenon still appears in some non-purely functional programs.

```
let capt_id_ref = λf.
    let id = λy.
        let r = ref(y) in
            either f (λz.either r (ref(y)); z);
        y
    in id(id)
```

The principal typing of $\lambda\mathtt{y}\ldots$ under the assumption $\mathtt{f} : t_n \multimap\!\langle u_n\rangle\!\rightarrowtail t_n$ results in the type $t'_n \multimap\!\langle u'_n\rangle\!\rightarrowtail t'_n$ under the constraints

$$t'_n \ \mathtt{ref} \triangleleft u_n,\ (t_n \multimap\!\langle u_n\rangle\!\rightarrowtail t_n) \triangleleft u'_n$$

The variable t'_n is dangerous in the typing environment, and therefore cannot be generalized. I think that this capture phenomenon could be avoided as in chapter 4: just as we can ignore the variables free in the closure type part of a function type if these variables are not free in the argument type or in the result type, I believe that it is correct to ignore the variables dangerous in the closure type part of a function type if these variables are not free in the argument type or in the result type. More formally, we would take:

$$\mathcal{D}(\sigma_1 \multimap\!\langle u\rangle\!\rightarrowtail \sigma_2 \,/\, C) = \mathcal{D}(u \,/\, C) \cap (\mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2)).$$

The intuition behind this definition is that the omitted dangerous variables correspond to references that are not accessible from outside the function. For instance, in the `capt_id_ref` example, t'_n would no longer be dangerous in the typing environment at generalization time. I haven't attempted to proof the soundness of this refinment of the system in chapter 4.

The `fake_ref` example, which consists in constraining the type of an exception to mimic the creation of a polymorphic reference, is rejected by my systems, but accepted by all others. This has absolutely no practical significance, but outlines the difference between a purely type-based approach, that leads to the rejection of those expressions whose type looks exactly like the type of a polymorphic reference creation, and the other approaches, that add various mechanisms to the type system to precisely better distinguish reference creation.

5.3 Ease of use and compatibility with modular programming

Programming in a typed language requires the programmer to have a good understanding of the type system: to write type declarations in the source code; to understand the type errors reported by the compiler; finally, to write type specifications in module interfaces. This argument favors type systems that are as simple as possible. It sets a limit to the search for more and more expressive type systems, that naturally tends to result in complicated type systems. (The progression from the system in chapter 1 to the system in chapter 4 illustrates this tendency; the other systems presented at the beginning of this chapter also follow this tendency.)

Type inference, as in ML, may seem to resolve this tension: if the types are left implicit in the source code and reconstructed by the compiler, then the programmer is unaffected by the

complexity of the type system. In fact, type inference makes the problem less acute, but does not make it disappear. Indeed, there remains a number of situations where the programmer must read or write type expressions directly.

First of all, when a type error is reported, the programmer has to decipher the inferred types, understand why they conflict, and locate where the error originates. The latter is often difficult, even with a simple type system. A more complex type system makes things worse, though not desperate. The compiler can indeed simplify the type expressions presented to the user, as long as the simplified types still reveal the type conflict. For instance, in the case of the systems that annotate function types with extra information (such as those in chapters 3 and 4, and also such as the effect systems presented above), this extra information need not be printed when reporting an application of a function to an object of the wrong type — the most common type error, by far. Moreover, truly subtle type errors, in particular those related to polymorphic references/channels/continuations, are unfrequent, both from novice programmers (who generally do not know how to mix full functionality with non-applicative features) and from experienced programmers (who scarcely do type errors). We can therefore tolerate that these subtle but unfrequent errors require some thinking from the programmer.

The situations where programmers must write type expressions by hand raise more serious issues. In ML, this occurs when declaring data structures (concrete types), since the programmer must give the types of the arguments to the constructors and labels, and also when writing module interfaces (or "signatures"), since the programmer must specify the types of the global identifiers exported by the module. In the case where only function types are enriched, as in all systems considered above, the problem with concrete type declarations is minor: in practice, concrete types containing functional values are unfrequent. The problem with module interfaces, however, is crucial: most values exported by a module are functions.

The issue of the practicality of the type systems considered here therefore reduces to the following issue: how hard is it to completely specify, in a module interface, the type of a function whose implementation is not known? (In the case where we write the implementation before the interface, we can always run the type inference system on the implementation, then copy the inferred type into the interface by "cutting and pasting". This practice goes against the principles of modular programming [70, 12], that require specifying before implementing. At any rate, it does not apply to the writing of interfaces for the parameters of parameterized modules, or "functors".)

On the systems presented in the previous section, it appears that this difficulty is roughly proportional to the expressiveness of the system: the more expressive the system, the more its type expressions reflect how a function is implemented, and the more difficult specifying an unimplemented function is.

This progression is striking in the case of the systems based on weak type variables. The CAML type system, the most restrictive of all, uses the same type algebra as the core ML language, and therefore turns out to be just as convenient for writing interfaces. The Standard ML system, more expressive, distinguishes between applicative and imperative variables; hence, to specify the type of a polymorphic function, one must know beforehand whether it will be implemented in the applicative style (in which case all variables in its type can be applicative) or in the imperative style (in which case some variables must be imperative). The introduction of weakness degrees,

as in SML-NJ, makes this problem even more acute: these levels highly depend on how curried functions interleave computations and parameter passing; as a consequence, the type specification of a curried, imperative, polymorphic function not only imposes the implementation style, but also the order in which computations and parameter passing are performed. Clearly, it is difficult to figure out these properties before the implementation is written.

Effect systems appear to follow a similar progression.[1] With a simple effect algebra, the type specification for a function reveals not only its implementation style, but also, in the case of higher-order functions, the time when the functional arguments are applied (this is apparent in the positions of the effect variables that stand for the latent effects of those functional arguments). With an effect algebra enriched with regions, the type specification must in addition specify the aliasing properties of the implementation. That's because the region parts of types express precisely how an implementation shares references between arguments and results. These sharing properties are highly dependent on the implementation, and therefore seems inappropriate for inclusion in the type specification.

My approach — closure typing — also adds major difficulties to the writing of module interfaces. Generally speaking, it is hard to imagine which closure types to put over the arrows in the type of a function not yet implemented. Moreover, two implementations of a curried function can opt for different interleavings of computations and parameter passing, resulting in two different types for two implementations that we would like to be equivalent. Notice that this problem does not occur in the case of monomorphic functions, the most frequent case by far. That's because we can omit all closed types from closure type specifications; hence, the specification of a monomorphic function contains only trivial closure types. This problem does not occur either for polymorphic, but uncurried functions: functions of the form `let f (x,y,z)` $= a$, where a does not return a functional value. Trivial closure types suffices for these functions, too. But I must admit that non-trivial (and even complicated) closure types are required for polymorphic curried functions, and in particular for those which take functional arguments. A quick look at some ML libraries [7, 99, 51] shows that many useful functions belong to this category. Most of these functions could be uncurried; but the curried form seems to be preferred by ML programmers. (I sympathize: I have myself loudly argued in favor of currying [49], and designed the Caml Light execution model so that curried functions are more efficient than their uncurried forms.) For this kind of functions, closure typing clearly conflicts with the requirements of modular programming.

[1] Since I was unable to get some practical experience with an implementation of a language with an effect system, I can only give a priori feelings here. The vast literature on effect systems [29, 54, 40, 100, 90] does not say anything about the problems related to modular programming.

Chapter 6

Polymorphism by name

In this chapter, we show that the difficulties encountered with the polymorphic typing of references, channels, and continuations are specific to the ML semantics for generalization and specialization: if these constructs are given alternate semantics, which I call polymorphism by name, the naive typing rules for references, channels, and continuations turn out to be sound. This result suggests that a variant of ML with polymorphism by name would probably be a better polymorphic language for references, channels, and continuations than ML. The end of the chapter discusses the pros and cons of this variant from a practical standpoint.

6.1 Informal presentation

6.1.1 The two semantics of polymorphism

Polymorphism is introduced by two basic operations: generalization, that transforms a term with type $\tau[\alpha]$, where α is a type parameter, into a term with type $\forall\alpha.\,\tau[\alpha]$; and specialization, that transforms a term with type $\forall\alpha.\,\tau[\alpha]$ into a term with type $\tau[\tau']$, for any given type τ'. In ML, these two operations are not explicit in the source program: they are performed implicitly at certain program points (the `let` construct, in the case of generalization; when accessing a variable, in the case of specialization). In other languages, these operations are explicit in the source program: the language provides syntactic constructs to generalize and to specialize. For instance, in the polymorphic lambda-calculus of Girard [30] and Reynolds [82], generalization is presented as an abstraction over a type variable. It is written $\Lambda\alpha.\,a$, by analogy with the notation for functions $\lambda x.\,a$. Symmetrically, specialization is presented as the application of a term to a type. It is written $a\langle\tau\rangle$, by analogy with function application. Some programming languages, such as Poly [57] and Quest [12], follow this approach.

Regardless of their syntactic presentation (explicit or implicit), generalization and specialization can be given two different semantics. The first possibility is to consider that these constructs have no computational content. Using the polymorphic lambda-calculus notation, this amounts to say

that $\Lambda\alpha.\, a$ evaluates like a, and similarly $a\langle\tau\rangle$ evaluates like a:

$$\frac{e \vdash a \Rightarrow r}{e \vdash \Lambda\alpha.\, a \Rightarrow r} \qquad \frac{e \vdash a \Rightarrow r}{e \vdash a\langle\tau\rangle \Rightarrow r}$$

Using the ML notation, and assuming strict semantics, this amounts to say that the expression `let` $x = a_1$ `in` a_2 evaluates a_1 once and for all, and shares the resulting value between all occurrences of x in a_2. This is so in ML, and in the small language studied in the previous chapters.

The alternate semantics consist in interpreting generalization as functional abstraction, and specialization as function application. In other terms, generalization suspends the evaluation of the expression whose type is generalized, and each specialization re-evaluates this expression. Using the polymorphic lambda-calculus notation, this means that Λ is interpreted as an actual abstraction (that builds a suspension), and $\langle\cdot\rangle$ is interpreted as an actual application (that evaluates the suspension):

$$e \vdash \Lambda\alpha.\, a \Rightarrow \mathtt{Susp}(a, e) \qquad \frac{e \vdash a \Rightarrow \mathtt{Susp}(a', e') \qquad e' \vdash a' \Rightarrow r}{e \vdash a\langle\tau\rangle \Rightarrow r}$$

That's the approach retained in Quest, for instance. The "generics" of CLU [53] or Ada [95], viewed as a restricted form of polymorphism, also follow this semantics.

I call POLYMORPHISM BY VALUE the former interpretation, and POLYMORPHISM BY NAME the latter, by analogy with the two well-known semantics for function application: call-by-value and call-by-name (as in Algol 60). [1]

6.1.2 Polymorphism by name in ML

At first sight, it might seem that polymorphism-by-name semantics requires polymorphism to be explicit in the syntax. In this chapter, I shall demonstrate that this is not true, by studying a variant of ML — with implicit polymorphism and type inference — where polymorphism is interpreted following the by-name semantics. (Rouaix has considered a similar language to study dynamic overloading [87, 86].) The starting point is to separate the two roles of the ML `let` construct: (1) introducing polymorphic types, (2) sharing the value of of an expression between several utilizations. Hence, we replace the `let` construct by two separate constructs. The first construct is a strict binding that performs no generalization:

$$\mathtt{let\ val}\ x = a_1\ \mathtt{in}\ a_2.$$

[1] In modern languages, call-by-name is often replaced by one of its variants, call-by-need (also called "lazy evaluation"), where the function arguments are evaluated the first time their values are needed, but the resulting values are reused for subsequent accesses. This strategy is used by most implementations of purely applicative languages, such as Haskell, Miranda, or Lazy ML [4, 72]. This strategy is more efficient than call-by-need, yet semantically equivalent to call-by-need in a purely applicative language. This is not true in a language with imperative features. In the remainder of this chapter, which studies polymorphism by name for a non-purely applicative language, it is crucial to re-evaluate polymorphic objects at each specialization, without sharing the values obtained between several specializations.

This construct evaluates a_1 exactly once, and shares the result between all occurrences of x in a_2. But it does not generalize the type of a_1. Hence, the `let val` expression above is just syntactic sugar for $(\lambda x.\, a_2)(a_1)$. (We shall keep the `let val` notation in the examples, for the sake of readability.) The other construct is a non-strict binding that performs generalization:

$$\texttt{let poly } x = a_1 \texttt{ in } a_2.$$

This construct generalizes the type of a_1 in the same way as the ML `let`. But it does not evaluate a_1 once and for all; instead, a_1 is re-evaluated each time x is accessed in a_2. In other terms, the `let poly` expression above evaluates as the textual substitution $[x \mapsto a_1](a_2)$.

Context. The `let poly` expression could also be typed (almost) as the textual substitution $[x \mapsto a_1](a_2)$. That's because of a well-known property of Milner's type system: in the non-degenerate case where x appears in a_2, the expression `let` $x = a_1$ `in` a_2 has type τ if and only if the textual substitution $[x \mapsto a_1](a_2)$ has type τ [66, section 4.7.2]. Some authors rely on this property to give a simplified presentation of Milner's type system that does not make use of type schemes [44, 101]. I chose not to follow this presentation in this work, because it does not directly lead to a reasonably efficient type inference algorithm. □

6.1.3 Polymorphism by name and imperative features

When polymorphism is interpreted according to the by-name semantics, the problems with the polymorphic typing of imperative constructs such as references, channels, and continuations disappear: the naive polymorphic typing for these constructs turns out to be sound. ("Naive typing" refers to the typings proposed in sections 2.1.3, 2.2.3 and 2.3.3.) The problems with combining polymorphism and imperative features are caused by the fact that a given reference/channel/continuation object can be used with several different types. But this cannot happen with polymorphism by name. To use one object with several different types, this object must be bound to a variable (so that we can reference it several times). If the object is bound by a λ or by a `let val`, the type of the variable remains monomorphic, hence the variable is always accessed with the same type. If the object is bound by a `let poly`, each access to the variable re-evaluates the expression to which it is bound, re-creating a different reference/channel/continuation object each time; hence, this does not allow accessing the same object with different types.

To illustrate these generalities, we are now going to reconsider some of the examples from chapters 2 and 3, in the setting of polymorphism by name.

Example. For starters, consider the *pons asinorum* of polymorphic references:

```
let r = ref(λx.x) in
    r := (λn.n + 1);
    if (!r)(true) then ... else ...
```

With polymorphism by name, the original `let` must be replaced either by a `let val` or by a `let poly`. If it is replaced by a `let val`, the type of r, $(\alpha \to \alpha)$ `ref`, is not generalized before typing the body of the `let val`. Typing the assignment instantiates α to `int`, hence the application `(!r)(true)` is rejected at compile-time as ill-typed. If we put a `let poly` instead, the program is well

typed (with `r` having type $\forall\alpha.(\alpha \to \alpha)$ `ref` in the body of the `let poly`). But, at run-time, each of the two accesses to `r` re-evaluates the expression `ref`$(\lambda$`x.x`$)$, resulting in two distinct references to the identity function. The assignment modifies one of these references, but the application (`!r`)(`true`) dereferences the other, which still points to the identity function. The example therefore evaluates without errors. □

Polymorphism by name prohibits pathological uses of polymorphic references. But this semantics does not preclude the consistent use of references inside generic functions.

Example. The function that reverses a list iteratively can be written as follows:

```
let poly reverse = λl.
    let val arg = ref(l) in
    let val res = ref(nil) in
        while not is_null(!arg) do
            res := cons(head(!arg), !res);
            arg := tail(!arg)
        done;
        !res
```

All accesses to `arg` must always return the same reference, and similarly for `res`; this is ensured by the two `let val`. All occurrences of `arg` and `res` inside the loop have the same type α `list ref` (assuming α `list` to be the type of `l`); the loop is therefore well-typed. The outer `let poly` assigns the expected polymorphic type to `reverse`: $\forall\alpha.\,\alpha$ `list` $\to\ \alpha$ `list`. The fact that the closure for λ`l`... is recomputed each time `reverse` is used, instead of being shared as in the case of polymorphism by value, does not affect the behavior of `reverse`. □

Example. The counterexample based on channels is similar to the one based on references:

```
let c = newchan() in (c!true) || (1 + c?)
```

If the `let` is turned into a `let val`, the program above is rejected as ill-typed. If the `let` is turned into a `let poly`, the two accesses to `c` return two different channels, hence the two processes cannot communicate: they remain blocked forever. The point is that no type violation occurs at run-time. (The deadlock situation is of course not what the author of the program had in mind; but this program is clearly erroneous, after all.) □

Example. Finally, consider again Harper and Lillibridge's counterexample for continuations:

```
let later =
    callcc(λk.
        (λx.x),
        (λf.throw(k, (f, λx.()))))
in
    print_string(first(later)("Hello!"));
    second(later)(λx.x + 1)
```

The program is not statically well-typed if `let` is replaced by `let val`. If `let` is replaced by `let poly`, the `callcc` expression is evaluated twice (each time `later` is accessed) instead of once (just before evaluating the body of the `let`), as in the case of polymorphism by value. The first evaluation of the `callcc` expression captures the continuation λ`later. print_string`...; this continuation is never activated. The second evaluation captures the continuation

$$\lambda\texttt{later.}\ \texttt{second(later)}(\lambda\texttt{x.}\ \texttt{x}+\texttt{1}).$$

The evaluation of `second(later)`(λ`x.` `x` $+$ `1`) restarts this continuation on the value (λ`x.` `x`$+1$, λ`x.` `()`). Hence this will re-evaluate `second(later)`(λ`x.` `x`$+$ `1`), this time in an environment where `later` is bound to (λ`x.` `x` $+$ `1`, λ`x.` `()`) — actually, to a suspension that evaluates into this pair of functions. The evaluation terminates by returning `()`. No run-time type violation has occurred. □

Context. It is fairly easy to convince oneself that the polymorphic typing of imperative constructs is sound if polymorphism is given the by-name semantics. It is not difficult to prove this result, as shown by the next section. Yet, this result seems often overlooked. It is alluded to in Gifford and Lucassen [29]. Cardelli also mentions this fact in his description of the Quest language Quest [12]. Both papers discuss languages with explicit polymorphism, and both tend to confuse explicit polymorphism with polymorphism by name. For instance, Cardelli writes [12, p. 24]:

> Mutability [in Quest] interacts very nicely with all the quantifiers, including polymorphism, showing that the functional-style approach suggested by type theory does not prevent the design of imperative languages.

(Well said.) And in a terse footnote, he adds:

> The problems encountered in ML are avoided by the use of explicit polymorphism.

This is not correct: if mutable data structures cause no difficulties in Quest, that's because polymorphism is interpreted following the by-name semantics, not because it is explicit in the source code. This confusion between the syntax and the semantics of polymorphism is encouraged by the fact that the by-name and by-value semantics for polymorphism are undistinguishable in the polymorphic lambda-calculus, due to the strong normalization property; the differences between the two semantics appear only when general recursion or non-purely applicative features are introduced. □

6.2 Operational semantics

In this section, we formally define the operational semantics for the three calculus (with references, with channels and with continuations) when polymorphism is interpreted by name. The three semantics are very close to the ones given in chapter 2. To simplify the presentation, we split variable identifiers into two classes: the strict identifiers (set `SVar`, typical element x_s), which are bound by function abstraction (f_s `where` $f_s(x_s) = a$), and the delayed identifiers (set `RVar`, typical element x_r), which are bound by the `let poly` construct.

Concerning the semantic objects, the main change is in the evaluation environments: they now map strict identifiers to values, and delayed identifiers to suspensions, that is, a pair (a, e) of an unevaluated expression a and an evaluation environment e. Concerning the evaluation rules, the only rules that differ from those in chapter 2 are the rules for the `let` binding and for variable access.

6.2.1 References

Here is a summary of the semantic objects used:

Responses:	r	::=	v/s	normal response
		|	`err`	error response
Values:	v	::=	cst	base value
		|	(v_1, v_2)	value pair
		|	(f_s, x_s, a, e)	functional value
		|	ℓ	memory location
Environments:	e	::=	$[x_s \mapsto v, \ldots, x_r \mapsto (a, e), \ldots]$	
Stores:	s	::=	$[\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n]$	

The evaluation predicate is defined by the same rules as in section 2.1.2, with the exception of the rules for the variables and the rule for the `let` construct. The two rules for variables are replaced by the four rules below:

$$\frac{x_s \in \mathrm{Dom}(e)}{e \vdash x_s/s \Rightarrow e(x_s)/s} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x/s \Rightarrow \texttt{err}} \quad (x \text{ is } x_s \text{ or } x_r)$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0/s \Rightarrow r}{e \vdash x_r/s \Rightarrow r}$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) \text{ does not match } (a, e)}{e \vdash x_r/s \Rightarrow \texttt{err}}$$

The two rules for the `let` construct are replaced by the following rule:

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2/s \Rightarrow r}{e \vdash (\texttt{let poly } x_r = a_1 \texttt{ in } a_2)/s \Rightarrow r}$$

6.2.2 Channels

Summary of the semantic objects used:

Responses:	r	::=	v	normal response (a value)
		|	`err`	error response
Valeurs:	v	::=	cst	base value
		|	(v_1, v_2)	pair of values
		|	(f_s, x_s, a, e)	functional value (closure)
		|	c	channel identifier
Environments:	e	::=	$[x_s \mapsto v, \ldots, x_r \mapsto (a, e), \ldots]$	
Events:	evt	::=	$c \,?\, v$	emission of a value
		|	$c \,!\, v$	reception of a value
Event sequences::	w	::=	ε	the empty sequence
		|	$evt \ldots evt$	

The evaluation rules are those from section 2.2.2, with the exception of the two rules for variable accesses, which now become:

$$\frac{x_s \in \mathrm{Dom}(e)}{e \vdash x_s \stackrel{\varepsilon}{\Rightarrow} e(x_s)} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x \stackrel{\varepsilon}{\Rightarrow} \mathtt{err}}$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0 \stackrel{u}{\Rightarrow} r}{e \vdash x_r \stackrel{u}{\Rightarrow} r}$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) \text{ does not match } (a, e)}{e \vdash x_r \stackrel{\varepsilon}{\Rightarrow} \mathtt{err}}$$

and of the two rules for the `let`, which are replaced by:

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2 \stackrel{u}{\Rightarrow} r}{e \vdash \mathtt{let\ poly}\ x_r = a_1\ \mathtt{in}\ a_2 \stackrel{u}{\Rightarrow} r}$$

6.2.3 Continuations

Summary of the semantic objects used:

Responses:	r	::=	v	normal response (a value)
		|	`err`	error response
Values:	v	::=	cst	base value
		|	(v_1, v_2)	pair of values
		|	(f_s, x_s, a, e)	functional value
		|	k	continuation
Environments:	e	::=	$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$	
Continuations:	k	::=	`stop`	end of the program
		|	`primc`(op, k)	before a primitive application
		|	`app1c`(a, e, k)	after the function part of an application
		|	`app2c`(f_s, x_s, a, e, k)	after the argument part of an application
		|	`pair1c`(a, e, k)	after the first component of a pair
		|	`pair2c`(v, k)	after the second component of a pair

Remark. The continuation `letc`(x, a, e, k) is no longer needed. □

The evaluation rules are those in section 2.3.2, except for the rules for variable accesses, which are replaced by:

$$\frac{x_s \in \mathrm{Dom}(e) \qquad \vdash e(x) \triangleright k \Rightarrow r}{e \vdash x_s; k \Rightarrow r} \qquad \frac{x \notin \mathrm{Dom}(e)}{e \vdash x; k \Rightarrow \mathtt{err}}$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0; k \Rightarrow r}{e \vdash x_r; k \Rightarrow r}$$

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) \text{ does not match } (a, e)}{e \vdash x_r \Rightarrow \texttt{err}}$$

and for the rule for the `let` construct, which becomes:

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2; k \Rightarrow r}{e \vdash (\texttt{let poly } x_r = a_1 \texttt{ in } a_2); k \Rightarrow r}$$

6.3 Soundness proofs

In this section, we show that the type system presented in chapter 1 is sound with respect to the three semantics given in the previous section. The proofs are an adaptation of the soundness proofs in section 3.3 to the by-name semantics for `let` and variable access. The overall approach is the same, but the proofs are considerably simplified by the fact that all values are now monomorphic: only suspensions can be considered with several types. As a consequence, we no longer need the (difficult) semantic generalization lemma.

6.3.1 References

We use the following semantic typing relations:

$S \models v : \tau$	the value v, considered in a store of type S, belongs to the type τ
$S \models (a, e) : \sigma$	the suspension (a, e), considered in a store of type S, belongs to all instances of the schema σ
$S \models e : E$	the values and suspensions contained in the evaluation environment e, considered in a store of type S, belong to the corresponding types and tupe schemes in E
$\models s : S$	the store s belongs to the store typing S.

Here are their formal definitions:

- $S \models cst : \texttt{unit}$ if cst is `()`
- $S \models cst : \texttt{int}$ if cst is an integer
- $S \models cst : \texttt{bool}$ if cst is `true` or `false`
- $S \models (v_1, v_2) : \tau_1 \times \tau_2$ if $S \models v_1 : \tau_1$ and $S \models v_2 : \tau_2$
- $S \models \ell : \tau \texttt{ ref}$ if $\ell \in \mathrm{Dom}(S)$ and $\tau = S(\ell)$
- $S \models (f_s, x_s, a, e) : \tau_1 \to \tau_2$ if there exists a typing environment E such that

$$S \models e : E \quad \text{and} \quad E \vdash (f_s \texttt{ where } f_s(x_s) = a) : \tau_1 \to \tau_2$$

- $S \models (a, e) : \forall\alpha_1 \ldots \alpha_n.\tau$ if for all substitutions φ whose domain is included in $\{\alpha_1 \ldots \alpha_n\}$, there exists a typing environment E such that

$$S \models e : E \quad \text{and} \quad E \vdash a : \varphi(\tau)$$

- $S \models e : E$ if $\mathrm{Dom}(e) = \mathrm{Dom}(E)$, and for all $x_s \in \mathrm{Dom}(e)$, $E(x_s)$ is a simple type τ such that $S \models e(x_s) : \tau$, and for all $x_r \in \mathrm{Dom}(e)$, $E(x_r)$ is a type scheme σ such that $S \models e(x_r) : \sigma$.

- $\models s : S$ if $\mathrm{Dom}(s) = \mathrm{Dom}(S)$, and for all $\ell \in \mathrm{Dom}(s)$, we have $S \models s(\ell) : S(\ell)$.

Proposition 6.1 (Strong soundness for references) *Let a be an expression, τ be a type, E be a typing environment, e be an evaluation environment, s be a store, S be a store typing such that:*

$$E \vdash a : \tau \quad \textit{and} \quad S \models e : E \quad \textit{and} \quad \models s : S.$$

If there exists a result r such that $e \vdash a/s \Rightarrow r$, then $r \neq \mathtt{err}$; instead, r is equal to v/s' for some v and s', and there exists a store typing S' such that:

$$S' \textit{ extends } S \quad \textit{and} \quad S' \models v : \tau \quad \textit{and} \quad \models s' : S'.$$

Proof: the proof is an inductive argument over the size of the evaluation derivation. We argue by case analysis over a, and therefore over the last typing rule used in the typing derivation. We just show the cases that differ from the corresponding cases in the proof of proposition 3.6.

• **Strict variables.**

$$\frac{\tau \leq E(x_s)}{E \vdash x_s : \tau}$$

By the typing, we know that x_s belongs to the domain of E, which is the same as the domain of e. Hence, the only possible evaluation is $e \vdash x_s/s \Rightarrow e(x_s)/s$. By hypothesis over e and E, we have $S \models e(x_s) : E(x_s)$. Since $E(x_s)$ is a simple type, we have $\tau = E(x_s)$. Hence $S \models e(x) : \tau$. We take $S' = S$.

• **Delayed variables.**

$$\frac{\tau \leq E(x_r)}{E \vdash x_r : \tau}$$

Write $E(x_r) = \forall\alpha_1 \ldots \alpha_n.\tau_x$. Let φ be the substitution over the α_i such that $\tau = \varphi(\tau_x)$. By hypothesis over e and E, we have $e(x_r) = (a_0, e_0)$, and there exists E_0 such that $S \models e_0 : E_0$ and $E_0 \vdash a_0 : \tau$. Since $e(x_r)$ is a suspension, the evaluation must end up with:

$$\frac{e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0/s \Rightarrow r}{e \vdash x_r/s \Rightarrow r}$$

We apply the induction hypothesis to the expression a_0, with type τ, in the environments e_0, E_0, s, S. It follows that $r = v'/s'$ and there exists S' extending S such that

$$S' \models v' : \tau \quad \text{and} \quad \models s' : S'.$$

This is the expected result.

• **`let poly` bindings.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x_r \mapsto \texttt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \texttt{let poly } x_r = a_1 \texttt{ in } a_2 : \tau_2}$$

The only evaluation rule that applies is:

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2/s \Rightarrow r}{e \vdash (\texttt{let poly } x_r = a_1 \texttt{ in } a_2)/s \Rightarrow r}$$

We first show $S \models (a_1, e) : \texttt{Gen}(\tau_1, E)$. We have $\texttt{Gen}(\tau_1, E) = \forall \alpha_1 \ldots \alpha_n . \tau_1$, for some variables α_i that are not free in E. Let φ be a substitution over the α_i. By proposition 1.2, we have $\varphi(E) \vdash a_1 : \varphi(\tau_1)$. Since $\varphi(E) = E$ and $S \models e : E$ by hypothesis, we can take E as the environment required by the definition of $\models$ over schemes. Hence $S \models (a_1, e) : \texttt{Gen}(\tau_1, E)$. Taking

$$e_1 = e + x \mapsto v_1 \qquad\qquad E_1 = E + x \mapsto \texttt{Gen}(\tau_1, E),$$

we therefore have $S_1 \models e_1 : E_1$. We apply the induction hypothesis to a_2, e_1, E_1, s_1, S_1. We get that r is equal to v_2/s_2, for some v_2 and s_2, and there exists S_2 such that

$$S_2 \models v_2 : \tau_2 \text{ and } \models s_2 : S_2 \text{ and } S_2 \text{ extends } S_1.$$

That's the expected result. □

6.3.2 Channels

The semantic typing relations used are as follows (Γ is a channel typing):

$\Gamma \models v : \tau$	the value v belongs to the type τ
$\Gamma \models (a, e) : \sigma$	the suspension (a, e) belongs to the type scheme σ
$\Gamma \models e : E$	the values and the suspensions contained in the evaluation environment e belong to the corresponding types and schemes in E
$\models u :? \Gamma$	the reception events $(c \,?\, v)$ contained in the event sequence u match the channel typing Γ
$\models u :! \Gamma$	the emission events $(c \,?\, v)$ contained in the event sequence u match the channel typing Γ

These relations are defined as follows:

- $\Gamma \models cst : \texttt{unit}$ if cst is `()`
- $\Gamma \models cst : \texttt{int}$ if cst is an integer
- $\Gamma \models cst : \texttt{bool}$ if cst is `true` or `false`
- $\Gamma \models (v_1, v_2) : \tau_1 \times \tau_2$ if $\Gamma \models v_1 : \tau_1$ and $\Gamma \models v_2 : \tau_2$

- $\Gamma \models c : \tau$ `chan` if $c \in \mathrm{Dom}(\Gamma)$ and $\tau = \Gamma(c)$
- $\models (f_s, x_s, a, e) : \tau_1 \to \tau_2$ if there exists a typing environment E such that
$$\Gamma \models e : E \quad \text{and} \quad E \vdash (f_s \texttt{ where } f_s(x_s) = a) : \tau_1 \to \tau_2$$
- $\Gamma \models (a, e) : \forall \alpha_1 \ldots \alpha_n . \tau$ if for all substitutions φ whose domain is included in $\{\alpha_1 \ldots \alpha_n\}$, there exists a typing environment E such that $\Gamma \models e : E$ and $E \vdash a : \varphi(\tau)$
- $S \models e : E$ if $\mathrm{Dom}(e) = \mathrm{Dom}(E)$, and for all $x_s \in \mathrm{Dom}(e)$, $E(x_s)$ is a simple type τ such that $S \models e(x_s) : \tau$, and for all $x_r \in \mathrm{Dom}(e)$, $E(x_r)$ is a type scheme σ such that $S \models e(x_r) : \sigma$.
- $\models u :? \Gamma$ if for all reception event $c ? v$ in the sequence u, we have $\Gamma \models v : \Gamma(c)$
- $\models u :! \Gamma$ if for all reception event $c ! v$ in the sequence u, we have $\Gamma \models v : \Gamma(c)$.

As in section 3.3.2, we assume given a closed, well-typed term a_0, where all `newchan`(a) subexpressions are distinct. Let $\mathcal{T}$ be a typing derivation for a_0, and $\mathcal{E}$ be an evaluation derivation for a_0. We construct a channel typing Γ suited to $\mathcal{T}$ and $\mathcal{E}$ as explained in section 3.3.2.

Proposition 6.2 (Strong soundness for channels) *Let $e \vdash a \stackrel{w}{\Longrightarrow} r$ be the conclusion of a sub-derivation of $\mathcal{E}$, and $E \vdash a : \tau$ be the conclusion of a sub-derivation of $\mathcal{T}$, for the same expression a. Assume $\Gamma \models e : E$.*

1. *If $\models w :? \Gamma$, then $r \neq$ `err`; instead, r is a value v such that $\Gamma \models v : \tau$, and moreover $\models w :! \Gamma$.*
2. *If $w = w'.c\,!\,v.w''$ and $\models u' :? \Gamma$, then $\Gamma \models v : \Gamma(c)$.*

Proof: by induction over the size of the evaluation derivation. We argue by case analysis over a, and thus over the last rule used in the typing derivation. Most cases are identical to those in the proof of proposition 3.9. The cases that differ are shown below.

• **Strict variables.**
$$\frac{\tau \leq E(x_s)}{E \vdash x_s : \tau}$$
By typing, we know that $x \in \mathrm{Dom}(E) = \mathrm{Dom}(e)$, hence the only evaluation rule that applies is $e \vdash x_s \stackrel{\varepsilon}{\Longrightarrow} e(x_s)$. By hypothesis over e and E, we have $\Gamma \models e(x_s) : E(x_s)$. Since $E(x_s)$ is a simple type, we have $\tau = E(x_s)$. Hence (1). (2) is trivially true, because $w = \varepsilon$ necessarily.

• **Delayed variables.**
$$\frac{\tau \leq E(x_r)}{E \vdash x_r : \tau}$$
Take $E(x_r) = \forall \alpha_1 \ldots \alpha_n . \tau_x$. Let φ be the substitution over the α_i such that $\tau = \varphi(\tau_x)$. By hypothesis over e and E, we know that $e(x_s) = (a_0, e_0)$, and there exists E_0 such that $S \models e_0 : E_0$ and $E_0 \vdash a_0 : \tau$. Since $e(x_r)$ is a suspension, only one evaluation rule applies:
$$\frac{e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0 \stackrel{w}{\Longrightarrow} r}{e \vdash x_r \stackrel{w}{\Longrightarrow} r}$$

We apply the induction hypothesis to the expression a_0, with type τ, in the environments e_0, E_0, and under the event sequence w. We obtain properties (1) and (2) for the evaluation of a_0; hence (1) and (2) for the evaluation of x_r.

• **`let poly` bindings.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x_r \mapsto \mathtt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \mathtt{let\ poly}\ x_r = a_1\ \mathtt{in}\ a_2 : \tau_2}$$

There's only one applicable rule:

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2 \stackrel{w}{\Longrightarrow} r}{e \vdash \mathtt{let\ poly}\ x_r = a_1\ \mathtt{in}\ a_2 \stackrel{w}{\Longrightarrow} r}$$

We show $\Gamma \models (a_1, e) : \mathtt{Gen}(\tau_1, E)$ as in the proof of proposition 6.1. Taking

$$e_1 = e + x \mapsto v_1 \qquad\qquad E_1 = E + x \mapsto \mathtt{Gen}(\tau_1, E),$$

we therefore have $\Gamma \models e_1 : E_1$. We apply the induction hypothesis to a_2, e_1, E_1 and w. The expected properties (1) and (2) follow. □

6.3.3 Continuations

We employ the following semantic typing relations:

$\models v : \tau$	the value v belongs to the type τ
$\models (a, e) : \sigma$	the suspension (a, e) belongs to all instances of the schema σ
$\models e : E$	the values and suspensions contained in the evaluation environment e belong to the corresponding types and schemes in the typing environment E
$\models k :: \tau$	the continuation k accepts all values belonging to the type τ

These relations are defined by:

- $\models cst : \mathtt{unit}$ if cst is `()`
- $\models cst : \mathtt{int}$ if cst is an integer
- $\models cst : \mathtt{bool}$ if cst is `true` or `false`
- $\models (v_1, v_2) : \tau_1 \times \tau_2$ if $\models v_1 : \tau_1$ and $\models v_2 : \tau_2$
- $\models k : \tau\ \mathtt{cont}$ if $\models k :: \tau$
- $\models (f_s, x_s, a, e) : \tau_1 \to \tau_2$ if there exists a typing environment E such that:

$$\models e : E \quad \text{and} \quad E \vdash (f_s\ \mathtt{where}\ f_s(x_s) = a) : \tau_1 \to \tau_2$$

- $\models (a, e) : \forall \alpha_1 \ldots \alpha_n.\, \tau$ if for all substitution φ whose domain is included in $\{\alpha_1 \ldots \alpha_n\}$, there exists a typing environment E such that $\models e : E$ and $E \vdash a : \varphi(\tau)$

- $\models e : E$ if $\mathrm{Dom}(e) = \mathrm{Dom}(E)$, and for all $x_s \in \mathrm{Dom}(e)$, $E(x_s)$ is a simple type τ such that $\models e(x_s) : \tau$, and for all $x_r \in \mathrm{Dom}(e)$, $E(x_r)$ is a schema σ such that $\models e(x_r) : \sigma$.
- $\models \mathtt{stop} :: \tau$ for all types τ
- $\models \mathtt{app1c}(a, e, k) :: \tau_1 \to \tau_2$ if there exists a typing environment E such that

$$E \vdash a : \tau_1 \quad \text{and} \quad \models e : E \quad \text{and} \quad \models k :: \tau_2$$

- $\models \mathtt{app2c}(f_s, x_s, a, e, k) :: \tau$ if there exists a typing environment E and a type τ' such that

$$E \vdash (f_s \ \mathtt{where}\ f_s(x_s) = a) : \tau \to \tau' \quad \text{and} \quad \models e : E \quad \text{and} \quad \models k :: \tau'$$

- $\models \mathtt{pair1c}(a, e, k) :: \tau$ if there exists a typing environment E and a type τ' such that

$$E \vdash a : \tau' \quad \text{and} \quad \models e : E \quad \text{and} \quad \models k :: \tau \times \tau'$$

- $\models \mathtt{pair2c}(v, k) :: \tau$ if there exists a type τ' such that

$$\models v : \tau' \quad \text{and} \quad \models k :: \tau' \times \tau$$

- $\models \mathtt{primc}(\mathtt{callcc}, k) :: \tau\ \mathtt{cont} \to \tau$ if $\models k :: \tau$
- $\models \mathtt{primc}(\mathtt{throw}, k) :: \tau\ \mathtt{cont} \times \tau$ for all τ.

Proposition 6.3 (Weak soundness for continuations)

1. *Let a be an expression, τ be a type, e be an evaluation environment, E be a typing environment, k be a continuation and r be a response such that*

$$E \vdash a : \tau \quad \textit{and} \quad \models e : E \quad \textit{and} \quad \models k :: \tau \quad \textit{and} \quad e \vdash a; k \Rightarrow r.$$

   Then $r \neq \mathtt{err}$.

2. *Let v be a value, k be a continuation, τ be a type and r be a response such that*

$$\models v : \tau \quad \textit{and} \quad \models k :: \tau \quad \textit{and} \quad \vdash v \triangleright k \Rightarrow r.$$

   Then $r \neq \mathtt{err}$.

Proof: the proof is an inductive argument over the size of the evaluation derivation. We argue, for (1), by case analysis over a and thus over the last typing rule used, and for (2), by case analysis over k. I only show the cases that are not identical to those in the proof of proposition 3.12.

• (1), strict variables.

$$\frac{\tau \leq E(x_s)}{E \vdash x_s : \tau}$$

The typing ensures that x_s belongs to the domain of E, hence by hypothesis $\models e : E$ we have $\mathrm{Dom}(E) = \mathrm{Dom}(e)$. Hence, the only evaluation possibility is:

$$\frac{x_s \in \mathrm{Dom}(e) \qquad \vdash e(x_s) \triangleright k \Rightarrow r}{e \vdash cst; k \Rightarrow r}$$

By hypothesis, we know that $\models e(x_s) : E(x_s)$, and $E(x_s) = \tau$ since $E(x_s)$ is a simple type. Hence $\models e(x_s) : \tau$. Applying the induction hypothesis (2) to the evaluation $\vdash e(x_s) \triangleright k \Rightarrow r$, we get $r \neq \mathtt{err}$.

• **(1), delayed variables.**

$$\frac{\tau \leq E(x_r)}{E \vdash x_r : \tau}$$

Write $E(x_r) = \forall \alpha_1 \ldots \alpha_n . \tau_x$. Let φ be the substitution over the α_i such that $\tau = \varphi(\tau_x)$. By hypothesis over e and E, we know that $e(x_s) = (a_0, e_0)$, and there exists E_0 such that $\models e_0 : E_0$ and $E_0 \vdash a_0 : \tau$. Since $e(x_r)$ is a suspension, the evaluation must end up with:

$$\frac{x_r \in \mathrm{Dom}(e) \qquad e(x_r) = (a_0, e_0) \qquad e_0 \vdash a_0; k \Rightarrow r}{e \vdash x_r; k \Rightarrow r}$$

We apply the induction hypothesis (1) to $a_0; k$ in E_0 and e_0. The expected result follows: $r \neq \mathtt{err}$.

• **(1), `let poly` bindings.**

$$\frac{E \vdash a_1 : \tau_1 \qquad E + x_r \mapsto \mathtt{Gen}(\tau_1, E) \vdash a_2 : \tau_2}{E \vdash \mathtt{let\ poly}\ x_r = a_1\ \mathtt{in}\ a_2 : \tau_2}$$

The evaluation must end up with:

$$\frac{e + x_r \mapsto (a_1, e) \vdash a_2; k \Rightarrow r}{e \vdash (\mathtt{let\ poly}\ x_r = a_1\ \mathtt{in}\ a_2); k \Rightarrow r}$$

We show $\models (a_1, e) : \mathtt{Gen}(\tau_1, E)$ as in the proof of proposition 6.1. Writing

$$e_1 = e + x_r \mapsto (a_1, e) \qquad\qquad E_1 = E + x_r \mapsto \mathtt{Gen}(\tau_1, E),$$

we have $\models e_1 : E_1$. Applying the induction hypothesis (1) to $a_2; k$, e_1, and E_1, it follows that $r \neq \mathtt{err}$, as expected. □

6.4 Discussion

Switching to polymorphism by name allows to correctly type references, channels, and continuations while keeping the ML typing rules and type algebra, which are simple and easy to understand — unlike the type systems for polymorphism by value, which require a richer type algebra and more

complex typing rules. In particular, writing types by hand in module interfaces remains easy, while this is difficult in the systems proposed in chapters 3 and 4, or in the effect systems presented in chapter 5. MLN, the variant of ML with polymorphism by name studied here, therefore looks like an interesting alternative to ML when non-purely applicative features are considered.

However, this variant is less expressive than ML on one point: it is not possible to define a polymorphic object once and for all, and have its value shared between all uses of the object. This can result in changes in semantics or in efficiency. The remainder of this chapter discusses the practical importance of these changes.

6.4.1 Semantic changes

Some programs do not have the same behavior in ML and in MLN: all programs where the computation of a polymorphic object has side-effects, or depends on the state. In the former case, the side-effects are performed once, at creation time, in ML, but zero, one, or several times in ML, once for each use of the object. Example: in ML, evaluating the following program:

```
let f = print("Hi!"); λx.x in f(f(f))
```

prints "`Hi!`" only once, while the corresponding MLN phrase,

```
let poly f = print("Hi!"); λx.x in f(f(f))
```

prints "`Hi!`" three times. Here is another example, that assumes defined a stamp generator `gensym`:

```
let stamper =
  let stamp = gensym() in λx.(x, stamp)
in ...
```

In ML, the `stamper` function thus defined takes as argument any object and pairs it with the mark already obtained — the same mark for each call. The straightforward translation to MLN behaves differently:

```
let poly stamper =
  let val stamp = gensym() in λx.(x, stamp)
in ...
```

Each call to `stamper` causes the re-evaluation of `gensym()`. Hence, a different mark is assigned each time. To keep the behavior of the original program, the program must be restructured as follows:

```
let val stamp = gensym() in
let poly stamper = λx.(x, stamp) in ...
```

In more complex (and more artificial) examples, deeper transformations might be required. My practical experience is that the situation shown above (defining a polymorphic object by an expression with side-effects) is very unfrequent. I have run about 10000 lines of ML through an experimental MLN compiler (see below, section 6.4.3), without encountering this situation. To translate the compiled programs, it has been sufficient to replace `let` by `let val` or `let poly`; no other transformation (as in the `stamper` example) was necessary.

Context. A language combining side-effects with a non-strict construct such as `let poly` might seem to encourage some programming errors: those where a side-effect does not occur at the expected time. It is often said that the most intuitive semantics for a language with side-effects is strict semantics with left-to-right evaluation, since this is the semantics where the sequencing of side-effects is closest to the order in which the side-effects appear in the source code. Some existing imperative languages do not follow this approach, however. The famous Algol 60 language uses call-by-name; so do some recent studies (Reynold's Forsythe language [84]). The Caml language mixes side-effects with lazy evaluation (with sharing of the evaluations) [99, chap. 4], making it very hard to predict when a side-effect inside a delayed expression occurs. In Caml, there's also the a `where` $x = b$ construct, where b is evaluated before a, even though it occurs after a in the source code. Finally, languages such as C, C++, Modula-3, Scheme, and even Caml Light, leave unspecified the order in which the subexpressions of an arithmetic expression are evaluated. My practical experience with C and Caml Light shows that this under-specification sometimes leads to tough errors, but has a beneficial effect on the readability of programs: it forces programmers to clearly indicate the side-effects, and the order in which they must occur. Adopting the non-strict semantics for polymorphism should have similar practical consequences. □

6.4.2 Efficiency changes

The two languages ML and MLN are equivalent when they are restricted to programs where all polymorphic objects are defined by expressions that always terminate, do not depend on the state, and have no observable side-effects. However, even in this case, we can observe differences in efficiency: in MLN, the polymorphic objects are not shared, but recomputed at each use; we might therefore fear that execution is slower than in ML.

In most programs, all polymorphic objects are function abstractions $\lambda x.\, a$. Evaluating these objects boils down to constructing a closure, which takes low, constant time. Reconstructing the closure each time the corresponding polymorphic identifier is used is therefore not costly, and does not change the time complexity of the program. Moreover, these extra closure constructions can be avoided in most cases by using the usual uncurrying techniques. Uncurrying consists in transforming (at compile-time) curried functions into functions with several arguments, which can be called more efficiently [2, section 6.2]. Here is a commonly encountered programming idiom:

```
let poly f = λx.a in
    ... f(1) ... f(true) ...
```

With polymorphism by name, the program above is compiled exactly like the ML program below:

```
let f = λ().λx.a in
    ... f()(1) ... f()(true) ...
```

The uncurrying optimization transforms `f` into a function with two arguments, `()` and `x`; the two applications of `f` are turned into two direct calls to `f`, almost as efficient as the simple applications `f(1)` and `f(true)`. The only additional cost comes from the passing of `()` as an extra argument; this cost drops to zero with some data representation techniques [50].

Some situations where curried functions are partially applied can result in polymorphic objects that are expensive to recompute, however. That's because a curried function can perform computations between the reception of its first argument and of its second argument. Binding (with a `let`)

the result of a partial application can therefore result in the sharing of these intermediate computations. With polymorphism by name, this sharing is lost if the result of the partial application must remain polymorphic. Here is an almost realistic example that demonstrates this situation; this is the only example I was able to come up with.

Example. Consider a function that sorts (key,data) pairs in increasing order of the keys. Assume that the keys and the associated data are not provided together, as an array of pairs for instance, but separately, as an array of keys and an array of data. The result of the function is the sorted array of data. To take advantage of partial application, the clever way to write this function is to compute the sorting permutation (an array of integers) as soon as the array of keys is passed, then return a function that simply applies the permutation over the given array of data:

```
let weird_sort =
  λorder. λkeys.
    let permut = ... in λitems. apply_permut(permut)(items)
```

This convoluted implementation is more efficient if we have to sort several arrays of data following the same array of keys:

```
let f = weird_sort (prefix <) (lots_of_integers) in
    ... f(lots_of_strings) ... f(lots_of_booleans) ...
```

In the phrase above, the intermediate function `f` is polymorphic (with type $\forall\alpha.\ \alpha$ `array` $\rightarrow$ α `array`), and therefore can be applied to arrays of different types — without having to sort again the array of keys. This last point holds in ML, but not in MLN: in order for `f` to be polymorphic, it must be bound by a `let poly` construct; then, each application `f(t)` evaluates as

```
weird_sort (prefix <) (lots_of_integers) (t)
```

Hence, all benefits of partial application are lost. On examples of this kind, MLN might turn out to be clearly less efficient than ML. □

6.4.3 An implementation

In an attempt at estimating the practical importance of the efficiency issue presented above, I have implemented a prototype compiler for MLP (ML with polymorphism by name), derived from my Caml Light system [51]. I then compared the efficiency of the code produced by this prototype compiler with the efficiency of the code produced by the original Caml Light system, which implements the by-value semantics for polymorphism.

6.4.3.1 The Caml Light execution model

Before commenting the results obtained, I briefly describe the Caml Light execution model, and how it can be adapted to polymorphism by name. (For more details, the reader is referred to [49, chapter 3].) The Caml Light execution model has a distinctive treatment of multiple applications, whose goal is to perform uncurrying "on-the-fly". I first show the compilation of the functional kernel of ML (with polymorphism by value), then extend these techniques to polymorphism by name.

The compilation process is formalized as two compilation schemes: the first, $CT(a)$, applies to expressions a in tail-call position; the other, $CN(a)$, applies to expressions not in tail-call position. A variable is compiled as an access instruction in the environment. (The actual structure of the environment is left unspecified in this presentation.)

$$CT(x) = \texttt{Access}(x) \qquad CN(x) = \texttt{Access}(x)$$

A function application to n (curried) arguments is compiled as follows:

$$\begin{aligned} CT(a(a_1)\ldots(a_n)) &= CN(a_n);\texttt{Push};\ldots;CN(a_1);\texttt{Push};CN(a);\texttt{Appterm} \\ CN(a(a_1)\ldots(a_n)) &= \texttt{Pushmark};CN(a_n);\texttt{Push};\ldots;CN(a_1);\texttt{Push};CN(a);\texttt{Apply} \end{aligned}$$

For a non-tail call, we first push a special value, the "mark", which separates the arguments passed to a from the other values on the stack. Then, we evaluate the arguments, from right to left, and push their values. Finally, the function part is evaluated into a closure, and control is transferred to the code part of this closure. For a tail call, we do not need to push a mark: the arguments passed to a are just added to the arguments that are already on the stack.

$$\begin{aligned} CT(\lambda x.\, a) &= \texttt{Grab}(x);CT(a) \\ CN(\lambda x.\, a) &= \texttt{Clos}(CT(\lambda x.\, a);\texttt{Return}) \end{aligned}$$

An abstraction over x in tail-call position translates to a $\texttt{Grab}(x)$ instruction. (For the sake of simplicity, I do not deal with recursive functions here.) At run-time, this instruction pops and tests the top of the argument stack. If the top of stack is a mark, this means that no more arguments have been provided to the function. The $\texttt{Grab}(x)$ instruction then builds a closure of the code $\texttt{Grab}(x);CT(a)$ with the current environment, and returns this closure to the caller. If the top of stack is not a mark, then it is the argument to which the abstraction is being applied. The $\texttt{Grab}(x)$ instruction then binds this value to the identifier x in the current environment, and proceeds sequentially with the evaluation of a (the body of the lambda-abstraction).

An abstraction not in tail-call position is never immediately applied, hence we compile a `Clos` instruction, that builds a closure of the code for $\lambda x.\, a$ with the current environment. The `Return` instruction that terminates the function code behaves symmetrically with `Grab`: if the top of the argument stack is a mark, this means that all arguments have been consumed, and the computed value is returned to the caller; if the top of stack is not a mark, some arguments remain to be consumed, hence `Return` applies (with a tail-call) the computed value (which can only be a closure) to the remaining arguments.

$$\begin{aligned} CT(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) &= CN(a_1);\texttt{Bind}(x);CT(a_2) \\ CN(\texttt{let}\ x = a_1\ \texttt{in}\ a_2) &= CN(a_1);\texttt{Bind}(x);CN(a_2);\texttt{Unbind}(x) \end{aligned}$$

The `let` binding is compiled classically in the two instructions `Bind` and `Unbind`, that add or remove a binding in the environment, without testing for a mark as `Grab` does.

The strength of this execution mechanism is its good behavior with respect to curried functions. For instance, the code fragment below executes without building any intermediate closure, and with only one jump/return between the caller and the callee.

```
let f = λx. λy. λz. x + y + z
in f(2)(3)(4)
```

In a more conventional model, such as for instance the SECD machine [46], we would have constructed two intermediate closures (corresponding to the partial applications `f(1)` and `f(1)(2)`), and we would have performed three jumps/returns. The Caml Light execution model therefore achieves almost the same efficiency as the compile-time uncurrying techniques: the three arguments to the curried application are passed to the function in one batch. (The only performance penalty is the tests on the top of stack performed by `Grab`.) This model turns out to be more powerful than compile-time uncurrying on curried functions that interleave parameter passing and internal computations:

```
let f = λx. let u = fib(x) in λy. let v = fact(y) in λz.u + v + z
in f(2)(3)(4)
```

In the example above, the three arguments are still passed in one batch, and only one jump/return is performed. The compile-time uncurrying techniques generally do not apply to the complex situation illustrated above.

6.4.3.2 Adaptation to polymorphism by name

The execution model summarized above clearly distinguishes between suspending the evaluation (instruction `Clos`) and passing a parameter (instruction `Grab`). It therefore provides good support for suspensions, and easily adapts to polymorphism by name.

The `let poly` binding is naturally compiled as creating a suspension and adding it to the environment.

$$\begin{aligned} CT(\texttt{let poly } x_r = a_1 \texttt{ in } a_2) &= \texttt{Clos}(CT(a_1);\texttt{Return});\texttt{Bind}(x_r);CT(a_2) \\ CN(\texttt{let poly } x_r = a_1 \texttt{ in } a_2) &= \texttt{Clos}(CT(a_1);\texttt{Return});\texttt{Bind}(x_r);CN(a_2);\texttt{Unbind}(x_r) \end{aligned}$$

Accessing a variable bound by a `let poly` is compiled as an application to zero arguments.

$$\begin{aligned} CT(x_r) &= \texttt{Access}(x_r);\texttt{Appterm} \\ CN(x_r) &= \texttt{Pushmark};\texttt{Access}(x_r);\texttt{Appterm} \end{aligned}$$

If this variable is immediately applied (which is often the case), we can combine the evaluation of the suspension and the application of the resulting value into a single application:

$$\begin{aligned} CT(x_r(a_1)\ldots(a_n)) &= CN(a_n);\texttt{Push};\ldots;CN(a_1);\texttt{Push};\texttt{Access}(x_r);\texttt{Appterm} \\ CN(x_r(a_1)\ldots(a_n)) &= \texttt{Pushmark};CN(a_n);\texttt{Push};\ldots;CN(a_1);\texttt{Push};\texttt{Access}(x_r);\texttt{Apply} \end{aligned}$$

The first line is an instance of the general rules for accessing a delayed variable and for tail function calls. The second line is not an instance of the general rules; it is a special optimization that avoid pushing two marks, and therefore to needlessly interrupt the evaluation of the suspension.

Test program	By-value semantics	By-name semantics	Slowdown by name/by value	Amount of polymorphism
Fibonacci	5.9 s	6.3 s	6 %	none
Church integers	2.5 s	2.9 s	16 %	high
Sieve	3.2 s	3.4 s	6 %	moderate
Word count	6.4 s	6.7 s	4 %	none
Boyer	16.0 s	18.0 s	12 %	low
Knuth-Bendix	7.9 s	8.3 s	5 %	moderate
Lexer generator	2.1 s	2.3 s	9 %	low
Caml Light compiler	7.1 s	8.3 s	16 %	low

Figure 6.1: Experimental comparison between polymorphism by name and polymorphism by value

As a consequence, applying a polymorphic function is just as efficient with polymorphism by name and with polymorphism by value. Consider the typical code fragment:

```
let poly f = λx.a in
    ... f(1) ...
    ... f(true) ...
```

The compiled code is:

```
Clos(Grab(x); CT(a)); Bind(f);
    ...; Pushmark; Const(1); Push; Access(f); Apply; ...
    ...; Pushmark; Const(true); Push; Access(f); Apply; ...
```

This code is exactly identical to the code produced by the compilation scheme for polymorphism by value when applied to the equivalent ML program:

```
let f = λx.a in
    ... f(1) ...
    ... f(true) ...
```

In this very common case, the by-name semantics for polymorphism introduces no performance penalty with respect to the by-value semantics.

6.4.3.3 Experimental results

Figure 6.1 compares the execution time of the modified Caml Light with polymorphism by name and the original Caml Light with polymorphism by value. The test programs comprise: some well-known toy programs (the Fibonacci sequence, operations on Church integers, Eratosthene's sieve over lists, a word counter); two medium-sized (100 and 500 lines) programs performing mostly symbolic processing, Boyer's simplified theorem prover and an implementation of the Knuth-Bendix completion algorithm; and two pieces of the Caml Light environment, ported to polymorphism by name and "bootstrapped", the lexical analyzer generator (1000 lines) and the compiler itself (8000 lines). Some of these programs are completely monomorphic (Fibonacci, word count). Others use polymorphic functions intensively (Church integers). The more realistic programs operate mostly

on monomorphic data structures, but make frequent use of generic functions over lists, hash tables, etc.

The execution times show that all programs, including the completely monomorphic ones, are slowed down when switching to polymorphism by name. The reason is that a minor optimization in the compiler no longer applies when we use the compilation scheme for polymorphism by name given above. Without polymorphism by name, all functions are always applied to at least one argument; hence, the initial `Grab` at the beginning of each function is omitted, and the `Apply` and `Appterm` instructions systematically add the first argument to the environment by themselves. With polymorphism by name, this is no longer the case: functions can be applied to zero arguments, hence the `Grab` at the beginning of each function cannot be omitted. This results in a performance penalty of about 5%, as can be deduced from the running times for the completely monomorphic examples (Fibonacci and word count).

In addition to this 5% slowdown for all programs, the programs that make use of polymorphism encounter a performance penalty from 1% to 10%, which represents the actual cost of polymorphism by name. There is no obvious correlation between the amount of polymorphic functions used in the program and the observed slowdown. A mostly monomorphic program such as the compiler is slowed down more than other programs that make heavy use of polymorphic functionals, such as the Knuth-Bendix implementation. It is true that I have estimates the static frequency of polymorphic functions (how many polymorphic function calls appear in the source code), but not the dynamic frequency (how many polymorphic function calls occur at run-time), which I was unable to measure.

These preliminary experimental results are encouraging: they demonstrate that an execution model initially designed for polymorphism by value can easily be adapted to polymorphism by name, without major efficiency lost. I believe these results are not specific to Caml Light, but should apply to any ML compiler equipped with good uncurrying mechanisms. At any rate, these results suffices to show that polymorphism by name cannot be dismissed easily on the grounds of inefficiency [29].

Conclusions

Having reached the end of this work, the least I can say is that polymorphic typing as in the ML language does not naturally extend from a purely applicative language to a language equipped with imperative features such as updatable data structures, communication channels as first-class values, or continuation objects.

This study reveals three desirable properties of such an extension which, empirically, seems very difficult to reconcile. First of all, the type system should be expressive. In particular, the non-applicative features should benefit from the full power of polymorphic typing. Second, the type algebra should remain simple enough to support the decomposition of programs into modules. In particular, the type specification for a function should indicate what result the function computes, but not how it computes this result. Finally, it would be nice to keep the ML semantics for polymorphism, that is, the by-value semantics. This semantics seems to be the most intuitive for a language with implicit polymorphism; it is also the most efficient in terms of execution speed.

All known approaches meet two of the requirements above, but not all three. The "historical" approach, followed by the standard ML, preserves the by-value semantics for polymorphism as well as reasonable compatibility with modular programming, but it sacrifices the expressiveness of the type system, making it unable to assign the most general polymorphic types to many useful generic functions. The type systems based on dangerous variables and closure typing, as introduced in this dissertation, as well as the most advanced effect systems, turn out to be highly expressive, and retain the by-value semantics for polymorphism; however, we can foresee major difficulties with modular programming, since the function types are much more precise than the ML types, hence they reveal too much information on how a function is implemented. Finally, polymorphism by name, as in chapter 6, is fully satisfactory from the standpoint of expressiveness and from the standpoint of modular programming, but it assigns a different semantics to polymorphism, a semantics that programmers might find less intuitive than the ML semantics.

A stool with three uneven legs is stable but uncomfortable. Similarly, each of the three partial solutions above is suited to many programming situations, but does not consitute a general solution. Sacrificing the expressiveness of the polymorphic typing of imperative constructs is acceptable for a "mostly applicative" programming style, where the non-applicative features are used rarely and locally. Complicating the type specifications is suited to the quick prototyping of small to medium-sized programs, for which the modular decomposition does not need to be made completely explicit. Finally, switching to by-name semantics for polymorphism is acceptable if full compatibility with ML is not required, and if we do not mind some efficiency loss in complicated situations. However,

the initial problem remains open: to propose a natural a fully satisfactory extension of the ML type system to the major features of algorithmic languages.

The present dissertation nonetheless contains two relatively important contributions to this problem. The first contribution is the type system based on dangerous variables and closure typing. In the setting of polymorphism by value, this system is, at the time of this writing, the most expressive of all known polymorphic type systems for references and continuations. Even more important than the fact that this system is highly expressive (the effect systems with regions and effect masking seems to give similar results on practical examples) is the way this expressiveness is obtained. This system relies only on the fact that type expressions describe what the values contain. Actually, this fact does not hold in the conventional type systems, and that's why closure typing had to be added. Yet, even when they are enriched by closure types, *the type expressions still express only static properties of the expressions* ("here is an approximation of what this expression computes"), *but no dynamic properties of the expressions* ("here is an approximation of how this expression computes"). The other approaches, from the imperative variables in Standard ML to the latest effect systems, all reflect in the types some information on the dynamic behavior of the expressions. I think this practice goes against some of the fundamental ideas behind type systems; in particular, it immediately leads to over-specification by the types in module interfaces. I am therefore glad to have demonstrated that it is not necessary to reflect the dynamic behavior of expressions in their types to correctly keep track of polymorphic references, channels, and continuations. (Unfortunately, this does not suffices to avoid the over-specification problem.)

The other contribution of this dissertation is to clearly propose polymorphism by name as an interesting alternative to ML when non-purely applicative features are considered. I do not claim originality for noticing that the polymorphic typing of references and continuations does not compromise soundness when polymorphism is given by-name semantics. This idea has been around for some time. The only merit I claim is to have written down the soundness proofs, thereby confirming the intuition, and to have demonstrated that polymorphism by name integrates pretty well in practice within an ML-like language. Yes, the by-name semantics for polymorphism is compatible with the implicit syntax and with type inference; we do not have to switch to the explicit syntax to benefit from polymorphism by name. No, polymorphism by name cannot be dismissed on the grounds of inefficiency: polymorphism by name can be compiled just as efficiently as polymorphism by value in most practical cases. It is true that the resulting language is not exactly ML; but it is a simple and highly practical solution to the problem of typing an algorithmic language with polymorphic types.

To finish with, I would like to conclude on the techniques I used to obtain these results. First of all, I have been able to reason about three features that are not easy to describe, since they are strongly "non-functional", by using only elementary techniques: the mathematical structures are term algebras; the specification tools are inference rules; the proof techniques are structural inductions. This approach is arguably less elegant than other formalisms that are mathematically richer; yet it results in proofs that are reasonably easy to follow, and that can be adapted to various settings. Just as relational semantics can almost directly be executed on a computer, I believe that the proofs I gave can almost directly be checked on a computer.

The methods leading to this results are simple: introducing indirections to describe cyclic situations (stores, for references; constraint sets, for closure types) by finite terms; arguing by structural

induction over the values; and treating closures and continuations as suspended expressions, whose semantic typing is essentially identical to their syntactic typing, instead of as value transformers, whose semantic typing relies on the continuity condition. Parts of these techniques can be found in Tofte's thesis [92]; I think I have carried them further, in particular by suppressing all proofs by co-induction.

This work has also revealed a few weaknesses of this approach based on relational semantics, however. For instance, it is not straightforward to add garbage collection rules to the calculus with references, describing the reclamation of unused memory cells. Also, proving a strong soundness property for a calculus with `callcc` in this formalism is still an open question. One may argue that the weak soundness property is sufficient evidence that the type system is sound; however, that's the strong soundness property which is at the basis of some type-based compilation techniques [50].

The main technical difficulty encountered in this work is the non-conservativity problem that is addressed in chapter 4. A similar problem appears with the effect systems that type allocation effects. I suspect this problem shows up each time function types are annotated by informations on the types of some of the objects appearing inside the function. The solution to this problem has a fairly simple semantic intuition, which essentially relies on the following claim: if an expression is well-typed assuming that two of its subexpressions have different types, then these subexpressions do not "communicate" at run-time; in particular, they cannot evaluate to the same reference. This property is obviously true if the two subexpressions belong to two different base types, such as `int` and `bool`; more interestingly, this property seems to hold even if the two subexpressions have types α and β (two distinct type variables). This "non-communication theorem" justifies the simplification operation over closure types consisting in erasing the closure type informations over type variables that are not free in the argument type or in the result type of the function.

What is surprising is that this non-communication result and the corresponding simplification operation over closure types do not appear anywhere in chapter 4. The system in chapter 4 relies on a refinement of type generalization, whose justification is purely syntactical. The labels put over function types can be viewed as type constructors with variable arity, and whose arguments can appear in any order. In contrast with the usual type constructors, with fixed arity, it is therefore correct to generalize over type variables that lie below a non-generalizable label. Each instantiation of the resulting generic type simply adds the produced instance to the non-generalizable label. The soundness proof essentially consists in checking that this operation correctly keeps track of all the types with which the values in the environment part of a closure are considered. The proof is a bit on the long side because the type system in chapter 4 is highly bureaucratic, yet that's the only result it establishes: I cannot see a non-communication result there, even between the lines.

It would certainly be illuminating to reformulate the conservative type system in chapter 4 in a way that remains closer to the initial semantic intuition. I have made several attempts in this direction, but they have all failed on the following point: the type systems obtained are not stable under substitution. That's because if the two variables α and β are identified, two expressions with types α and β respectively, which therefore are detected as non-communicating, now have the same type after substitution, and therefore appear to be communicating. In other terms, the non-communication criteria is most precise when principal typings are considered; it becomes less and less precise when the typings are weakened. To the best of my knowledge, it is an open issue to take into account the fact that a typing is principal for the proof of a semantic property.

Bibliography

[1] Hassan Aït-Kaci and Roger Nasr.
Integrating logic and functional programming.
Lisp and Symbolic Computation, 2(1):51–89, 1989.

[2] Andrew W. Appel.
Compiling with continuations.
Cambridge University Press, 1992.

[3] Andrew W. Appel and David B. MacQueen.
Standard ML reference manual (preliminary).
AT&T Bell Laboratories, 1989.
Included in the Standard ML of New Jersey distribution.

[4] Lennart Augustsson.
A compiler for lazy ML.
In *Lisp and Functional Programming 1984*, pages 218–227. ACM Press, 1984.

[5] Henry G. Baker.
Unify and conquer (garbage, updating, aliasing, ...) in functional languages.
In *Lisp and Functional Programming 1990.* ACM Press, 1990.

[6] Jean-Pierre Banâtre and Daniel Le Métayer.
A new computational model and its discipline of programming.
Research report 566, INRIA, 1986.

[7] Dave Berry.
The Edinburgh SML library.
Technical report ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[8] Dave Berry, Robin Milner, and David N. Turner.
A semantics for ML concurrency primitives.
In *Principles of Programming Languages 1992*, pages 119–129. ACM Press, 1992.

[9] Gérard Berry and Gérard Boudol.
The chemical abstract machine.
In *Principles of Programming Languages 1990.* ACM Press, 1990.

[10] Bernard Berthomieu.
Implementing CCS: the LCS experiment.
Technical report 89425, LAAS, December 1989.

[11] Luca Cardelli.
Basic polymorphic typechecking.
Science of Computer Programming, 8(2):147–172, 1987.

[12] Luca Cardelli.
Typeful programming.
In Erich J. Neuhold and Manfred Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, 1989.

[13] Luca Cardelli and John C. Mitchell.
Operations on records.
In *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 22–52, 1989.

[14] Luca Cardelli and Peter Wegner.
On understanding types, data abstraction, and polymorphism.
Computing surveys, 17(4), 1985.

[15] Nicholas Carriero and David Gelerntner.
Linda in context.
Communications of the ACM, 4(32):444–458, 1989.

[16] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, and Gilles Kahn.
Natural semantics on the computer.
Research report 416, INRIA, 1985.

[17] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn.
A simple applicative language: Mini-ML.
Technical Report 529, INRIA, 1986.

[18] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny.
The categorical abstract machine.
Science of Computer Programming, 8(2):173–202, 1987.

[19] Guy Cousineau and Gérard Huet.
The CAML primer.
Technical report 122, INRIA, 1990.

[20] Patrick Cousot and Radhia Cousot.
Inductive definitions, semantics and abstract interpretation.
In *Principles of Programming Languages 1992*, pages 83–94. ACM Press, 1992.

[21] Luis Damas.
Type assignment in programming languages.
PhD thesis, University of Edinburgh, 1985.

[22] Luis Damas and Robin Milner.
Principal type-schemes for functional programs.
In *Principles of Programming Languages 1982*, pages 207–212. ACM Press, 1982.

[23] Daniel de Rauglaudre.
An implementation of monomorphic dynamics in ML using closures and references.
Note, INRIA, projet Formel, 1991.

[24] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Christine Paulin-Mohring, and Benjamin Werner.
The COQ proof assistant user's guide: version 5.6.
Technical report 134, INRIA, 1991.

[25] Bruce F. Duba, Robert Harper, and David MacQueen.
Typing first-class continuations in ML.
In *Principles of Programming Languages 1991*, pages 163–173. ACM Press, 1991.

[26] Matthias Felleisen and Daniel P. Friedman.
A syntactic theory of sequential state.
Theoretical Computer Science, 69(3):243–287, 1989.

[27] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba.
A syntactic theory of sequential control.
Theoretical Computer Science, 52(3):205–237, 1987.

[28] You-Chin Fuh and Prateek Mishra.
Type inference with subtypes.
In *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

[29] David K. Gifford and John M. Lucassen.
Integrating functional and imperative programming.
In *Principles of Programming Languages 1986*, pages 28–38. ACM Press, 1986.

[30] Jean-Yves Girard.
Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.
Thèse d'État, Université Paris VII, 1972.

[31] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth.
Edinburgh LCF, volume 78 of *Lecture Notes in Computer Science.*
Springer-Verlag, 1979.

[32] Carl A. Gunter and Didier Rémy.
Ravl: A language for programming with records and variants.
Submitted for publication, 1992.

[33] Carl A. Gunter and Dana S. Scott.
Semantic domains.
In van Leeuwen [96], pages 635–674.

[34] Robert Harper.
Introduction to Standard ML.
Technical report ECS-LFCS-86-14, University of Edinburgh, 1986.

[35] Robert Harper and Mark Lillibridge.
ML with `callcc` is unsound.
Message sent to the `sml` mailing list, July 1991.

[36] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand.
Obtaining coroutines from continuations.
Computer Languages, 11(3/4):143–153, 1986.

[37] C. A. R. Hoare.
Communicating sequential processes.
Prentice-Hall, 1985.

[38] Paul Hudak, Simon Peyton Jones, and Philip Wadler.
Report on the programming language Haskell, version 1.1.
Technical report, Yale University, 1991.

[39] Lalita A. Jategaonkar and John C. Mitchell.
ML with extended pattern matching and subtypes.
In *Lisp and Functional Programming 1988*, pages 198–211. ACM Press, 1988.

[40] Pierre Jouvelot and David K. Gifford.
Algebraic reconstruction of types and effects.
In *Principles of Programming Languages 1991*, pages 303–310. ACM Press, 1991.

[41] Gilles Kahn.
The semantics of a simple language for parallel programming.
In *Information processing 74*, pages 471–475. North-Holland, 1974.

[42] Gilles Kahn.
Natural semantics.
In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–257. Elsevier, 1988.

[43] Gilles Kahn and David MacQueen.
Coroutines and networks of parallel processes.
Research report 202, INRIA, 1976.

[44] Paris Kanellakis and John C. Mitchell.
Polymorphic unification and ML typing.
In *Principles of Programming Languages 1989*. ACM Press, 1989.

[45] Yves Lafont.
Interaction nets.
In *Principles of Programming Languages 1990*. ACM Press, 1990.

[46] P. J. Landin.
The mechanical evaluation of expressions.
The Computer Journal, 6:308–320, 1964.

[47] P. J. Landin.
The next 700 programming languages.
Communications of the ACM, 9(3):157–165, 1966.

[48] Xavier Leroy.
Une extension de Caml vers la programmation logique.
Mémoire de première année, Ecole Normale Supérieure, 1988.

[49] Xavier Leroy.
The ZINC experiment: an economical implementation of the ML language.
Technical report 117, INRIA, 1990.

[50] Xavier Leroy.

Unboxed objects and polymorphic typing.
In *Principles of Programming Languages 1992*, pages 177–188. ACM Press, 1992.

[51] Xavier Leroy and Michel Mauny.
The Caml Light system, version 0.5 — documentation and user's guide.
Technical report L-5, INRIA, 1992.

[52] Xavier Leroy and Pierre Weis.
Polymorphic type inference and assignment.
In *Principles of Programming Languages 1991*, pages 291–302. ACM Press, 1991.

[53] Barbara Liskov, Russell Atkinson, and Toby Bloom.
CLU reference manual, volume 114 of *Lecture Notes in Computer Science*.
Springer-Verlag, 1981.

[54] John M. Lucassen and David K. Gifford.
Polymorphic effect systems.
In *Principles of Programming Languages 1988*, pages 47–57. ACM Press, 1988.

[55] David MacQueen.
Modules for Standard ML.
In *Lisp and Functional Programming 1984*, pages 198–207. ACM Press, 1984.

[56] David MacQueen, Gordon Plotkin, and Ravi Sethi.
An ideal model for recursive polymorphic types.
Information and Control, 71:95–130, 1986.

[57] David C. J. Matthews.
Poly manual.
Technical Report 63, University of Cambridge, 1985.

[58] Robin Milner.
A theory of type polymorphism in programming.
Journal of Computer and System Sciences, 3(17):348–375, 1978.

[59] Robin Milner.
Communication and Concurrency.
Prentice-Hall, 1990.

[60] Robin Milner.
Message sent to the `sml` mailing list, July 1991.

[61] Robin Milner, Joachim Parrow, and David Walker.
A calculus of mobile processes: part 1.
Research report ECS-LFCS-89-85, University of Edinburgh, 1989.

[62] Robin Milner and Mads Tofte.
Co-induction in relational semantics.
Theoretical Computer Science, 87:209–220, 1991.

[63] Robin Milner and Mads Tofte.
Commentary on Standard ML.
The MIT Press, 1991.

[64] Robin Milner, Mads Tofte, and Robert Harper.
The definition of Standard ML.
The MIT Press, 1990.

[65] John C. Mitchell.
Coercion and type inference.
In *Principles of Programming Languages 1984*, pages 175–185. ACM Press, 1984.

[66] John C. Mitchell.
Type systems for programming languages.
In van Leeuwen [96], pages 367–458.

[67] Vladimir Nabokov.
Ada or ardor, a family chronicle.
McGraw-Hill, 1969.

[68] Flemming Nielson.
The typed λ-calculus with first-class processes.
In *PARLE '89*, volume 366 of *Lecture Notes in Computer Science*, pages 357–373, 1989.

[69] Atsushi Ohori and Peter Buneman.
Type inference in a database language.
In *Lisp and Functional Programming 1988*, pages 174–183. ACM Press, 1988.

[70] D. L. Parnas.
On the criteria to be used in decomposing systems into modules.
Communications of the ACM, 15(12):1053–1058, 1972.

[71] Lawrence C. Paulson.
ML for the working programmer.
Cambridge University Press, 1991.

[72] Simon Peyton-Jones.
The implementation of functional programming languages.
Prentice-Hall, 1987.

[73] Gordon D. Plotkin.
A structural approach to operational semantics.
Technical Report DAIMI FN-19, Aarhus University, 1981.

[74] Vincent Poirriez.
Intégration de fonctionnalités logiques dans un langage fonctionnel fortement typé: MLOG, une extension de ML.
Thèse de doctorat, Université Paris VII, 1991.

[75] Carl G. Ponder, Patrick C. McGeer, and Antony P.C. Ng.
Are applicative languages inefficient?
SIGPLAN Notices, 23(6), 1988.

[76] Jonathan A. Rees and William Clinger.
Revised^3 report on the algorithmic language Scheme.
ACM Sigplan Notices, 21(12), December 1986.

[77] Didier Rémy.

Records and variants as a natural extension of ML.
In *Principles of Programming Languages 1989*. ACM Press, 1989.

[78] Didier Rémy.
Algèbres touffues. Application au typage polymorphe des objets enregistrements dans les langages fonctionnels.
Thèse de doctorat, Université Paris VII, 1991.

[79] Didier Rémy.
Type inference for records in a natural extension of ML.
Research report 1431, INRIA, 1991.

[80] John H. Reppy.
First-class synchronous operations in Standard ML.
Technical Report TR 89-1068, Cornell University, 1989.

[81] John H. Reppy.
CML: a higher-order concurrent language.
SIGPLAN Notices, 6(26):294–305, 1991.

[82] John C. Reynolds.
Toward a theory of type structure.
In *Programming Symposium, Paris, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[83] John C. Reynolds.
Three approaches to type structure.
In *Mathematical Foundations of Software Development (TAPSOFT 85)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer-Verlag, 1985.

[84] John C. Reynolds.
Preliminary design of the programming language Forsythe.
Technical report CMU-CS-88-159, Carnegie Mellon University, 1988.

[85] J.A. Robinson.
A machine-oriented logic based on the resolution principle.
Journal of the ACM, 12(1):23–41, 1965.

[86] François Rouaix.
ALCOOL-90: Typage de la surcharge dans un langage fonctionnel.
Thèse de doctorat, Université Paris 7, 1990.

[87] François Rouaix.
Safe run-time overloading.
In *Principles of Programming Languages 1990*. ACM Press, 1990.

[88] Robert Sedgewick.
Algorithms.
Addison-Wesley, second edition, 1988.

[89] Gert Smolka.
FRESH: a higher-order language with unification and multiple results.
In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 469–524. Prentice-Hall, 1986.

[90] Jean-Pierre Talpin and Pierre Jouvelot.
The type and effect discipline.
In *Logic in Computer Science 1992*. IEEE Computer Society Press, 1992.

[91] Bent Thomsen.
A calculus of higher-order communicationg systems.
In *Principles of Programming Languages 1989*. ACM Press, 1989.

[92] Mads Tofte.
Operational semantics and polymorphic type inference.
PhD thesis, University of Edinburgh, 1987.

[93] Mads Tofte.
Type inference for polymorphic references.
Information and Computation, 89(1), 1990.

[94] David A. Turner.
Miranda, a non-strict functional language with polymorphic types.
In *Functional Programming Languages and Computer Architecture 1985*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

[95] U.S. Department of Defense.
Reference manual for the ADA programming language, 1983.
ANSI-MIL-STD 1815A.

[96] Jan van Leeuwen, editor.
Handbook of Theoretical Computer Science, volume B.
The MIT Press/Elsevier, 1990.

[97] Mitchell Wand.
Continuation-based multiprocessing.
In *Lisp and Functional Programming 1980*, pages 19–28. ACM Press, 1980.

[98] Mitchell Wand.
Complete type inference for simple objects.
In *Logic in Computer Science 1987*, pages 37–44. IEEE Computer Society Press, 1987.

[99] Pierre Weis.
The CAML reference manual, version 2.6.1.
Technical report 121, INRIA, 1990.

[100] Andrew K. Wright.
Typing references by effect inference.
In *European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[101] Andrew K. Wright and Matthias Felleisen.
A syntactic approach to type soundness.
Technical report TR91-160, Rice University, 1991.

Contents