THESIS

presented at

UNIVERSITÉ PARIS VII

to obtain the title of

DOCTOR OF UNIVERSITÉ PARIS VII

Specialty:

COMPUTER SCIENCE

by

François POTTIER

Subject of the thesis:

TYPE INFERENCE IN THE PRESENCE OF SUBTYPING: FROM THEORY TO PRACTICE

Defended on July 3rd, 1998 against a jury composed of:

Messrs.	**Jean-Pierre**	**JOUANNAUD**	President
	Claude	**KIRCHNER**	Reviewers
	Jens	**PALSBERG**	
	Dominique	**BOLIGNANO**	Examiners
	Guy	**COUSINEAU**	
	Michel	**MAUNY**	
	Didier	**RÉMY**	

Contents

Acknowledgements

First, I wish to express particular thanks to Mr. Didier Rémy for accepting to advise me during this work, and taking this task very much to heart. With amazing intuition and confidence, Didier was able to prompt a sometimes perplexed student to overcome each difficulty.

Me:	It doesn't work. (*a nebulous technical explanation follows*)
Him:	Yet a clear intuition exists...
	(*sketching a mysterious diagram*) It must work.
Me:	... !

With him, I believe I have learned to consider things no longer from a purely technical point of view, but with hindsight, by first rephrasing each problem so as to highlight its essential questions, before attempting to answer anything. In short, his experience allowed me to acquire—or so I hope—the ways of a researcher.

I extend my thanks to all members of the jury—in particular to Messrs. Claude Kirchner and Jens Palsberg, who accepted the burden of reviewing the manuscript—for the interest and time they granted to this work.

Lastly, thanks to all members of the Cristal and Coq projects at INRIA Rocquencourt. Throughout these four years, they have provided a work environment remarkable both for its high scientific level and for its relaxed atmosphere.

Introduction

Computer programming is often considered, at worst, as an obscure practice, or at best, as an art. One of the reasons for this misconception is the difficulty of writing an error-free program. So, to turn programming into a science, researchers have attempted, for a few decades, to develop rigorous methods to guarantee the absence of errors in a program.

In its broadest meaning, the notion of error refers to the famous "bugs": a program contains an error if its behavior does not correspond exactly to its author's expectations. Thus, to guarantee the absence of such errors, one must first give a description, or *specification*, of the intended behavior, and then prove, in a mathematical way, that the program abides by it. The specification cannot, in general, be generated automatically, because it depends on the desired goal. As for the proof, it essentially contains an explanation of the program's functioning; hence, it may be very complex, and it is difficult to have it written by a machine. So, *proving programs* essentially remains a human activity, even though a proof can be *checked* mechanically.

There exists a less general, but still interesting, notion: the *execution error*. A program contains an execution error if, when the machine executes this program, it encounters a meaningless instruction, such as adding an integer to a boolean, or read a piece of data at a non-existent address. Being able to guarantee the absence of such errors is interesting: even though we still will not be certain that the program computes the expected result, we shall at least be assured that it won't "crash". Thus, by considering a more restricted class of errors, we have weakened our result; but, on the other hand, we can now hope to establish this result automatically, thanks to the *typing* mechanism.

Typing—type inference

Thus, typing is a simple form of program proof. As such, it contains the two steps mentioned above: specification and proof.

Here, a program's specification is its *type*. For instance, the elementary operation `+` can be given the type $\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$, thus indicating that this function accepts two integer arguments and returns an integer result. This type partly describes the behavior of the `+` function. In particular, it allows detecting the execution error caused by computing `3 + true`, since the pair `(3, true)`, of type $\texttt{int} \times \texttt{bool}$, is not a suitable argument for `+`. However, this is only a partial specification, since other functions, such as `-`, have the same type. If the programmer confuses `+` with `-`, the error shall not be reported by the typing process; it is not an execution error.

To show that a given program admits a given type, one uses a set of *typing rules*. These rules are often syntax-directed, that is, one rule is associated to each of the language's syntactic constructs. For instance, the rule associated to function application typically has the following form:

$$\frac{f : \tau \to \tau' \qquad e : \tau}{f(e) : \tau'}$$

This rule tells that if the expression e has the type τ expected by the function f, then the application $f(e)$ is legal—it shall not cause any execution error—and its type is f's result type, namely τ'. By combining instances of the typing rules, one creates *typing derivations*; a derivation whose conclusion is $e : \tau$ proves that the program e admits type τ.

Different type systems have different types and typing rules. However, all systems share a common property: correctness, that is, the guarantee that if a program admits a type—in other words, if it is *well-typed*—then it shall not cause any execution error. Of course, this property has to be formally stated and proved. To state it, one has to define the notion of execution, by supplying a *semantics* of the programming language of interest.

Let us note, in passing, that this definition of typing is sometimes called *strong* and *static*. The former term refers to the correctness property mentioned above. Thus, if a type system is not strong, it is simply "broken", since it does not fulfill its mission, which is to detect all errors. The latter is opposed to *dynamic* typing, where errors are detected not at compile-time, but during execution, as they occur. This technique is, of course, more flexible, but much less reliable, since the absence of errors is no longer guaranteed; errors shall then merely cause an abrupt error message, rather than a true "crash". So, we consider that any type system should be strong and static.

Finally, when we dismissed the general notion of program proof in favor of typing, we mentioned that the latter's advantage is its possible automation. Indeed, most type systems are simple enough for *typechecking* to be decidable. This means that if the programmer supplies enough information about the type of the objects he works with, then the machine is able to automatically verify that this information is correct, and that the program indeed admits the specified type. This property sounds natural; yet, a few classic systems, such as some variants of $F_{\leq}$ [41], do not enjoy it. Better yet, one might want to perform *type inference*, that is, given a program without type annotations, automatically reconstruct these annotations. Thus, if the program is well-typed, the machine shall compute its type. In type systems equipped with *polymorphism*, a single program can have several types; the inference algorithm shall then produce the most general one, that is, the one which gives the most precise specification of the program. Type inference can be a difficult problem; for instance, in system F [23], typechecking is decidable, but type inference isn't. However, when it is possible, the latter provides an undeniable comfort, since it allows the programmer to omit all type annotations, and concentrate on his program's logic. Among the languages with type inference used in practice, the most important are probably those of the ML family [12].

In this thesis, we take interest in a type system where type inference is decidable. We recall the type inference algorithm, and we focus on *simplifying* inferred types, to enhance the algorithm's efficiency, as well as the readability of its results. Among this type system's characteristic features is *subtyping*.

Subtyping

We have stated that a type system, in order to have practical interest, must be decidable and correct. However, it is well known that determining whether a program contains an error is an undecidable problem. Hence, well-typed programs do not coincide with correct ones; in other words, there necessarily exist error-free programs which are ill-typed.

Thus, no system is perfect; all systems reject programs which are correct, but too complex to be recognized as such. This is one of the reasons why there are so many different systems:

there is a quest for finer and finer systems, so as to restrict the user's freedom as little as possible.

Among the various features developed to make type systems more and more expressive, let us briefly cite *parametric polymorphism*, that is, the ability to abstract a type with respect to a *type variable*. For instance, the identity function $\lambda x.x$ can be given the *type scheme* $\forall\alpha.\alpha \rightarrow \alpha$, which means that it has, at the same time, the types $\texttt{int} \rightarrow \texttt{int}$, $\texttt{bool} \rightarrow \texttt{bool}$, etc. Thus, it can be passed an argument of any type. This feature is present, in a very general form, in system F; unfortunately, type inference isn't decidable in it. To allow inference, the ML language adopts a restriction to *first-order polymorphism*, where the $\forall$ quantifier can only appear in front of a type, not inside it. The system presented here shall use the same form of polymorphism.

The introduction of *subtyping*, suggested independently by several authors, such as Cardelli [11] and Mitchell [35], is another way of enhancing a type system's expressiveness. An ordering on types is given; one says that τ is a *subtype* of τ' when $\tau \leq \tau'$. Then, a new rule is added to the system, which usually has the following aspect:

$$\frac{e : \tau \qquad \tau \leq \tau'}{e : \tau'}$$

This rule, called the *subtyping rule*, tells that if an expression e has type τ, then it also has all types τ' of which τ is a subtype. In other words, this rule allows weakening an expression's type, by replacing it with a less precise type. For instance, if we use the base types `int` and `float` to represent integers and floating-point numbers, then we can set $\texttt{int} \leq \texttt{float}$, that is, declare that `int` is a subtype of `float`. The subtyping rule then allows considering any expression of type `int` as also having type `float`. This allows us to pass an integer as argument to a function which expects a floating-point number, without having to perform an explicit conversion: for instance, if `x` is an integer, one can write `log x` instead of `log (float_of_int x)`.

Besides, the relationship $\texttt{int} \leq \texttt{float}$ induces relationships on composite types. For instance, a function capable of handling a real argument can pretend that it accepts integers only; and, on the contrary, a function which computes an integer result can pretend that it returns a real. Thus, we can set $\texttt{float} \rightarrow \texttt{int} \leq \texttt{int} \rightarrow \texttt{float}$. More generally, we shall have $\tau_0 \rightarrow \tau_1 \leq \tau_0' \rightarrow \tau_1'$ if and only if $\tau_0' \leq \tau_0$ and $\tau_1 \leq \tau_1'$. Notice that the relationship's direction is reversed between τ_0 and τ_0'; one says that the $\rightarrow$ constructor is *contravariant* with respect to its first argument and *covariant* with respect to the second one.

Up to here, the subtyping relation is non-trivial only on base types, and then extends in a regular way to composite types. Such a setup is sometimes called *atomic subtyping*. Type inference in such a system has been thoroughly studied [35, 28, 45, 20]; in particular, various complexity bounds are known, depending on the form of the subtyping relation. Fuh and Mishra [21, 22] took interest in atomic subtyping in conjunction with ML polymorphism, and in simplifying typings in this setting. However, from a practical point of view, limiting subtyping to relationships between base types, such as $\texttt{int} \leq \texttt{float}$, can seem rather uninteresting. If the notion of subtyping has been so successful in the past few years, it is mainly because it seems to play a central role in the definition of type systems for *object oriented languages*.

These languages offer a philosophy which differs slightly from that proposed by classic languages. Instead of handling data, one deals with *objects*. An object is simply an entity capable of answering certain *messages* with a value. All objects do not answer the same messages; in particular, sending a message causes an execution error if the object does not know how to answer it. A classic way to encode objects consists in representing them using plain *records* where each field corresponds to a message. Thus, to reuse a famous example, a "point" object shall have type `{ x: int }`, whereas a "colored point" object shall have type `{ x: int; c: color }`. However, at the heart of object oriented programming is the idea that objects of different natures can be handled uniformly, provided they are only sent messages common to all of them. For instance, one wishes to be able to apply the function $\lambda p.(p.\mathrm{x})$, which sends the message `x` to a point p, to any point, colored or not. In an ML-like system, it would then be necessary that this function's domain be unifiable with `{ x: int }` and with `{ x: int; c: color }`, which is not the case. Subtyping provides a solution to this problem: one declares that `{ x: int; c: color }` is a subtype of `{ x: int }`. The function $\lambda p.(p.\mathrm{x})$ has type `{ x: int }` $\rightarrow$ `int`. So, its domain is a common supertype of `{ x: int }` and of `{ x: int; c: color }`, which indicates that it can be applied to points as well as to colored points. Generally speaking, the subtyping relation allows to "forget" that an object accepts certain messages, and thus, to mix it with other objects, which otherwise would have had different types. Thus, these objects can be passed to a single function, or stored inside a single heterogeneous data structure. The term *polymorphism* is sometimes used to refer to this flexibility of object-oriented type systems.

Note that in this case, the subtyping relation isn't atomic any more. When subtyping was generated solely by relationships between base types, if two types without variables were in the subtyping relation, then they necessarily had the same structure; that is, if seen as trees, they would differ only by their leaves. However, this is no longer the case here, since we now have `{ x: int; c: color }` $\leq$ `{ x: int }`, where the left-hand tree has two branches and the right-hand one has a single one. Actually, from a theoretical point of view, it is possible to define a non-atomic subtyping relation without needing to introduce these record types. It suffices to equip the set of types with a smallest element $\perp$ and a greatest element $\top$. We then have, for instance, $\perp \leq \top \rightarrow \perp$, so subtyping isn't atomic. Under these conditions, a constraint of the form $\alpha \leq \beta_0 \rightarrow \beta_1$ does not allow us to state that α can be written $\alpha_0 \rightarrow \alpha_1$, because α could also be $\perp$. This means that a subtyping constraint can no longer be decomposed into a set of constraints between variables and base types, as in the atomic case.

Next, note that typing objects typically requires handling recursive types, because of the inheritance mechanism, which allows a method to refer to the receiver object through the keyword `self`. By comparison, in the ML language, any recursive unification constraint is illegal. Recursive types must be explicitly declared and named. Consequently, to allow inference, a given record label must appear in at most one type definition. However, this restriction is not acceptable when programming in object oriented style, since one precisely wishes that objects of different types accept common messages. So, we shall not use named types, and we shall accept recursive subtyping constraints. Note that, hence, the user shall not need to supply any type declarations.

Thus, we are led to study a system where subtyping is not atomic, and where subtyping constraints are possibly recursive. Various languages and type systems come to mind; however, the type inference algorithms associated to these choices all rely on the same basic idea.

Type inference with subtyping

How does one perform type inference with subtyping? Most of the existing inference algorithms follow a common approach, based on *constraints*. Its principle is simple. Besides, ML's inference algorithm can itself be considered a particular case of this general method. So, rather than considering these algorithms as entirely new, one can view them as generalizing a well-known algorithm.

Here is the principle, stated abstractly. (The problem of polymorphism, introduced by the `let` construct, is ignored here.) First, a type variable is associated with each node of the program, i.e. with each sub-expression. Then, from the program's syntactic structure, one derives a series of relationships, or *constraints*, between these variables. If the constraint set thus obtained admits a solution, then the program is well-typed; otherwise, it is rejected.

In the case of ML, constraints are equalities between types, and a constraint is generated at each function application node. Indeed, the type of the argument supplied to a function must coincide with the type of the expected parameter. In other words, if we consider the application $(e_1^{\alpha_1}\ e_2^{\alpha_2})^\alpha$, where each sub-expression is annotated with its associated type variable, then we necessarily have $\alpha_1 = \alpha_2 \rightarrow \alpha$. Thus, analyzing the program generates as many constraints as there are application nodes. There remains to determine whether these constraints admit a solution, which the unification algorithm does in a particularly efficient way. Furthermore, this algorithm computes the most general unifier, which describes the set of solutions; thus, constraints are never dealt with explicitly.

When the subtyping rule is added, things become slightly more complex. Indeed, when a function is applied, it is no longer necessary that the supplied type coincide with the expected type; it is enough for the former to be a subtype of the latter. Thus, the application $(e_1^{\alpha_1}\ e_2^{\alpha_2})^\alpha$ leads to the constraint $\alpha_1 \leq \alpha_2 \rightarrow \alpha$. This time, we have a subtyping constraint, rather than an equality constraint. So, analyzing the program generates a conjunction of such constraints, and it suffices to determine whether it has a solution to check that the program is well-typed. Unfortunately—this is the main novelty—the unification algorithm no longer applies. A *closure* algorithm is used to determine whether the constraints are solvable. Besides, the constraints do not necessarily admit a principal solution; thus, the only known way of describing the set of all solutions is to supply the constraint set itself. Hence, the principal type scheme associated to the program shall explicitly contain a constraint set. Type schemes shall be of the form $\forall\bar{\alpha} \mid C.\tau$; such a scheme indicates that the program has type τ, provided the variables $\bar{\alpha}$ satisfy the conjunction of constraints C.

So far, we have discussed typing and type inference only. Yet, it is important to realize that the same algorithms could be described in terms of abstract interpretation. Indeed, since each function application gives birth to a constraint, the constraint graph is in fact an approximate representation of the program's data flow. For instance, in the absence of polymorphism, our system would be equivalent to a 0-CFA, as pointed out by Palsberg, O'Keefe and Smith [39, 40]. One can also compare our type inference system to a modular abstract analysis, such as SBA [25, 19]. In both cases, the analysis yields a description of the program's behavior, as a set of constraints. Typing is traditionally distinguished from abstract interpretation, because of its greater simplicity and lesser precision; here, the type systems at hand are precise enough that the distinction blurs, and both points of view are possible. Besides, note that this parallel with abstract interpretation allows understanding some of our simplification algorithms in a more graphic way: for instance, *garbage collection* simply consists in simulating a flow of data through the constraint graph, and in destroying all unused edges.

Thus, the use of constraints is a common feature of the various type inference systems with subtyping. However, these systems exhibit many differences. First, the programming language of interest can vary. Then, the syntax of constraints, as well as their semantics, differ. Thus, the algorithms used to solve and to simplify constraints, which directly depend on them, shall vary largely across systems.

Which system to choose?

We explained our interest in performing type inference in a system where subtyping constraints are non-atomic and recursive. A rather diverse palette of choices exists; here are a few of its most prominent members.

In this thesis, we choose to take interest in a λ-calculus with `let`, that is, an extremely simple functional core—identical to the one used in ML. Indeed, the notion of subtyping is not restricted to object oriented languages; thus, it seems natural to study it in the well-known setting of functional languages, rather than choose one of the various proposals for object oriented calculi. Furthermore, it is important not to dismiss the notion of function, since it requires mastering the contravariance phenomenon. The type system is reduced as much as possible: ground types are formed using the constructors $\bot$, $\top$, and $\rightarrow$. This system, although restricted, already has a reasonable intrinsic complexity; thus, our results shall easily carry over to a richer system, equipped with base types, extensible record and variant types, reference types, etc. One can then come back to the notion of object, either through an encoding, or by applying our techniques to a language equipped with primitive object oriented constructs. The problem of type inference for this system was initially solved by Eifrig, Smith and Trifonov [15], who wished to later apply it to object oriented languages.

On the contrary, Abadi and Cardelli [1, 2] propose what could be considered a canonical object oriented language. It is an extremely simple calculus, where the notion of object is primitive. Abadi and Cardelli successively study different systems, of increasing complexity, for this language. All of them have a subtyping rule, which allows, as expected, to "forget" the presence of certain methods, and is thus non atomic. The problem of type inference, for the most powerful of the so-called *first-order* systems, has been solved by Palsberg [38], who uses an algorithm based on the same principle as that of Eifrig, Smith and Trifonov. However, the two systems exhibit a fundamental difference: whereas Eifrig *et al.*'s $\rightarrow$ constructor is contravariant with respect to its first argument and covariant with respect to the second one, Abadi and Cardelli's object types are *invariant*; that is, a subtyping relationship between two object types entails the *equality* of their common components. As shown by Henglein [26], this peculiarity allows enhancing the inference algorithm's efficiency. To simulate function types in a satisfactory way, Abadi and Cardelli introduce universally and existentially quantified types, but in doing so, lose type inference.

Aiken, Wimmers and Lakshman [6, 7] also take interest in λ-calculus with `let`, but adopt a vastly different type system. In our system, as in ML, ground types are terms, ordered by the subtyping relation. Aiken *et al.* choose to use the *ideal model* [34], that is, ground types are subsets of the model, ordered by set-theoretic inclusion. In both cases, type inference involves constraint solving; however, in the former case, constraints are written in a dedicated formalism, whereas in the latter, the general theory of set constraints is used. As a result, the system is more expressive, and more complex. The simplicity of our system allows us to formalize our simplification methods rather easily, whereas the algorithms implemented in Illyria [3] remain undocumented. As for the loss of expressiveness, it is partly compensated by the addition of complex types to our system. Still, Aiken *et al.*'s system remains more precise; for instance, we are currently unable to type pattern matchings as precisely as done in [7].

Felleisen and Flanagan [18, 19, 17] also manipulate set constraints. Here, the issue is not typing any more, but a *set-based analysis*; however, as pointed out above, the distinction is not necessarily useful. Felleisen and Flanagan's system shares several aspects with ours; actually, our *minimization* algorithm is inspired by it. The main difference probably lies in the treatment of functions. Indeed, in our system, a function's domain is the type of its *formal* argument, that is, the type it is able to handle; so, the $\rightarrow$ constructor must be contravariant with respect to its first argument. In Felleisen and Flanagan's system, on the contrary, a function's domain represents its *actual* argument, that is, the values passed to this function during the program's execution; so, the dom destructor is covariant. Furthermore, the semantics of constraints allows applying this destructor to objects other than functions. Thus, the constraints' solvability no longer implies the program's correctness; an additional check shall be necessary to guarantee the latter. These two peculiarities of the dom destructor significantly modify the theory; in particular, in Felleisen and Flanagan's work, any constraint set admits a smallest solution, and entailment is decidable.

Müller, Niehren and Podelski [36] take interest in static analysis of the language Oz. The set of each program variable's possible values is approximated by a set of infinite terms. Once again, these sets are related by inclusion constraints, which have to be solved in order to determine if the program is well-typed. For the program to be accepted, there must not merely exist a solution, but one that associates a non-empty set to each variable. For this reason, Müller *et al.* interpret constraints in the model of *non-empty* sets of terms. This decision modifies the logical properties of the system—in particular, entailment—and yields greater simplicity. This system presents, in principle, common points with those mentioned previously; however, note that all constructors are covariant in it.

Lastly, even though we presented subtyping as a mechanism to allow typing object oriented languages, it is not the only possibility. Indeed, it is possible to handle objects in a classic language such as ML, provided the type system is enriched with *row variables* [43]. This solution is adopted by Objective Caml [44]. Besides, the two solutions are not mutually exclusive; in section 14.5, we discuss the interaction between row variables and subtyping.

Simplification

The details of the inference algorithm, for the system we're interested in, were initially given by Eifrig, Smith and Trifonov [15], approximately at the time this thesis work was started. However, the algorithm was still far from usable in practice. Indeed, the number of generated constraints is linear in the program size, since each function application creates one. (It is even exponential in theory, since the `let` construct allows duplicating the constraint set.) Since the closure algorithm works in cubic time, this makes type inference very slow. Furthermore, the inferred type schemes contain constraint sets, and are thus illegible. So, it becomes difficult for the user to use the inference engine as a tool to help understand his program.

Thus, it was necessary, to enhance efficiency and readability, to develop methods of *simplifying* constraints. A few methods already existed, but only for significantly simpler systems, such as Fuh and Mishra's [22], where it is possible to deal with constraints between variables and base types only. Thus, we had to design new simplification algorithms to fit our more complex system; and also, in order to make defining and proving these algorithms easier, to formulate our system in as simple and elegant a way as possible.

In its current form, the system is based on a *polymorphic type scheme comparison* relation, denoted by $\leq^\forall$. Its definition uses the semantics of constraints, i.e. it is expressed in terms of the solutions of the constraint set which appears in each type scheme. This relation appears in the subtyping rule, which could be written, in a slightly simplified way:

$$\frac{e : \sigma \qquad \sigma \leq^\forall \sigma'}{e : \sigma'}$$

So, it intuitively plays the same role as the subtyping relation explained above, but at the level of type schemes; that is, it accounts for the universal quantifiers which appear in σ and σ'. This relation constitutes the theoretical justification for all of our simplification methods: indeed, to "simplify" a type scheme σ, it suffices to replace it with a scheme σ' which is equivalent according to this relation, i.e. such that $\sigma =^\forall \sigma'$. Which criteria determine whether σ' is "simpler" than σ? If the goal is readability, the criterion shall probably be the size of σ''s textual representation. However, if the goal is efficiency, the most succinct representation is not necessarily easiest to deal with. So, one shall still attempt to minimize

σ''s size, but one shall preserve certain invariants which are useful for our algorithms. Thus, it is important to separate these two goals.

Hence, a simplification method is simply an algorithm capable of transforming a given type scheme into an equivalent type scheme. We shall present three methods, which naturally fit together into an efficient and powerful combination. *Canonization* makes sure that each variable admits at most one constructed bound, by introducing auxiliary variables. This property is indispensable to formulate the minimization algorithm; furthermore, canonization can be viewed as a way of improving sharing between terms. *Garbage collection* determines, through a fix-point computation, which constraints really influence the type scheme's denotation, and eliminates all others. Finally, *minimization* establishes an (almost) optimal sharing between the constraint graph's nodes, through a process similar to minimization of finite automata.

To operate at their best, these algorithms use two invariants. The *small terms invariant* imposes rather drastic restrictions on the form of the terms we deal with. Intuitively, it requires that each node, in each type term, be labeled with a type variable. Thus, sharing between nodes is achieved simply by identifying variables; a similar approach is used e.g. when dealing with unification problems, which are presented as multi-equations. The *mono-polarity invariant* requires that a single type variable not be used at the same time to represent a parameter and a result. It has various beneficial effects, in theory as well as in practice. To preserve each of these invariants, it suffices to slightly modify the type inference rules, which we shall do when the time comes.

Besides, we study the semantics of constraints, that is, how to decide various kinds of logical assertions related to solutions of constraint sets. During this study, we design three algorithms. The first one allows deciding whether a given conjunction of constraints admits a solution; it serves in determining, once the constraints have been obtained by analyzing the program, whether the latter is well-typed. The second one attempts to decide constraint entailment; the third one, which generalizes it, to decide the $\leq^{\forall}$ relation. However, the last two are incomplete. They are not used in practice; they serve, on paper, to prove the correctness of the above simplification methods. Thus, our simplification methods directly build a simplified scheme, equivalent to the original one. This is a vast improvement with respect to the heuristic approach, where a type scheme is produced in a "random" way, and one of these two algorithms is used to determine whether it is equivalent to the original scheme. Still, the last algorithm is used to compare inferred type schemes to user-supplied signatures, when separate compilation is desired.

It is interesting to notice that the part of the theory described above, that is, the whole constraint simplification system, stems entirely from the semantics of constraints, that is, from their interpretation in the model of ground types. In particular, it is independent of the chosen language and typing rules. Thus, it is immediately applicable to other languages, provided the semantics of constraints is unchanged.

So, the language-dependent aspects of our theory are simply the definition of the language itself, its semantics and its typing rules, together with their proof of correctness. In our case, the latter is rather simple, provided it is stated in an appropriate way—the power of the $\leq^{\forall}$ relation poses a few problems, and we shall be led to introduce an auxiliary set of typing rules to work around them.

Thus, the contribution of this thesis is mainly in the area of simplification. The initial algorithm [15] proved that inference was decidable, but was not directly useable in practice. Here, we propose an efficient and homogeneous system, made up of several complementary algorithms, working with data under an appropriate form. Historically, our research was led in parallel with that of Trifonov and Smith; so, they complement each other. Some ideas were suggested entirely independently, while in other cases, one team would use the other's results and improve them. This thesis presents the most advanced form of the system, while trying to indicate who the main concepts are due to. To sum up, in [42], we suggest *eliminating unreachable constraints*, and we introduce the entailment relation, together with its axiomatization. In [46], Trifonov and Smith reformulate the type system using λ-lifting, which allows them to generalize both notions, yielding garbage collection and the scheme subsumption relation. Furthermore, they present the canonization algorithm. Finally, in this thesis, we introduce the minimization algorithm, and deal with a number of points which significantly contribute to obtaining a complete and homogeneous simplification system. In particular, we enhance the canonization algorithm's efficiency by combining it with a garbage collection phase; we reformulate the inference rules so as to preserve the small terms invariant and the mono-polarity invariant, and we show that efficiency and readability are conflicting goals.

Outline

The first part of this thesis presents the type system. As explained above, the system is itself made up of two essentially independent components: a constraint language and a programming language, each defined by its syntax and semantics. The latter is accompanied by a set of typing rules, whose formulation makes use of the former.

The constraint language can be viewed as a logical system: it allows expressing formulas, which are to be interpreted in a model. The model is the set of *ground types*, that is, types without variables, made into a lattice by the subtyping relation. By introducing *types* with variables, then *constraints* between types and *type schemes*, we shall be able to express formulas concerning the existence of a solution, the entailment of constraints, or the comparison of two type schemes.

Once the constraint language is defined, we can present the *type system* proper. It comprises the definition of the language and of its semantics, the presentation of the typing rules and of the type inference algorithm, and the proof of correctness of typing with respect to the semantics.

The second part of this thesis contains the core of our theory. There, we first study the decidability of the three logical problems mentioned above: solvability of constraints, entailment of constraints and comparison of type schemes. Next, comes the presentation of the three main simplification methods, namely garbage collection, canonization and minimization. Finally, the small terms invariant and the mono-polarity invariant each receive a dedicated chapter. Each of these invariants is introduced at the most appropriate time, and is subsequently assumed to be in force, which contributes to simplify the theory. This helpful hypothesis could sometimes have been avoided, thus yielding slightly more general results; however, we preferred to favor the simplicity of the proofs.

Finally, the third part examines the problems under a more practical light. It first reviews a number of extensions of the system, left outside of the theoretical presentation for the sake of simplicity, but which pose no particular difficulty. It then describes the way our theoretical results lead to an implementation, and measures its performance. Lastly, a chapter is dedicated to studying several remaining problems, among which the treatment of imperative programs and a certain lack of efficiency, and to suggesting several solutions.

Part I

Presentation of the system

Chapter 1

Ground types

Among the various kinds of types introduced in this thesis, the notion of *ground type* is the simplest and most essential. Ground types are the regular trees built with the elementary constructors $\bot$, $\top$ and $\rightarrow$. The set of ground types is equipped with a partial ordering, called *subtyping*, which makes it a lattice.

In order to allow type inference, we shall later define the notion of *type*, where we introduce *type variables* as well as two additional constructors, $\sqcup$ and $\sqcap$. Types shall be manipulated formally through *constraint graphs*. In logical terms, constraint graphs are formulas, interpreted inside the model of ground types.

To deal with polymorphism, we shall next introduce *type schemes*. Roughly speaking, a type scheme can be interpreted as the set of its ground instances, that is, as a subset of the set of ground types. A type scheme is *more general* than another one if the former's interpretation, in the set of ground types, is a superset of the latter's. Thus, the lattice of ground types forms the logical basis of our theory.

In order to simplify our theory, we reduce the ground lattice as much as possible. This decreases the apparent complexity of statements and proofs, without affecting their principle. A much richer ground lattice is described, with fewer details, in chapter 14.

We first define the set of ground types (section 1.1), then the subtyping relation on this set (section 1.2).

1.1 Ground types

This section defines ground types as regular trees. Why choose regular trees, rather than finite trees? In ML, for instance, ground types are finite. Recursive unification constraints are rejected by the *occur check*, and recursive data types must be explicitly declared by the user. However, it would be easy (although not necessarily desirable, from a practical point of view) to extend ML's type theory to the case of regular ground types. So, for ML, both choices are possible. In our case, though, subtyping constraints replace unification constraints. So, it becomes possible for a recursive constraint to have a finite solution. Rejecting constraints with no finite solutions becomes more difficult and does not seem natural any more. Besides, from a practical point of view, being able to infer types for programs which deal with recursive data structures, without requiring explicit type declarations, seems interesting. For instance, it helps when doing type inference for object-oriented languages.

One can also ask the opposite question: why choose regular trees, rather than arbitrary infinite trees? In fact, it is possible to show that using the latter would not modify the basic logical properties of our system. That is, a constraint graph has a regular solution if and only if it has a (possibly irregular) solution. Better yet, entailment assertions have the same boolean value in both models. The proofs of these facts are beyond the scope of this dissertation; they are, however, rather simple and use topological properties of trees similar to those developed in section 2.6.1. In conclusion, it seems that we lose no power by restricting the model to regular trees, while gaining simplicity.

Definition 1.1 *Let the* ground signature Σ_g *consist of* $\bot$ *and* $\top$ *with arity* 0 *and* $\rightarrow$ *with arity* 2. *A* path p *is a finite string of* 0*'s and* 1*'s, i.e. an element of* $\{0,1\}^*$. ϵ *denotes the empty path. The length of a path* p *is denoted by* $|\,p\,|$. *Its* parity $\pi(p)$ *is the number of* 0*'s it contains, taken modulo* 2. *A* ground tree τ *is a partial function from paths into* Σ_g, *whose domain is non-empty and prefix-closed, and such that* $\tau(p0)$ *and* $\tau(p1)$ *are defined iff* $\tau(p) = \rightarrow$. *We denote the set of ground trees by* $\mathbb{T}_\infty$. *Given* $p \in \mathrm{dom}(\tau)$, *the* subtree of τ rooted at p *is the tree* $q \mapsto \tau(pq)$. *A tree is* finite *iff its domain is finite. A tree is* regular *iff it has a finite number of subtrees. A* ground type *is a regular ground tree. We denote the set of ground types by* $\mathbb{T}$.

Regular trees can be finitely represented using term automata. This equivalence shall be used in the formal definition of the $\sqcup$ and $\sqcap$ operators on ground types.

Definition 1.2 *A* term automaton *over* Σ_g *is a tuple* $\mathcal{A} = (Q, q_0, \delta, l)$ *where:*

- Q *is a finite set of* states,
- $q_0 \in Q$ *is the* start state,
- $\delta : Q \times \{0,1\} \rightarrow Q$ *is a (partial)* transition function,
- $l : Q \rightarrow \Sigma_g$ *is a* labeling function,

such that for any state $q \in Q$ *and for any* $i \in \{0,1\}$, $\delta(q,i)$ *is defined iff* $l(q) = \rightarrow$.

δ *can be naturally extended to a partial function* $\hat{\delta} : Q \times \{0,1\}^* \rightarrow Q$. *The term* $\tau_{\mathcal{A}}$ *associated to the automaton* $\mathcal{A}$ *is* $p \mapsto l(\hat{\delta}(q_0, p))$.

Proposition 1.1 *A tree* τ *is regular iff there exists an automaton* $\mathcal{A}$ *such that* $\tau = \tau_{\mathcal{A}}$.

Proof. By establishing an isomorphism between the states of $\mathcal{A}$ and the subtrees of $\tau_{\mathcal{A}}$. □

Considering ground types as trees, or automata, is cumbersome, so we introduce a few usual notations for types.

Definition 1.3 $\bot$ *(resp.* $\top$*) stands for the tree* τ *such that* $\mathrm{dom}(\tau) = \{\epsilon\}$ *and* $\tau(\epsilon) = \bot$ *(resp.* $\top$*). If* τ_0 *and* τ_1 *are trees,* $\tau_0 \rightarrow \tau_1$ *stands for the tree* τ *defined by* $\tau(\epsilon) = \rightarrow$, $\tau(0p) = \tau_0(p)$ *and* $\tau(1p) = \tau_1(p)$.

1.2 Ground subtyping

We have now defined the set of ground types. We shall now equip it with a partial order, called *subtyping*, and show that it is a lattice. Technically, there are several equivalent ways to define subtyping on regular trees: by finite approximations, by co-induction or, as done here, by quantifying over paths. Each of these choices leads to the same order, which is characterized by proposition 1.2. Similarly, the technical definition of $\sqcup$ and $\sqcap$ is of little interest; their behavior is characterized by proposition 1.5.

Let us first define subtyping.

Definition 1.4 *Define an ordering on* Σ_g *by* $\bot \leq_g \rightarrow \leq_g \top$*. Equipped with this ordering,* Σ_g *becomes a lattice and we denote its least upper bound and greatest lower bound operators by* $\sqcup_g$ *and* $\sqcap_g$*, respectively. We further define* $\leq_g^0$ *as* $\leq_g$ *and* $\leq_g^1$ *as its reverse.*

Given two ground types τ *and* τ'*, we say that* τ *is a* subtype *of* τ'*, and we write* $\tau \leq \tau'$*, iff*

$$\forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') \quad \tau(p) \leq_g^{\pi(p)} \tau'(p)$$

As mentioned above, a better intuition is given by the following property. It states that $\bot$ is the least element, $\top$ is the greatest element, and subtyping propagates structurally along $\rightarrow$. Of course, $\rightarrow$ is contravariant in its domain and covariant in its codomain.

Proposition 1.2 $\tau \leq \tau'$ *holds iff at least one of the following is true:*

- $\tau = \bot$*;*
- $\tau' = \top$*;*
- $\exists \tau_0 \tau_1 \tau_0' \tau_1' \quad \tau = \tau_0 \rightarrow \tau_1$*,* $\tau' = \tau_0' \rightarrow \tau_1'$*,* $\tau_0' \leq \tau_0$ *and* $\tau_1 \leq \tau_1'$*.*

Proof. We shall first assume that $\tau = \tau_0 \rightarrow \tau_1$ and $\tau' = \tau_0' \rightarrow \tau_1'$. Then $\tau \leq \tau'$ is, by definition, equivalent to

$$\forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') \quad \tau(p) \leq_g^{\pi(p)} \tau'(p)$$

This can be written

$$\forall i \in \{0, 1\} \quad \forall q \in \mathrm{dom}(\tau_i) \cap \mathrm{dom}(\tau_i') \quad \tau(iq) \leq_g^{\pi(iq)} \tau'(iq)$$

Since $\tau(iq) = \tau_i(q)$ and $\tau'(iq) = \tau_i'(q)$, this is itself equivalent to $\tau_0' \leq \tau_0 \wedge \tau_1 \leq \tau_1'$ (set i to 0 and 1 successively).

Back to the general case, consider given τ and τ'. There are 9 possible values for the pair $(\tau(\epsilon), \tau'(\epsilon))$. Out of these 9 cases, 4 are cases where $\tau \leq \tau'$ does not hold, 4 are cases where $\tau = \bot$ or $\tau' = \top$, and the last one we just treated. □

Let us now give an alternate characterization of subtyping, based on a sequence of finite approximations, which shall be useful in several proofs.

Definition 1.5 *Let* $\leq_0$ *be the binary relation which is uniformly true on ground types. For* $k \geq 1$*, define* $\leq_k$ *by*

- *If* $\tau = \tau_0 \rightarrow \tau_1$ *and* $\tau' = \tau_0' \rightarrow \tau_1'$*, then* $\tau \leq_k \tau'$ *iff* $\tau_0' \leq_{k-1} \tau_0$ *and* $\tau_1 \leq_{k-1} \tau_1'$*.*
- *Otherwise,* $\tau \leq_k \tau'$ *iff* $\tau(\epsilon) \leq_g \tau'(\epsilon)$*.*

It is easy to verify that $(\leq_k)_{k \geq 0}$ is a decreasing sequence of pre-orders. It turns out that its intersection is precisely $\leq$, as stated by the following proposition.

Proposition 1.3 $\tau \leq \tau'$ *is equivalent to*

$$\forall k \geq 0 \quad \tau \leq_k \tau'$$

For this reason, we shall also denote $\leq$ *by* $\leq_\infty$*.*

Proof. We shall prove (by induction) that for all $k \geq 0$, $\tau \leq_k \tau'$ is equivalent to

$$\forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') \quad |\,p\,| < k \Rightarrow \tau(p) \leq_g^{\pi(p)} \tau'(p)$$

where $|\,p\,|$ is the length of the path p. This is immediate for $k = 0$. Assume it holds at rank $k-1$ where $k \geq 1$. There are two cases to consider.

First, assume $\tau = \tau_0 \to \tau_1$ and $\tau' = \tau'_0 \to \tau'_1$. Then $\tau \leq_k \tau'$ is equivalent to $\tau'_0 \leq_{k-1} \tau_0$ and $\tau_1 \leq_{k-1} \tau'_1$. By induction hypothesis, the former is equivalent to

$$\forall q \in \mathrm{dom}(\tau'_0) \cap \mathrm{dom}(\tau_0) \quad |\,q\,| < k-1 \Rightarrow \tau'_0(q) \leq_g^{\pi(q)} \tau_0(q)$$

The latter assertion can be rewritten in a similar way, and we can then combine them again to form

$$\forall i \in \{0,1\} \quad \forall q \in \mathrm{dom}(\tau_i) \cap \mathrm{dom}(\tau'_i) \quad |\,iq\,| < k \Rightarrow \tau(iq) \leq_g^{\pi(iq)} \tau'(iq)$$

which is equivalent to the goal.

Otherwise, $\tau \leq_k \tau'$ is, by definition, equivalent to $\tau(\epsilon) \leq_g \tau'(\epsilon)$. Besides, at least one of $\mathrm{dom}(\tau)$ and $\mathrm{dom}(\tau')$ is equal to $\{\epsilon\}$, so the goal is also equivalent to $\tau(\epsilon) \leq_g \tau'(\epsilon)$.

From this result, it is obvious that the intersection of $(\leq_k)_{k\geq 0}$ coincides with the subtyping relation. □

A fundamental property of the subtyping relation is the following:

Proposition 1.4 *The set of ground types $\mathbb{T}$, equipped with the subtyping relation, is a lattice. We denote its least upper bound and greatest lower bound operators by $\sqcup$ and $\sqcap$, respectively.*

Proof. Let us first show that $\leq$ is an order, which we haven't done yet. $\leq$ is the intersection of $(\leq_k)_{k\geq 0}$ which is a sequence of pre-orders, so it is a pre-order. To show that it is antisymmetric, assume $\tau \leq \tau' \leq \tau$. This translates to $\forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') \quad \tau(p) = \tau'(p)$, i.e. τ and τ' agree on the intersection of their domains. Let us show that their domains coincide. Assume that $\mathrm{dom}(\tau) \setminus \mathrm{dom}(\tau')$ is non-empty (the other case is symmetric). We can pick a path p of minimal length in it. Since $\epsilon \in \mathrm{dom}(\tau')$, we have $|\,p\,| > 0$ and we can write $p = qi$ where $i \in \{0,1\}$. Because $\mathrm{dom}(\tau)$ is prefix-closed, q must belong to it; and since p was chosen minimal, necessarily $q \in \mathrm{dom}(\tau')$. It follows that $\tau(q) = \tau'(q)$. Now, recall that any tree σ must satisfy the following property: for any path r, $\sigma(r0)$ and $\sigma(r1)$ are defined iff $\sigma(r) = \to$. Since $\tau(qi)$ is defined, it follows that $\to = \tau(q) = \tau'(q)$ so $\tau'(qi) = \tau'(p)$ must be defined. We have a contradiction with $p \notin \mathrm{dom}(\tau')$. Thus, $\mathrm{dom}(\tau) = \mathrm{dom}(\tau')$, and $\tau = \tau'$. $\leq$ is antisymmetric, and it is a partial order.

We shall now give explicit definitions of $\sqcup$ and $\sqcap$ and subsequently prove that they are least upper bound and greatest lower bound operators. We would like to define these operators using a few intuitive equations, but these equations would be recursive and thus wouldn't constitute a proper definition. Another idea that comes to mind is to define them using a sequence of finite approximations, but then we would have to prove that the limit of the sequence is a regular tree. So, the simplest way to define them is, given two regular trees τ_1 and τ_2, to construct $\tau_1 \sqcup \tau_2$ and $\tau_1 \sqcap \tau_2$ as term automata.

For $j \in \{1,2\}$, let τ_j be the regular tree given by the term automaton $\mathcal{A}_j = (Q_j, q_j^0, \delta_j, l_j)$. One can assume—by adding them if necessary—that $\mathcal{A}_j$ has a state whose label is $\top$ and one whose label is $\bot$; we refer to them as $\top$ and $\bot$, respectively, by abuse of language. We introduce some notations: $\sqcup^n$ shall stand for $\sqcup$ whenever n is an even integer and for $\sqcap$ whenever n is an odd integer. A similar convention is adopted for $\sqcup_g^n$. Define $Q = \{\sqcup, \sqcap\} \times Q_1 \times Q_2$. Consider a state $q \in Q$ of the form $(\sqcup^n, q_1, q_2)$. Its label $l(q)$ is defined as $l_1(q_1) \sqcup_g^n l_2(q_2)$. The transitions out of state q are defined by:

- If $l(q) = \rightarrow$, then $\delta(q,i) = (\sqcup^{n+1+i}, \delta_1(q_1,i), \delta_2(q_2,i))$ for $i \in \{0,1\}$. Here, $\delta_j(q_j,i)$ is undefined whenever $l_j(q_j) \neq \rightarrow$; it shall then be read as $\bot$ when $n+1+i$ is even and $\top$ when it is odd.

- Otherwise, $\delta(q,i)$ is undefined for $i \in \{0,1\}$.

For $s \in \{\sqcup, \sqcap\}$, define $q_0^s = (s, q_1^0, q_2^0)$. We can finally define $\tau_0 \, s \, \tau_1$ as the regular tree associated to the automaton $\mathcal{A}^s = (Q, q_0^s, \delta, l)$.

We must now verify that $\sqcup$ and $\sqcap$ are indeed least upper bound and greatest lower bound operators for $\leq$.

First, let us show that $\tau_1 \sqcup \tau_2$ is an upper bound for τ_j ($j \in \{1,2\}$). Let $p \in \mathrm{dom}(\tau_1 \sqcup \tau_2) \cap \mathrm{dom}(\tau_j)$. Because $p \in \mathrm{dom}(\tau_1 \sqcup \tau_2)$, we have $l(q) \notin \{\bot, \top\}$ for any state $q = \delta(q_0^\sqcup, r)$ associated to a strict prefix r of p. By looking up the definition of $l(q)$ for such a q, and by remembering that $p \in \mathrm{dom}(\tau_j)$, we obtain that the $(1+j)^{\text{th}}$ component of the state $\delta(q_0^\sqcup, p)$ is $\hat{\delta}_j(q_j^0, p)$. According to the definition of the labeling function l, this state's label, which is $(\tau_1 \sqcup \tau_2)(p)$, is greater than $l_j(\hat{\delta}_j(q_j^0, p))$, which is $\tau_j(p)$. We have thus shown that

$$\forall p \in \mathrm{dom}(\tau_1 \sqcup \tau_2) \cap \mathrm{dom}(\tau_j) \quad \tau_j(p) \leq_g^{\pi(p)} (\tau_1 \sqcup \tau_2)(p)$$

which is, by definition, equivalent to $\tau_j \leq \tau_1 \sqcup \tau_2$.

Reciprocally, let τ be an upper bound of τ_1 and τ_2. We have

$$\forall j \in \{1,2\} \quad \forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau_j) \quad \tau_j(p) \leq_g^{\pi(p)} \tau(p)$$

Now, consider a path $p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau_1 \sqcup \tau_2)$. Take $j \in \{1,2\}$. If $p \in \mathrm{dom}(\tau_j)$, then $\tau_j(p) \leq_g^{\pi(p)} \tau(p)$ according to the above. Otherwise, it is easy to verify that the label of the $(1+j)^{\text{th}}$ component of the state $\hat{\delta}(q_0^\sqcup, p)$ is the least element of the ordering $\leq_g^{\pi(p)}$. In both cases, the label of this $(1+j)^{\text{th}}$ component is smaller (for $\leq_g^{\pi(p)}$) than $\tau(p)$. Since this holds for all $j \in \{1,2\}$, the label $l(\hat{\delta}(q_0^\sqcup, p) = (\tau_1 \sqcup \tau_2)(p)$, which is the least upper bound of these labels, is also less than $\tau(p)$. Since this holds for all $p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau_1 \sqcup \tau_2)$, we have shown that $\tau_1 \sqcup \tau_2 \leq \tau$.

The two previous paragraphs, taken together, prove that $\sqcup$ is the least upper bound operation for the ordering $\leq$. Similarly, one proves that $\sqcap$ is the greatest lower bound operation. $\square$

As in any lattice, $\sqcup$ and $\sqcap$ are associative and commutative. In addition, we give a few equations which fully characterize them. The first four equations below define the behavior of $\top$ and $\bot$, while the last two state that $\sqcup$ and $\sqcap$ are distributive over $\rightarrow$.

Proposition 1.5 *The following are identities:*

$$\begin{gathered}
\bot \sqcup \tau = \tau \qquad \bot \sqcap \tau = \bot \\
\top \sqcup \tau = \top \qquad \top \sqcap \tau = \tau \\
(\tau_1 \rightarrow \tau_2) \sqcup (\tau_1' \rightarrow \tau_2') = (\tau_1 \sqcap \tau_1') \rightarrow (\tau_2 \sqcup \tau_2') \\
(\tau_1 \rightarrow \tau_2) \sqcap (\tau_1' \rightarrow \tau_2') = (\tau_1 \sqcup \tau_1') \rightarrow (\tau_2 \sqcap \tau_2')
\end{gathered}$$

Proof. Straightforward by using the definition of $\sqcup$ and $\sqcap$ as products of term automata (see the proof of proposition 1.4). $\square$

Finally, the following proposition, of mostly technical interest, states that $\sqcup$ and $\sqcap$ also have a nice behavior with respect to $\leq_k$.

Proposition 1.6 *Let $k \in \mathbb{N}^+$. For any ground types τ_0, τ_1 and τ, we have*

$$\begin{aligned} \tau_0 \leq_k \tau \wedge \tau_1 \leq_k \tau &\iff \tau_0 \sqcup \tau_1 \leq_k \tau \\ \tau \leq_k \tau_0 \wedge \tau \leq_k \tau_1 &\iff \tau \leq_k \tau_0 \sqcap \tau_1 \end{aligned}$$

Proof. Consider the first line (the second one is symmetric). As for the direct implication, we have already proved the case $k = \infty$ as part of proposition 1.4; for an arbitrary k, the proof is essentially identical (it suffices to consider only paths of length less than k). As for its reciprocal, the result is immediate since $\leq$ is finer than $\leq_k$. $\square$

Chapter 2

Types

Having introduced ground types, which constitute the logical model of our system, we must now define types, which are the basic components of our logical formulas. The structure of types is similar to that of ground types, with a few differences. First, a formula must be finite; so, types are *finite terms*, whereas ground types are potentially infinite trees. Next, interesting formulas contain variables; so, we introduce *type variables*. Finally, we allow types to contain symbolic expressions built with the $\sqcup$ and $\sqcap$ constructors.

We first comment on the last of these decisions (section 2.1). Then, we move on to the central part of this chapter, that is, the precise definition of the notion of type sketched above (section 2.2). Some technical definitions and developments follow (sections 2.3 to 2.6).

2.1 About the $\sqcup$ and $\sqcap$ constructors

The main reason why we introduce these constructors is to allow encoding a conjunction of constraints, of the form $\bigwedge_{i\in I}(\tau_i \leq \alpha)$, into a single constraint $(\sqcup_{i\in I}\tau_i) \leq \alpha$. (The case of the $\sqcap$ constructor is symmetric.) Thus, in a constraint graph (see definition 3.5), each variable shall have exactly one lower bound and one upper bound. In the absence of these constructors, constraint graphs would have to keep track of a set of bounds for each variable.

In addition to introducing $\sqcup$ and $\sqcap$ constructors into the type language, we define a set of rewriting rules, which allow computing the *normal form* of a type (see definition 2.1). For instance, the type

$$(\alpha \rightarrow \beta) \sqcup (\gamma \rightarrow \bot)$$

shall be immediately reduced to $(\alpha \sqcap \gamma) \rightarrow \beta$.

Note that $\sqcup$ and $\sqcap$ shall only be used in a restricted way. Indeed, they only serve to encode conjunctions of constraints. Thus, in a constraint graph, any variable α has a lower bound of the form $\sqcup_{i\in I}\tau_i$, where the τ_i's contain no occurrences of $\sqcup$ or $\sqcap$. This type can be written in several different ways, thanks to the aforementioned rewriting rules. However, all of them have a property in common: the $\sqcup$ constructor only appears in *positive* positions, and the $\sqcap$ constructor in *negative* ones. The case of α's upper bound is symmetric.

The *closure* computation (see definition 7.1) combines the various constraints generated by analyzing the program. During this phase, the bounds of each variable are computed and reduced to normal form, as explained above. Next, comes the *canonization* phase (see chapter 11), which eliminates all occurrences of the $\sqcup$ and $\sqcap$ constructors.

Combining and normalizing bounds enables better sharing, as well as a more compact representation of constraints. For instance, according to the computational rules associated to the $\sqcup$ and $\sqcap$ constructors, the term $(\alpha_0 \rightarrow \alpha_1)\sqcup(\beta_0 \rightarrow \beta_1)$ reduces to $(\alpha_0\sqcap\beta_0) \rightarrow (\alpha_1\sqcup\beta_1)$, where the $\to$ constructor is shared. Better yet, this computation can cause some terms to disappear; for instance, $(\alpha_0 \to \alpha_1) \sqcup \top$ is rewritten to $\top$.

If we had chosen to keep track of each bound individually, inside a set, then this normal form computation would no longer occur during the closure phase; it would only be performed as part of canonization. Hence, introducing the $\sqcup$ and $\sqcap$ constructors, together with their rewriting rules, amounts to performing part of the canonization work, incrementally, during the closure phase. In practice, this allows manipulating more compact graphs. Regardless of this decision, once canonization is over, each variable has exactly one lower bound and one upper bound, both free of any occurrences of $\sqcup$ or $\sqcap$. Thus, only the "front-end" of the system is affected by this decision.

Example. Imagine that a variable α has lower bounds $\beta_1 \to \beta_2$ and $\gamma_1 \to \gamma_2$. As explained, we combine these two types into a single bound, $(\beta_1 \sqcap \gamma_1) \to (\beta_2 \sqcup \gamma_2)$. Now, suppose that, during the program's analysis, a third lower bound for α appears, namely $\beta_1 \to \gamma_2$. We combine this new bound with the previous one by computing

$$((\beta_1 \sqcap \gamma_1) \to (\beta_2 \sqcup \gamma_2)) \sqcup (\beta_1 \to \gamma_2)$$

The result of the computation is precisely

$$(\beta_1 \sqcap \gamma_1) \to (\beta_2 \sqcup \gamma_2)$$

which means that the new bound actually contains no new information.

If we had chosen to represent α's previous lower bound by the set $\{\beta_1 \to \beta_2, \gamma_1 \to \gamma_2\}$, then the new bound $\beta_1 \to \gamma_2$ would not have been a member of this set, and it would have been added to the set of α's lower bounds. Thus, the sets of bounds could grow in a redundant way, causing a loss of efficiency in terms of space and time.

From a theoretical point of view, the presentation of closure is made slightly more complex by this decision, as expected, while the description of the canonization algorithm is simplified. So, the influence of this choice on the theory is rather limited; our decision was made primarily in light of its practical advantages.

Trifonov and Smith [46] do not allow $\sqcup$ and $\sqcap$ constructors to appear in types. As explained above, their system is nonetheless equivalent in power. Aiken, Wimmers and Lakshman [6, 7] propose a system where the $\cup$ and $\cap$ constructors are used in a less restrictive way. (These constructors represent set-theoretic union and intersection, not the type lattice operations, but they behave, at first sight, in a comparable way.) An expression $\tau_1 \cup \tau_2$ is allowed to appear in negative position, provided it is a disjoint union (i.e. $\tau_1 \cap \tau_2 = \varnothing$ for all assignments of the free variables). An expression $\tau_1 \cap \tau_2$ is allowed to appear in positive position, provided τ_2 has no free variables and is *upward closed*. These possibilities are not available in our system; however, note that part of their power can be recovered using variant types, to be introduced in chapter 14. For instance, in [6], when a union $\tau_1 \cup \tau_2$ appears in negative position, one typically has $\tau_1 = a(\tau_1')$ and $\tau_2 = b(\tau_2')$, where a and b are distinct data constructors, since the union must be disjoint. Then, the type $\tau_1 \cup \tau_2$ corresponds, in our system, to the type `[ a:` τ_1' `| b:` τ_2' `]`, using the notation of section 14.4. When an intersection $\tau_1 \cap \tau_2$ appears in positive position, one typically has $\tau_2 = \neg a(1)$, where a is a data constructor and 1 is equivalent to our type $\top$. Then, the type $\tau_1 \cap \tau_2$ can be written, in our system, `[ a: Abs;` ρ `]`, provided we add the constraint $\tau_1 \leq$ `[ a: Pre` $\top$`;` ρ `]`, which allows "subtracting" the field a from τ_1 and representing all other fields via the row variable ρ. (Variant types with row variables are presented in section 14.5.2.) Thus, our system is rather expressive. However, some of the possibilities offered by [7], such as the use of union and *conditional* types to obtain a very fine typing of pattern matchings, can not currently be expressed in our system.

2.2 Types

Let us now move on to the definition of types. It is done in two stages. First, we introduce terms called *pretypes*. Then, to account for the properties of the $\sqcup$ and $\sqcap$ constructors, the set of pretypes is viewed modulo a certain congruence relation. This gives birth to *types*.

We are led to introduce various kinds of (pre)types, which differ only by the positions in which the $\sqcup$ and $\sqcap$ constructors are allowed to appear. *Simple* types are not allowed to contain these constructors; once the canonization phase is over, we shall work solely with these types. The lower and upper bound of a type variable in a constraint graph shall be respectively a *pos-type* and a *neg-type*. Finally, bi-pretypes, which place no restrictions on the use of $\sqcup$ and $\sqcap$, shall be used only in section 2.6. We shall not attempt to view them modulo a congruence, as mentioned above.

Definition 2.1 *Let $\mathcal{V}$ be a denumerable set of* type variables*, denoted by α, β, etc. The set of* simple pretypes*, denoted by $p\mathcal{T}$, is made up of the terms defined by the following grammar:*

$$\tau ::= \alpha \mid \bot \mid \top \mid \tau \rightarrow \tau$$

The set of pos-pretypes*, denoted by $p\mathcal{T}^+$, is defined by*

$$\tau^+ ::= \alpha \mid \bot \mid \top \mid \tau^- \rightarrow \tau^+ \mid \sqcup\{\tau^+, \ldots, \tau^+\}$$

The set of neg-pretypes*, denoted by $p\mathcal{T}^-$, is defined by*

$$\tau^- ::= \alpha \mid \bot \mid \top \mid \tau^+ \rightarrow \tau^- \mid \sqcap\{\tau^-, \ldots, \tau^-\}$$

Lastly, the set of bi-pretypes*, denoted by $p\mathcal{T}^\pm$, is defined by*

$$\tau^\pm ::= \alpha \mid \bot \mid \top \mid \tau^\pm \rightarrow \tau^\pm \mid \sqcup\{\tau^\pm, \ldots, \tau^\pm\} \mid \sqcap\{\tau^\pm, \ldots, \tau^\pm\}$$

We shall simply refer to pretypes *when the distinction is irrelevant or clear from the context.*

Rather than making $\sqcup$ and $\sqcap$ binary syntactic constructs, we make them unary, with a set of terms as argument. This helps deal with associativity and commutativity problems. The notation $\tau \sqcup \tau'$ shall be syntactic sugar for $\sqcup\{\tau, \tau'\}$.

In the language of pretypes, there are numerous terms which, intuitively, describe the same type. For instance, the term

$$\alpha \sqcup (\sqcup\{\alpha, \bot \rightarrow \beta, \gamma \rightarrow \beta\})$$

can (still informally) be simplified to $\alpha \sqcup (\bot \rightarrow \beta)$. These terms are two different pretypes; yet we wish to identify them. To this end, we shall now define a congruence relation on pretypes, and use it to define types as congruence classes.

Definition 2.2 *Let $\equiv$ be the congruence generated by the following identities:*

$$\begin{aligned}
\sqcup\varnothing &\equiv \bot \\
\sqcup\{\tau\} &\equiv \tau \\
\sqcup(\{\sqcup S\} \cup S') &\equiv \sqcup(S \cup S') \\
\sqcup(\{\bot\} \cup S) &\equiv \sqcup S \\
\sqcup(\{\top\} \cup S) &\equiv \top \\
\sqcup(\{\tau_0 \rightarrow \tau_1, \tau'_0 \rightarrow \tau'_1\} \cup S) &\equiv \sqcup(\{(\tau_0 \sqcap \tau'_0) \rightarrow (\tau_1 \sqcup \tau'_1)\} \cup S)
\end{aligned}$$

(plus 6 symmetric identities concerning $\sqcap$).

The first two identities cover the special cases where the operator has zero or one argument. The third one deals with associativity. The last three are computational rules coming from proposition 1.5.

Definition 2.3 *The set of* simple types*, denoted by $\mathcal{T}$, is the quotient $p\mathcal{T}/\equiv$. The sets of* pos-types *and* neg-types*, denoted respectively by $\mathcal{T}^+$ and $\mathcal{T}^-$, are defined similarly.*

We shall simply refer to types *when the distinction is irrelevant or clear from the context.*

Thus, formally speaking, a type is a congruence class of pretypes; it may be denoted by any of its elements. However, in order to be able to reason about types, we shall isolate, inside each class, a particular element called a *normal form*.

Proposition 2.1 *A rewriting system is defined by orienting each of the equations in definition 2.2 from left to right. Then, this system is confluent and Noetherian. That is, any type has a normal form.*

Proof. To prove that this rewriting system is locally confluent, we use the automatic tool RRL [31]. In addition to our term syntax, we have to define a few notions of set theory. Here is our RRL definition file:

```
;; ------------------------------------------------------------------------
;; An RRL source file used to prove that types have a normal form with
;; respect to the union/intersection rewriting rules.

;; ------------------------------------------------------------------------
;; A tiny chunk of set theory.
;; One also needs to declare that cup (the set union operator) is
;; associative and commutative using RRL's interactive interface.

[empty : set]
[s : term -> set]
[cup : set, set -> set]

cup(X, empty) := X

;; ------------------------------------------------------------------------
;; Next, our rewriting rules.

[glb : set -> term]
[lub : set -> term]
[arr : term, term -> term]
[bot : term]
[top : term]

lub(empty) := bot
lub(s(X)) := X
lub(cup(s(lub(X)),Y)) := lub(cup(X,Y))
lub(cup(s(bot), X)) := lub(X)
lub(cup(s(top), X)) := top
lub(cup(cup(s(arr(X0, X1)), s(arr(Y0, Y1))), Z)) :=
  lub(cup(s(arr(glb(cup(s(X0),s(Y0))),lub(cup(s(X1),s(Y1))))),Z))

glb(empty) := top
glb(s(X)) := X
glb(cup(s(glb(X)),Y)) := glb(cup(X, Y))
glb(cup(s(top), X)) := glb(X)
```

```
glb(cup(s(bot), X)) := bot
glb(cup(cup(s(arr(X0, X1)), s(arr(Y0, Y1))), Z)) :=
  glb(cup(s(arr(lub(cup(s(X0),s(Y0))),glb(cup(s(X1),s(Y1))))),Z))
```

When fed this input, RRL asks the user to manually orient 4 equations, and then succeeds, which proves that the system is locally confluent. To prove that no infinite rewriting sequence exists, we define the size of a syntactic type by

$$\begin{aligned}
\mathrm{size}(\bot) &= 1\\
\mathrm{size}(\top) &= 1\\
\mathrm{size}(\alpha) &= 1\\
\mathrm{size}(\tau_0 \rightarrow \tau_1) &= 5 + \mathrm{size}(\tau_0) + \mathrm{size}(\tau_1)\\
\mathrm{size}(\sqcup S) &= 2 + \Sigma_{\tau \in S}\, \mathrm{size}(\tau)\\
\mathrm{size}(\sqcap S) &= 2 + \Sigma_{\tau \in S}\, \mathrm{size}(\tau)
\end{aligned}$$

It is then easy to verify that each of the rewriting rules causes the size of the type to decrease strictly. Hence, the rewriting system is Noetherian; since it is locally confluent, it is also confluent. (Thanks to Cesar Muñoz for suggesting—and helping with—the use of RRL.) □

Normal forms can be characterized as follows.

Proposition 2.2 *A pretype τ is a normal form iff every occurrence in τ of the $\sqcup$ and $\sqcap$ constructors satisfies the following conditions:*

- *the constructor has $n \geq 2$ arguments;*
- *at least $n - 1$ are type variables;*
- *if one of the arguments isn't a type variable, then it is a type term whose head constructor is not among $\bot$, $\top$, $\sqcup$ and $\sqcap$.*

Proof. It is easy to see that if one of the conditions above is violated, then one of the rewriting rules applies; and conversely, that if all conditions above are satisfied, then none of the rules applies. □

From now on, when we reason on the form of a type (that is, on its structure as a term), its normal form shall be implicitly meant. Thus, functions defined on pretypes by structural analysis extend straightforwardly to types.

2.3 Auxiliary definitions

The notion of free variables of a type is defined in a classic way. However, we shall sometimes distinguish those that appear at a positive position from those that occur at a negative one.

Definition 2.4 *The set of* positive *(resp.* negative*)* free type variables *of a pretype τ, denoted by* $\mathrm{fv}^+(\tau)$ *(resp.* $\mathrm{fv}^-(\tau)$*), is defined by*

$$\begin{aligned}
\mathrm{fv}^+(\alpha) &= \{\alpha\} & \mathrm{fv}^-(\alpha) &= \varnothing\\
\mathrm{fv}^+(\bot) &= \varnothing & \mathrm{fv}^-(\bot) &= \varnothing\\
\mathrm{fv}^+(\top) &= \varnothing & \mathrm{fv}^-(\top) &= \varnothing\\
\mathrm{fv}^+(\tau_0 \rightarrow \tau_1) &= \mathrm{fv}^-(\tau_0) \cup \mathrm{fv}^+(\tau_1) & \mathrm{fv}^-(\tau_0 \rightarrow \tau_1) &= \mathrm{fv}^+(\tau_0) \cup \mathrm{fv}^-(\tau_1)\\
\mathrm{fv}^+(\sqcup S) &= \cup_{\tau \in S}\, \mathrm{fv}^+(\tau) & \mathrm{fv}^-(\sqcup S) &= \cup_{\tau \in S}\, \mathrm{fv}^-(\tau)\\
\mathrm{fv}^+(\sqcap S) &= \cup_{\tau \in S}\, \mathrm{fv}^+(\tau) & \mathrm{fv}^-(\sqcap S) &= \cup_{\tau \in S}\, \mathrm{fv}^-(\tau)
\end{aligned}$$

The set of free type variables *of* τ*, denoted by* $\mathrm{fv}(\tau)$*, is defined by*

$$\mathrm{fv}(\tau) = \mathrm{fv}^+(\tau) \cup \mathrm{fv}^-(\tau)$$

The height of a term is defined in a usual way. Note that the $\sqcup$ and $\sqcap$ constructors have no intrinsic height, since in fact they stand for height-preserving operations on ground terms.

Definition 2.5 *The* height *of a pretype* τ*, denoted by* $\mathrm{height}(\tau)$*, is defined by*

$$\begin{aligned}
\mathrm{height}(\alpha) &= 0 \\
\mathrm{height}(\bot) &= 0 \\
\mathrm{height}(\top) &= 0 \\
\mathrm{height}(\tau_0 \rightarrow \tau_1) &= 1 + \max\{\mathrm{height}(\tau_0), \mathrm{height}(\tau_1)\} \\
\mathrm{height}(\sqcup S) &= \max\{\mathrm{height}(\tau)\,;\, \tau \in S\} \\
\mathrm{height}(\sqcap S) &= \max\{\mathrm{height}(\tau)\,;\, \tau \in S\}
\end{aligned}$$

The following definition shall be used intensively in section 2.6, as well as in certain statements concerning entailment.

Definition 2.6 *The* depth of the nearest variable *in a pretype* τ*, denoted by* $\mathrm{dnv}(\tau)$*, is defined by*

$$\begin{aligned}
\mathrm{dnv}(\alpha) &= 0 \\
\mathrm{dnv}(\bot) &= \infty \\
\mathrm{dnv}(\top) &= \infty \\
\mathrm{dnv}(\tau_0 \rightarrow \tau_1) &= 1 + \min\{\mathrm{dnv}(\tau_0), \mathrm{dnv}(\tau_1)\} \\
\mathrm{dnv}(\sqcup S) &= \min\{\mathrm{dnv}(\tau)\,;\, \tau \in S\} \\
\mathrm{dnv}(\sqcap S) &= \min\{\mathrm{dnv}(\tau)\,;\, \tau \in S\}
\end{aligned}$$

The next definition recalls the notion of head constructor. If the head constructor of a type τ is neither $\sqcup$ or $\sqcap$, nor a variable, then any ground substitution maps τ to a ground type with the same head constructor; τ is then said to be *constructed.*

Definition 2.7 *The* head constructor *of a pretype* τ *is* $\bot$*,* $\rightarrow$ *or* $\top$*, when* τ *is of the form* $\bot$*,* $\tau_0 \rightarrow \tau_1$*, or* $\top$*, respectively; it is undefined otherwise. It is denoted by* $\mathrm{head}(\tau)$*.* τ *is said to be* constructed *iff* $\mathrm{head}(\tau)$ *is defined.*

2.4 Ground substitutions and renamings

We now define *ground substitutions.* They establish a link between types and ground types, and thus give a meaning to all structures based on types: constraint graphs, type schemes, etc.

Definition 2.8 *A* ground substitution *is a (partial) mapping from type variables to ground trees. It is said to be* regular *if its image contains only regular trees.*

From here on, we shall always work with total, regular ground substitutions (i.e. whose domain is $\mathcal{V}$ and whose image is a subset of $\mathbb{T}$), unless explicitly mentioned otherwise.

We have defined ground substitutions on variables. Let us now extend them to pretypes, then to types.

Definition 2.9 *Let ρ be a ground substitution, of domain $V \subseteq \mathcal{V}$. It is extended to the set of pretypes τ such that* $\mathrm{fv}(\tau) \subseteq V$ *by setting*

$$\begin{aligned}
\rho(\alpha) &= \rho(\alpha) \\
\rho(\bot) &= \bot \\
\rho(\top) &= \top \\
\rho(\tau_0 \to \tau_1) &= \rho(\tau_0) \to \rho(\tau_1) \\
\rho(\sqcup S) &= \sqcup \rho(S) \\
\rho(\sqcap S) &= \sqcap \rho(S)
\end{aligned}$$

ρ is then extended to types by defining the image of a type as the image of its normal form.

Proposition 2.3 *Let τ and τ' be two pretypes of the same class, i.e. such that $\tau \equiv \tau'$. Let ρ be a ground substitution. Then $\rho(\tau)$ and $\rho(\tau')$, if defined, are equal. It follows that the identities given in definition 2.9 are also valid on types.*

Proof. It suffices to verify that the equations which define the congruence (given in definition 2.2) are identities on ground types. □

Definition 2.10 *A* renaming *is a bijection between two subsets of $\mathcal{V}$.*

Renamings are straightforwardly extended to pretypes and to types.

2.5 Type containment

One might consider that the purpose of introducing $\sqcup$ and $\sqcap$ constructors is to encode a set of types into a single type. Indeed, any pos-type (resp. neg-type) can be written $\sqcup S$ (resp. $\sqcap S$), where S is a set of simple types. (To verify this, it suffices to push the $\sqcup$ and $\sqcap$ constructors towards the top of the term.)

The $\cup$ operation on these sets of terms corresponds to the $\sqcup$ operation on pos-types and to $\sqcap$ on neg-types. Which is the analogue of set-theoretic inclusion between sets of terms? We define it here. It shall be useful, in particular, when defining the closure of a constraint graph.

Definition 2.11 *A pos-type τ* contains *a pos-type τ' iff $\tau \sqcup \tau' = \tau$. Symmetrically, a neg-type τ contains a neg-type τ' iff $\tau \sqcap \tau' = \tau$. In both cases, one shall write $\tau' \lessdot \tau$.*

Recall that types are congruence classes. So, comparison between types is done by first computing their normal forms, and then comparing the latter. For instance, $\alpha \sqcup (\beta \to (\gamma \sqcup \delta))$ contains $\bot$, α, and $\beta \to \delta$.

Proposition 2.4 *$\lessdot$ is an ordering on $\mathcal{T}^+$ (respectively, $\mathcal{T}^-$).*

Proof. Let us show that $\lessdot$ is an ordering on $\mathcal{T}^+$. It is reflexive, because $\tau \sqcup \tau = \tau$. It is anti-symmetric, because if $\tau \lessdot \tau'$ and $\tau' \lessdot \tau$, then $\tau \sqcup \tau' = \tau = \tau'$. It is transitive, because if $\tau \lessdot \tau'$ and $\tau' \lessdot \tau''$, then

$$\begin{aligned}
\tau \sqcup \tau'' &= \tau \sqcup (\tau' \sqcup \tau'') \\
&= (\tau \sqcup \tau') \sqcup \tau'' \\
&= \tau' \sqcup \tau'' \\
&= \tau''
\end{aligned}$$

so $\tau \lessdot \tau''$. The result concerning $\mathcal{T}^-$ is symmetric. □

2.6 Systems of type equations

The goal of this section is to study certain systems of type equations, and in particular, to establish an isomorphism between ground types and *contractive* equation systems. This is a classic result [13]. However, it is slightly generalized here, since $\sqcup$ and $\sqcap$ constructors are allowed in equations. For this reason, we give a full proof of it. It is still a classic proof.

We first introduce a metric space structure on ground trees. Then, we define contractive systems of equations and establish the aforementioned isomorphism.

2.6.1 Metric properties of ground trees

Definition 2.12 *The set $\mathbb{T}_\infty$ of ground trees is equipped with a distance by setting $d(\tau, \tau') = 2^{-l}$, where $l = \min\{|p| \, ; \, \tau(p) \neq \tau'(p)\}$, with the convention that $2^{-\infty} = 0$. Furthermore, given a finite set I, a distance (also denoted by d) is defined over $\mathbb{T}_\infty^I$ by*

$$d((\tau_i)_{i\in I}, (\tau'_i)_{i\in I}) = \max\{d(\tau_i, \tau'_i) \, ; \, i \in I\}$$

Let V be a finite subset of $\mathcal{V}$. The set of (possibly irregular) ground substitutions of domain V is $\mathbb{T}_\infty^V$. Thus, this definition also equips it with a distance.

Equipped with this distance, $\mathbb{T}_\infty$ and $\mathbb{T}_\infty^I$ are complete metric spaces. The set of ground trees $\mathbb{T}_\infty$ is the (topological) closure of the set of finite ground trees.

Proof. See [10, 13]. □

Definition 2.13 *Given $l \in \mathbb{N}^+ \cup \{\infty\}$, two ground trees τ and τ' are said to* coincide up to depth l *iff*

$$\forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') \quad |p| < l \Rightarrow \tau(p) = \tau'(p)$$

Proposition 2.5 *Two trees τ and τ' coincide up to depth l iff $d(\tau, \tau') \leq 2^{-l}$.*

Proof. Immediate. □

Proposition 2.6 *The functions $\sqcup$ and $\sqcap$, initially defined as elements of $\mathbb{T}^2 \rightarrow \mathbb{T}$, can be extended by continuity to $\mathbb{T}_\infty^2 \rightarrow \mathbb{T}_\infty$. They are then 1-contractive.*

Any subtyping statement on ground types, involving the operators $\sqcup$, $\sqcap$, and $\rightarrow$, carries over to ground trees. In particular, $\mathbb{T}_\infty$ is a lattice.

Proof. Recall that a function $f : E \mapsto F$ (where E and F are metric spaces) is said to be 1-contractive iff

$$\forall x, x' \in E \quad d_F(f(x), f(x')) \leq d_E(x, x')$$

Let us first prove that $\sqcup$ and $\sqcap$ are uniformly continuous over $\mathbb{T}^2$. More precisely, we shall show that for any regular trees τ_0, τ'_0, τ_1 and τ'_1,

$$d(\tau_0 \sqcup \tau_1, \tau'_0 \sqcup \tau'_1) \leq d((\tau_0, \tau_1), (\tau'_0, \tau'_1))$$
$$d(\tau_0 \sqcap \tau_1, \tau'_0 \sqcap \tau'_1) \leq d((\tau_0, \tau_1), (\tau'_0, \tau'_1))$$

By definition of the distance,

$$d((\tau_0, \tau_1), (\tau'_0, \tau'_1)) = \max\{d(\tau_0, \tau'_0), d(\tau_1, \tau'_1)\}$$

So, our goal is equivalent to the following statement: "for any $l \in \mathbb{N}^+ \cup \{\infty\}$, if τ_0 coincides with τ'_0 up to depth l and τ_1 coincides with τ'_1 up to depth l, then $\tau_0 \sqcup \tau_1$ coincides with $\tau'_0 \sqcup \tau'_1$ up to depth l, and $\tau_0 \sqcap \tau_1$ coincides with $\tau'_0 \sqcap \tau'_1$ up to depth l."

This statement can be easily verified by considering the definition of $\sqcup$ and $\sqcap$ as automata products. The transitions of the product automaton depend directly on those of the input automata, so when fed a path p, the product automaton outputs a label which depends only on the labels output by the input automata when fed prefixes of p.

So, $\sqcup$ and $\sqcap$ are 1-contractive over $\mathbb{T}^2$. Hence, they can be extended by continuity to its closure, which is $\mathbb{T}^2_\infty$. The extended functions are also 1-contractive.

The function $\rightarrow$, a binary function from ground trees to ground trees, is $\frac{1}{2}$-contractive. Note that $\leq$, viewed as a binary function from ground trees to booleans, is not continuous. However, one easily shows that it is lower semi-continuous.

This implies that any subtyping assertion on ground types, involving the operators $\sqcup$, $\sqcap$, and $\rightarrow$, carries over to ground trees. $\square$

2.6.2 Contractive systems of equations

While studying contractive systems, we shall often consider irregular ground substitutions. Up to now, these were defined only on variables. (Indeed, the extension performed in definition 2.9 only deals with regular ground substitutions.) We can now extend them to pretypes, thanks to our extension of the $\sqcup$ and $\sqcap$ operations (see proposition 2.6).

Besides, there is no reason here to restrict the use of the $\sqcup$ and $\sqcap$ constructors, so we shall work with bi-pretypes.

Lemma 2.7 *Let τ be a bi-pretype. Let ρ, ρ' be two (possibly irregular) ground substitutions whose domain contain* $\mathrm{fv}(\tau)$*. Then*

$$d(\rho(\tau), \rho'(\tau)) \leq 2^{-\mathrm{dnv}(\tau)}.\, d(\rho, \rho')$$

Proof. By induction on the structure of τ.

- τ is of the form α. Then the goal becomes $d(\rho(\alpha), \rho'(\alpha)) \leq d(\rho, \rho')$, which is a direct consequence of definition 2.12.
- τ is equal to $\bot$ or $\top$. Then $\rho(\tau) = \rho'(\tau)$, so the left-hand side of the goal is 0 and the goal holds.
- τ is of the form $\tau_0 \rightarrow \tau_1$. Then

$$\begin{aligned} d(\rho(\tau), \rho'(\tau)) &= d(\rho(\tau_0 \rightarrow \tau_1), \rho'(\tau_0 \rightarrow \tau_1)) \\ &= d(\rho(\tau_0) \rightarrow \rho(\tau_1), \rho'(\tau_0) \rightarrow \rho'(\tau_1)) \\ &= \tfrac{1}{2} \max\{d(\rho(\tau_0), \rho'(\tau_0)), d(\rho(\tau_1), \rho'(\tau_1))\} \\ &\leq \tfrac{1}{2} \max\{2^{-\mathrm{dnv}(\tau_0)}, 2^{-\mathrm{dnv}(\tau_1)}\}.\, d(\rho, \rho') \\ &= 2^{-\mathrm{dnv}(\tau_0 \rightarrow \tau_1)}.\, d(\rho, \rho') \\ &= 2^{-\mathrm{dnv}(\tau)}.\, d(\rho, \rho') \end{aligned}$$

  The induction hypothesis is used to go from line 3 to line 4.

- τ is of the form $\sqcup S$. According to proposition 2.6, $\sqcup$ is a 1-contractive binary function over ground trees, and it is associative. Associativity allows us to easily extend the 1-contractiveness property to $\sqcup$ viewed as an n-ary function, for any $n \in \mathbb{N}^+$. This argument is used, with $n = |\, S\,|$, to go from line 1 to line 2 in the following:

$$\begin{aligned} d(\rho(\tau), \rho'(\tau)) &= d(\sqcup\rho(S), \sqcup\rho'(S)) \\ &\leq \max\{d(\rho(\tau), \rho'(\tau))\,;\, \tau \in S\} \\ &\leq \max\{2^{-\mathrm{dnv}(\tau)}\,;\, \tau \in S\}.\, d(\rho, \rho') \\ &= 2^{-\mathrm{dnv}(\tau)}.\, d(\rho, \rho') \end{aligned}$$

The induction hypothesis is used to go from line 2 to line 3.

- τ is of the form $\sqcap S$. This case is symmetric to the previous one. □

Definition 2.14 *Let V be a finite subset of $\mathcal{V}$. A* contractive system *of domain V is a mapping S from V to $p\mathcal{T}^{\pm}$, such that for any $\alpha \in V$, $S(\alpha)$ is a constructed bi-type with variables in V. A (possibly irregular) ground substitution ρ is a* solution *of S iff* $\mathrm{dom}(\rho) = V$ *and*

$$\forall \alpha \in V \quad \rho(\alpha) = \rho(S(\alpha))$$

Proposition 2.8 *Let S be a contractive system. Then, any solution of S is regular.*

Proof. Let ρ be a solution of S and $\alpha \in V = \mathrm{dom}(S)$. It is easy to verify that any subtree of $\rho(\alpha)$ is the image by ρ of some sub-term of $S(\beta)$, for some $\beta \in V$. Hence, $\rho(\alpha)$ has a finite number of subtrees, and ρ is a regular ground substitution. □

Theorem 2.1 *Let S be a contractive system of domain V. Then S has a unique solution.*

Proof. Define S_∞ by

$$\begin{aligned} \mathbb{T}_\infty^V &\to \mathbb{T}_\infty^V \\ (\tau_\alpha)_{\alpha \in V} &\mapsto (S(\alpha)[\alpha \leftarrow \tau_\alpha]_{\alpha \in V})_{\alpha \in V} \end{aligned}$$

We will now show that S_∞ is a $\frac{1}{2}$-contractive map. By definition of S_∞,

$$d(S_\infty((\tau_\alpha)_{\alpha \in V}), S_\infty((\tau'_\alpha)_{\alpha \in V})$$

is equal to

$$\max\{d(S(\alpha)[\alpha \leftarrow \tau_\alpha]_{\alpha \in V}, S(\alpha)[\alpha \leftarrow \tau'_\alpha]_{\alpha \in V})\,;\, \alpha \in V\}$$

We can now apply lemma 2.7 to the term $S(\alpha)$ and to the (possibly irregular) ground substitutions $\rho = [\alpha \leftarrow \tau_\alpha]_{\alpha \in V}$ and $\rho' = [\alpha \leftarrow \tau'_\alpha]_{\alpha \in V}$. We obtain that the above expression is less than, or equal to,

$$\max\{2^{-\mathrm{dnv}(S(\alpha))}\,;\, \alpha \in V\}.\, d(\rho, \rho')$$

Now, per definition 2.14, each $S(\alpha)$ is a constructed bi-pretype, so $\mathrm{dnv}(S(\alpha))$ is at least 1. Thus, the expression is less than, or equal to,

$$\tfrac{1}{2} d(\rho, \rho')$$

Finally, by definition of ρ and ρ', we have

$$\begin{aligned} d(\rho, \rho') &= d([\alpha \leftarrow \tau_\alpha]_{\alpha \in V}, [\alpha \leftarrow \tau'_\alpha]_{\alpha \in V}) \\ &= \max\{d(\tau_\alpha, \tau'_\alpha)\,;\, \alpha \in V\} \\ &= d((\tau_\alpha)_{\alpha \in V}, (\tau'_\alpha)_{\alpha \in V}) \end{aligned}$$

We have thus shown

$$d(S_\infty((\tau_\alpha)_{\alpha \in V}), S_\infty((\tau'_\alpha)_{\alpha \in V}) \leq \tfrac{1}{2} d((\tau_\alpha)_{\alpha \in V}, (\tau'_\alpha)_{\alpha \in V})$$

which states that S_∞ is $\frac{1}{2}$-contractive.

S_∞ is a contractive map from a complete metric space to itself, so it has a unique fix-point. To conclude, it remains to notice that

$$\begin{aligned} & \rho \text{ is a solution of the system } S \\ \iff\; & \forall \alpha \in V \quad \rho(\alpha) = \rho(S(\alpha)) \\ \iff\; & \forall \alpha \in V \quad \rho(\alpha) = S(\alpha)[\alpha \leftarrow \rho(\alpha)]_{\alpha \in V} \\ \iff\; & (\rho(\alpha))_{\alpha \in V} = S_\infty((\rho(\alpha))_{\alpha \in V}) \\ \iff\; & (\rho(\alpha))_{\alpha \in V} \text{ is a fix-point of } S_\infty \end{aligned}$$

□

To obtain an isomorphism between contractive systems and ground types, it only remains to prove the reciprocal statement, which poses no difficulty.

Proposition 2.9 *Let ρ be a regular ground substitution of finite domain V. Then, there exists a contractive system S whose domain contains V and whose unique solution, restricted to V, coincides with ρ.*

Proof. Each $\rho(\alpha)$, where $\alpha \in V$, is a regular tree, so it has a finite number of sub-trees. Let us associate a type variable to each of these sub-trees, while making sure that α is associated to the sub-tree $\rho(\alpha)$. Then, define a contractive system by stating the relationship between each node and its sons. Obviously the solution of S must coincide with ρ on V. $\square$

Chapter 3

Constraints

A constraint is simply a formal inequation between two types. It requires the former to be a subtype of the latter, and thus *constrains* the variables which appear in these types, by restricting the sets of their possible values. The notion of constraint is central to our theory, since we shall use a conjunction of constraints to approximate a program's data flow. The program shall be considered correct if and only if these constraints have a solution.

We first define the notion of constraint (section 3.1). We then introduce a *decomposition* operation on constraints (section 3.2), which reduces a complex constraint into a set of so-called elementary ones. Then, we explain how to represent constraints efficiently in a *constraint graph* (section 3.3). Finally, section 3.4 introduces a syntactic criterion, called *closure*, which allows determining whether a given graph admits a solution.

3.1 Definitions

Definition 3.1 *A* constraint *is a pair of a pos-type $\tau \in \mathcal{T}^+$ and of a neg-type $\tau' \in \mathcal{T}^-$, written $\tau \leq \tau'$.*

Definition 3.2 *Let $k \in \mathbb{N}^+ \cup \{\infty\}$. A ground substitution ρ is a k-*solution *of the constraint $\tau \leq \tau'$ iff $\rho(\tau) \leq_k \rho(\tau')$. A* solution *is an ∞-solution. A k-solution of a constraint set is a k-solution of each of its elements. A constraint, or a constraint set, is* solvable *iff it admits a solution.*

Finally, let us mention that some elementary functions on types are extended to constraints:

Definition 3.3 *Define*

$$\begin{aligned} \mathrm{fv}(\tau \leq \tau') &= \mathrm{fv}(\tau) \cup \mathrm{fv}(\tau') \\ \mathrm{height}(\tau \leq \tau') &= \max\{\mathrm{height}(\tau), \mathrm{height}(\tau')\} \\ \mathrm{dnv}(\tau \leq \tau') &= \min\{\mathrm{dnv}(\tau), \mathrm{dnv}(\tau')\} \end{aligned}$$

3.2 Decomposing constraints

The subtyping relation on ground types is induced by their term structure: comparing two trees amounts to performing an elementary comparison on their head constructors, then comparing their sub-trees (see proposition 1.2). It follows that any constraint $\tau \leq \tau'$

involving two constructed types is either clearly not solvable (if $\mathrm{head}(\tau) \not\leq_g \mathrm{head}(\tau')$), or equivalent to a (possibly empty) set of constraints between the sub-terms of τ and τ'. This remark allows any constraint to be reduced to a set of so-called *elementary* constraints. This process shall be called *structural decomposition.*

Definition 3.4 *A constraint $\tau \leq \tau'$ is* elementary *iff the following conditions are met:*

- *τ or τ' are type variables or constructed types;*
- *at least one of $\{\tau, \tau'\}$ is a type variable.*

Example. $\alpha \leq \beta$ et $\alpha \leq \beta \to \top$ are elementary. $\alpha \sqcup \beta \leq \gamma$ and $\alpha_0 \to \alpha_1 \leq \beta_0 \to \beta_1$ are not; they can be decomposed, as shown by the following definition.

Proposition 3.1 *The following rules—whose order is significant—define a function* subc*, which maps any solvable constraint c to a set of elementary constraints:*

$$\begin{aligned}
\mathrm{subc}(\sqcup S \leq \tau') &= \cup_{\tau \in S}\, \mathrm{subc}(\tau \leq \tau') \\
\mathrm{subc}(\tau \leq \sqcap S) &= \cup_{\tau' \in S}\, \mathrm{subc}(\tau \leq \tau') \\
\mathrm{subc}(c) &= \{c\} \textit{ if } c \textit{ is elementary} \\
\mathrm{subc}(\bot \leq \tau) &= \varnothing \\
\mathrm{subc}(\tau \leq \top) &= \varnothing \\
\mathrm{subc}(\tau_0 \to \tau_1 \leq \tau_0' \to \tau_1') &= \mathrm{subc}(\tau_0' \leq \tau_0) \cup \mathrm{subc}(\tau_1 \leq \tau_1')
\end{aligned}$$

Proof. This definition is well-founded. Indeed, if c is a solvable constraint, then the constraints which appear as arguments to subc in the right-hand sides are of a smaller size and are also solvable. Furthermore, the rules cover all possible cases; if no rule applies to the constraint $\tau \leq \tau'$, then necessarily $\mathrm{head}(\tau) \not\leq_g \mathrm{head}(\tau')$, which contradicts the assumption that the constraint is solvable. □

Proposition 3.2 *For all $k \in \mathbb{N}^+ \cup \{\infty\}$, any k-solution of* $\mathrm{subc}(c)$ *is a $(k+d)$-solution of c, where $d = \mathrm{dnv}(c)$. The converse is true when $k = \infty$, i.e. any solution of c is a solution of* $\mathrm{subc}(c)$.

Proof. Let $k \in \mathbb{N}^+ \cup \{\infty\}$. We will now show, by induction on the structure of c, that any k-solution of $\mathrm{subc}(c)$ is a $(k + \mathrm{dnv}(c))$-solution of c. The following cases arise, in order:

- c is of the form $\sqcup S \leq \tau'$ or $\tau \leq \sqcap S$. According to proposition 2.2, at least one element of S is a type variable. Hence, $\mathrm{dnv}(c) = 0$ and the result is immediate.
- c is elementary. Then one of its members is a type variable. It follows that $\mathrm{dnv}(c) = 0$ and we conclude in the same way.
- c is of the form $\bot \leq \tau$ or $\tau \leq \top$. Any ground substitution is a j-solution of c, regardless of the value of j, so the result is immediate.
- c is of the form $\tau_0 \to \tau_1 \leq \tau_0' \to \tau_1'$. Let c_0 be the constraint $\tau_0' \leq \tau_0$ and c_1 be $\tau_1 \leq \tau_1'$. We have

$$\begin{aligned}
d &= \min\{\mathrm{dnv}(\tau_0 \to \tau_1), \mathrm{dnv}(\tau_0' \to \tau_1')\} \\
&= 1 + \min\{\mathrm{dnv}(\tau_0), \mathrm{dnv}(\tau_1), \mathrm{dnv}(\tau_0'), \mathrm{dnv}(\tau_1')\} \\
&= 1 + \min\{\mathrm{dnv}(c_0), \mathrm{dnv}(c_1)\} \\
&= 1 + \min\{d_0, d_1\}
\end{aligned}$$

By induction hypothesis, for $i \in \{0,1\}$, any k-solution of $\mathrm{subc}(c_i)$ is a $(k+d_i)$-solution of c_i. Hence, we have

$$\begin{array}{ll} & \rho \text{ is a } k\text{-solution of } \mathrm{subc}(c) \\ \iff & \forall i \in \{0,1\} \quad \rho \text{ is a } k\text{-solution of } \mathrm{subc}(c_i) \\ \Rightarrow & \forall i \in \{0,1\} \quad \rho \text{ is a } (k+d_i)\text{-solution of } c_i \\ \Rightarrow & \rho \text{ is a } (k+d)\text{-solution of } c \end{array}$$

The proof of the reciprocal statement, in the case where $k = \infty$, presents no difficulties. □

3.3 Constraint graphs

While analyzing a program, we shall of course need to manipulate conjunctions of constraints. The simplest approach, used in numerous presentations of constrained type systems, is to group constraints inside a set. However, we have seen that it is possible to reduce any constraint to a conjunction of elementary constraints (see section 3.2), and that two bounds on a single variable can be combined using the $\sqcup$ and $\sqcap$ constructors (see section 2.1). We shall take advantage of these observations to represent conjunctions of constraints inside a better adapted structure, called a *constraint graph.*

Definition 3.5 *A* constraint graph C*, of domain* $V \subseteq \mathcal{V}$*, is made up of*

- *a reflexive relation between elements of* V*, denoted by* $\leq_C$*;*
- *for any* $\alpha \in V$*, a constructed pos-type* $C^{\downarrow}(\alpha) \in \mathcal{T}^+$ *and a constructed neg-type* $C^{\uparrow}(\alpha) \in \mathcal{T}^-$*, whose variables are in* V*.*

Let $k \in \mathbb{N}^+ \cup \{\infty\}$*. A ground substitution* ρ *is a* k-solution *of* C *iff for all* $\alpha, \beta \in V$*:*

- $\alpha \leq_C \beta$ *implies* $\rho(\alpha) \leq_k \rho(\beta)$*;*
- $\rho(C^{\downarrow}(\alpha)) \leq_k \rho(\alpha) \leq_k \rho(C^{\uparrow}(\alpha))$*.*

We shall write $\rho \vdash C$ *to indicate that* ρ *is a solution of* C*.*

Having defined solutions, we can now define entailment. This notion is rather central, although it is partially superseded by the more general notion of scheme subsumption. It shall be studied in detail in chapter 8.

Definition 3.6 *Let* $k \in \mathbb{N}^+ \cup \{\infty\}$*. A constraint graph* C k-entails *a constraint* c *iff any* k*-solution of* C *is a* k*-solution of* c*. We denote this fact by* $C \Vdash_k c$*.* Entailment *is, by definition,* ∞*-entailment and is denoted by* $\Vdash$*.*

The following property allows establishing an entailment assertion through a series of finite approximations.

Proposition 3.3 *If* $C \Vdash_k c$ *holds for all finite* k*, then* $C \Vdash c$*.*

Proof. Let ρ be a solution of c and let $k \in \mathbb{N}^+$. ρ is, in particular, a k-solution of C, and $C \Vdash_k c$, so ρ is a k-solution of c. Since this holds for all k, ρ is a solution of c. □

While working with constraint graphs, we shall need to be able to determine whether a given graph "contains" a given constraint, and also to "add" a given constraint to a given graph. If we had chosen to work with constraint sets, plain set-theoretic membership and union would do. Here, we have to give slightly more complex definitions. However, they are still based on simple, syntactic criteria. The rest of this section is dedicated to their definition.

Definition 3.7 *A constraint graph C* contains *an elementary constraint c iff one of the following conditions holds:*

- *c is of the form $\alpha \leq \beta$, and $\alpha \leq_C \beta$;*
- *c is of the form $\alpha \leq \tau$, where τ is a constructed neg-type, and $C^{\uparrow}(\alpha)$ contains τ;*
- *c is of the form $\tau \leq \alpha$, where τ is a constructed pos-type, and $C^{\downarrow}(\alpha)$ contains τ.*

C contains an arbitrary constraint c iff $\mathrm{subc}(c)$ *is defined and C contains each of its elements.*

Note that this definition uses the notion of type containment introduced by definition 2.11.

The following proposition confirms that this relation is, as expected, analogous to set membership.

Proposition 3.4 *Let $k \in \mathbb{N}^+ \cup \{\infty\}$. If a constraint graph C contains a constraint c, then any k-solution of C is a $(k+d)$-solution of c, where $d = \mathrm{dnv}(c)$. In particular, for $k = \infty$, we obtain $C \Vdash c$.*

Proof. Let ρ be a k-solution of C. Let us first assume that c is elementary. Then $\mathrm{dnv}(c) = 0$, so we have to show that ρ is a k-solution of c. Three cases arise, depending on the form of c, as in definition 3.7.

- c is of the form $\alpha \leq \beta$. Then, $\alpha \leq_C \beta$, according to definition 3.7. So $\rho(\alpha) \leq_k \rho(\beta)$, and ρ is a k-solution of c.
- c is of the form $\alpha \leq \tau$, where τ is a constructed neg-type. Then $C^{\uparrow}(\alpha)$ contains τ; that is, $C^{\uparrow}(\alpha) \sqcap \tau = C^{\uparrow}(\alpha)$. This implies $\rho(C^{\uparrow}(\alpha)) \sqcap \rho(\tau) = \rho(C^{\uparrow}(\alpha))$, which can be rewritten $\rho(C^{\uparrow}(\alpha)) \leq \rho(\tau)$. Besides, because ρ is a k-solution of C, we have $\rho(\alpha) \leq_k \rho(C^{\uparrow}(\alpha))$. $\leq_k$ contains $\leq_\infty$ and is transitive, so $\rho(\alpha) \leq_k \rho(\tau)$, and ρ is a k-solution of c.
- c is of the form $\tau \leq \alpha$, where τ is a constructed pos-type. This case is symmetric to the previous one.

We have shown that the proposition holds for any elementary c. Let us now move on to the general case. Then, $\mathrm{subc}(c)$ is defined and C contains any $c' \in \mathrm{subc}(c)$. The proposition applies to each c', so ρ is a k-solution of $\mathrm{subc}(c)$. According to proposition 3.2, ρ is then a $(k+d)$-solution of c. $\square$

Let us now move on to the second problem mentioned above, namely the addition of a constraint to a graph.

Definition 3.8 *The* addition *of an elementary constraint c to a constraint graph C, denoted by $C + c$, is the constraint graph D defined as follows:*

- *If c is of the form $\alpha \leq \beta$, then $\leq_D = \{(\alpha, \beta)\} \cup \leq_C$, $D^{\uparrow} = C^{\uparrow}$ and $D^{\downarrow} = C^{\downarrow}$.*
- *If c is of the form $\alpha \leq \tau$ where τ is a constructed neg-type, then $\leq_D = \leq_C$, $D^{\uparrow} = C^{\uparrow} + [\alpha \mapsto \tau \sqcap C^{\uparrow}(\alpha)]$ and $D^{\downarrow} = C^{\downarrow}$.*
- *If c is of the form $\tau \leq \alpha$ where τ is a constructed pos-type, then $\leq_D = \leq_C$, $D^{\uparrow} = C^{\uparrow}$ and $D^{\downarrow} = C^{\downarrow} + [\alpha \mapsto \tau \sqcup C^{\downarrow}(\alpha)]$.*

(If $\mathrm{fv}(c) \not\subseteq V$*, where* $V = \mathrm{dom}(C)$*, then* C *is first extended to* $\mathrm{fv}(c) \cup V$ *by defining* $\alpha \leq_C \alpha$*,* $C^{\downarrow}(\alpha) = \bot$ *and* $C^{\uparrow}(\alpha) = \top$ *for any* $\alpha \in \mathrm{fv}(c) \setminus V$*.)*

If c_1 *and* c_2 *are elementary constraints, then* $(C + c_1) + c_2 = (C + c_2) + c_1$*. This allows us to define the addition of an arbitrary constraint* c *to a constraint graph* C*, also denoted by* $C + c$*, as the addition of all elements of* $\mathrm{subc}(c)$ *to* C*.*

The following proposition formalizes the intuition that this operation is analogous to set union.

Proposition 3.5 *The solutions of* $C + c$ *are exactly the solutions common to* C *and* c*.*

Proof. Easy case analysis. □

Finally, we extend renamings to constraint graphs in the (very predictable) following way:

Definition 3.9 *Let* ρ *be a renaming and* C *a constraint graph of domain* V*, such that* $V \subseteq \mathrm{dom}(\rho)$*. Then* $\rho(C)$ *is the constraint graph* D *of domain* $\rho(V)$ *defined as follows:*

- $\alpha \leq_D \beta \iff \rho^{-1}(\alpha) \leq_C \rho^{-1}(\beta)$*;*
- $D^{\uparrow} = \rho \circ C^{\uparrow} \circ \rho^{-1}$ *and* $D^{\downarrow} = \rho \circ C^{\downarrow} \circ \rho^{-1}$*.*

3.4 Solving constraint graphs

We are now interested in determining whether a given constraint graph admits a solution. This operation is fundamental to our type system, since a program shall be considered well-typed if and only if its associated constraint graph is solvable.

We first define a property of constraint graphs, called *closure*, which is sufficient to guarantee the existence of a solution. Then, we show that if a constraint graph is solvable, then there exists a closed graph which is equivalent to it. This assertion is effective; so, it provides a decision algorithm for the solvability problem.

Note that what we call a "closed" graph is usually termed "closed and consistent" in the literature [15, 38, 40]. Here, closure and consistency are combined into a single notion. This makes our theory slightly simpler, but entails no practical difference.

Definition 3.10 *A constraint graph* C *of domain* V *is* closed *iff the following conditions are met:*

- $\leq_C$ *is transitive;*
- *for all* $\alpha, \beta \in V$ *such that* $\alpha \leq_C \beta$*,* $C^{\downarrow}(\beta)$ *contains* $C^{\downarrow}(\alpha)$ *and* $C^{\uparrow}(\alpha)$ *contains* $C^{\uparrow}(\beta)$*;*
- *for all* $\alpha \in V$*,* C *contains* $C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$*.*

We shall sometimes refer to plain closure*, rather than closure, to emphasize the difference with the less restrictive notions (weak closure, simple closure) to be developed later.*

In the above definition, the first two conditions correspond to a transitivity property, while the third one is a combination of transitivity and structural decomposition.

Actually, the transitivity of $\leq_C$ is not used in the proof of theorem 3.1, which states that any closed graph is solvable. Nevertheless, we include it in the definition of closed graphs, because it is important from an implementation's point of view. In particular, it allows implementing the closure algorithm in a more efficient way. Besides, still from the implementor's point of view, it is a prerequisite for the canonization phase (see chapter 11), which follows the closure phase. Thus, it is interesting to be able to compose the two phases without any intermediate computation.

If interested in a weaker solvability criterion, the reader is referred to the definition of *weak closure* (definition 9.4). The latter involves more elaborate mechanisms, such an axiomatization of entailment, and shall only be useful from a theoretical point of view.

Theorem 3.1 *Any closed constraint graph admits a solution.*

Proof. The principle of the proof is to build a system of equations which is stronger than the supplied constraint graph, then to produce a solution of it and to verify that it is also a solution of the constraint graph. Here, we choose to equate each type variable with its constructed lower bound. Other choices are possible. For instance, one could choose to equate each variable with its constructed upper bound, or in general, with any constructed type which is provably comprised between these two bounds. Thus, our choice is arbitrary. This is not a problem, since our goal is to produce an arbitrary solution, not to enumerate all of them, which would be a much more difficult task.

Let C be a closed constraint graph of domain V. Then $C^{\downarrow}$ is a contractive system of domain V; according to theorem 2.1, it admits a solution ρ. ρ is a ground substitution of domain V. We extend it to $\mathcal{V}$ arbitrarily. Let us now verify that ρ is a solution of C.

First, let $\alpha, \beta \in V$ such that $\alpha \leq_C \beta$. Since C is closed, $C^{\downarrow}(\beta)$ contains $C^{\downarrow}(\alpha)$. That is, $C^{\downarrow}(\beta) \sqcup C^{\downarrow}(\alpha) = C^{\downarrow}(\beta)$. By applying ρ to this equation, we obtain

$$\begin{array}{cc} & \rho(C^{\downarrow}(\beta)) \sqcup \rho(C^{\downarrow}(\alpha)) = \rho(C^{\downarrow}(\beta)) \\ \iff & \rho(C^{\downarrow}(\alpha)) \leq \rho(C^{\downarrow}(\beta)) \\ \iff & \rho(\alpha) \leq \rho(\beta) \end{array}$$

Next, let $\alpha \in V$. It is immediate that $\rho(C^{\downarrow}(\alpha)) \leq \rho(\alpha)$, since these expressions are equal. There remains to show that $\rho(C^{\downarrow}(\alpha)) \leq \rho(C^{\uparrow}(\alpha))$. We cannot do this directly; instead, we shall show, by induction on $k \in \mathbb{N}^+$, that

$$\forall \alpha \in V \quad \rho(C^{\downarrow}(\alpha)) \leq_k \rho(C^{\uparrow}(\alpha))$$

The assertion holds for $k = 0$, since $\leq_0$ is uniformly true. Assume it holds for a given $k \in \mathbb{N}^+$. Then ρ is a k-solution of C. Let $\alpha \in V$. Since C is closed, C contains the constraint $C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$, which we shall call c. According to proposition 3.4, ρ is a $(k + d)$-solution of c, where $d = \mathrm{dnv}(c)$. However, since $C^{\downarrow}(\alpha)$ and $C^{\uparrow}(\alpha)$ are constructed types, we have $d \geq 1$, so ρ is a $(k + 1)$-solution of c. The induction is complete.

Thanks to proposition 1.3, the above result implies that

$$\forall \alpha \in V \quad \rho(C^{\downarrow}(\alpha)) \leq \rho(C^{\uparrow}(\alpha))$$

whence we conclude that ρ is a solution of C. □

This theorem forms a theoretical basis for the closure algorithm, which allows deciding whether a given constraint graph is solvable. Rather than giving this algorithm here, we delay its description until chapter 7. Indeed, the algorithm's proof uses the small terms invariant, introduced in chapter 6.

The closure property is so called because of the way it is defined, that is, by deriving new logical consequences from the constraints and making sure that they are in fact already contained within the graph. However, we could also have talked about *solved* constraint graphs, not only because any closed constraint graph has a solution, but also because such a graph is in fact the best description of its own set of solutions. In other words, there is no known, finitary way to enumerate all solutions of a closed constraint graph, other than to produce the graph itself.

Chapter 4

Type schemes

Like that of ML, our type system allows polymorphism, introduced by the `let` construct. Thus, a typing judgement shall associate a program not simply with a type, but with a *type scheme*, which (roughly speaking) stands for the set of its *ground instances*.

However, following Trifonov and Smith [46], our presentation departs significantly from that of ML. Our goal is for *all* type variables of any type scheme to be universally quantified. In ML, newly introduced variables are not quantified; they become so only after an explicit *generalization* operation, associated with the `let` construct. Here, any type scheme is, so to speak, generalized to the maximum, and the `let` operation only moves an existing scheme into the environment, without requiring any generalization.

The advantage of this decision is to give birth to a system *without shared variables*. For instance, since a type scheme has no free variables, one can define its denotation, that is (approximately) the set of its ground instances, regardless of the environment. To compare two type schemes, i.e. to determine whether one is more general than the other, it suffices to compare their denotations, still independently of any environment. Furthermore, we shall notice that in our type system (introduced in chapter 5), two distinct branches of a typing derivation share no type variables. Thus, the system becomes more modular, more elementary. This leads to a considerable simplification, and a better understanding, of the theory.

In ML, it is incorrect to generalize a variable if it appears free in the environment. So, how can we hope to be able to generalize all variables? The solution is to move the environment and make it an integral part of the type scheme. This presentation is known as λ-*lifting*; its functioning shall be detailed by the typing rules (see chapter 5). More precisely, the type information concerning `let`-bound variables remains stored inside an external environment, while information about λ-bound variables appears in a *context* which is part of the type scheme.

We first define type schemes (section 4.1), then we introduce a *subsumption* relation between them (section 4.2).

4.1 Type schemes

Definition 4.1 *Assume given a denumerable set of* λ-identifiers*, denoted by* $x, y, \dots$ Contexts *are defined by*

$$\begin{array}{rcl} A & ::= & \varnothing \\ & | & A; x : \tau \end{array}$$

where τ is a neg-type.

Let us define a few classic notations:

Definition 4.2 *Set*

$$(A; x : \tau)(x) = \tau \qquad (A; x : \tau) \setminus x = A$$
$$(A; y : \tau)(x) = A(x) \qquad (A; y : \tau) \setminus x = (A \setminus x); y : \tau$$

$$\mathrm{fv}(\varnothing) = \varnothing \qquad \mathrm{dom}(\varnothing) = \varnothing$$
$$\mathrm{fv}(A; x : \tau) = \mathrm{fv}(A) \cup \mathrm{fv}(\tau) \qquad \mathrm{dom}(A; x : \tau) = \mathrm{dom}(A); x$$

Definition 4.3 *A* type scheme *is of the form*

$$\sigma ::= A \Rightarrow \tau \mid C$$

where A is a context, τ is a pos-type and C is a constraint graph whose domain contains $\mathrm{fv}(A) \cup \mathrm{fv}(\tau)$. *(The symbol $\mid$ should here be interpreted as a literal, not as a choice.) σ is* closed *iff C is closed. σ is* simple *iff it contains only simple types, i.e. iff it contains no occurrences of $\sqcup$ or $\sqcap$.*

Intuitively speaking, all variables which appear in a type scheme are to be considered here as universally quantified. However, we shall not use any explicit quantifier. Formally speaking, no implicit α-conversion is allowed on type schemes; α-conversion shall be dealt with explicitly. This decision has several advantages: it brings more clarity, hence more rigor, and it allows an explicit description of the way fresh variables and renamings are handled.

Every type scheme contains a context, which describes the assumptions made by the expression of interest about its environment. In other words, if we typecheck an open expression, i.e. one that contains free program variables, then the inferred context shall indicate which types these variables must have for the expression to be correct. The most trivial example is the open expression `x`, which can be given the type scheme

$$(\varnothing; x : \alpha) \Rightarrow \alpha \mid \varnothing$$

This scheme expresses a tautology: if `x` has type α, then `x` has type α. Note that this holds for any α; thus, it is correct to consider this type scheme as universally quantified. For more details about the mechanics of contexts, please refer to the description of the typing rules in section 5.3.

4.2 Scheme subsumption

We shall now define a *polymorphic comparison* relation between type schemes, called *subsumption*. It is based on ground subtyping, but accounts for the fact that a type scheme contains universally quantified variables. This relation plays a fundamental role in our theory, since it shall allow justifying all of the transformations and simplifications to be introduced on type schemes.

This relation has a very simple definition. It comes from the idea that a type scheme is in fact just a way of describing a set of ground typings, which we shall call its *denotation*. To compare two schemes, it suffices to compare their denotations, using plain set-theoretic inclusion.

What is the denotation of a scheme $\sigma = A \Rightarrow \tau \mid C$? In other words, which ground typings does this type scheme represent? At least all of its ground instances, that is, all ground typings obtained through substitutions. Of course, applying a ground substitution to σ is only legal if it satisfies the constraints in C; so, the ground instances of σ are the $\rho(A \Rightarrow \tau)$, where ρ ranges over the solutions of C. Furthermore, notice that ground typings are ordered by subtyping; given a correct ground typing, the cone generated by it also contains only correct typings.

Let us now formalize this discussion.

Definition 4.4 Ground contexts *are defined by*

$$\begin{array}{rcl} A & ::= & \varnothing \\ & \mid & A; x : \tau \end{array}$$

where τ is a ground type. The subtyping relation is extended, point-wise, to ground contexts of identical domains. A ground typing *is of the form*

$$A \Rightarrow \tau$$

where A is a ground context and τ is a ground type. The subtyping relation is extended to ground typings, in a contravariant way for the context, and in a covariant way for the type.

Definition 4.5 *Let $A \Rightarrow \tau \mid C$ be a type scheme. Its* denotation *is the union of the cones generated by its ground instances, that is*

$$\{A' \Rightarrow \tau' \,;\, \exists \rho \vdash C \quad \rho(A \Rightarrow \tau) \leq A' \Rightarrow \tau'\}$$

where $\rho(A \Rightarrow \tau)$ stands for the ground typing $\rho(A) \Rightarrow \rho(\tau)$.

Definition 4.6 *Given two type schemes σ_1 and σ_2, the former is said to be* more general *than the latter iff the former's denotation contains the latter's. This fact is denoted by $\sigma_1 \leq^\forall \sigma_2$.*

In other words, σ_1 is more general than σ_2 iff for any ground instance of σ_2, there exists a ground instance of σ_1 which is smaller. Formally,

$$(A_1 \Rightarrow \tau_1 \mid C_1) \leq^\forall (A_2 \Rightarrow \tau_2 \mid C_2)$$

is thus equivalent to

$$\forall \rho_2 \vdash C_2 \quad \exists \rho_1 \vdash C_1 \quad \rho_1(A_1 \Rightarrow \tau_1) \leq \rho_2(A_2 \Rightarrow \tau_2)$$

We shall write $\sigma =^\forall \sigma'$ when $\sigma \leq^\forall \sigma'$ and $\sigma' \leq^\forall \sigma$.

Finally, in order to designate in a concise way the constraints generated by comparing two type schemes, we give the following definition:

Definition 4.7 *The notation $A \leq A'$, where A and A' are two contexts with the same domain, is defined by*

$$\begin{gathered} \varnothing \leq \varnothing = \varnothing \\ (A; x : \tau) \leq (A'; x : \tau') = (A \leq A') \cup \mathrm{subc}(\tau \leq \tau') \end{gathered}$$

Let $A \Rightarrow \tau \mid C$ and $A' \Rightarrow \tau' \mid C'$ be type schemes. The notation $A \Rightarrow \tau \leq A' \Rightarrow \tau'$ shall stand for the set $(A' \leq A) \cup \mathrm{subc}(\tau \leq \tau')$.

The definition of the relation $\leq^\forall$ is due to Trifonov and Smith [46]. Note that it generalizes both solvability and entailment assertions. Indeed, the assertion $\exists\rho \vdash C$, which states that C is solvable, can be written $\top \mid C \leq^\forall \top \mid \varnothing$. The assertion $C \Vdash \alpha \leq \beta$ is equivalent to $\gamma \rightarrow \gamma \mid \varnothing \leq^\forall \alpha \rightarrow \beta \mid C$. Since the decidability of entailment is currently an open problem, the same is true of comparison between type schemes. However, in chapter 9, we shall give an incomplete decision algorithm for the latter. Unsurprisingly, this algorithm shall generalize the closure algorithm, which decides the solvability of a constraint graph, and the (incomplete) entailment algorithm.

In [42], we presented a less powerful comparison relation, which didn't account for the fact that some variables were universally quantified. It was defined in terms of the entailment relation. It is now unused, except in the proof of correctness of the typing rules, where it plays an auxiliary role (see definition 5.8). It is sufficient to justify substitution-based simplifications. However, since it does not consider variables as quantified, it prohibits numerous operations; for instance, renamings, but also the elimination of unreachable variables, which was justified in [42] by rewriting the whole typing derivation. This is why Trifonov and Smith introduced the relation $\leq^\forall$: renamings, as well as garbage collection (which generalizes the elimination of unreachable variables) can now be justified with a single application of the subtyping rule. Thus, former meta-theoretic properties are now integrated into the system.

Chapter 5

The type system

A type system's foremost goal is to eliminate all programs which cause errors at execution time. (A secondary goal could be to document a program, by using its type as a description of its functionality—type simplification then comes into play—but that aspect shall not interest us in this chapter.)

A type system usually consists of a predicate called *typing judgement*, with three parameters Γ, e and σ, denoted by $\Gamma \vdash e : \sigma$. It is defined as the smallest predicate stable through a certain set of *typing rules*. That is, a typing judgement is valid if and only if it can be *derived* using these rules. A program e shall be considered correct if and only if it is well typed in the empty environment, i.e. if the predicate $\varnothing \vdash e : \sigma$ holds for some σ.

A type system must be *safe*: if a program is found correct, it must cause no execution errors. Of course, to give a meaning to this sentence, we must give a formal specification of execution. So, we give an *operational semantics*, which describes execution as a series of *reductions*, i.e. rewritings of the program text.

Finally, to have practical interest, a type system must be *decidable*: one wants to be able to verify automatically that a program is well typed. Better yet, here, we wish to do *type inference*, i.e. determine whether a program is well typed and discover its type without any help from the programmer. The type system must then be accompanied by a description of the inference algorithm. In order to allow modular programming, the inferred type must be the (or rather, a) most general type for the supplied program. This requires such a type to exist in general: that is, the type system must have *principal types*.

In this chapter, we describe our type system and show that it meets all of the above criteria. We first give a brief definition of the language (section 5.1). Next, after some technical preliminaries (section 5.2), comes the description of the type system (section 5.3). Section 5.4 describes another type system, called "simple", equivalent to the former in terms of accepted programs, but less powerful in terms of valid typing judgements. This system shall be used in the proof of correctness. Section 5.5 describes a third system, also equivalent to its predecessor but more restricted, which in fact constitutes a specification of the type inference algorithm. Once we have introduced all three systems, we show that they are equivalent (section 5.6); furthermore, we prove that principal types exist and are computed by the inference algorithm. Lastly, section 5.7 deals with defining the semantics and proving that the type system respects it.

5.1 Language

The language we are interested in is core ML, that is, a λ-calculus equipped with a `let` construct. For the sake of simplicity, we separate λ-bound identifiers from `let`-bound ones, by placing them in two distinct syntactic classes.

Definition 5.1 *Assume given a denumerable set of* `let`-identifiers*, denoted by* $X, Y, \ldots$ Expressions *are defined by*

$$e ::= x \mid \lambda x.e \mid e\,e \mid X \mid \texttt{let}\, X = e \,\texttt{in}\ e$$

5.2 Preliminary definitions

Our typing rules deal with the environment in a slightly unusual way. The part of the environment that pertains to `let`-bound variables appears, in the classic way, left of the $\vdash$ symbol in a typing judgement. However, the part of the environment that concerns λ-bound variables is considered part of the type scheme; thus, it appears right of the $\vdash$ symbol. This presentation, sometimes called λ-*lifting*, requires some preliminary definitions, to which this section is dedicated.

We shall not choose the simplest possible presentation of λ-lifting. A slightly less verbose description is possible, provided one makes the hypothesis that two distinct λ binders bind distinct variables. Given any program, it is possible, through renamings, to produce an equivalent program which satisfies this hypothesis. However, this property is not preserved by β-reduction. Thus, this presentation is not suitable, should one wish to prove the type system's correctness with respect to the semantics; this is why we discard it. Let us, however, describe it tersely. (For more details, one can consult [46].) As is the case here, environments only deal with `let`-bound variables X. The inferred contexts mention only the variables actually used by the expression, as opposed to all λ-bound variables currently in scope, as is the case in our system. The "lift" and "unlift" operations disappear, since the capture phenomenon has been eliminated. So, this presentation is less "bureaucratic"; as a matter of fact, we adopt it in our implementation, for efficiency. It suffices, in the implementation, to rename some variables on the fly to satisfy the requirement that λ-bound variables should have unique names.

Definition 5.2 Environments *are defined by*

$$\begin{array}{rcl} \Gamma & ::= & \varnothing \\ & \mid & \Gamma; x \\ & \mid & \Gamma; X : \sigma \end{array}$$

An environment Γ associates a type scheme σ to any `let`-bound variable X. Note that it does not associate any type information to λ-bound variables x; the presence of these variables in the environment only serves to handle bindings correctly. Association of a type to λ-bound variables is done inside the type scheme.

Definition 5.3 *The* domain *and the* λ-domain *of an environment are defined by*

$$\begin{array}{rclcrcl} \mathrm{dom}(\varnothing) & = & \varnothing & \qquad & \mathrm{dom}_\lambda(\varnothing) & = & \varnothing \\ \mathrm{dom}(\Gamma; x) & = & \mathrm{dom}(\Gamma); x & & \mathrm{dom}_\lambda(\Gamma; x) & = & \mathrm{dom}_\lambda(\Gamma); x \\ \mathrm{dom}(\Gamma; X : \sigma) & = & \mathrm{dom}(\Gamma); X & & \mathrm{dom}_\lambda(\Gamma; X : \sigma) & = & \mathrm{dom}_\lambda(\Gamma) \end{array}$$

The following auxiliary functions are necessary when one wishes to introduce or eliminate a λ construct.

Definition 5.4 *Define*

$$
\begin{aligned}
\mathrm{lift}_x(\varnothing) &= \varnothing \\
\mathrm{lift}_x(\Gamma; y) &= \mathrm{lift}_x(\Gamma); y \\
\mathrm{lift}_x(\Gamma; X : A \Rightarrow \tau \mid C) &= \mathrm{lift}_x(\Gamma); X : (A; x : \top) \Rightarrow \tau \mid C \\
\mathrm{unlift}_x(\varnothing) &= \varnothing \\
\mathrm{unlift}_x(\Gamma; y) &= \mathrm{unlift}_x(\Gamma); y \\
\mathrm{unlift}_x(\Gamma; X : A \Rightarrow \tau \mid C) &= \mathrm{unlift}_x(\Gamma); X : (A \setminus x) \Rightarrow \tau \mid C
\end{aligned}
$$

Definition 5.5 *The* subtraction *of a λ-identifier x from an environment Γ, denoted by $\Gamma \setminus x$, is defined by*

$$
\begin{aligned}
(\Gamma; x) \setminus x &= \Gamma \\
(\Gamma; y) \setminus x &= (\Gamma \setminus x); y \\
(\Gamma; X : \sigma) \setminus x &= (\Gamma \setminus x); X : \sigma
\end{aligned}
$$

Given an environment Γ, a variable x and two types τ and τ', one wishes to define a context $\mathrm{single}_{(x,\tau,\tau')}(\Gamma)$, with the same domain as Γ, which maps x to τ and any other variable to τ'. This definition shall be used in the type inference rule (VAR$_\mathrm{I}$).

Definition 5.6 *Given a variable x and two types τ, τ', define*

$$
\begin{aligned}
\mathrm{single}_{(x,\tau,\tau')}(\Gamma; x) &= \mathrm{uni}_{\tau'}(\Gamma); x : \tau \\
\mathrm{single}_{(x,\tau,\tau')}(\Gamma; y) &= \mathrm{single}_{(x,\tau,\tau')}(\Gamma); y : \tau' \\
\mathrm{single}_{(x,\tau,\tau')}(\Gamma; X : \sigma) &= \mathrm{single}_{(x,\tau,\tau')}(\Gamma)
\end{aligned}
$$

where $\mathrm{uni}_{\tau'}$ *is given by*

$$
\begin{aligned}
\mathrm{uni}_{\tau'}(\varnothing) &= \varnothing \\
\mathrm{uni}_{\tau'}(\Gamma; x) &= \mathrm{uni}_{\tau'}(\Gamma); x : \tau' \\
\mathrm{uni}_{\tau'}(\Gamma; X : \sigma) &= \mathrm{uni}_{\tau'}(\Gamma)
\end{aligned}
$$

5.3 Typing rules

The typing rules are given by figure 5.1 on the following page.

Typing judgements are of the form $\Gamma \vdash e : \sigma$. One verifies that all contexts A which appear in this judgement, be it inside Γ or inside σ, have domain $\mathrm{dom}_\lambda(\Gamma)$. The reason behind the lift_x operation used in rule (ABS) is precisely to enforce this invariant.

The typing rules describe how to analyze a program and generate subtyping constraints. There remains to verify that these constraints admit a solution, as specified by the following definition.

Definition 5.7 *An expression e is* well typed *in an environment Γ iff there exists a type scheme σ, whose denotation is non-empty, such that $\Gamma \vdash e : \sigma$.*

Recall that the denotation of a type scheme $A \Rightarrow \tau \mid C$ is non-empty if and only if C admits a solution. Thus, to determine whether a program is well-typed, one must not only build a typing derivation, but also make sure that it yields a solvable constraint graph. To this end, an algorithm, based on a *closure* computation, shall be given in chapter 7.

$$\frac{\mathrm{dom}(A) = \mathrm{dom}_\lambda(\Gamma)}{\Gamma \vdash x : A \Rightarrow A(x) \mid C} \quad (\textsc{Var})$$

$$\frac{\mathrm{lift}_x(\Gamma); x \vdash e : (A; x : \tau) \Rightarrow \tau' \mid C}{\Gamma \vdash \lambda x.e : A \Rightarrow (\tau \to \tau') \mid C} \quad (\textsc{Abs})$$

$$\frac{\Gamma \vdash e_1 : A \Rightarrow (\tau_2 \to \tau) \mid C \qquad \Gamma \vdash e_2 : A \Rightarrow \tau_2 \mid C}{\Gamma \vdash e_1\, e_2 : A \Rightarrow \tau \mid C} \quad (\textsc{App})$$

$$\frac{\Gamma(X) = \sigma}{\Gamma \vdash X : \sigma} \quad (\textsc{LetVar})$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash e_2 : A_2 \Rightarrow \tau_2 \mid C_2 \qquad \sigma_1 \leq^\forall A_2 \Rightarrow \top \mid C_2}{\Gamma \vdash \mathtt{let}\, X = e_1\ \mathtt{in}\ e_2 : A_2 \Rightarrow \tau_2 \mid C_2} \quad (\textsc{Let})$$

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \leq^\forall \sigma'}{\Gamma \vdash e : \sigma'} \quad (\textsc{Sub})$$

Figure 5.1: Typing rules

One rule is associated to each syntactic construct; in addition, rule (SUB), called the "subtyping rule", allows reformulating the type scheme at any point, with great flexibility, since it uses the powerful scheme subsumption relation. In particular, this rule allows arbitrary α-conversions, in accordance with the fact that all variables of a type scheme should be regarded as universally quantified. Besides, all of our simplification methods shall be presented as algorithms which accept a type scheme σ and produce an equivalent scheme σ'; thus, a single use of rule (SUB) shall suffice to prove their correctness. Of course, on the other hand, the power of rule (SUB) poses a problem when trying to prove that the type system is correct: because of this rule, it is very difficult to show directly that reduction preserves typings. To work around this problem, we shall introduce a second set of rules, called "simple" typing rules, with a much weaker subtyping rule.

These typing rules are very close to those of Trifonov and Smith [46]. The main differences are in our treatment of λ-lifting, which is heavier but better suited to the subject reduction proof, and in our formulation of rule (LET).

The first of these points has been mentioned in section 5.2, where we briefly discussed the differences between the two possible presentations of λ-lifting. However, we did not explain the very principle of λ-lifting, that is, the way this mechanism allows the type system to emulate ML's behavior, while using universally quantified variables exclusively. So, let us give a brief example of its functioning. The question is, how can we give an expression a "monomorphic" type, since all variables must be universally quantified? We shall now answer it, in a slightly informal way. Consider the expression

$$\lambda x.\mathtt{let}\, Y = x\ \mathtt{in}\ (Y, Y)$$

Let us type this expression in ML. Y's type is a monomorphic variable α. So, the two uses of Y do not require any instantiation, and the expression's type is $\alpha \to \alpha \times \alpha$. In our system, on the contrary, Y's type is $(x : \alpha) \Rightarrow \alpha$, according to rule (VAR). Here, α is (implicitly) universally quantified. So, if one were free to use rule (SUB) to perform renamings, the two uses of Y could yield two distinct schemes $(x : \beta) \Rightarrow \beta$ and $(x : \gamma) \Rightarrow \gamma$. However, the typing rule for pairs requires that its two branches share the same context. (Our language actually does not have pairs; but the rule for pairs can be derived from the application rule, where the same condition is found: both branches must use the same context A.) So, necessarily, β and γ must be the same variable, and the pair (Y, Y) has type $(x : \beta) \Rightarrow \beta \times \beta$. Once the λ-abstraction is performed, the whole expression receives type $\beta \rightarrow \beta \times \beta$, as expected. To sum up, all variables which appear in the context actually behave monomorphically; this is caused by a sharing constraint on contexts, which is enforced whenever two branches of the inference tree are brought together. So, the system is correct; nonetheless, it is able to eliminate the notion of unquantified type variable.

The second point is our formulation of rule (LET). Trifonov and Smith [46] propose a simpler rule:

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \mathtt{let}\, X = e_1 \;\mathtt{in}\; e_2 : \sigma_2} \quad (\text{LET'})$$

This rule is simple and elegant. Unfortunately, it is incorrect with respect to the call-by-value semantics. The problem shows up when the variable X is unused in expression e_2. For instance, in the expression

$$\lambda x.\mathtt{let}\, X = x + 1 \;\mathtt{in}\; x$$

X receives the type scheme $(x : \mathtt{int}) \Rightarrow \mathtt{int}$, which accounts for the fact that x must be an integer. However, since X is unused in the body of the `let` construct, this information is lost, and the whole expression receives the type scheme $\alpha \rightarrow \alpha$, which is incorrect. To avoid this problem, Trifonov and Smith [46] suggest replacing the construct $\mathtt{let}\, X = e_1 \;\mathtt{in}\; e_2$ with $\mathtt{let}\, X = e_1 \;\mathtt{in}\; (\lambda\_.e_2)\, X$ whenever X does not appear in e_2. Thus, a new occurrence of X is explicitly added, which does not affect the semantics, but leads to a correct typing. This solution is acceptable in practice, but once again, is cumbersome in the subject reduction proofs. So, we prefer to introduce a slightly heavier, but correct, (LET) rule. To explain its formulation, it suffices to notice that it corresponds exactly to what we would obtain if we used (LET'), (SUB), etc. to type the expression $\mathtt{let}\, X = e_1 \;\mathtt{in}\; (\lambda\_.e_2)\, X$.

5.4 "Simple" typing rules

The typing rules introduced above are powerful. In particular, rule (SUB), based on polymorphic scheme subsumption, is very flexible. Paradoxically, this causes a problem when one wishes to prove that the type system is correct with respect to the semantics. Indeed, rule (SUB) is so powerful that it is not trivial to verify its correctness; the problem shows up when trying to prove the β-reduction lemma.

The type inference rules, which we shall introduce soon, are less flexible; in particular, they have no equivalent of rule (SUB). So, one might imagine to prove first that the type inference rules are correct, and then go back from there to the initial system. However, the type inference rules are rather complex, because they must describe the inference algorithm with precision. That would make the correctness proof unnecessarily complex.

So, we now introduce a third set of rules, called "simple", given by figure 5.2 on the next page. They are just as terse as the typing rules, but less powerful, because they do not use polymorphic scheme subsumption.

This new set of rules is based on the following remark. The original (SUB) rule has two main uses: renaming a type scheme and modifying its constraint graph. (A third, and

$$\frac{\mathrm{dom}(A) = \mathrm{dom}_\lambda(\Gamma)}{\Gamma \vdash_{\mathrm{S}} x : A \Rightarrow A(x) \mid C} \quad (\textsc{Var}_{\mathrm{S}})$$

$$\frac{\mathrm{lift}_x(\Gamma); x \vdash_{\mathrm{S}} e : (A; x : \tau) \Rightarrow \tau' \mid C}{\Gamma \vdash_{\mathrm{S}} \lambda x.e : A \Rightarrow (\tau \to \tau') \mid C} \quad (\textsc{Abs}_{\mathrm{S}})$$

$$\frac{\Gamma \vdash_{\mathrm{S}} e_1 : A \Rightarrow (\tau_2 \to \tau) \mid C \qquad \Gamma \vdash_{\mathrm{S}} e_2 : A \Rightarrow \tau_2 \mid C}{\Gamma \vdash_{\mathrm{S}} e_1\, e_2 : A \Rightarrow \tau \mid C} \quad (\textsc{App}_{\mathrm{S}})$$

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ renaming of } \sigma}{\Gamma \vdash_{\mathrm{S}} X : \rho(\sigma)} \quad (\textsc{LetVar}_{\mathrm{S}})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{S}} e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash_{\mathrm{S}} e_2 : A_2 \Rightarrow \tau_2 \mid C_2 \\ \sigma_1 \leq^m A_2 \Rightarrow \top \mid C_2\end{array}}{\Gamma \vdash_{\mathrm{S}} \mathtt{let}\, X = e_1\, \mathtt{in}\ e_2 : A_2 \Rightarrow \tau_2 \mid C_2} \quad (\textsc{Let}_{\mathrm{S}})$$

$$\frac{\Gamma \vdash_{\mathrm{S}} e : \sigma \qquad \sigma \leq^m \sigma'}{\Gamma \vdash_{\mathrm{S}} e : \sigma'} \quad (\textsc{Sub}_{\mathrm{S}})$$

Figure 5.2: “Simple” typing rules

fundamental, use of polymorphic subsumption is simplification of the type scheme; but this is of no interest here since it doesn't affect the set of typable programs.) These two uses can be separated and described in a more elementary way. Renaming can be explicitly made part of the environment access rule, (LetVar$_{\mathrm{S}}$). Modifying the constraint graph can be done with a weakened (Sub$_{\mathrm{S}}$) rule, where polymorphic comparison of type schemes is replaced with a new comparison relation, called *monomorphic*, defined as follows.

Definition 5.8 Monomorphic comparison between type schemes*, denoted by* $\leq^m$*, is defined as follows. Given two type schemes* $\sigma_1 = (A_1 \Rightarrow \tau_1 \mid C_1)$ *and* $\sigma_2 = (A_2 \Rightarrow \tau_2 \mid C_2)$*,* $\sigma_1 \leq^m \sigma_2$ *holds iff*

$$C_2 \Vdash C_1 + (A_1 \Rightarrow \tau_1 \leq A_2 \Rightarrow \tau_2)$$

Lemma 5.1 $\sigma_1 \leq^m \sigma_2$ *implies* $\sigma_1 \leq^\forall \sigma_2$*.*

Proof. Assume $\sigma_1 \leq^m \sigma_2$. Let ρ_2 be a solution of C_2. Then, according to the definition of entailment, ρ_2 also satisfies C_1 and $A_1 \Rightarrow \tau_1 \leq A_2 \Rightarrow \tau_2$. Thus, ρ_2 is a witness to the assertion $\sigma_1 \leq^\forall \sigma_2$. □

Note that rule (Sub$_{\mathrm{S}}$) is identical to the subtyping rule proposed in [42] (except that we now use λ-lifting).

5.5 Type inference rules

The typing rules introduced above do not define an algorithm. First, they are not syntax directed, since rule (Sub) can be applied at any time. Second, rule (App) places sharing

$$\frac{\alpha \notin F \qquad A = \text{single}_{(x,\alpha,\top)}(\Gamma)}{[F]\;\Gamma \vdash_{\mathrm{I}} x : [F \cup \{\alpha\}]\; A \Rightarrow \alpha \mid \varnothing} \quad (\textsc{Var}_{\mathrm{I}})$$

$$\frac{[F]\;\text{lift}_x(\Gamma); x \vdash_{\mathrm{I}} e : [F']\; (A; x : \tau) \Rightarrow \tau' \mid C}{[F]\;\Gamma \vdash_{\mathrm{I}} \lambda x.e : [F']\; A \Rightarrow (\tau \to \tau') \mid C} \quad (\textsc{Abs}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}[F]\;\Gamma \vdash_{\mathrm{I}} e_1 : [F']\; A_1 \Rightarrow \tau_1 \mid C_1 \\ [F']\;\Gamma \vdash_{\mathrm{I}} e_2 : [F'']\; A_2 \Rightarrow \tau_2 \mid C_2 \qquad \alpha \notin F''\end{array}}{\begin{array}{c}[F]\;\Gamma \vdash_{\mathrm{I}} e_1\, e_2 : [F'' \cup \{\alpha\}]\; (A_1 \sqcap A_2) \Rightarrow \alpha \mid C \\ \text{where } C = (C_1 \cup C_2) + (\tau_1 \leq \tau_2 \to \alpha)\end{array}} \quad (\textsc{App}_{\mathrm{I}})$$

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ renaming of } \sigma \qquad \text{rng}(\rho) \cap F = \varnothing}{[F]\;\Gamma \vdash_{\mathrm{I}} X : [F \cup \text{rng}(\rho)]\; \rho(\sigma)} \quad (\textsc{LetVar}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}[F]\;\Gamma \vdash_{\mathrm{I}} e_1 : [F']\; A_1 \Rightarrow \tau_1 \mid C_1 \\ [F']\;\Gamma; X : A_1 \Rightarrow \tau_1 \mid C_1 \vdash_{\mathrm{I}} e_2 : [F'']\; A_2 \Rightarrow \tau_2 \mid C_2\end{array}}{[F]\;\Gamma \vdash_{\mathrm{I}} \texttt{let}\, X = e_1\, \texttt{in}\; e_2 : [F'']\; (A_1 \sqcap A_2) \Rightarrow \tau_2 \mid C_1 \cup C_2} \quad (\textsc{Let}_{\mathrm{I}})$$

Figure 5.3: Type inference rules

constraints between its premises: A, τ_2 and C appear in both premises. This requires making an appropriate choice ahead of time when applying rule (Var).

We now give a set of type inference rules. These are syntax directed, and present no sharing constraints. Furthermore, "fresh variables" are dealt with explicitly. We shall verify that, thanks to these properties, the inference rules directly describe an algorithm.

Note that, although our typing rules are very close to those of Trifonov and Smith [46], our type inference rules exhibit more noticeable differences. Indeed, we formulate them in a more rigorous way, thus directly describing an algorithm. Furthermore, we shall refine them twice, in chapters 6 and 12, so as to preserve certain invariants.

The type inference rules are given in figure 5.3. Let us comment on the differences with the typing rules. (Please ignore the "$[F]$" annotations for the time being.) In short, the only fundamental difference is the disappearance of the subtyping rule, which has been built into the application rule.

- Rule ($\textsc{Var}_{\mathrm{I}}$) places the hypothesis $x : \alpha$ into the context. Under this hypothesis, the expression x has type α. Variables other than x are not used, so they receive type $\top$ in the inferred context.

- The λ-abstraction rule types the abstraction's body in an extended environment. Note that this new environment does not associate any type information to x. On the contrary, the inferred context states which type τ is expected for x. Rule ($\textsc{Abs}_{\mathrm{I}}$) removes this type from the context and moves it to the left of the $\to$ symbol. This rule is essentially identical to (Abs).

- Rule ($\textsc{App}_{\mathrm{I}}$) infers a type scheme separately for each of the two pieces of code that make up the expression. Then, it brings their requirements together by computing the greatest lower bound of their contexts and the union of their constraint graphs (this is simply set-theoretic union, since the graphs share no type variables, as we shall see). Then, it adds the constraint $\tau_1 \leq \tau_2 \rightarrow \alpha$. Why? The type inferred for the left-hand operand is τ_1. On the other hand, this piece of code is passed a value of type τ_2, and is expected to return a value of type α, since α is the type chosen for the whole expression. So, the left-hand operand has type τ_1, but it is used with type $\tau_2 \rightarrow \alpha$. Since we have subtyping, we do not need to require that τ_1 be equal to $\tau_2 \rightarrow \alpha$. Instead, it is enough to require that these types be in the subtyping relation, which we do by adding the appropriate constraint to the constraint graph.

- Rule (LETVAR$_\mathrm{I}$) is essentially identical to rule (LETVAR$_\mathrm{S}$): it fetches the type scheme from the environment, and makes a copy of it to allow polymorphism. The copy operation must be explicitly specified here, just as in the "simple" typing rules, because (SUB) (which allowed renaming at any point) has disappeared.

- Lastly, the typing rule for the `let` construct has been modified in the same way as the application rule: it computes the intersection of the contexts and the union of the constraint graphs of the two branches. In the original (LET) rule, this operation was encoded in the third premise. Recall that this third premise could be naturally obtained by typing the expression $\texttt{let}\, X = e_1 \,\texttt{in}\, (\lambda\_.e_2)\, X$ with a simplified (LET') rule. The same remark applies here: the context intersection and graph union operations essentially come from the (APP$_\mathrm{I}$) rule used when typing the application $(\lambda\_.e_2)\, X$.

Let us now discuss the "$[F]$" annotations. Type inference judgements are of the form

$$[F]\, \Gamma \vdash_\mathrm{I} e : [F']\, \sigma$$

As the reader might have guessed, these annotations are used to handle "fresh" variables explicitly. F and F' are (finite) sets of type variables which have been used already and must not be re-used. F is an input of the type inference algorithm, while F' is an output of it. Thanks to these annotations, the treatment of fresh variables is fully formal. Although they are rather cumbersome, their use is straightforward. Recall that no implicit α-conversion of type schemes is allowed. This is necessary for the annotations to make sense, and it allows the type inference rules to reflect the actual implementation very closely.

Since F is, by hypothesis, the input set of "dirty" type variables, picking a "fresh" variable simply means picking it outside of F; hence the premises of the form $\alpha \notin F$. On the other hand, once a fresh variable α has been used, it must not be re-used elsewhere. So, the new set of "dirty" variables shall be $F \cup \{\alpha\}$.

Note that in the application rule (APP$_\mathrm{I}$), the set of dirty variables F' output by the first premise is passed as input to the second premise. (Rule (LET$_\mathrm{I}$) works in a similar way.) As a consequence, two distinct sub-trees of the inference tree never share any type variables. This property, formalized by lemma 5.2 below, is in contrast with a classic type inference system such as ML's, where the *environment* is an *input* shared by distinct sub-trees. Here, on the contrary, the *context* is an *output* and there is no sharing. This property is essential to the simplification of type schemes, because it allows us to work solely with type schemes bereft of free type variables. Thus, simplification only has to deal with a single, self-contained type scheme and does not have to take an environment of shared type variables into account.

To conclude this description of the type inference rules, let us demonstrate their functioning on an example.

Example. We wish to infer a type scheme for the λ-term $\lambda fx.f\,(f\,x)$, which composes a function f with itself. Since the term is a λ-abstraction with respect to the variables f and x, the derivation shall end with two instances of rule (ABS$_\mathrm{I}$); the program body, that is, the expression $f\,(f\,x)$, shall thus be typed in the environment $\Gamma = (\varnothing; f; x)$.

First, we use ($\textsc{Var}_\mathrm{I}$) to type f's first occurrence:

$$[\varnothing]\ \Gamma \vdash_\mathrm{I} f : [\{v_1\}]\ (\varnothing; f : v_1; x : \top) \Rightarrow v_1 \mid \varnothing$$

Then, we do likewise for f's second occurrence, and for x. We are careful, each time, to keep track of the set of "dirty" variables provided by previous computations; thus, we always obtain fresh variables.

$$[\{v_1\}]\ \Gamma \vdash_\mathrm{I} f : [\{v_1, v_2\}]\ (\varnothing; f : v_2; x : \top) \Rightarrow v_2 \mid \varnothing$$
$$[\{v_1, v_2\}]\ \Gamma \vdash_\mathrm{I} x : [\{v_1, v_2, v_3\}]\ (\varnothing; f : \top; x : v_3) \Rightarrow v_3 \mid \varnothing$$

We can now use rule ($\textsc{App}_\mathrm{I}$) to type the application $(f\,x)$. This generates a first subtyping constraint. (For the sake of brevity, the constraint graph is here represented by a set.)

$$[\{v_1\}]\ \Gamma \vdash_\mathrm{I} (f\,x) : [\{v_1, v_2, v_3, v_4\}]\ (\varnothing; f : v_2; x : v_3) \Rightarrow v_4 \mid \{v_2 \leq v_3 \rightarrow v_4\}$$

We then move on to the second application, which creates a new constraint:

$$\begin{gathered}[\varnothing]\ \Gamma \vdash_\mathrm{I} f\,(f\,x) : [V]\ (\varnothing; f : v_1 \sqcap v_2; x : v_3) \Rightarrow v_5 \mid C \\ \text{where } V = \{v_1, v_2, v_3, v_4, v_5\} \\ \text{and } C = \{v_2 \leq v_3 \rightarrow v_4,\ v_1 \leq v_4 \rightarrow v_5\}\end{gathered}$$

We can now apply rule ($\textsc{Abs}_\mathrm{I}$) twice; it removes the last entry from the context and uses it to create a function type:

$$[\varnothing]\ (\varnothing; f) \vdash_\mathrm{I} \lambda x.f\,(f\,x) : [V]\ (\varnothing; f : v_1 \sqcap v_2) \Rightarrow v_3 \rightarrow v_5 \mid C$$
$$[\varnothing]\ \varnothing \vdash_\mathrm{I} \lambda f x.f\,(f\,x) : [V]\ \varnothing \Rightarrow (v_1 \sqcap v_2) \rightarrow v_3 \rightarrow v_5 \mid C$$

To make sure that the λ-term is well typed, there only remains to verify that the constraint graph C admits a solution, which is immediate, since it is closed. Hence, provided we prove that the typing rules are correct, we are guaranteed that this program cannot cause any runtime errors. Note that the inferred type is not very readable, which shows that simplification is necessary. Once we apply internal (canonization, garbage collection) and external (i.e. reserved for display, see section 15.2) simplification methods, we shall obtain the typing judgement

$$\varnothing \vdash \lambda f x.f\,(f\,x) : (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \mid \{\beta \leq \alpha\}$$

This judgement gives a clear specification of our λ-term: it expects a function which can be composed with itself (as stated by the constraint $\beta \leq \alpha$), and returns a function with the same type.

5.6 Equivalence of the three sets of rules

In this section, we prove that the three type systems presented above are equivalent. This implies, in particular, that they accept the same set of programs. More precisely, we verify that each system is *correct* and *complete* with respect to its predecessors. From this, we deduce that in the original type system, any expression has a *principal type*, which is computed by the inference algorithm.

The equivalence of the three systems stems from three theorems: correctness of the simple typing rules with respect to the typing rules, correctness of the type inference rules with respect to the simple typing rules, and completeness of the type inference rules with respect to the typing rules.

Theorem 5.1 *The simple typing rules are correct with respect to the typing rules; that is,*

$$\Gamma \vdash_{\mathrm{S}} e : \sigma$$

implies

$$\Gamma \vdash e : \sigma$$

Proof. By induction on the structure of the input derivation. There is one case per simple typing rule. Cases ($\textsc{Var}_\mathrm{S}$), ($\textsc{Abs}_\mathrm{S}$) and ($\textsc{App}_\mathrm{S}$) are immediate. For case ($\textsc{LetVar}_\mathrm{S}$), remark that for any renaming ρ, $\sigma \leq^\forall \rho(\sigma)$ holds; hence, the goal can be obtained by applying ($\textsc{Sub}$) immediately after ($\textsc{LetVar}$). Cases ($\textsc{Let}_\mathrm{S}$) and ($\textsc{Sub}_\mathrm{S}$) stem from the fact that $\leq^m$ is included in $\leq^\forall$ (lemma 5.1). □

Theorem 5.2 *The type inference rules are correct with respect to the simple typing rules; that is,*

$$[F]\,\Gamma \vdash_{\mathrm{I}} e : [F']\,\sigma$$

implies

$$\Gamma \vdash_{\mathrm{S}} e : \sigma$$

Proof. By induction on the structure of the input derivation. There is one case per type inference rule. Each case uses exactly the notations of figure 5.3, which allows us to omit the hypothesis.

- ($\textsc{Var}_\mathrm{I}$) It suffices to check that $\mathrm{single}_{(x,\alpha,\top)}(\Gamma)$ has domain $\mathrm{dom}_\lambda(\Gamma)$, and that it maps x to α, which is immediate according to definition 5.6.
- ($\textsc{Abs}_\mathrm{I}$) Immediate, by applying the induction hypothesis to the premise, then applying ($\textsc{Abs}_\mathrm{S}$).
- ($\textsc{App}_\mathrm{I}$) By applying the induction hypothesis to each of the premises, we obtain $\Gamma \vdash_{\mathrm{S}} e_1 : A_1 \Rightarrow \tau_1 \mid C_1$ and $\Gamma \vdash_{\mathrm{S}} e_2 : A_2 \Rightarrow \tau_2 \mid C_2$. For the sake of brevity, define $A = A_1 \sqcap A_2$ and $C = (C_1 \cup C_2) + (\tau_1 \leq \tau_2 \rightarrow \alpha)$. Then, it is easy to verify that

  $$\begin{array}{l} A_1 \Rightarrow \tau_1 \mid C_1 \leq^m A \Rightarrow \tau_2 \rightarrow \alpha \mid C \\ A_2 \Rightarrow \tau_2 \mid C_2 \leq^m A \Rightarrow \tau_2 \mid C \end{array}$$

  This allows us to apply ($\textsc{Sub}_\mathrm{S}$) twice:

  $$\frac{\Gamma \vdash_{\mathrm{S}} e_1 : A_1 \Rightarrow \tau_1 \mid C_1}{\Gamma \vdash_{\mathrm{S}} e_1 : A \Rightarrow \tau_2 \rightarrow \alpha \mid C}\ (\textsc{Sub}_\mathrm{S})$$

  $$\frac{\Gamma \vdash_{\mathrm{S}} e_2 : A_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash_{\mathrm{S}} e_2 : A \Rightarrow \tau_2 \mid C}\ (\textsc{Sub}_\mathrm{S})$$

  We can then conclude with

  $$\frac{\Gamma \vdash_{\mathrm{S}} e_1 : A \Rightarrow \tau_2 \rightarrow \alpha \mid C \qquad \Gamma \vdash_{\mathrm{S}} e_2 : A \Rightarrow \tau_2 \mid C}{\Gamma \vdash_{\mathrm{S}} e_1\, e_2 : A \Rightarrow \alpha \mid C}\ (\textsc{App}_\mathrm{S})$$

- ($\textsc{LetVar}_\mathrm{I}$) Immediate.
- ($\textsc{Let}_\mathrm{I}$) Similar to case ($\textsc{App}_\mathrm{I}$). □

Before proving the completeness theorem, we must introduce a few technical lemmas. The first one formalizes the use of F and F' to deal with fresh variables.

Lemma 5.2 *If* $[F]\,\Gamma \vdash_{\mathrm{I}} e : [F']\,\sigma$*, then* $\mathrm{fv}(\sigma) \subseteq F' \setminus F$.

Proof. Straightforward induction on the structure of the input derivation. □

The following lemma states that if an expression is typable in a certain environment, then it is *a fortiori* typable in any finer environment.

Lemma 5.3 *If* $\Gamma \vdash e : \sigma$ *and* $\Gamma' \leq^{\forall} \Gamma$*, then* $\Gamma' \vdash e : \sigma$.

Proof. ($\Gamma' \leq^{\forall} \Gamma$ is point-wise comparison between environments with identical domains.) By induction on the structure of the input derivation. Let us treat one case explicitly:

- (LetVar) We have $\Gamma(X) = \sigma$. Since $\Gamma' \leq^{\forall} \Gamma$, this implies $\Gamma'(X) = \sigma'$ where $\sigma' \leq^{\forall} \sigma$. We can write

  $$\frac{\Gamma'(X) = \sigma'}{\Gamma' \vdash X : \sigma'}\ (\textsc{LetVar})$$

  and conclude with

  $$\frac{\Gamma' \vdash X : \sigma' \qquad \sigma' \leq^{\forall} \sigma}{\Gamma' \vdash X : \sigma}\ (\textsc{Sub})$$

The other cases are immediate. □

We can now establish the main theorem:

Theorem 5.3 *The type inference rules are complete with respect to the typing rules. That is, if*

$$\Gamma \vdash e : \sigma$$

then, for any finite $F \subseteq \mathcal{V}$*, there exists a finite* $F' \subseteq \mathcal{V}$ *and a type scheme* $\sigma' \leq^{\forall} \sigma$ *such that*

$$[F]\,\Gamma \vdash_{\mathrm{I}} e : [F']\,\sigma'$$

Proof. By induction on the structure of the input derivation. There is one case per typing rule. Each case uses exactly the notations of figure 5.1, which allows us to omit the hypothesis.

- (Var) Let $\alpha \notin F$; define $F' = F \cup \{\alpha\}$, $A' = \mathrm{single}_{(x,\alpha,\top)}(\Gamma)$ and $\sigma' = A' \Rightarrow \alpha \mid \varnothing$. Then

  $$\frac{\alpha \notin F \qquad A' = \mathrm{single}_{(x,\alpha,\top)}(\Gamma)}{[F]\,\Gamma \vdash_{\mathrm{I}} x : [F']\,\sigma'}\ (\textsc{Var}_{\mathrm{I}})$$

  There remains to verify that $\sigma' \leq^{\forall} A \Rightarrow A(x) \mid C$. According to the definition of scheme subsumption, and to definition 5.6, this assertion is equivalent to: for any solution ρ of C, there exists a ground substitution ρ', of domain $\{\alpha\}$, such that $\rho(A(x)) \leq \rho'(\alpha)$ and $\rho'(\alpha) \leq \rho(A(x))$. This assertion holds; the witness is $\rho' : \alpha \mapsto \rho(A(x))$.

- (Abs) By applying the induction hypothesis to the premise, we obtain

  $$[F]\,\mathrm{lift}_x(\Gamma); x \vdash_{\mathrm{I}} e : [F']\,\sigma_0$$

where $\sigma_0 \leq^\forall (A; x : \tau) \Rightarrow \tau' \mid C$. Thus, σ_0 is necessarily of the form $(A_0; x : \tau_0) \Rightarrow \tau'_0 \mid C_0$. We can write

$$\frac{[F]\ \mathrm{lift}_x(\Gamma); x \vdash_{\mathrm{I}} e : [F']\ (A_0; x : \tau_0) \Rightarrow \tau'_0 \mid C_0}{[F]\ \Gamma \vdash_{\mathrm{I}} \lambda x.e : [F']\ A_0 \Rightarrow \tau_0 \rightarrow \tau'_0 \mid C_0}\ (\textsc{Abs}_{\mathrm{I}})$$

There remains to check that

$$A_0 \Rightarrow \tau_0 \rightarrow \tau'_0 \mid C_0 \leq^\forall A \Rightarrow \tau \rightarrow \tau' \mid C$$

which is a straightforward consequence of our hypothesis

$$(A_0; x : \tau_0) \Rightarrow \tau'_0 \mid C_0 \leq^\forall (A; x : \tau) \Rightarrow \tau' \mid C$$

- (App) By applying the induction hypothesis to the first premise, we obtain $[F]\ \Gamma \vdash_{\mathrm{I}} e_1 : [F']\ \sigma'_1$ where $\sigma'_1 \leq^\forall A \Rightarrow \tau_2 \rightarrow \tau \mid C$. By applying the induction hypothesis to the second premise (this time with F' as input, in lieu of F), we obtain $[F']\ \Gamma \vdash_{\mathrm{I}} e_2 : [F'']\ \sigma'_2$ where $\sigma'_2 \leq^\forall A \Rightarrow \tau_2 \mid C$. For $i \in \{1, 2\}$, the type scheme σ'_i is of the form $A'_i \Rightarrow \tau'_i \mid C'_i$. Let $\alpha \notin F''$. We can write

$$\frac{[F]\ \Gamma \vdash_{\mathrm{I}} e_1 : [F']\ \sigma'_1 \qquad [F']\ \Gamma \vdash_{\mathrm{I}} e_2 : [F'']\ \sigma'_2 \qquad \alpha \notin F''}{[F]\ \Gamma \vdash_{\mathrm{I}} e_1\, e_2 : [F'' \cup \{\alpha\}]\ A' \Rightarrow \alpha \mid C'}\ (\textsc{App}_{\mathrm{I}})$$

where $A' = A'_1 \sqcap A'_2$ and $C' = (C'_1 \cup C'_2) + (\tau'_1 \leq \tau'_2 \rightarrow \alpha)$. There remains to show that

$$A' \Rightarrow \alpha \mid C' \leq^\forall A \Rightarrow \tau \mid C$$

Let ρ be a solution of C. According to the two subsumption assertions established above, there exist solutions ρ'_i of C'_i, for $i \in \{1, 2\}$, such that

$$\rho'_1(A'_1 \Rightarrow \tau'_1) \leq \rho(A \Rightarrow \tau_2 \rightarrow \tau)$$
$$\rho'_2(A'_2 \Rightarrow \tau'_2) \leq \rho(A \Rightarrow \tau_2)$$

Now, two consecutive applications of lemma 5.2 show that $\mathrm{fv}(\sigma_1)$, $\mathrm{fv}(\sigma_2)$ and $\{\alpha\}$ are pairwise disjoint. Consequently, there exists a ground substitution ρ', which coincides with ρ'_i on $\mathrm{fv}(\sigma_i)$ for $i \in \{1, 2\}$, and whose value on α is $\rho(\tau)$. We shall now verify that ρ' is the desired witness, i.e. ρ' is a solution of C' and

$$\rho'(A' \Rightarrow \alpha) \leq \rho(A \Rightarrow \tau)$$

For $i \in \{1, 2\}$, ρ' satisfies C'_i because it coincides with ρ'_i on $\mathrm{fv}(\sigma_i)$. Furthermore, we have $\rho'_1(\tau'_1) \leq \rho(\tau_2 \rightarrow \tau)$ and $\rho'_2(\tau'_2) \leq \rho(\tau_2)$. By combining these inequations, we obtain $\rho'_1(\tau'_1) \leq \rho'_2(\tau'_2) \rightarrow \rho(\tau)$. Since $\rho(\tau) = \rho'(\alpha)$, this is precisely $\rho'(\tau'_1) \leq \rho'(\tau'_2 \rightarrow \alpha)$, which means that ρ' satisfies $\tau'_1 \leq \tau'_2 \rightarrow \alpha$. We have shown that ρ' is a solution of C'.

We have $\rho(A) \leq \rho'_1(A'_1)$ and $\rho(A) \leq \rho'_2(A'_2)$. These can be rewritten as $\rho(A) \leq \rho'(A'_1)$ and $\rho(A) \leq \rho'(A'_2)$, which imply $\rho(A) \leq \rho'(A')$, since $A' = A'_1 \sqcap A'_2$. Finally, $\rho'(\alpha) \leq \rho(\tau)$ is satisfied by definition of ρ'.

- (LetVar) We have $\Gamma(X) = \sigma$. Let ρ be a renaming of σ such that $\mathrm{rng}(\rho) \cap F = \varnothing$. Then

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ renaming of } \sigma \qquad \mathrm{rng}(\rho) \cap F = \varnothing}{[F]\ \Gamma \vdash_{\mathrm{I}} X : [F \cup \mathrm{rng}(\rho)]\ \rho(\sigma)}\ (\textsc{LetVar}_{\mathrm{I}})$$

There remains to verify $\rho(\sigma) \leq^\forall \sigma$, which is immediate since α-conversion is contained within the scheme subsumption relation.

- (LET) By applying the induction hypothesis to the first premise, we obtain $[F]\,\Gamma \vdash_{\mathrm{I}} e_1 : [F']\,\sigma_1'$, where $\sigma_1' \leq^{\forall} \sigma_1$. By applying lemma 5.3 to the second premise, we obtain $\Gamma; X : \sigma_1' \vdash e_2 : A_2 \Rightarrow \tau_2 \mid C_2$. By applying the induction hypothesis to this judgement, we obtain $[F']\,\Gamma; X : \sigma_1' \vdash_{\mathrm{I}} e_2 : [F'']\,\sigma_2'$, where $\sigma_2' \leq^{\forall} A_2 \Rightarrow \tau_2 \mid C_2$. We can then write

$$\frac{[F]\,\Gamma \vdash_{\mathrm{I}} e_1 : [F']\,\sigma_1' \qquad [F']\,\Gamma; X : \sigma_1' \vdash_{\mathrm{I}} e_2 : [F'']\,\sigma_2'}{[F]\,\Gamma \vdash_{\mathrm{I}} \mathtt{let}\, X = e_1\, \mathtt{in}\ e_2 : [F'']\,(A_1' \sqcap A_2') \Rightarrow \tau_2' \mid C_1' \cup C_2'}\ (\textsc{Let}_{\mathrm{I}})$$

where $\sigma_i' = (A_i' \Rightarrow \tau_i' \mid C_i')$ for $i \in \{1, 2\}$. There remains to verify that

$$(A_1' \sqcap A_2') \Rightarrow \tau_2' \mid C_1' \cup C_2' \leq^{\forall} A_2 \Rightarrow \tau_2 \mid C_2$$

From $\sigma_1' \leq^{\forall} \sigma_1$ and from the third premise, we deduce, by transitivity of $\leq^{\forall}$, that $\sigma_1' \leq^{\forall} A_2 \Rightarrow \top \mid C_2$. Besides, recall that $\sigma_2' \leq^{\forall} A_2 \Rightarrow \tau_2 \mid C_2$. The result follows straightforwardly (as in the case of (APP), lemma 5.2 must be applied).

- (SUB) Applying the induction hypothesis yields $[F]\,\Gamma \vdash_{\mathrm{I}} e : [F']\,\sigma''$, where $\sigma'' \leq^{\forall} \sigma$. Besides, we have $\sigma \leq^{\forall} \sigma'$, whence $\sigma'' \leq^{\forall} \sigma'$ by transitivity of $\leq^{\forall}$. □

The completeness theorem indicates that if a program is typable and has a typing σ, then there exists a type inference derivation which yields a finer typing. This result can be strengthened by noticing that the type inference rules are deterministic, i.e. any two derivations (for the same program) yield equivalent type schemes:

Lemma 5.4 *If $[F_1]\,\Gamma \vdash_{\mathrm{I}} e : [F_1']\,\sigma_1$ and $[F_2]\,\Gamma \vdash_{\mathrm{I}} e : [F_2']\,\sigma_2$, then $\sigma_1 =^{\forall} \sigma_2$.*

Proof. Left to the reader. □

Here is now the enhanced theorem:

Theorem 5.4 *Let Γ be an environment and e an expression. Then,*

- *either e is not typable in Γ. Then there exists no type inference derivation for e in Γ.*
- *or e is typable in Γ. Then there exists a type inference derivation for e in Γ; furthermore, any such derivation yields a type scheme which is optimal for e in Γ, according to relation $\leq^{\forall}$.*

Proof. If e isn't typable, then no type inference derivation exists, otherwise the type inference rules would be not correct with respect to the typing rules (they are, according to the combination of theorems 5.1 and 5.2).

Otherwise, we have $\Gamma \vdash e : \sigma$ for a certain type scheme σ. According to theorem 5.3, there exists a type inference derivation

$$[\varnothing]\,\Gamma \vdash_{\mathrm{I}} e : [F']\,\sigma'$$

where $\sigma' \leq^{\forall} \sigma$.

Imagine σ'' is such that $\Gamma \vdash e : \sigma''$. Then, the above reasoning can be applied to σ'' as well. However, according to lemma 5.4, all type inference derivations for e in Γ yield type schemes which are equivalent to σ'; it follows that $\sigma' \leq^{\forall} \sigma''$. Hence, σ' is an optimal type scheme for e in Γ. □

This last theorem shows that the type inference rules directly describe a syntax-directed inference algorithm. The algorithm rejects the program if it is not typable, and produces an optimal typing otherwise.

5.7 Safety of the type system

There remains to show that the type system is safe, i.e. that a well typed program cannot cause execution errors. We shall first give a formal definition of execution by supplying an operational semantics for our language. Then, we shall show that reduction preserves typing, and deduce that the type system is safe.

5.7.1 Operational semantics

A small-step operational semantics simply consists in a syntactic description of program execution, that is, a set of rewriting rules on programs. The two fundamental rules are reduction rules for β and `let` redexes. Then, a third rule allows reduction under a context, and at the same time specifies the evaluation strategy; we use a call-by-value strategy.

Definition 5.9 Values *form a subset of irreducible expressions, defined by*

$$v ::= \lambda x.e$$

Since our language is restricted to pure λ-calculus, one notes that irreducible expressions without free variables coincide with values. This isn't true in richer languages; for instance, if integer constants are added to the language, then the expression $2\,(\lambda x.x)$ is irreducible, but it isn't a value; it corresponds to an execution error. In our restricted language, no execution error is possible. However, this does not affect the subject reduction proof, so we did not deem it necessary to enrich the language. The addition of constants to the language shall be briefly discussed when proving the system's safety (see theorem 5.6).

Definition 5.10 *The* capture-free substitution *of an expression e' for a variable x in an expression e, denoted by $[e'/x]\,e$, is defined by*

$$\begin{aligned}
[e'/x]\,x &= e' \\
[e'/x]\,y &= y \\
[e'/x]\,X &= X \\
[e'/x]\,\lambda x.e &= \lambda x.e \\
[e'/x]\,\lambda y.e &= \lambda y.[e'/x]\,e \\
[e'/x]\,(e_1\,e_2) &= ([e'/x]\,e_1)\,([e'/x]\,e_2) \\
[e'/x]\,(\mathtt{let}\,X = e_1\,\mathtt{in}\;e_2) &= \mathtt{let}\,X = [e'/x]\,e_1\,\mathtt{in}\;[e'/x]\,e_2
\end{aligned}$$

Similarly, the capture-free substitution of an expression e' for a variable X in an expression e, denoted by $[e'/X]\,e$, is defined by

$$\begin{aligned}
[e'/X]\,x &= x \\
[e'/X]\,X &= e' \\
[e'/X]\,Y &= Y \\
[e'/X]\,\lambda x.e &= \lambda x.[e'/X]\,e \\
[e'/X]\,(e_1\,e_2) &= ([e'/X]\,e_1)\,([e'/X]\,e_2) \\
[e'/X]\,(\mathtt{let}\,X = e_1\,\mathtt{in}\;e_2) &= \mathtt{let}\,X = [e'/X]\,e_1\,\mathtt{in}\;e_2 \\
[e'/X]\,(\mathtt{let}\,Y = e_1\,\mathtt{in}\;e_2) &= \mathtt{let}\,Y = [e'/X]\,e_1\,\mathtt{in}\;[e'/X]\,e_2
\end{aligned}$$

Definition 5.11 *One defines the following fundamental* reduction rules*:*

$$\begin{aligned}
(\lambda x.e)\,v &\longrightarrow [v/x]\,e \\
\mathtt{let}\,X = v\,\mathtt{in}\;e &\longrightarrow [v/X]\,e
\end{aligned}$$

to which a third rule, called reduction under a context, *is added:*

$$e \longrightarrow e' \Rightarrow E[e] \longrightarrow E[e']$$

Reduction contexts *are defined so as to reflect the call-by-value evaluation strategy:*

$$E ::= [] \mid E\, e \mid v\, E \mid \texttt{let}\, X = E\, \texttt{in}\ e$$

5.7.2 Stability of typing by reduction

The goal of this section is to prove that typing is preserved by reduction. This is an essentially technical section. We shall conclude and comment the result in the next section.

Let us first give, without demonstration, a very simple closure lemma, which states that closed terms can be typed independently of the environment.

Lemma 5.5 *Let e be a closed term (i.e. without any free variables). Let Γ, A, τ, C be such that* $\mathrm{dom}(A) = \mathrm{dom}_\lambda(\Gamma)$. *Then*

$$\Gamma \vdash_{\mathrm{S}} e : A \Rightarrow \tau \mid C \iff \varnothing \vdash_{\mathrm{S}} e : \varnothing \Rightarrow \tau \mid C$$

We shall now establish a lemma regarding the stability of typing through one step of β-reduction. Because we are interested in the "simple" type system, which has a weakened subtyping rule, this result is relatively straightforward. It is indeed classic; a similar statement can be found in [15], although bereft of a proof.

Lemma 5.6 *Let e be an expression and v be a value such that $\Gamma \vdash_{\mathrm{S}} e : A \Rightarrow \tau \mid C$ and $\varnothing \vdash_{\mathrm{S}} v : \varnothing \Rightarrow \tau_{\mathrm{in}} \mid C_{\mathrm{in}}$. Let $x \in \mathrm{dom}(A)$. One assumes $C \Vdash C_{\mathrm{in}} + (\tau_{\mathrm{in}} \leq A(x))$. Then*

$$\mathrm{unlift}_x(\Gamma) \setminus x \vdash_{\mathrm{S}} [v/x]\, e : (A \setminus x) \Rightarrow \tau \mid C$$

Proof. By induction on the structure of e's typing derivation. Thus, there is one case per "simple" typing rule. For the sake of brevity, define $\Gamma' = \mathrm{unlift}_x(\Gamma) \setminus x$.

- ($\textsc{Var}_{\mathrm{S}}$) The expression is a variable. There are two cases to consider, depending on whether this variable is x.
  - $e = x$. Then $\tau = A(x)$ and $[v/x]\, e = v$. According to lemma 5.5, we have

    $$\Gamma' \vdash_{\mathrm{S}} v : (A \setminus x) \Rightarrow \tau_{\mathrm{in}} \mid C_{\mathrm{in}}$$

    whence one can deduce, using ($\textsc{Sub}_{\mathrm{S}}$),

    $$\Gamma' \vdash_{\mathrm{S}} v : (A \setminus x) \Rightarrow \tau \mid C$$

  - $e = y \neq x$. Then $\tau = A(y) = (A \setminus x)(y)$, so rule ($\textsc{Var}_{\mathrm{S}}$) yields $\Gamma' \vdash_{\mathrm{S}} y : (A \setminus x) \Rightarrow \tau \mid C$.

- ($\textsc{Abs}_{\mathrm{S}}$) The expression e is of the form $\lambda y.a$, and its typing derivation ends with

  $$\frac{\mathrm{lift}_y(\Gamma); y \vdash_{\mathrm{S}} a : (A; y : \tau_0) \Rightarrow \tau_1 \mid C}{\Gamma \vdash_{\mathrm{S}} \lambda y.a : A \Rightarrow \tau \mid C}\ (\textsc{Abs}_{\mathrm{S}})$$

  where $\tau = \tau_0 \rightarrow \tau_1$. There are again two cases, depending on whether y is equal to x.

– $y = x$. Thus, the variable x appears twice in the environment $\mathrm{lift}_x(\Gamma); x$, in which the premise was typed. So, the first occurrence is useless. A capture lemma (a close cousin of lemma 5.5's) allows us to rewrite the premise

$$\mathrm{lift}_x(\mathrm{unlift}_x(\Gamma) \setminus x); x \vdash_{\mathrm{S}} a : (A \setminus x; x : \tau_0) \Rightarrow \tau_1 \mid C$$

from which we can deduce, thanks to rule ($\textsc{Abs}_{\mathrm{S}}$),

$$\Gamma' \vdash_{\mathrm{S}} \lambda x.a : (A \setminus x) \Rightarrow \tau \mid C$$

– $y \neq x$. Applying the induction hypothesis to the premise yields

$$\mathrm{unlift}_x(\mathrm{lift}_y(\Gamma); y) \setminus x \vdash_{\mathrm{S}} [v/x]\, a : ((A; y : \tau_0) \setminus x) \Rightarrow \tau_1 \mid C$$

Slightly rewritten, this assertion allows us to apply ($\textsc{Abs}_{\mathrm{S}}$) again:

$$\frac{\mathrm{lift}_y(\Gamma'); y \vdash_{\mathrm{S}} [v/x]\, a : ((A \setminus x); y : \tau_0) \Rightarrow \tau_1 \mid C}{\Gamma' \vdash_{\mathrm{S}} [v/x]\, e : (A \setminus x) \Rightarrow \tau \mid C} \; (\textsc{Abs}_{\mathrm{S}})$$

- ($\textsc{App}_{\mathrm{S}}$) The expression e is of the form $e_1\, e_2$, and its typing derivation ends with

$$\frac{\Gamma \vdash_{\mathrm{S}} e_1 : A \Rightarrow \tau_2 \to \tau \mid C \qquad \Gamma \vdash_{\mathrm{S}} e_2 : A \Rightarrow \tau_2 \mid C}{\Gamma \vdash_{\mathrm{S}} e : A \Rightarrow \tau \mid C} \; (\textsc{App}_{\mathrm{S}})$$

Let us apply the hypothesis induction to each premise. We can then write

$$\frac{\begin{array}{c}\Gamma' \vdash_{\mathrm{S}} [v/x]\, e_1 : (A \setminus x) \Rightarrow \tau_2 \to \tau \mid C \\ \Gamma' \vdash_{\mathrm{S}} [v/x]\, e_2 : (A \setminus x) \Rightarrow \tau_2 \mid C\end{array}}{\Gamma' \vdash_{\mathrm{S}} [v/x]\, e : (A \setminus x) \Rightarrow \tau \mid C} \; (\textsc{App}_{\mathrm{S}})$$

- ($\textsc{LetVar}_{\mathrm{S}}$) e is a variable X. The type scheme $A \Rightarrow \tau \mid C$ can then be written $\rho(\Gamma(X))$ for a certain renaming ρ. By definition of Γ', this implies that $\rho(\Gamma'(X))$ is precisely $(A \setminus x) \Rightarrow \tau \mid C$. Thus, one can use ($\textsc{LetVar}_{\mathrm{S}}$) to obtain $\Gamma' \vdash_{\mathrm{S}} X : (A \setminus x) \Rightarrow \tau \mid C$.

- ($\textsc{Let}_{\mathrm{S}}$) The expression e is of the form $\texttt{let}\, X = e_1 \,\texttt{in}\, e_2$, and its typing derivation ends with

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{S}} e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash_{\mathrm{S}} e_2 : \sigma_2 \\ \sigma_1 \leq^m A \Rightarrow \top \mid C\end{array}}{\Gamma \vdash_{\mathrm{S}} e : \sigma_2} \; (\textsc{Let}_{\mathrm{S}})$$

where $\sigma_2 = A \Rightarrow \tau \mid C$. Let us write $\sigma_1 = A_1 \Rightarrow \tau_1 \mid C_1$. From $\sigma_1 \leq^m A \Rightarrow \top \mid C$, we deduce $C \Vdash A(x) \leq A_1(x)$. Together with $C \Vdash C_{\mathrm{in}} + (\tau_{\mathrm{in}} \leq A(x))$, this implies $C \Vdash C_{\mathrm{in}} + (\tau_{\mathrm{in}} \leq A_1(x))$. Thus, we can apply the induction hypothesis to the first premise, which yields

$$\Gamma' \vdash_{\mathrm{S}} [v/x]\, e_1 : (A_1 \setminus x) \Rightarrow \tau_1 \mid C_1$$

Now, let us apply the induction hypothesis to the second premise. We find

$$\mathrm{unlift}_x(\Gamma; X : \sigma_1) \setminus x \vdash_{\mathrm{S}} [v/x]\, e_2 : (A \setminus x) \Rightarrow \tau \mid C$$

Note that the environment used in this assertion is precisely $\Gamma'; X : (A_1 \setminus x) \Rightarrow \tau_1 \mid C_1$. We can now conclude:

$$\frac{\begin{array}{c}\Gamma' \vdash_{\mathrm{S}} [v/x]\, e_1 : (A_1 \setminus x) \Rightarrow \tau_1 \mid C_1 \\ \Gamma'; X : (A_1 \setminus x) \Rightarrow \tau_1 \mid C_1 \vdash_{\mathrm{S}} [v/x]\, e_2 : (A \setminus x) \Rightarrow \tau \mid C \\ (A_1 \setminus x) \Rightarrow \tau_1 \mid C_1 \leq^m (A \setminus x) \Rightarrow \top \mid C\end{array}}{\Gamma' \vdash_{\mathrm{S}} [v/x]\, e : (A \setminus x) \Rightarrow \tau \mid C} \; (\textsc{Let}_{\mathrm{S}})$$

- ($\textsc{Sub}_\textsc{S}$) e's typing derivation ends with

$$\frac{\Gamma \vdash_{\mathrm{S}} e : A' \Rightarrow \tau' \mid C' \qquad A' \Rightarrow \tau' \mid C' \leq^m A \Rightarrow \tau \mid C}{\Gamma \vdash_{\mathrm{S}} e : A \Rightarrow \tau \mid C} \; (\textsc{Sub}_\textsc{S})$$

The second premise, together with the hypothesis $C \Vdash C_{\text{in}} + (\tau_{\text{in}} \leq A(x))$, implies $C \Vdash C_{\text{in}} + (\tau_{\text{in}} \leq A'(x))$. So, we can apply the induction hypothesis to the first premise. We shall then conclude as follows:

$$\frac{\begin{array}{c}\Gamma' \vdash_{\mathrm{S}} [v/x]\,e : (A' \setminus x) \Rightarrow \tau' \mid C' \\ (A' \setminus x) \Rightarrow \tau' \mid C' \leq^m (A \setminus x) \Rightarrow \tau \mid C\end{array}}{\Gamma' \vdash_{\mathrm{S}} [v/x]\,e : (A \setminus x) \Rightarrow \tau \mid C} \; (\textsc{Sub}_\textsc{S})$$

This ends the β-reduction lemma. □

We now wish to establish a similar result with respect to `let`-reduction. We begin with a lemma which allows renaming type variables. The statement is similar to the one found in the theory of ML, but the proof is almost trivial here, thanks for our simpler treatment of quantification.

(Note that this lemma is immediate for the typing rules, since polymorphic scheme subsumption contains renamings. However, we are interested here in the so-called "simple" typing rules, which are less powerful. A proof by induction is then necessary.)

Lemma 5.7 *Assume $\Gamma \vdash_{\mathrm{S}} e : \sigma$. Let ρ be a renaming of Γ and of σ. Then*

$$\rho(\Gamma) \vdash_{\mathrm{S}} e : \rho(\sigma)$$

Proof. By induction on the structure of the input derivation. Thus, there is one case per "simple" typing rule.

- ($\textsc{Var}_\textsc{S}$) The expression e is a variable x, and σ is of the form $A \Rightarrow A(x) \mid C$. Applying ($\textsc{Var}_\textsc{S}$) yields $\rho(\Gamma) \vdash_{\mathrm{S}} e : \rho(A) \Rightarrow \rho(A(x)) \mid \rho(C)$.

- ($\textsc{LetVar}_\textsc{S}$) The expression e is a variable X, and σ is of the form $\psi(\Gamma(X))$ where ψ is a renaming. Then, $\rho(\sigma)$ can be written $\rho\psi\rho^{-1}(\rho\Gamma(X))$. $\rho\psi\rho^{-1}$ is a renaming of $\rho\Gamma(X)$, so rule ($\textsc{LetVar}_\textsc{S}$) can be applied and yields $\rho(\Gamma) \vdash_{\mathrm{S}} e : \rho(\sigma)$.

- ($\textsc{Let}_\textsc{S}$) This case is dealt with by simple application of the induction hypothesis. We shall however expose it in detail to emphasize the difference with the case of the ML language.

  e's typing derivation ends with a rule of the form

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{S}} e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash_{\mathrm{S}} e_2 : \sigma \\ \sigma_1 \leq^m A \Rightarrow \top \mid C\end{array}}{\Gamma \vdash_{\mathrm{S}} e : \sigma} \; (\textsc{Let}_\textsc{S})$$

  where $\sigma = A \Rightarrow \tau \mid C$. ρ might be (partly) undefined on $\mathrm{fv}(\sigma_1)$; we then extend it so that it becomes a renaming defined on all of $\mathrm{fv}(\sigma_1)$. Next, let us apply the induction hypothesis to the first premise; we obtain

$$\rho(\Gamma) \vdash_{\mathrm{S}} e_1 : \rho(\sigma_1)$$

Reiterate with the second premise:

$$\rho(\Gamma); X : \rho(\sigma_1) \vdash_{\mathrm{S}} e_2 : \rho(\sigma)$$

Finally, from the third premise, we easily deduce

$$\rho(\sigma_1) \leq^m \rho(A \Rightarrow \top \mid C)$$

Having established these three assertions, we can apply ($\textsc{Let}_{\mathrm{S}}$), and we obtain the expected result: $\rho(\Gamma) \vdash_{\mathrm{S}} e : \rho(\sigma)$.

- Cases ($\textsc{Abs}_{\mathrm{S}}$), ($\textsc{App}_{\mathrm{S}}$) and ($\textsc{Sub}_{\mathrm{S}}$) are obtained, similarly, by applying the induction hypothesis. □

We can now prove that typing is preserved by `let`-reduction:

Lemma 5.8 *Let e be an expression and v be a value such that $\Gamma_1; X : \sigma_X; \Gamma_2 \vdash_{\mathrm{S}} e : \sigma$ and $\varnothing \vdash_{\mathrm{S}} v : \varnothing \Rightarrow \tau_X \mid C_X$, where $\sigma_X = A_X \Rightarrow \tau_X \mid C_X$ and $X \notin \mathrm{dom}(\Gamma_2)$. Then*

$$\Gamma_1; \Gamma_2 \vdash_{\mathrm{S}} [v/X]\, e : \sigma$$

Proof. By induction on e's typing derivation. The non-trivial cases are:

- ($\textsc{LetVar}_{\mathrm{S}}$) There are two cases, depending on whether e is precisely X. Only the first case is worthy of interest; thus, let us assume $e = X$. Then $\sigma = \rho(\sigma_X)$ for a certain renaming ρ. Besides, $[v/X]\, e$ is equal to v. By applying lemma 5.7 to our second hypothesis, we find $\varnothing \vdash_{\mathrm{S}} v : \varnothing \Rightarrow \rho(\tau_X) \mid \rho(C_X)$. Lemma 5.5 then yields $\Gamma_1; \Gamma_2 \vdash_{\mathrm{S}} v : \rho(A_X) \Rightarrow \rho(\tau_X) \mid \rho(C_X)$ which is precisely the expected result.

- ($\textsc{Let}_{\mathrm{S}}$) The expression e is of the form $\texttt{let}\, Y = e_1 \,\texttt{in}\, e_2$. Its typing derivation ends with a rule of the form

$$\frac{\Gamma_1; X : \sigma_X; \Gamma_2 \vdash_{\mathrm{S}} e_1 : \sigma_1 \qquad \Gamma_1; X : \sigma_X; \Gamma_2; Y : \sigma_1 \vdash_{\mathrm{S}} e_2 : \sigma \\ \sigma_1 \leq^m A \Rightarrow \top \mid C}{\Gamma_1; X : \sigma_X; \Gamma_2 \vdash_{\mathrm{S}} e : \sigma} \ (\textsc{Let}_{\mathrm{S}})$$

  where $\sigma = A \Rightarrow \tau \mid C$. Applying the induction hypothesis to the first premise yields $\Gamma_1; \Gamma_2 \vdash_{\mathrm{S}} [v/X]\, e_1 : \sigma_1$. We now have to distinguish two cases:

  - $Y = X$. Then the environment which appears in the second premise above contains two occurrences of X. According to a capture lemma (close to lemma 5.5), the first one is superfluous, and we have $\Gamma_1; \Gamma_2; X : \sigma_1 \vdash_{\mathrm{S}} e_2 : \sigma$. We can then conclude by applying ($\textsc{Let}_{\mathrm{S}}$):

$$\Gamma_1; \Gamma_2 \vdash_{\mathrm{S}} \texttt{let}\, X = [v/X]\, e_1 \,\texttt{in}\, e_2 : \sigma$$

  - $Y \neq X$. We can then apply the induction hypothesis to the second premise, which yields $\Gamma_1; \Gamma_2; Y : \sigma_1 \vdash_{\mathrm{S}} [v/X]\, e_2 : \sigma$. Then, we apply ($\textsc{Let}_{\mathrm{S}}$) and obtain

$$\Gamma_1; \Gamma_2 \vdash_{\mathrm{S}} \texttt{let}\, Y = [v/X]\, e_1 \,\texttt{in}\, [v/X]\, e_2 : \sigma$$

This ends the `let`-reduction lemma. □

Equipped with the preceding lemmas, we are now able to show that "simple" typing is preserved by reduction.

Lemma 5.9 *If* $\Gamma \vdash_S e : \sigma$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash_S e' : \sigma$.

Proof. By induction on the structure of the derivation of the assertion $e \longrightarrow e'$. In each of the cases below, we shall assume that e's typing derivation does not end with (SUB$_S$); the extension to the general case is then immediate.

- $e = (\lambda x.a)\, v$ and $e' = [v/x]\, a$. We have mentioned that e's typing derivation does not end with rule (SUB$_S$). Thus, it is necessarily of the form

$$\frac{\Gamma \vdash_S \lambda x.a : A \Rightarrow \tau_0 \rightarrow \tau \mid C \qquad \Gamma \vdash_S v : A \Rightarrow \tau_0 \mid C}{\Gamma \vdash_S e : A \Rightarrow \tau \mid C}\ (\text{APP}_S)$$

  where $\sigma = A \Rightarrow \tau \mid C$. What's more, rule (SUB$_S$) can commute freely with rule (ABS$_S$); thus, we can assume (by rewriting it if necessary) that the typing derivation of $\lambda x.a$ is of the form

$$\frac{\text{lift}_x(\Gamma); x \vdash_S a : (A; x : \tau_0) \Rightarrow \tau \mid C}{\Gamma \vdash_S \lambda x.a : A \Rightarrow \tau_0 \rightarrow \tau \mid C}\ (\text{ABS}_S)$$

  v is a closed term, so lemma 5.5 yields $\varnothing \vdash_S v : \varnothing \Rightarrow \tau_0 \mid C$. Lastly, notice that $C \Vdash C + (\tau_0 \leq (A; x : \tau_0)(x))$ —this is a tautology. Hence, we can apply lemma 5.6. It yields

$$\text{unlift}_x(\text{lift}_x(\Gamma); x) \setminus x \vdash_S [v/x]\, a : ((A; x : \tau_0) \setminus x) \Rightarrow \tau \mid C$$

  which can be rewritten

$$\Gamma \vdash_S e' : A \Rightarrow \tau \mid C$$

- $e = \texttt{let}\, X = v\, \texttt{in}\ a$ and $e' = [v/X]\, a$. e's typing derivation must be of the form

$$\frac{\Gamma \vdash_S v : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash_S a : \sigma \qquad \sigma_1 \leq^m A \Rightarrow \top \mid C}{\Gamma \vdash_S e : \sigma}\ (\text{LET}_S)$$

  where $\sigma = A \Rightarrow \tau \mid C$. Let us write $\sigma_1 = A_1 \Rightarrow \tau_1 \mid C_1$. According to lemma 5.5, we have $\varnothing \vdash_S v : \varnothing \Rightarrow \tau_1 \mid C_1$. We can then apply lemma 5.8, and we find

$$\Gamma \vdash_S [v/X]\, a : \sigma$$

- $e = E[e_0]$, $e' = E[e_0']$ and $e_0 \longrightarrow e_0'$, where E is a reduction context. e's typing tree contains a typing judgement for e_0. By applying the induction hypothesis to it, we replace it with an identical judgement concerning e_0'. The whole typing tree then becomes a derivation of $\Gamma \vdash_S E[e_0'] : \sigma$. □

Finally, here comes the main result of this section: the preservation of typing by reduction, in the case of the original typing rules.

Theorem 5.5 *If* $\Gamma \vdash e : \sigma$ *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' : \sigma$.

Proof. Assume $\Gamma \vdash e : \sigma$. The combination of theorems 5.3 and 5.2 indicates that the "simple" typing rules are complete with respect to the typing rules. Hence, there exists a type scheme σ' such that $\sigma' \leq^\forall \sigma$ and $\Gamma \vdash_S e : \sigma'$. Lemma 5.9 then gives $\Gamma \vdash_S e' : \sigma'$. The "simple" typing rules are correct with respect to the typing rules (theorem 5.1), so this entails $\Gamma \vdash e' : \sigma'$. By applying rule (SUB), we then obtain $\Gamma \vdash e' : \sigma$. □

5.7.3 Correctness of typing with respect to semantics

We have shown, in the previous section, that typing is preserved by reduction. The correctness of typing with respect to semantics is based on this fundamental result. To conclude, all we need to do is prove the following proposition:

Proposition 5.10 *Let e be an irreducible expression, well typed in the empty environment, i.e. such that $\varnothing \vdash e : \sigma$. Then e is a value.*

Proof. By induction on the structure of the term e.

- If e is a variable, then e cannot be well typed in the empty environment; case eliminated.
- If e is a λ-abstraction, then e is a value.
- If e is an application $e_1\, e_2$, then e_1 itself is irreducible, because $[]\, e_2$ is a reduction context (definition 5.11). According to the induction hypothesis, e_1 is a value. This implies that $e_1\, []$ is also a reduction context, and, similarly, that e_2 is a value. Because any value is a λ-abstraction (definition 5.9), e is a β-redex. Hence, e is not irreducible; case eliminated.
- If e is of the form $\texttt{let}\, X = e_1 \,\texttt{in}\, e_2$, then we show, as above, that e_1 is a value. Consequently, e is a `let`-redex, so it is not irreducible; case eliminated.

The proof is complete. As a matter of fact, this result is trivial, because we have considered pure λ-calculus, in which any closed irreducible expression is a value. In other words, there are no possible execution errors in pure λ-calculus. This proposition becomes non-trivial if the calculus and the type system are extended. For instance, let us show what the demonstration would become if we introduced integer constants, together with a base type `int`. (No primitive operations on integers are introduced here. Thus, this extension has little practical interest, but it suffices to introduce the possibility of an error.)

A new case appears in the proof, where e is a constant. Constants are values, so the result is immediate. The other cases are unchanged, except the case where e is an application. Then, we can no longer assert that e_1 is a λ-abstraction; we also have to examine the case where e_1 is a constant. Since the application is well typed, this constant has been given, in the empty environment, a type scheme of the form $\varnothing \Rightarrow \tau_0 \rightarrow \tau_1 \mid C$. However, any typing derivation for an integer constant is necessarily formed of an instance of the newly added typing rule for constants, possibly followed by one or more applications of rule (Sub). Consequently, we have

$$\varnothing \Rightarrow \texttt{int} \mid \varnothing \leq^{\forall} \varnothing \Rightarrow \tau_0 \rightarrow \tau_1 \mid C$$

Recall that for a program to be considered well typed, there is an (implicit) condition: its associated type scheme must have at least one instance. So, in the present case, C admits a solution ρ. By definition of polymorphic scheme subsumption, we thus have

$$\texttt{int} \leq \rho(\tau_0) \rightarrow \rho(\tau_1)$$

But this is impossible; the subtyping relation has been defined so as to make the constructors `int` and $\rightarrow$ incompatible. Case eliminated. □

We can now assert that our type system is safe.

Theorem 5.6 *Let e be an expression, well typed in the empty environment, i.e. such that $\varnothing \vdash e : \sigma$. Assume that some evaluation of e terminates, i.e. $e \longrightarrow^* e'$ where e' is irreducible. Then e' is a value and has the same type as e, i.e. $\varnothing \vdash e' : \sigma$.*

Proof. The assertion $\varnothing \vdash e' : \sigma$ follows from theorem 5.5. Thus, e' is both irreducible and well typed in the empty environment. According to proposition 5.10, e' is a value (as opposed to an irreducible, erroneous expression, which would constitute a runtime error). $\square$

Part II

Simplification

Chapter 6

The small terms invariant

This short chapter intends to show that it is possible, without loss of power, to enforce quite drastic restrictions on the shape of the type schemes we work with. Adopting these restrictions makes several aspects of our theoretical presentation significantly simpler, and also benefits our implementation.

The idea of working solely with small terms is not new, at least from a theoretical point of view. It is to be found, for instance, in the theory of unification [30], where problems are presented as sets of multi-equations of depth 1 at most, rather than as equations between arbitrary terms. Among works more closely related to ours, those of Aiken and Wimmers [5] and of Palsberg [38] use a similar convention. However, it is often a mere theoretical convenience. Here, the invariant shall also help improve sharing between nodes, through the minimization algorithm, and is thus useful from an implementor's point of view.

We begin by defining the so-called small terms invariant (section 6.1). Then, we show how to enforce it at a low cost (section 6.2). Furthermore, section 6.3 describes an algorithm which transforms any type scheme into a compliant one. Finally, in section 6.4, we study the benefits reaped from this choice.

6.1 Definition

Let us first define these restrictions:

Definition 6.1 *A type term is said to be a* leaf term *iff it has height* 0 *and it is not constructed. Equivalently, iff it is a combination using* $\sqcup$ *or* $\sqcap$ *of one or more type variables.*

A type term is said to be a small term *iff it has height* 1 *at most, it is constructed, and its sub-terms are leaf terms.*

Definition 6.2 *A constraint graph* C *is said to satisfy the* small terms invariant *iff for all* $\alpha \in \mathrm{dom}(C)$*,* $C^{\downarrow}(\alpha)$ *and* $C^{\uparrow}(\alpha)$ *are small terms.*

A type scheme $A \Rightarrow \tau \mid C$ *is said to satisfy the small terms invariant iff*

- *all types appearing in* A *are leaf terms;*
- τ *is a leaf term;*
- C *verifies the small terms invariant.*

6.2 Enforcing the invariant

It is very easy to require that all type schemes produced by our type inference rules satisfy this invariant. A quick review of the rules shows that only two of them are liable to break it: ($\mathrm{V{\scriptstyle AR}_I}$) and ($\mathrm{A{\scriptstyle BS}_I}$). The former introduces a context A which maps all variables other than x to $\top$; it suffices to replace $\top$ with a fresh variable β. The latter builds a constructed type using the $\rightarrow$ constructor; it suffices to move this term into the constraint graph by adding an appropriate constraint.

$$\frac{\alpha, \beta \notin F \qquad A = \mathrm{single}_{(x,\alpha,\beta)}(\Gamma)}{[F]\ \Gamma \vdash_{\mathrm{I}} x : [F \cup \{\alpha, \beta\}]\ A \Rightarrow \alpha \mid \varnothing} \quad (\mathrm{V{\scriptstyle AR}_I})$$

$$\frac{[F]\ \mathrm{lift}_x(\Gamma); x \vdash_{\mathrm{I}} e : [F']\ (A; x : \tau) \Rightarrow \tau' \mid C \qquad \alpha \notin F'}{[F]\ \Gamma \vdash_{\mathrm{I}} \lambda x.e : [F' \cup \{\alpha\}]\ A \Rightarrow \alpha \mid C + (\tau \rightarrow \tau' \leq \alpha)} \quad (\mathrm{A{\scriptstyle BS}_I})$$

Figure 6.1: Type inference rules, compliant with the small terms invariant

The modified rules are given in figure 6.1. In each case, one easily verifies that the type scheme produced by the new version of the rule is equivalent (with respect to scheme subsumption) to that produced by the old one. Thus, all properties of the type inference system are preserved.

Theorem 6.1 *If $[F]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ \sigma$ is derivable in the modified type inference system, then σ verifies the small terms invariant.*

Proof. By induction on the structure of the derivation. The proof is straightforward, and left to the reader. □

6.3 Explicit normalization

Thus, it is possible to enforce the small terms invariant without adding a dedicated phase to the inference process; using the modified rules is enough. Nevertheless, we shall now develop a "normalization" algorithm, which transforms any type scheme into an equivalent one that verifies the invariant. From a theoretical point of view, the existence of such an algorithm confirms that our restrictions on the shape of type schemes do not cause any loss of expressivity. From a practical point of view, this algorithm shall be used to normalize user-supplied type schemes, such as those found in module signatures.

Theorem 6.2 *Let σ be a type scheme. Then there exists a type scheme σ', which verifies the small terms invariant, such that $\sigma =^{\forall} \sigma'$.*

Proof. The proof is entirely trivial; the only interesting point is that it suffices to add inequations, rather than equations, to the constraint graph. A term which appears at a positive (resp. negative) position in the constraint graph is replaced with a fresh variable and becomes its lower (resp. upper) bound.

Given a pair of a pos-type τ and a constraint graph C, define its *pos-shielding* as the pair (α, C'), where $\alpha \notin \mathrm{dom}(C)$ and $C' = C + (\tau \leq \alpha)$. Its *neg-shielding* is defined symmetrically. Note that this operation produces a leaf term.

Given a pair of a constructed pos-type τ and a constraint graph C, define its *pos-chopping* as the pair

- (τ, C) if τ is of height 0;
- $(\tau_0' \to \tau_1', C'')$ if $\tau = \tau_0 \to \tau_1$, (τ_0', C') is the neg-shielding of (τ_0, C), and (τ_1', C'') is the pos-shielding of (τ_1, C').

Its *neg-chopping* is defined symmetrically. Note that this operation produces a small term.

We then define a relation $\rightsquigarrow$ between types schemes as follows. Let $\sigma = A \Rightarrow \tau \mid C$ be a type scheme.

1. If τ is not a leaf term, let (τ', C') be the pos-shielding of (τ, C); then $\sigma \rightsquigarrow A \Rightarrow \tau' \mid C'$.
2. If A contains an entry of the form $x : \tau_x$ where τ_x is not a leaf term, let (τ_x', C') be the neg-shielding of (τ_x, C); let A' be identical to A, except that the entry $x : \tau_x$ is replaced with $x : \tau_x'$; then $\sigma \rightsquigarrow A' \Rightarrow \tau \mid C'$.
3. If, for some $\alpha \in \mathrm{dom}(C)$, $C^{\downarrow}(\alpha)$ is not a small term, let (τ', C') be the pos-chopping of $(C^{\downarrow}(\alpha), C)$; let C'' be identical to C' except that $C''^{\downarrow}(\alpha) = \tau'$. Then $\sigma \rightsquigarrow A \Rightarrow \tau \mid C''$.
4. If, for some $\alpha \in \mathrm{dom}(C)$, $C^{\uparrow}(\alpha)$ is not a small term, let (τ', C') be the neg-chopping of $(C^{\uparrow}(\alpha), C)$; let C'' be identical to C' except that $C''^{\uparrow}(\alpha) = \tau'$. Then $\sigma \rightsquigarrow A \Rightarrow \tau \mid C''$.

We now verify that whenever $\sigma \rightsquigarrow \sigma'$, then $\sigma =^{\forall} \sigma'$. There is one case for each of the above situations.

1. Assume ρ' is a solution of C'. Then ρ' is also a solution of C, because C' contains C. We claim that ρ' is a witness to $\sigma \leq^{\forall} \sigma'$. It suffices to verify that $\rho'(\tau) \leq \rho'(\tau')$, which holds because C' contains the constraint $\tau \leq \tau'$.

   Reciprocally, assume ρ is a solution of C. Define ρ' as $\rho + [\tau' \mapsto \rho(\tau)]$ (τ' is a type variable). Then ρ' is a solution of C', because $\tau \leq \tau'$ is satisfied by equality. It is a witness to $\sigma' \leq^{\forall} \sigma$, because $\rho'(\tau') \leq \rho'(\tau)$ also holds.
2. The other cases are based on the same simple principle, and left to the reader.

It is easy to verify that given a type scheme σ, any rewriting sequence for $\rightsquigarrow$ starting at σ is finite; indeed, step 1 can only be applied once, step 2 once for each $x \in \mathrm{dom}(A)$; steps 3 and 4 strictly decrease the number of sub-terms which violate the invariant in the constraint graph.

Finally, any type scheme σ' which is a normal form for $\rightsquigarrow$ verifies the small terms invariant. Thus, the algorithm simply consists in applying $\rightsquigarrow$ as many times as possible. □

Example. Consider the type scheme

$$(\varnothing; x : \alpha \to \beta) \Rightarrow \alpha \to \top$$

It does not respect the small terms invariant, because constructed terms appear in its context and in its body. The algorithm described above replaces these terms with fresh variables, while adding appropriate constraints. Note that the term $\alpha \to \top$ is not a small term, and must itself be decomposed. Thus, we obtain

$$(\varnothing; x : \gamma) \Rightarrow \delta \mid \{\gamma \leq \alpha \to \beta,\ \alpha \to \epsilon \leq \delta,\ \top \leq \epsilon\}$$

6.4 Consequences

The most obvious benefit of the small terms invariant is the simplification of the theory. The structure of type schemes is now very simple, which shall drastically facilitate presentation, proof and implementation of several algorithms (closure, scheme subsumption, canonization, minimization).

The closure algorithm (see chapter 7) and the scheme subsumption algorithm (see chapter 9) have simpler definitions (as far as the latter is concerned) and proofs. However, they are fundamentally independent from this invariant.

The canonization algorithm (see chapter 11) eliminates all occurrences of $\sqcup$ and $\sqcap$ from a type scheme. Thanks to the small terms invariant, if such a constructor appears, then all of its arguments must be type variables. The "heterogeneous" case, where one of the arguments is a constructed type and the others are variables, is thus eliminated, which simplifies the theory and the implementation of the algorithm.

Most of all, the minimization algorithm (see chapter 13) would probably be significantly more difficult to define without this hypothesis. This algorithm strives to detect sub-terms (i.e. nodes) of the constraint graph which play identical roles, so as to identify them. Now, adopting the small terms invariant essentially amounts to tagging each node of the graph with a type variable, as shown by the way the above normalization algorithm works. So, the minimization algorithm can content itself with reasoning on type variables: equivalence relation between variables, polarity of a variable, merging two variables, and so on. Without the invariant, we would have to extend these notions to nodes, which would be redundant, and in some cases, difficult. Thus, the invariant provides a simple way of expressing *sharing* between nodes; therein lies its main interest. The gain is not just theoretical; it is also practical, since the minimization algorithm is then able to identify arbitrary nodes, thus achieving better sharing.

The idea of working solely with small terms is not new. It is to be found, in particular, in the theory of unification [30]. Rather than using equations between arbitrary terms, problems are presented as sets of multi-equations whose members are all variables, except possibly one constructed term, of depth 1 at most. The benefits are the same; namely, a simpler theory and more sharing opportunities. Among works more closely related to ours, Aiken and Wimmers [5] and Palsberg [38] use an analogous technique.

From now on, we shall thus work solely with type schemes that satisfy the small terms invariant. This means that the transformations to be introduced in the future must preserve this invariant. This shall always be the case, because none of them shall *build* new types.

Note that adopting this invariant outlaws the classic simplification method which consists in replacing a variable with its unique bound [14, 4, 8, 42], when the latter is a constructed term. Indeed, the substitution would break the invariant, since it eliminates a variable, thus creating unlabeled nodes. Thus, this simplification is opposed to efficiency, and we shall use it for display purposes only. More shall be said about this topic in section 15.2.

Chapter 7

Deciding solvability

Here are the definition and the proof of the closure algorithm. This algorithm allows deciding whether a given constraint graph admits a solution. It is based on the fact that any solvable constraint graph is equivalent to a closed graph. Thus, the problem becomes, given an arbitrary graph, to complete it and obtain a closed graph. If the process fails, then the graph of interest is insolvable; otherwise, it admits a solution, which can be read off its closed form, according to theorem 3.1.

Interestingly, the algorithm is *incremental*; that is, it expects a closed graph C and a new constraint c, and produces (if at all possible) a closed graph which is equivalent to $C + c$. Thus, in an implementation, constraints can be added as they are created by the type inference engine, without having to reconstruct the closure from scratch after each step.

We could have presented this algorithm in chapter 3, since it logically follows theorem 3.1. We have moved it here because its proof makes use of the small terms invariant, which had not been introduced yet. We shall assume, in this chapter, that all constraint graphs satisfy the small terms invariant.

We begin with a few preliminary lemmas (section 7.1). Then, we define the algorithm (section 7.2) and prove its correctness (section 7.3).

7.1 Preliminaries

These lemmas are based on the small terms invariant. They allow stating the algorithm's invariant in a simpler way, as well as proving it more easily.

Lemma 7.1 *Let $(\tau_i)_{i\in I}$ (resp. $(\tau'_j)_{j\in J}$) be a family of small pos-types (resp. neg-types). Then*

$$\mathrm{subc}(\sqcup_{i\in I}\tau_i \leq \sqcap_{j\in J}\tau'_j) = \bigcup_{(i,j)\in I\times J} \mathrm{subc}(\tau_i \leq \tau'_j)$$

Proof. (By this statement, we mean that both expressions are defined at the same time, and are equal when defined.) The result is immediate in the case of leaf terms. It is then generalized to small terms, which requires a tedious, but straightforward, case analysis. □

From this result, we can deduce, in particular:

Lemma 7.2 *Let τ_1 and τ'_1 be two small pos-types such that τ'_1 contains τ_1. Let τ_2 be a neg-type. Then, the assertion*

$$\mathrm{subc}(\tau_1 \leq \tau_2) \subseteq \mathrm{subc}(\tau'_1 \leq \tau_2)$$

holds, provided its right-hand side is defined.

Proof. Assume $\mathrm{subc}(\tau_1' \leq \tau_2)$ is defined. According to definition 2.11, we have $\tau_1' = \tau_1 \sqcup \tau_1'$. Thus, according to lemma 7.1, $\mathrm{subc}(\tau_1' \leq \tau_2)$ is equal to $\mathrm{subc}(\tau_1 \leq \tau_2) \cup \mathrm{subc}(\tau_1' \leq \tau_2)$. In other words, $\mathrm{subc}(\tau_1 \leq \tau_2) \subseteq \mathrm{subc}(\tau_1' \leq \tau_2)$ which is the expected result. □

Of course, there exists a symmetric variant of the above lemma, which we shall not state, for the sake of brevity.

7.2 Algorithm

Here comes the definition of the incremental closure algorithm.

Definition 7.1 *A* state *of the algorithm is a pair (C, Q), where C is a constraint graph and Q is a set of constraints of the form $\alpha \leq \beta$, $\alpha \leq \tau$ or $\tau \leq \beta$, where α, β are type variables and τ is a small term. Furthermore, one requires $\mathrm{fv}(Q) \subseteq \mathrm{dom}(C)$.*

The algorithm switches *from a state (C, Q) to a new state as follows. If the waiting queue Q is empty, terminate the algorithm and return C. Otherwise, pick a constraint c in Q and set $Q' = Q \setminus \{c\}$. Three cases are then to be considered, depending on the form of c.*

- *c is of the form $\alpha \leq \beta$. If $\alpha \leq_C \beta$, switch to the state (C, Q'). Otherwise, switch to the state (D, Q''), where D and Q'' are defined as follows:*
  - $\leq_D = \leq_C \cup \{(\alpha', \beta')\,;\, \alpha' \leq_C \alpha \wedge \beta \leq_C \beta'\}$;
  - *If $\beta \leq_C \beta'$, then $D^\downarrow(\beta') = C^\downarrow(\beta') \sqcup C^\downarrow(\alpha)$, otherwise $D^\downarrow(\beta') = C^\downarrow(\beta')$;*
  - *If $\alpha' \leq_C \alpha$, then $D^\uparrow(\alpha') = C^\uparrow(\alpha') \sqcap C^\uparrow(\beta)$, otherwise $D^\uparrow(\alpha') = C^\uparrow(\alpha')$;*
  - $Q'' = Q' \cup \mathrm{subc}(C^\downarrow(\alpha) \leq C^\uparrow(\beta))$.
- *c is of the form $\alpha \leq \tau$. If $C^\uparrow(\alpha)$ contains τ, switch to the state (C, Q'). Otherwise, switch to the state (D, Q''), where D and Q'' are defined as follows:*
  - $\leq_D = \leq_C$;
  - $D^\downarrow = C^\downarrow$;
  - *If $\alpha' \leq_C \alpha$, then $D^\uparrow(\alpha') = C^\uparrow(\alpha') \sqcap \tau$, otherwise $D^\uparrow(\alpha') = C^\uparrow(\alpha')$;*
  - $Q'' = Q' \cup \mathrm{subc}(C^\downarrow(\alpha) \leq \tau)$.
- *c is of the form $\tau \leq \beta$. If $C^\downarrow(\beta)$ contains τ, switch to the state (C, Q'). Otherwise, switch to the state (D, Q''), where D and Q'' are defined as follows:*
  - $\leq_D = \leq_C$;
  - *If $\beta \leq_C \beta'$, then $D^\downarrow(\beta') = C^\downarrow(\beta') \sqcup \tau$, otherwise $D^\downarrow(\beta') = C^\downarrow(\beta')$;*
  - $D^\uparrow = C^\uparrow$;
  - $Q'' = Q' \cup \mathrm{subc}(\tau \leq C^\uparrow(\beta))$.

Note that the algorithm can fail if the function subc is evaluated outside of its domain. This means that an insolvable constraint has been found.

Example. To demonstrate the incremental closure algorithm's functioning, let us compute, using the type inference rules, the type of the application $(\lambda x.x)\,3$. We assume given the type scheme of the identity function $\lambda x.x$, which is $\gamma \mid C_1$, where $C_1 = \varnothing + (\alpha \to \beta \leq \gamma) + (\alpha \leq \beta)$; as well as that of the expression 3, which is $\epsilon \mid C_2$, where $C_2 = \varnothing + (\mathtt{int} \leq \epsilon)$. To type the application, we must apply rule (App$_\mathrm{I}$). It states that the type scheme associated to the result is $\delta \mid C$, where $C = (C_1 \cup C_2) + (\gamma \leq \epsilon \to \delta)$. The union $C_1 \cup C_2$ requires no computation, since these two constraint graphs have disjoint domains. There remains to use the incremental closure algorithm to add the constraint $\gamma \leq \epsilon \to \delta$.

Our description of the algorithm is functional, that is, each step produces a new constraint graph. For the sake of simplicity, this example shall be presented in an imperative way, that is, we shall assume that the algorithm modifies an existing graph.

So, initially, let C be the graph defined by $\alpha \leq_C \beta$, $C^\downarrow(\gamma) = \alpha \to \beta$ and $C^\downarrow(\epsilon) = \mathtt{int}$. The waiting queue Q contains the constraint $\gamma \leq \epsilon \to \delta$.

The algorithm's first step removes this constraint from the queue. It updates γ's upper bound by setting $C^\uparrow(\gamma) = \epsilon \to \delta$; then, it adds to the waiting queue the constraints $\mathrm{subc}(\alpha \to \beta \leq \epsilon \to \delta)$, i.e. $\epsilon \leq \alpha$ and $\beta \leq \delta$, obtained by transitivity on γ.

The second step removes the constraint $\epsilon \leq \alpha$ from the waiting queue. (One could start with the other constraint; the final result would be the same.) It adds its consequences on variables: $\epsilon \leq_C \alpha$ and $\epsilon \leq_C \beta$. Then, since ϵ has a non-trivial constructed lower bound, this bound also constrains α and β: so, we set $C^\downarrow(\alpha) = C^\downarrow(\beta) = \mathtt{int}$. Lastly, the waiting queue must receive the constraints $\mathrm{subc}(C^\downarrow(\epsilon) \leq C^\uparrow(\alpha))$; but $\mathrm{subc}(\mathtt{int} \leq \top) = \varnothing$, so there is nothing no add.

The third step removes the constraint $\beta \leq \delta$ from the waiting queue. It adds its consequences on variables, that is, $\epsilon \leq_C \delta$, $\alpha \leq_C \delta$ and $\beta \leq_C \delta$. β's constructed lower bound is now $\mathtt{int}$; so, we must set $C^\downarrow(\delta) = \mathtt{int}$. Lastly, once more, we find that no constraints have to be added to the waiting queue. So, the algorithm terminates, and yields a closed graph C.

Recall that, according to rule (App$_\mathrm{I}$), the type scheme associated to the complete λ-term is $\delta \mid C$. Thus, δ is the type of this term, and C contains the constraint $\mathtt{int} \leq \delta$; this indicates that the term is an integer. If, by anticipating somewhat, we apply the garbage collection algorithm (presented in chapter 10) to this type scheme, we find that all other constraints are eliminated. Thus, they have played an intermediary role during the closure computation, but are no longer necessary once we obtain the relevant information, namely $\mathtt{int} \leq \delta$.

7.3 Correctness

Proposition 7.3 *The closure algorithm terminates.*

Proof. Let $(C_n, Q_n)_n$ be a transition sequence. Note that all C_n's have the same domain V and satisfy the small terms invariant. The set of constraint graphs which have domain V and satisfy this invariant is finite. We order it by setting that C is less than, or equal to, D iff the following conditions hold:

- $\leq_C \subseteq \leq_D$;
- for all $\alpha \in V$, $D^\downarrow(\alpha)$ contains $C^\downarrow(\alpha)$ and $D^\uparrow(\alpha)$ contains $C^\uparrow(\alpha)$.

The set of states is ordered lexicographically, the order on the first components being the one defined above, and the order on the second components being the reverse of set-theoretic inclusion. Then, it is easy to verify, according to the algorithm's definition, that the sequence $(C_n, Q_n)_n$ is increasing. Its first component has values inside a finite set; thus, it is constant after a certain rank. From that rank on, the set Q_n decreases; hence, the sequence is finite. The algorithm terminates. $\square$

Lemma 7.4 *If the algorithm switches from a state (C, Q) to a state (C', Q'), then the solutions common to C and Q are exactly the solutions common to C' and Q'.*

Proof. This is easily verified by considering the algorithm's definition. □

Lemma 7.5 *Assume that a state (C, Q) verifies the following properties:*

- *$\leq_C$ is transitive;*
- *if $\alpha \leq_C \beta$, then $C^{\downarrow}(\beta)$ contains $C^{\downarrow}(\alpha)$ and $C^{\uparrow}(\alpha)$ contains $C^{\uparrow}(\beta)$;*
- *for all $\gamma \in \mathrm{dom}(C)$, the set $\mathrm{subc}(C^{\downarrow}(\gamma) \leq C^{\uparrow}(\gamma))$ is defined and is a subset of $\leq_C \cup Q$.*

If the algorithm switches from this state to a new state, then the latter verifies the same properties.

Proof. (To properly understand the third property above, note that, thanks to the small terms invariant, the set $\mathrm{subc}(C^{\downarrow}(\gamma) \leq C^{\uparrow}(\gamma))$ only contains constraints between variables. Thus, each of them must be either part of C or waiting inside Q.)

Let us consider the first property. We shall deal with the case where the constraint c picked from Q is of the form $\alpha \leq \beta$, since the result is immediate in the other cases. Assume $\alpha' \leq_D \beta' \leq_D \gamma'$. There are four sub-cases, depending on whether each of these two relationships is already present in $\leq_C$.

- $\alpha' \leq_C \beta' \leq_C \gamma'$. The result stems from the transitivity of $\leq_C$ and from the fact that $\leq_D$ contains $\leq_C$.
- $\alpha' \leq_C \beta'$, $\beta' \leq_C \alpha$ and $\beta \leq_C \gamma'$. By transitivity of $\leq_C$, we have $\alpha' \leq_C \alpha$, whence $\alpha' \leq_D \gamma'$ by definition of $\leq_D$.
- The last two sub-cases are similar to the previous one.

Let us move on to the second property. We shall only consider the first of the two assertions it contains, since the second is symmetric. There are three cases, according to the form of the constraint c picked from Q. In the first case, c is of the form $\alpha \leq \beta$. Assume $\alpha' \leq_D \beta'$. We now distinguish two sub-cases, depending on whether this relationship already exists in $\leq_C$.

- $\alpha' \leq_C \beta'$. We have to show that $D^{\downarrow}(\beta') \sqcup D^{\downarrow}(\alpha') = D^{\downarrow}(\beta')$. This is easy, by coming back to the definition of $D^{\downarrow}$ and noticing that if $\beta \leq_C \alpha'$, then $\beta \leq_C \beta'$ by transitivity of $\leq_C$. (Of course, one shall take advantage of the fact that the property is verified by C.)
- $\alpha' \leq_C \alpha$ and $\beta \leq_C \beta'$. We have to show that $C^{\downarrow}(\beta') \sqcup C^{\downarrow}(\alpha)$ contains $D^{\downarrow}(\alpha')$. According to the definition of $D^{\downarrow}$, $D^{\downarrow}(\alpha')$ is equal either to $C^{\downarrow}(\alpha')$, or to $C^{\downarrow}(\alpha') \sqcup C^{\downarrow}(\alpha)$. To conclude, it suffices to verify that $C^{\downarrow}(\alpha)$ contains $C^{\downarrow}(\alpha')$, which stems from $\alpha' \leq_C \alpha$ and from our hypothesis on C.

The second case is trivial, since $D^{\downarrow} = C^{\downarrow}$. The third case is easily obtained, through a reasoning similar to the above.

Let us finally consider the third property. Let $\gamma \in \mathrm{dom}(D)$. We distinguish three cases, according to the form of the constraint c picked from Q. In the first case, c is of the form $\alpha \leq \beta$. Four sub-cases then appear, depending on whether the conditions $\gamma \leq_C \alpha$ and $\beta \leq_C \gamma$ are verified. These sub-cases are handled in similar ways; let us focus, for instance, on the sub-case where $\gamma \leq_C \alpha$ and $\beta \not\leq_C \gamma$. We then have $D^{\downarrow}(\gamma) = C^{\downarrow}(\alpha)$ and $D^{\uparrow}(\gamma) = C^{\uparrow}(\gamma) \sqcap C^{\uparrow}(\beta)$. According to lemma 7.1, we can thus write

$$\mathrm{subc}(D^{\downarrow}(\gamma) \leq D^{\uparrow}(\gamma)) = \mathrm{subc}(C^{\downarrow}(\gamma) \leq C^{\uparrow}(\gamma)) \cup \mathrm{subc}(C^{\downarrow}(\gamma) \leq C^{\uparrow}(\beta))$$

According to our hypothesis, the first of the right-hand sets is defined and is a subset of $\leq_C \cup Q$. *A fortiori*, it is a subset of $\leq_D \cup Q''$. Let us now consider the second one. By hypothesis, $C^{\downarrow}(\alpha)$ contains $C^{\downarrow}(\gamma)$. According to lemma 7.2, this implies

$$\mathrm{subc}(C^{\downarrow}(\gamma) \leq C^{\uparrow}(\beta)) \subseteq \mathrm{subc}(C^{\downarrow}(\alpha) \leq C^{\uparrow}(\beta))$$

provided the latter is defined. Indeed, it is defined; it is actually a subset of Q'', by definition of Q''. Hence, we are done with this sub-case. The second and third cases are dealt with similarly. □

Thanks to the preceding lemmas, we can now establish the following result:

Theorem 7.1 *Let C be a closed constraint graph. Let c be a constraint of the form $\alpha \leq \beta$, $\alpha \leq \tau$ or $\tau \leq \beta$, where α, β are type variables and τ is a small term, and such that $\mathrm{fv}(c) \subseteq \mathrm{dom}(C)$. The closure algorithm is executed with initial state $(C, \{c\})$. Then*

- *if $C + c$ is solvable, then the algorithm produces a closed constraint graph which is equivalent to $C + c$;*
- *otherwise, the algorithm signals a failure.*

Proof. First, one verifies that $(C, \{c\})$ is a state, as defined by definition 7.1. (Note that the condition $\mathrm{fv}(c) \subseteq \mathrm{dom}(C)$ isn't restrictive in practice, since C's domain can be extended first.)

According to proposition 7.3, the algorithm terminates. Consequently, it either produces a result or reports a failure.

If it produces a result, then it has reached a state of the form $(D, \varnothing)$. By applying lemma 7.4 repeatedly, we obtain an equivalence between $C + c$ and D. Furthermore, since C is closed, the initial state $(C, \{c\})$ verifies the invariant specified by lemma 7.5. By applying this lemma repeatedly, it follows that D itself is closed. Since every closed graph is solvable, D is solvable, and so is $C + c$.

If, on the contrary, the algorithm signals a failure, then an insolvable constraint has been found. However, all constraints manipulated here are logically entailed by $C + c$. Thus, the latter is also insolvable. □

Thus, we have designed an incremental closure algorithm, which allows adding an arbitrary constraint to a closed graph, yielding a new closed graph. This algorithm can of course be used in a non-incremental way; the closure of a constraint set, or graph, is computed by adding all of its constraints, one by one, to the empty graph. Hence, the solvability problem for conjunctions of constraints is decidable.

We shall not prove some properties, such as the fact that the algorithm is deterministic (i.e. its result does not depend on the order in which constraints are picked from the queue), or the fact that it yields a minimal result (i.e. it is the smallest closed graph containing the original graph and waiting queue). These properties hold, but shall not be useful here.

Chapter 8

Deciding entailment

We attempt, in this chapter, to design an algorithm which determines whether a given constraint graph entails a given constraint. In other words, we would like to decide the entailment relation given by definition 3.6.

We originally published this algorithm in [42]. It then played a key role in the type simplification process. Indeed, the type system proposed in that paper was rather close to our so-called "simple" type system (see section 5.4); in particular, it was based on entailment. Furthermore, simplification was based on heuristics; thus, we had to determine, at runtime, whether the typing judgements produced by these heuristics were valid. This is why we developed this algorithm. Unfortunately, it turns out to be incomplete.

Thus, we studied monomorphic comparison between type schemes, which is based on entailment. Trifonov and Smith [46] independently develop a very similar algorithm, then generalize it to the case of polymorphic comparison, that is, scheme subsumption. We give a formal description and a proof of this new algorithm in chapter 9. It is also incomplete; at the time of writing, the decidability of these two relations remains an open problem.

Currently, neither algorithm is actually used in the simplification process, since we do not rely on heuristics any more. However, the second one is still used to match inferred type schemes against user-declared specifications. Besides, it has theoretical importance, since it is the fundamental explanation of the garbage collection process, introduced in chapter 10. As for the first algorithm, it is still interesting, because it helps understand the more general one, and also because it actually serves to define it.

This chapter is organized as follows. Section 8.1 proposes an axiomatization of entailment, that is, a logic framework within which certain entailment assertions can be derived. We first prove its correctness (section 8.2); then, we verify that it specifies an algorithm (section 8.4). In section 8.5, we show this axiomatization to be incomplete, and develop several counter-examples. Finally, section 8.3 gives a simplified formulation of the axiomatization, suitable when dealing with simple types.

8.1 Axiomatization

Let us first state the problem. Given a constraint graph C of domain V and a constraint c such that $\mathrm{fv}(c) \in V$, does $C \Vdash c$ hold?

Clearly, if C has no solution, then the answer is yes. According to theorem 7.1, this case is decidable; we can thus assume that C is a closed constraint graph. (Note that this closure hypothesis is important to make the algorithm as powerful as possible in practice; however, it is not used in the correctness proof.)

Under these assumptions, we attempt to axiomatize entailment as follows:

$$\frac{(\tau \leq \tau') \in H}{C, H \vdash \tau \leq \tau'} \qquad (\textsc{Hist})$$

$$\frac{\alpha \in S \qquad \alpha' \in S' \qquad \alpha \leq_C \alpha'}{C, H \vdash \sqcap S \leq \sqcup S'} \qquad (\textsc{Tauto})$$

$$\frac{\begin{array}{c}(S \cup S') \cap \mathcal{V} \neq \varnothing \\ H' = H \cup \{\sqcap S \leq \sqcup S'\} \\ C, H' \vdash \sqcap\{C^{\uparrow}(\tau)\,;\, \tau \in S\} \leq \sqcup\{C^{\downarrow}(\tau')\,;\, \tau' \in S'\}\end{array}}{C, H \vdash \sqcap S \leq \sqcup S'} \qquad (\textsc{V-Elim})$$

$$C, H \vdash \bot \leq \tau' \qquad (\bot\textsc{-Elim})$$

$$C, H \vdash \tau \leq \top \qquad (\top\textsc{-Elim})$$

$$\frac{C, H \vdash \tau_0' \leq \tau_0 \qquad C, H \vdash \tau_1 \leq \tau_1'}{C, H \vdash \tau_0 \to \tau_1 \leq \tau_0' \to \tau_1'} \qquad (\to\textsc{-Elim})$$

Figure 8.1: Axiomatization of entailment

$$\frac{(\tau \leq \tau') \in H}{C, H \vdash \tau \leq \tau'} \qquad (\textsc{Hist})$$

$$\frac{\alpha \leq_C \alpha'}{C, H \vdash \alpha \leq \alpha'} \qquad (\textsc{Tauto})$$

$$\frac{\begin{array}{c}\{\tau, \tau'\} \cap \mathcal{V} \neq \varnothing \\ H' = H \cup \{\tau \leq \tau'\} \qquad C, H' \vdash C^{\uparrow}(\tau) \leq C^{\downarrow}(\tau')\end{array}}{C, H \vdash \tau \leq \tau'} \qquad (\textsc{V-Elim})$$

$$C, H \vdash \bot \leq \tau' \qquad (\bot\textsc{-Elim})$$

$$C, H \vdash \tau \leq \top \qquad (\top\textsc{-Elim})$$

$$\frac{C, H \vdash \tau_0' \leq \tau_0 \qquad C, H \vdash \tau_1 \leq \tau_1'}{C, H \vdash \tau_0 \to \tau_1 \leq \tau_0' \to \tau_1'} \qquad (\to\textsc{-Elim})$$

Figure 8.2: Restriction to simple types

Definition 8.1 *We set up a formal derivation system, defined by the rules given in figure 8.1 on the facing page. A derivation for this system shall be called an* entailment derivation. *Judgments are of the form $C, H \vdash \tau \leq \tau'$ where C is a closed constraint graph, $(\tau, \tau') \in \mathcal{T}^- \times \mathcal{T}^+$, $H \subseteq \mathcal{T}^- \times \mathcal{T}^+$, and all type variables appearing in the judgement are elements of C's domain. We shall write $C \vdash \tau \leq \tau'$ for $C, \varnothing \vdash \tau \leq \tau'$.*

In rules (Tauto) and (V-Elim), we assume that S and S' contain only type variables or constructed types, as allowed by proposition 2.2. We shall allow S and S' to have any cardinality, so as to group several cases into one rule. Additionally, $C^\uparrow$ (resp. $C^\downarrow$), which has been defined on type variables, is extended here to constructed types by setting $C^\uparrow(\tau) = \tau$ (resp. $C^\downarrow(\tau) = \tau$) when τ is constructed.

These deduction rules are extremely simple. To better understand them, one can first look at the case where the $\sqcup$ and $\sqcap$ constructors are proscribed. The rules can then be simplified, yielding the system given in figure 8.2.

Note that we generalize Amadio and Cardelli's [9] algorithm for the comparison of two recursive types. In both cases, the problem consists in determining whether, under a certain set of assumptions, some type τ is a subtype of some type τ'. When comparing two recursive types, the set of assumptions is simply a system of equations, whose unique solution defines both types. Here, the set of assumptions is a constraint graph, whose set of solutions is *a priori* complex, whence a greater difficulty. Nevertheless, as we shall see, our rules remain extremely close to Amadio and Cardelli's.

In the latter, judgements are of the form $\Sigma, \epsilon \supset \tau \leq \tau'$. Our judgements have an essentially identical form, namely $C, H \vdash \tau \leq \tau'$. The assumptions are represented by the equation set ϵ in the first case, and by the constraint graph C in the second case. The parameters Σ and H play the role of a *history*; that is, they serve to memorize previously encountered goals, so as to guarantee the algorithm's termination.

Let us now comment our rules:

- Rule (Hist) allows accepting any goal present in the history, that is, any previously encountered goal. Thus, this rule allows the algorithm to reason by co-induction: if a previous goal appears again, then it is possible to build an infinite derivation for it. We then accept it immediately.

- Rule (Tauto) simply allows using a constraint between variables present in the assumptions set.

- Rule (V-Elim) comes into play when a variable appears at depth 0 in the goal, as indicated by the condition $(S \cup S') \cap \mathcal{V} \neq \varnothing$. For instance, let us consider the simple case where the goal is of the form $\alpha \leq \tau_0 \rightarrow \tau_1$. The goal cannot be decomposed structurally, since the variable α appears at toplevel. So, the most natural solution is to use the hypotheses contained in C. If we can prove $C^\uparrow(\alpha) \leq \tau_0 \rightarrow \tau_1$, then the goal shall follow, by transitivity of subtyping. So, the variable α is replaced with its constructed upper bound. (It would be useless to replace α with a variable β such that $\alpha \leq_C \beta$, since β would then itself have to be replaced with one of its upper bounds.) In the general case, all variables appearing on the left-hand (resp. right-hand) side are replaced with their constructed upper (resp. lower) bound.

  Besides, note that the rule adds its goal to the history. It is the only rule which performs this operation. Indeed, it need not be also performed by the structural decomposition rules; this simply means that a few more decomposition steps may be necessary before a previously encountered goal is recognized.

- Rules ($\bot$-Elim), ($\top$-Elim) and ($\rightarrow$-Elim) are called structural decomposition rules; that is, they break their goal down into zero or more sub-goals, when both sides of the goal are constructed types. (Actually, rules ($\perp$-ELIM) and ($\top$-ELIM) also apply when τ or τ' is not constructed, but they could be restricted to this case without loss of power.) If type constructors other than $\perp$, $\rightarrow$ and $\top$ are added to the language, then corresponding structural decomposition rules should also be added.

Example. Let F be a covariant type operator, distinct from identity. (For instance, one may choose $F : \tau \mapsto \epsilon \rightarrow \tau$.) Let C be the constraint graph of domain $\{\alpha, \beta\}$ defined by $C^{\uparrow}(\alpha) = F\,\alpha$ and $C^{\downarrow}(\beta) = F\,\beta$. Then, our axiomatization can be used to show that $C \Vdash \alpha \leq \beta$.

Indeed, given the goal $\alpha \leq \beta$, rule (V-ELIM) can be used twice to replace each variable with its constructed bound. Thus, the goal becomes $F\,\alpha \leq F\,\beta$. Furthermore, since this rule adds its conclusion to the history, the constraint $\alpha \leq \beta$ becomes part of the history.

Since F is, by hypothesis, a covariant operator, applying a sufficient number of structural decomposition rules yields the goal $\alpha \leq \beta$ again. (Carrying on with our example, if we chose $F : \tau \mapsto \epsilon \rightarrow \tau$, then we can apply rule ($\rightarrow$-ELIM). We obtain the goal $\epsilon \leq \epsilon$, which is trivially justified by (TAUTO), and the expected goal, namely $\alpha \leq \beta$.)

This is where rule (HIST) comes into play: since the goal $\alpha \leq \beta$ appears in the history, it is immediately accepted. Without this rule, we would be able to build an infinite derivation of our goal, but no finite one. Thus, this rule allows encoding an infinite, regular proof into a finite proof.

So, this axiomatization of entailment is very simple, and close to Amadio and Cardelli's system. Unfortunately, the completeness property is lost; that is, not all true statements can be proved. This is due to the excessive simplicity of rule (V-ELIM). Indeed, in Amadio and Cardelli's case, replacing a variable with its bound is not a source of incompleteness, since the variable and its bound are related by an equation. Here, we only have an inequation; the replacement remains correct, but completeness is lost. No complete axiomatization is currently known. This problem shall be discussed in further detail in section 8.5.

8.2 Soundness

We shall now prove that this axiomatization is correct with respect to entailment.

Obviously, we cannot prove that $C, H \vdash \tau \leq \tau'$ implies $C \Vdash \tau \leq \tau'$ without some hypothesis on H, because of the presence of rule (HIST). This rule allows any goal found in H to be declared true, so it is unsound in the absence of an invariant on H. Rather than trying to give such an invariant, we will indeed build a proof where rule (HIST) is considered unsound, but we shall show that its use can be delayed as much as desired.

Let us first give a technical definition:

Definition 8.2 *The* depth *of an entailment derivation is the number of structural decomposition rules appearing in the shortest branch which ends with a* (HIST) *rule. (If no branch at all ends with a* (HIST) *rule, the depth of the derivation is ∞.)*

We can now state the following proposition, which forms the core of the soundness proof:

Proposition 8.1 *Assume given a derivation of $C, H \vdash \tau \leq \tau'$ of depth k. Then*

$$\forall j \leq k \quad C \Vdash_j \tau \leq \tau'$$

Proof. By induction on the structure of the input entailment derivation. There is one case per derivation rule. For the sake of brevity, in each case, we use exactly the same notations as in figure 8.1, which allows us not to write the assumption explicitly.

- (HIST) The derivation has depth 0. Since $\leq_0$ is uniformly true, so is $\Vdash_0$. So, $C \Vdash_0 \tau \leq \tau'$ holds.

- (TAUTO) The derivation has depth ∞. Let $j \in \mathbb{N}^+ \cup \{\infty\}$, and let ρ be a j-solution of C. In particular, $\rho(\alpha) \leq_j \rho(\alpha')$. Since $\alpha \in S$ and $\alpha' \in S'$, we have $\rho(\sqcap S) \leq \rho(\alpha)$ and $\rho(\alpha') \leq \rho(\sqcup S')$. Finally, since $\leq$ is finer than $\leq_j$, and since $\leq_j$ is transitive, this implies $\rho(\sqcap S) \leq_j \rho(\sqcup S')$. So, ρ is a j-solution of $\sqcap S \leq \sqcup S'$.

- (V-ELIM) The depth k of this derivation coincides with the depth of its premise. By applying the induction hypothesis, we obtain

$$\forall j \leq k \quad C \Vdash_j \sqcap\{C^{\uparrow}(\tau)\,;\,\tau \in S\} \leq \sqcup\{C^{\downarrow}(\tau')\,;\,\tau' \in S'\}$$

Let ρ be a j-solution of C. Then, according to the above statement, we have

$$\rho(\sqcap\{C^{\uparrow}(\tau)\,;\,\tau \in S\}) \leq_j \rho(\sqcup\{C^{\downarrow}(\tau')\,;\,\tau' \in S'\})$$

which is equivalent to

$$\sqcap\{\rho(C^{\uparrow}(\tau))\,;\,\tau \in S\} \leq_j \sqcup\{\rho(C^{\downarrow}(\tau'))\,;\,\tau' \in S'\}$$

Now, for any type variable α, we have

$$\rho(C^{\downarrow}(\alpha)) \leq_j \rho(\alpha) \leq_j \rho(C^{\uparrow}(\alpha))$$

because ρ is a j-solution of C. Furthermore, for any constructed type τ,

$$\rho(C^{\downarrow}(\tau)) \leq_j \rho(\tau) \leq_j \rho(C^{\uparrow}(\tau))$$

trivially holds because of our definition of $C^{\uparrow}$ and $C^{\downarrow}$ on constructed types (see definition 8.1). So, by transitivity, our assertion implies

$$\sqcap\{\rho(\tau)\,;\,\tau \in S\} \leq_j \sqcup\{\rho(\tau')\,;\,\tau' \in S'\}$$

which can easily be rewritten as

$$\rho(\sqcap S) \leq_j \rho(\sqcup S')$$

so ρ is a j-solution of $\sqcap S \leq \sqcup S'$.

- ($\perp$-ELIM) This derivation has infinite depth. For any $j \in \mathbb{N}^+ \cup \{\infty\}$, it is clear that $C \Vdash_j \perp \leq \tau'$.

- ($\top$-ELIM) Similar to the previous case.

- ($\rightarrow$-ELIM) Let k be the depth of this derivation and let $j \leq k$. The case $j = 0$ is immediate, since $\Vdash_0$ always holds, so let us assume $j \leq 1$. The depth of each premise is at least $k - 1$. So, by applying the induction hypothesis to each premise and by specializing the result for $j - 1$, we obtain $C \Vdash_{j-1} \tau_0' \leq \tau_0$ and $C \Vdash_{j-1} \tau_1 \leq \tau_1'$. From there, we easily deduce that $C \Vdash_j \tau_0 \rightarrow \tau_1 \leq \tau_0' \rightarrow \tau_1'$. □

We have just showed that if rule (HIST) is not used until depth k, then k-entailment holds. That is, the deeper rule (HIST) appears, the closer to full entailment we get. We will now show that uses of (HIST) can be delayed as far as desired, provided we start out with an empty history set.

Proposition 8.2 *Assume there exists a derivation of $C, \varnothing \vdash \tau \leq \tau'$ which has finite depth. Then, there exists a derivation of the same assertion which has strictly greater depth.*

Proof. Let D be the original derivation, and let k be its depth. Since k is finite, D contains at least one branch ending with rule (HIST) at depth k. Consider one such occurrence of (HIST). Its conclusion $\tau \leq \tau'$ comes from the history set. The history set is $\varnothing$ at the bottom of the derivation D, and it can only be augmented by uses of rule (V-ELIM). Hence, $\tau \leq \tau'$ must appear somewhere in this branch as the conclusion of a (V-ELIM) rule.

Generally speaking, rule (V-ELIM) has some interesting properties:

- Its premise is always a constraint between two constructed types. Indeed, $C^{\downarrow}(\tau)$ (resp. $C^{\uparrow}(\tau)$) is always a constructed type; and a combination of several constructed types using $\sqcup$ (resp. $\sqcap$) is also a constructed type.
- Its conclusion always involves at least one type variable, thanks to the condition $(S \cup S') \cap \mathcal{V} \neq \varnothing$.
- It follows that its premise always differs from its conclusion.

Back to our particular case, the occurrence of (V-ELIM) cannot be directly preceded by the occurrence of (HIST), because that would require (V-ELIM) to have a premise identical to its conclusion. Hence, there must be some other rule above the occurrence of (V-ELIM). This other rule has two constructed types as its conclusion and has at least one premise, since there is a (HIST) rule somewhere above it. Hence, it must be an occurrence of rule ($\rightarrow$-ELIM) (or, more generally, of a structural decomposition rule).

Now, consider the subtree rooted at our occurrence of (V-ELIM). We can replace our occurrence of (HIST) with a copy of this subtree (this might involve using larger history sets within the copy than within the original subtree, but this is not a problem), and still obtain a valid derivation. Since we have proved the presence of a structural decomposition rule next to its root, this subtree has depth 1 at least, and we have replaced a branch of depth k with a branch of depth strictly greater than k, while keeping a valid derivation.

Since there can only be a finite number of occurrences of rule (HIST) at depth k, iterating this process eventually yields a derivation of depth strictly greater than k. $\square$

We can now bring these two propositions together:

Theorem 8.1 *The axiomatization is correct with respect to entailment; that is, if $C \vdash \tau \leq \tau'$, then $C \Vdash \tau \leq \tau'$.*

Proof. Consider a derivation of $C, \varnothing \vdash \tau \leq \tau'$. If it has infinite depth, then $\forall k \quad C \Vdash_k \tau \leq \tau'$ holds, according to proposition 8.1. If it has finite depth, then according to proposition 8.2, for any finite k, there exists a derivation of $C, \varnothing \vdash \tau \leq \tau'$ which has depth greater than k. Applying proposition 8.1 to this derivation and to k, we obtain $C \Vdash_k \tau \leq \tau'$.

In both cases, we have shown $C \Vdash_k \tau \leq \tau'$ for any finite k, which is actually stronger than $C \Vdash \tau \leq \tau'$, according to proposition 3.3. $\square$

8.3 A specialized axiomatization

The axiomatization developed so far works with a full constraint graph as hypothesis and a (neg-type, pos-type) pair as goal. That is, the hypothesis and the goal are allowed to contain occurrences of the $\sqcup$ and $\sqcap$ constructs at certain positions. It is possible to restrict the axiomatization to situations where these constructs are entirely disallowed; we obtain the rules given in figure 8.2 on page 84, which are slightly easier to understand.

This specialization does not incur any loss of intrinsic power, because any entailment problem can be translated to an equivalent problem which has no occurrences of $\sqcup$ and $\sqcap$. This fact has a simple proof, although we haven't yet developed all of the necessary tools. The entailment assertion $C \Vdash \tau \leq \tau'$ is equivalent to $\alpha \rightarrow \alpha \leq^{\forall} \tau \rightarrow \tau' \mid C$, which is a subsumption assertion between polymorphic type schemes. Besides, we shall see in chapter 11 that any occurrences of $\sqcup$ and $\sqcap$ can be removed from a type scheme, yielding an equivalent scheme. By applying this statement to the right-hand scheme and coming back, we obtain an entailment assertion which involves simple types only.

Although allowing $\sqcup$ and $\sqcap$ in our entailment algorithm does not inherently augment its power, it does not make more complex either, so it seems worthwhile. We shall see, in contrast, that allowing them in the polymorphic subsumption algorithm (which we shall define in chapter 9) would not be natural, because it would lead to malformed constraints such as $\alpha \leq \beta \sqcup \gamma$.

8.4 Algorithm

Our axiomatization of entailment can easily be read as a decision algorithm. However, we have to verify that this algorithm terminates. Furthermore, for efficiency reasons, it is best for it not to be faced with any choices while building a derivation. This allows it to terminate as soon as a failure is detected; otherwise, backtracking would be necessary. Both properties are verified below.

Proposition 8.3 *There exists an algorithm which, given a goal of the form $C, H \vdash \tau \leq \tau'$, gives a derivation of it if one exists and reports failure otherwise.*

Proof. The algorithm considers the goal and tries to build a derivation for it. One verifies that no goal is such that two "recursive" rules can be applied to it. Indeed, (V-ELIM) requires a type variable to appear on at least one side of the goal, while ($\rightarrow$-ELIM) (and, more generally speaking, any structural decomposition rule) requires two constructed types. As a consequence, no backtracking is required in the algorithm.

Besides, when several rules apply to the goal at hand, the algorithm shall of course choose a non-recursive rule—in particular, rule (HIST)—rather than the recursive one. This is indeed necessary to ensure termination.

Let us now show that the process terminates. If it does not, then the algorithm builds an infinite derivation branch; let us show that this is impossible. Consider such a branch. It must consist solely of (V-ELIM) and of structural decomposition rules, since these are the only recursive rules. It cannot consist solely of structural decomposition rules, because these rules strictly reduce the size of the goal. Hence, an infinite number of (V-ELIM) rules must appear in the branch.

The derivation rules are such that in any derivation of $C, H \vdash \tau \leq \tau'$, the operands of any goal are combinations using $\sqcup$ or $\sqcap$ of terms which appear in C, τ or τ'. There is a finite number of such terms, so there is a finite number of subgoals in any derivation.

In particular, our infinite family of (V-ELIM) occurrences only has a finite number of conclusions. Rule (V-ELIM) adds its conclusion to the history; so, eventually, all of them must appear in the history set. At this point, it is possible to end the branch with a (HIST) rule. Since the algorithm gives higher priority to (HIST) than to other rules, it could not possibly build this infinite branch. □

To assess the algorithm's complexity, we shall assume that the constraint graph at hand has no occurrences of $\sqcup$ or $\sqcap$, and that it verifies the small terms invariant—which we haven't used yet in this chapter. This hypothesis, as explained above, does not lessen the algorithm's power, but makes the analysis simpler.

Under this condition, the goals encountered by the algorithm are made up of either two small terms, or two variables. Given a goal made up of two variables, the operations to be carried out are invariably as follows. First, check whether (HIST) or (TAUTO) applies. If not, apply (V-ELIM), which produces a goal made up of two small terms. If this goal is insolvable, then the algorithm fails; otherwise, one of the structural decomposition rules applies and leads, again, to a number of goals made up of two variables.

Thus, the history set contains only goals made up of two variables. So, if the constraint graph contains n variables, the size of the history set is bounded by $O(n^2)$. In addition, if all possible goals appear in the history set, then (HIST) must apply, and the current branch succeeds. Hence, the depth of a branch is also bounded by $O(n^2)$. It follows that the algorithm has an exponential complexity bound.

The same reasoning can be applied to Amadio and Cardelli's algorithm [9]. However, the problem studied by them can be solved in time $O(n^2)$, as pointed out by Kozen, Palsberg and Schwartzbach [32], who propose an algorithm based on automata. The same enhancement is possible in our case; here is its principle, explained in a particularly simple way.

When we defined our axiomatization, we could easily have avoided using a history set and introducing rule (HIST). In that case, we would simply have worked with *regular* proof trees, rather than demanding the existence of a *finite* derivation. In fact, introducing a goal g into the history set corresponds to introducing a μg binder, and a use of g through the (HIST) rule is essentially a reference to the node where g was bound. Hence, the algorithm we described builds a regular proof tree, described in a finite way using μ binders. However, this description method isn't optimal; it is better to build a *term automaton*, which enables a more compact representation.

Building an automaton which describes the proof tree is very simple. The automaton has n^2 states, which are the goals made up of two variables; the initial state is the initial goal. To build the transitions, we first consider the initial state. We perform the elementary step described above. That is, we first check whether this state has already been dealt with; if not, we apply either (TAUTO), or (V-ELIM) followed by a structural decomposition rule. If the latter produces any sub-goals, then we create a transition towards each of them, labeled 0 or 1, as appropriate; then, we repeat the same process for each of these states. If, during the process, an insolvable constraint is encountered, then the algorithm fails; otherwise, we have built a regular derivation, described by an automaton. Each state, or goal, is dealt with at most once, so the algorithm's complexity is $O(n^2)$.

The principle of this enhancement is identical to the one used by Kozen, Palsberg and Schwartzbach to improve Amadio and Cardelli's algorithm. However, it seemed interesting to point out that, in each case, the naive and the efficient versions of the algorithm actually compute the same object, namely a regular derivation, but produce different representations of it.

Finally, note that the efficient version can be informally described in a couple of words. Just implement the naive version, as given by figure 8.2, and then refine it by sharing the history set between branches. In other words, make the history set a global variable, where goals are only *added*; it shall no longer shrink upon return of the recursive calls. This fact was remarked by Trifonov and Smith [46], but without this justification.

8.5 Incompleteness

Unfortunately, we have not proved entailment to be decidable, because our axiomatization turns out not be complete.

Proposition 8.4 *The axiomatization of entailment is incomplete. That is,* $C \Vdash \tau \leq \tau'$ *does* not *imply* $C \vdash \tau \leq \tau'$.

Proof. We give several counter-examples below. □

Example. Let C be the constraint graph associated to $\{\alpha \rightarrow \bot \leq \alpha\}$. Let τ be the type $(\bot \rightarrow \top) \rightarrow \bot$.

We claim that $C \Vdash \tau \leq \alpha$. Indeed, consider a solution ρ of C. ρ satisfies $\alpha \rightarrow \bot \leq \alpha$, so it must map α to a ground type whose head constructor is $\top$ or $\rightarrow$. In the former case, ρ also satisfies $\tau \leq \alpha$. In the latter case, $\rho(\alpha)$ is less than the greatest function type: $\rho(\alpha) \leq \bot \rightarrow \top$. Besides, we have $\rho(\alpha) \rightarrow \bot \leq \rho(\alpha)$. By transitivity, it follows that $(\bot \rightarrow \top) \rightarrow \bot \leq \rho(\alpha)$, so ρ satisfies $\tau \leq \alpha$.

On the other hand, one verifies that $C \Vdash \tau \leq \alpha \rightarrow \bot$ does not hold. A ground substitution ρ which maps α to $\top$ is a solution of C; yet it is not a solution of $\tau \leq \alpha \rightarrow \bot$.

Now, it is easy to verify that $C \vdash \tau \leq \alpha$ does not hold. While trying to build a derivation, the algorithm applies rule (V-ELIM), yielding $\tau \leq \alpha \rightarrow \bot$ as a new goal. But we have just shown this goal to be false in the model; the existence of a derivation for it would contradict theorem 8.1. Hence, the algorithm fails.

We have just shown that rule (V-ELIM) is essentially incomplete. It is easy to verify that it is actually the sole source of incompleteness. That is, in all other rules, the premises are equivalent (with respect to entailment) to the conclusion.

Example. One could think that the incompleteness, in the above example, comes from the fact that a type variable appears in a contravariant position in its own bound. However, here a slight variation of the same counter-example where this is no longer the case.

Let C be the empty constraint graph of domain $\{\alpha\}$. Let τ be defined as in the previous example. Then, one verifies that $C \Vdash \tau \leq \alpha \sqcup (\alpha \rightarrow \bot)$ holds; the proof is essentially identical to the previous one.

However, applying rule (V-ELIM) to this goal produces the new goal $\tau \leq \alpha \rightarrow \bot$, because α has been replaced with its lower bound in C, which is $\bot$. Again, this new goal is not entailed by C.

So, rule (V-ELIM) remains incomplete even in the absence of recursive constraints (or of any constraints at all, for that matter) in the constraint graph. Here, the incompleteness seems to come from the fact that the algorithm cannot predict the value of $\alpha \sqcup (\alpha \rightarrow \bot)$ in a fine way when α varies. An algorithm based on a case analysis (considering that any solution of C must map α to either $\bot$, $\top$, or a function type, and analyzing each case separately) would not have this problem. However, guaranteeing its termination appears to be difficult. In the case where α stands for a function type, the algorithm introduces two fresh type variables to stand for its domain and its codomain. Thus, the size of the problem can increase as the analysis proceeds, and finding a smart termination criterion is non-trivial.

Example. Finally, by looking at the previous counter-examples, one might think that the incompleteness stems from the presence of a contravariant type constructor. To dispel this belief, let us give a counter-example which does not require a contravariant type constructor. It was devised by Joachim Niehren.

Assume that the type language contains a unary, covariant type constructor F (see section 14.3). Let C be the constraint graph associated to

$$\{F(\alpha) \leq \beta, \alpha \leq F(\beta)\}$$

Then, we claim that $C \Vdash \alpha \leq \beta$. To prove it, we shall verify, by induction on k, that $C \Vdash_k \alpha \leq \beta$ holds for all $k \in \mathbb{N}^+$. The case $k = 0$ is immediate. Assume the result holds for a given k, and let ρ be a $(k+1)$-solution of C. If ρ maps α to $\bot$ or β to $\top$, then ρ verifies $\alpha \leq \beta$ and we have the desired result. Otherwise, since $k \geq 0$, ρ is a 1-solution of

$\alpha \leq F(\beta)$, and it follows that $\rho(\alpha)$ is of the form $F(\tau_\alpha)$, where τ_α is some ground type. Likewise, $\rho(\beta)$ is of the form $F(\tau_\beta)$. From the fact that ρ is a $(k+1)$-solution of C, we deduce that $F(\tau_\alpha) \leq_k \tau_\beta$ and $\tau_\alpha \leq_k F(\tau_\beta)$. In other words, a ground substitution ρ' which maps α to τ_α and β to τ_β is a k-solution of C. According to the induction hypothesis, ρ' k-satisfies $\alpha \leq \beta$, that is, $\tau_\alpha \leq_k \tau_\beta$. It follows that $F(\tau_\alpha) \leq_{k+1} F(\tau_\beta)$, i.e. ρ $(k+1)$-satisfies $\alpha \leq \beta$. The induction is complete; we have shown $C \Vdash \alpha \leq \beta$.

Now, it is easy to verify that no derivation exists for this goal. Rule (V-ELIM) turns $\alpha \leq \beta$ into $F(\beta) \leq F(\alpha)$, which is in turn rewritten as $\beta \leq \alpha$ by the structural decomposition rule associated to the covariant type constructor F. However, $C \Vdash \beta \leq \alpha$ is false, since a ground substitution which maps α to $\bot$ and β to $\top$ is a solution of C. So, once again, rule (V-ELIM) is incomplete.

The essence of the incompleteness is difficult to grasp here. One notes that the previously mentioned case-based approach would work perfectly, since decomposing the problem yields another instance of the same problem, with renamed variables, which is easy to detect. However, there are situations where such a decomposition yields a "trail" of garbage type variables, and detecting a repeating pattern is non-trivial.

So, we leave the decidability of entailment as an open problem. Only a complexity lower bound is known: Henglein and Rehof [27] show the problem to be PSPACE-hard.

Chapter 9

Deciding scheme subsumption

The aim of this section is to present an algorithm which decides whether two given type schemes are in the polymorphic subsumption relation; or, in other words, whether the former is more general than the latter. This algorithm is due to Trifonov and Smith [46]; however, they omit its detailed proof. It is a generalization of the entailment algorithm developed in chapter 8; as a matter of fact, it can be seen as a combination of it with the closure algorithm. It suffers from the same incompleteness problems as the entailment algorithm.

This algorithm is used, in practice, to compare the type scheme inferred for a certain expression with the one given by the user in the module signature. Besides, the algorithm also provides a theoretical basis for one of our simplification methods, namely garbage collection.

For the sake of simplicity, we shall require the right-hand scheme to be *simple*, i.e. to not contain any occurrences of $\sqcup$ or $\sqcap$. The algorithm cannot be naturally extended to the general case. This restriction causes no loss of power, since any type scheme is equivalent to some simple type scheme (see chapter 11).

We begin with some rather unimportant, technical preliminaries (section 9.1). Then, we introduce the notion of *weakly closed extension*, which shall form the theoretical basis for our algorithm (section 9.2). Section 9.3 explains, in a mostly informal way, the ideas which lead to its design. Finally, the algorithm itself is defined and proved in section 9.4.

9.1 Preliminaries

This section presents a slight extension of the axiomatization of entailment developed in chapter 8, which allows it to deal with a larger class of goals. It is of little complexity as well as of little interest.

Section 9.2 introduces the notion of *weak closure*. To this end, it shall need to deal with constraints of the form $\tau \leq \tau'$, where τ and τ' are *arbitrary* small terms. That is, each of τ and τ' is allowed to be a pos-type or a neg-type. So, let us define

Definition 9.1 *A* leaf constraint *is a pair of two leaf terms τ and τ', written $\tau \leq \tau'$. A* small constraint *is a pair of two small terms τ and τ', written $\tau \leq \tau'$.*

Recall that the original axiomatization deals with goals of the form $\tau \leq \tau'$, where $(\tau, \tau') \in \mathcal{T}^- \times \mathcal{T}^+$. Here, we wish to be able to specify leaf and small constraints as goals. This is quite straightforward, since these new goals have very restricted shapes.

We extend the axiomatization to leaf constraints by adding the two rules given in figure 9.1. Note that this extension of the axiomatization is not "recursive". That is, given a

$$\frac{\forall \alpha \in V \quad C, H \vdash \alpha \leq \tau'}{C, H \vdash \sqcup V \leq \tau'} \quad (\sqcup\text{-Elim})$$

$$\frac{\forall \alpha' \in V' \quad C, H \vdash \tau \leq \alpha'}{C, H \vdash \tau \leq \sqcap V'} \quad (\sqcap\text{-Elim})$$

Figure 9.1: Axiomatization of entailment, extended to leaf constraints

leaf constraint as goal, each of the rules ($\sqcup$-Elim) and ($\sqcap$-Elim) can be applied at most once at the bottom of the derivation, and nowhere else. So, a derivation in this extended system is simply a family of derivations in the original system, possibly brought together using the new rules. In short, our extension is only a commodity which allows us to encode a conjunction of goals into a single one.

To prove that the algorithm is still correct, it suffices to verify that proposition 8.1 still holds; everything else follows. So, here is a new version of it.

Proposition 9.1 *Assume given a derivation of $C, H \vdash \tau \leq \tau'$, of depth k, where $\tau \leq \tau'$ is a leaf constraint. Then*

$$\forall j \leq k \quad C \Vdash_j \tau \leq \tau'$$

Proof. It suffices to consider the two new rules; since they are symmetric, let us treat ($\sqcup$-Elim). Assume $\forall \alpha \in V \quad C \Vdash_j \alpha \leq \tau'$; we must prove $C \Vdash_j \sqcup V \leq \tau'$. This is a straightforward consequence of proposition 1.6. □

We have thus extended the axiomatization of entailment to leaf constraints. Extending it to small constraints is now immediate. Indeed, given a small constraint, applying ($\bot$-Elim), ($\top$-Elim) or ($\rightarrow$-Elim) decomposes it into 0 or more leaf constraints. Let us re-state the correction theorem:

Theorem 9.1 *Let C be a constraint graph. Let c be a leaf or small constraint such that*

$$C \vdash c$$

Then

$$\forall k \in \mathbb{N}^+ \cup \{\infty\} \quad C \Vdash_k c$$

Proof. Identical to that of theorem 8.1, using proposition 9.1 instead of proposition 8.1. □

9.2 Weak closure

Definition 9.2 *Let C' be a constraint graph of domain V'. An* extension *of C' is a constraint graph C of domain $V \cup V'$ such that $\leq_C$, $C^\downarrow$ and $C^\uparrow$, when restricted to V', coincide with $\leq_{C'}$, $C'^\downarrow$, and $C'^\uparrow$, respectively.*

The aim of this section is to give a sufficient condition, given such an extension, for any solution of C' to be extensible to a solution of C. This condition will then be used as the theoretical basis of the subsumption algorithm.

The condition given below is a kind of closure condition, similar to that given in definition 3.10, with two main differences.

First, we are trying to find as weak a requirement as possible. So, we replace transitivity and containment with provable entailment assertions, that is, judgements derived in our axiomatization of entailment. The latter are simple, syntactic criteria, while the former are finer and allow more flexibility.

Second, the definition of closure deals with finding a solution for a constraint graph; here, we deal with a more general problem, that is, extending a given solution to a larger constraint graph. Because of this, variables of V and V' play different roles, and this distinction is used to weaken again our requirements.

Definition 9.3 *Let C' be a constraint graph of domain V'. Let C be an extension of C' to $V \cup V'$, where V and V' are disjoint. In the following, variables of V shall be denoted by $\alpha, \beta, \ldots$ while variables of V' shall be denoted by $\alpha', \beta', \ldots$*

C is said to be a weakly closed extension *of C' iff for all α, β, α', β', γ':*

- $\alpha' \leq_C \beta$ *and* $\beta \leq_C \gamma'$ *imply* $C \vdash \alpha' \leq \gamma'$;
- $\alpha' \leq_C \alpha$ *and* $\alpha \leq_C \beta$ *imply* $\alpha' \leq_C \beta$;
- $\alpha \leq_C \beta$ *and* $\beta \leq_C \beta'$ *imply* $\alpha \leq_C \beta'$;
- $\alpha \leq_C \alpha'$ *implies* $\exists \beta \geq_C \alpha \quad C \vdash C^{\downarrow}(\beta) \leq C^{\downarrow}(\alpha')$;
- $\alpha' \leq_C \alpha$ *implies* $\exists \beta \leq_C \alpha \quad C \vdash C^{\uparrow}(\alpha') \leq C^{\uparrow}(\beta)$;
- $\alpha \leq_C \beta$ *implies* $C \vdash C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$ *and* $C \vdash C^{\uparrow}(\alpha) \leq C^{\uparrow}(\beta)$;
- $C \vdash C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$.

Note that, in the conditions above, some entailment assertions make use of the extended axiomatization of entailment given in section 9.1. For instance, in the assertion $C \vdash C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$, both sides of the goal are pos-types, so the axiomatization given in chapter 8 does not suffice. Thanks to the small terms invariant, $C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$ is a small constraint, so the assertion fits within our extension.

We shall now proceed to prove the main property of weakly closed extensions. We begin with a variant of proposition 3.4:

Proposition 9.2 *Let $k \in \mathbb{N}^+$. If a constraint graph C provably entails a small constraint c (i.e. $C \vdash c$), then any k-solution of C is a $(k+1)$-solution of c.*

Proof. Let $k \in \mathbb{N}^+$. Since c involves two small terms, it is (by structural decomposition) equivalent to a set S of leaf constraints. Take $c' \in S$. Since the entailment algorithm also works by structural decomposition, $C \vdash c$ implies $C \vdash c'$. According to theorem 9.1, this implies $C \Vdash_k c'$. Thus, we have proved that any k-solution of C is a k-solution of S. However, any k-solution of S is a $(k+1)$-solution of c, by structural decomposition. □

Theorem 9.2 *Assume C is a weakly closed extension of C'. Then every solution of C' can be extended to a solution of C.*

Proof. The proof is similar, in principle, to that of theorem 3.1, with some important differences. First, since the definition of a weakly closed extension involves provable entailment rather than containment, proposition 9.2 shall be used instead of proposition 3.4. Second and foremost, since we prove a more general result with as few conditions as possible, the details are more delicate. However, the general idea remains the same: exhibit a ground substitution, and prove that it is a k-solution of C, for all $k \in \mathbb{N}^+$, by induction on k.

Let $V' = \mathrm{dom}(C')$ and $V \cup V' = \mathrm{dom}(C)$, where V and V' are disjoint. In the following, variables of V shall be denoted by $\alpha, \beta, \ldots$ while variables of V' shall be denoted by $\alpha', \beta', \ldots$

Let ρ' be a solution of C'. According to proposition 2.9, there exists a contractive system S', whose domain contains V', and whose unique solution coincides with ρ' on V'. Obviously, one can require $\mathrm{dom}(S') \cap V = \varnothing$.

Consider, then, the contractive system S, of domain $\mathrm{dom}(S') \cup V$, defined as follows:

$$\begin{aligned} \alpha' \in \mathrm{dom}(S') &\mapsto S'(\alpha') \\ \alpha \in V &\mapsto C^{\downarrow}(\alpha) \sqcup (\sqcup\{S'(\alpha')\,;\, \alpha' \in V' \wedge \alpha' \leq_C \alpha\}) \end{aligned}$$

S is indeed contractive, because any type in its image is constructed. Let ρ be the unique solution of S. ρ is in particular a solution of S', so it coincides with ρ' on V'; it is an extension of ρ'.

Note that for any $\alpha \in V$, we have

$$\begin{aligned} \rho(\alpha) &= \rho(C^{\downarrow}(\alpha)) \sqcup (\sqcup\{\rho(S'(\alpha'))\,;\, \alpha' \in V' \wedge \alpha' \leq_C \alpha\}) \\ &= \rho(C^{\downarrow}(\alpha)) \sqcup (\sqcup\{\rho(\alpha')\,;\, \alpha' \in V' \wedge \alpha' \leq_C \alpha\}) \end{aligned}$$

We shall now verify, by induction on $k \in \mathbb{N}^+$, that ρ is a k-solution of C. The result is immediate for $k = 0$; assume it holds for a given $k \in \mathbb{N}^+$.

First, consider two variables in the $\leq_C$ relation. There are four sub-cases to consider, depending on whether each of them is in V or V'.

1. $\alpha' \in V'$ and $\beta' \in V'$. Then $\alpha' \leq_{C'} \beta'$ (because C is an extension of C'). ρ is an extension of ρ', so it satisfies $\alpha' \leq \beta'$.

2. $\alpha' \in V'$ and $\alpha \in V$. Then, according to the above note,

$$\begin{aligned} \rho(\alpha) &= \rho(C^{\downarrow}(\alpha)) \sqcup (\sqcup\{\rho(\alpha')\,;\, \alpha' \in V' \wedge \alpha' \leq_C \alpha\}) \\ &\geq \rho(\alpha') \end{aligned}$$

   so ρ satisfies $\alpha' \leq \alpha$.

3. $\alpha \in V$ and $\alpha' \in V'$. We have

$$\rho(\alpha) = \rho(C^{\downarrow}(\alpha)) \sqcup (\sqcup\{\rho(\gamma')\,;\, \gamma' \in V' \wedge \gamma' \leq_C \alpha\})$$

   so it suffices to show that

$$\rho(C^{\downarrow}(\alpha)) \leq_{k+1} \rho(\alpha') \qquad (1)$$

$$\forall \gamma' \in V' \quad \gamma' \leq_C \alpha \Rightarrow \rho(\gamma') \leq_{k+1} \rho(\alpha') \qquad (2)$$

   C is a weakly closed extension of C' and $\alpha \leq_C \alpha'$; hence, there exists $\beta \in V$ such that $\alpha \leq_C \beta$ and $C \vdash C^{\downarrow}(\beta) \leq C^{\downarrow}(\alpha')$. Since $\alpha \leq_C \beta$, we use once again the fact that C is a weakly closed extension of C', and we obtain $C \vdash C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$. ρ is a k-solution of C; so, according to proposition 9.2, ρ is a $(k+1)$-solution of these two constraints. Hence,

$$\rho(C^{\downarrow}(\alpha)) \leq_{k+1} \rho(C^{\downarrow}(\alpha'))$$

   In addition, because $\alpha' \in V'$ and ρ is a solution of C',

$$\rho(C^{\downarrow}(\alpha')) \leq \rho(\alpha')$$

   By transitivity, we obtain (1). Now, consider $\gamma' \in V'$ such that $\gamma' \leq_C \alpha$. Since $\alpha \leq_C \alpha'$ and C is a weakly closed extension of C', we have $C \vdash \gamma' \leq \alpha'$. Consider the derivation of this assertion. Necessarily, one of the following must hold:

- $\gamma' \leq_C \alpha'$. Then, since $\gamma' \in V'$ and $\alpha' \in V'$, and since C is an extension of C', we have $\gamma' \leq_{C'} \alpha'$, and ρ satisfies $\gamma' \leq \alpha'$.
- $C \vdash C^\uparrow(\gamma') \leq C^\downarrow(\alpha')$. Then, since ρ is a k-solution of C, it is a $(k+1)$-solution of this constraint between constructed terms. Besides, since ρ is a solution of C', it satisfies $\gamma' \leq C'^\uparrow(\gamma') = C^\uparrow(\gamma')$ and $C^\downarrow(\alpha') = C'^\downarrow(\alpha') \leq \alpha'$. It follows that ρ $(k+1)$-satisfies $\gamma' \leq \alpha'$.

In both cases, ρ $(k+1)$-satisfies $\gamma' \leq \alpha'$, so we have proved (2).

4. $\alpha \in V$ and $\beta \in V$. We have

$$\rho(\alpha) = \rho(C^\downarrow(\alpha)) \sqcup (\sqcup\{\rho(\alpha')\,;\,\alpha' \in V' \wedge \alpha' \leq_C \alpha\})$$
$$\rho(\beta) = \rho(C^\downarrow(\beta)) \sqcup (\sqcup\{\rho(\beta')\,;\,\beta' \in V' \wedge \beta' \leq_C \beta\})$$

We shall show that each member of the $\sqcup$ expression on the first line above is less (at rank $k+1$) than some member of the $\sqcup$ expression on the second line.

Since $\alpha \leq_C \beta$ and C is a weakly closed extension of C', we have $C \vdash C^\downarrow(\alpha) \leq C^\downarrow(\beta)$. As shown before, because ρ k-satisfies C, it $(k+1)$-satisfies this constraint.

Now, consider $\alpha' \in V'$ such that $\alpha' \leq_C \alpha$. Since $\alpha \leq_C \beta$ and C is a weakly closed extension of C', we have $\alpha' \leq_C \beta$. Then α' is also one of the β' above.

We have verified that each element of the first line is less, at rank $k+1$, than some element of the second line; hence $\rho(\alpha) \leq_{k+1} \rho(\beta)$.

We have proved that ρ is a $(k+1)$-solution of any constraint in $\leq_C$.

Now, consider $\alpha \in V$. We shall show that ρ is a $(k+1)$-solution of $C^\downarrow(\alpha) \leq \alpha \leq C^\uparrow(\alpha)$. (This assertion is true for $\alpha' \in V'$, because ρ extends ρ', which is why we consider only $\alpha \in V$.) The first inequality is a simple consequence of the definition of $\rho(\alpha)$. Let us now consider the second one. We have

$$\rho(\alpha) = \rho(C^\downarrow(\alpha)) \sqcup (\sqcup\{\rho(\alpha')\,;\,\alpha' \in V' \wedge \alpha' \leq_C \alpha\})$$

We shall verify that each element of this $\sqcup$ expression is less, at rank $k+1$, than $\rho(C^\uparrow(\alpha))$.

Because C is a weakly closed extension of C', we have $C \vdash C^\downarrow(\alpha) \leq C^\uparrow(\alpha)$. As shown before, it follows that ρ is a $(k+1)$-solution of this constraint.

Now, consider $\alpha' \in V'$ such that $\alpha' \leq_C \alpha$. Because C is a weakly closed extension of C', there exists $\beta \in V$ such that $\beta \leq_C \alpha$ and $C \vdash C^\uparrow(\alpha') \leq C^\uparrow(\beta)$. Since $\beta \leq_C \alpha$ and C is a weakly closed extension of C', we also have $C \vdash C^\uparrow(\beta) \leq C^\uparrow(\alpha)$. Again, ρ $(k+1)$-satisfies these constraints. Besides, because $\alpha' \in V'$, ρ satisfies $\alpha' \leq C'^\uparrow(\alpha') = C^\uparrow(\alpha')$. Hence, ρ $(k+1)$-satisfies $\alpha' \leq C^\uparrow(\alpha)$.

We have now proved that ρ is a $(k+1)$-solution of C; the induction is complete, and ρ is a solution of C. □

It is interesting to note that as a corollary, we obtain a weaker condition for a constraint graph to admit a solution:

Definition 9.4 *A constraint graph C of domain V is said to be* weakly closed *iff for all $\alpha, \beta \in V$:*

- *$\alpha \leq_C \beta$ implies $C \vdash C^\downarrow(\alpha) \leq C^\downarrow(\beta)$ and $C \vdash C^\uparrow(\alpha) \leq C^\uparrow(\beta)$;*
- *$C \vdash C^\downarrow(\alpha) \leq C^\uparrow(\alpha)$.*

A type scheme $A \Rightarrow \tau \mid C$ is weakly closed iff C is.

Proposition 9.3 *Any weakly closed constraint graph admits a solution.*

Proof. Let C be a weakly closed constraint graph. C is a weakly closed extension of the empty graph, hence the result. □

The notion of weak closure shall be used only to prove the correctness of the scheme subsumption algorithm. It shall not otherwise be useful, because closure, which has a simpler definition, shall fit our purposes. (The proof of the canonization algorithm, in chapter 11, is an exception; it shall require a third notion of closure, intermediate between closure and weak closure.)

From an implementation point of view, the idea of using weak closure, rather than plain closure, might come to mind. Indeed, the former puts fewer requirements on the constraint graph; so, one might imagine that a weak closure computation produces a smaller graph, thus helping the simplification process. Actually, it is much easier, and much more efficient, to work with plain closure, for several reasons.

The first reason, which isn't the best one, but came up first historically, is in the difficulty of designing an incremental weak closure algorithm. Indeed, it is easy to *check* that a given constraint graph is weakly closed, thanks to the entailment algorithm; but it is difficult, given a constraint graph, to compute the smallest equivalent weakly closed graph (insofar as such a thing exists, which is probably false in general). In the case of closure, we have a simple, eager algorithm (see definition 7.1). It keeps adding constraints that *must* be present for the graph to be closed, until it reaches a fix-point. In the case of weak closure, the use of entailment makes things trickier. It might appear necessary to add a constraint, because it is not entailed by the graph in its current state; but that constraint might later turn out to be unnecessary, because it can be deduced from other constraints which have been added after it. In other words, we are trying to compute a fix-point for a function which is not monotonous. Working around this problem might be possible, but definitely not worth the trouble.

Indeed, the second reason is simple and definitive. We shall see, in chapter 12, that when the *mono-polarity invariant* is enforced, any *garbage-collected* constraint graph is automatically closed and weakly closed. Thus, there is no practical difference between the two notions anymore. Furthering our investigation of weak closure is not warranted.

9.3 Principle

Equipped with the theoretical tools developed in the previous section, we are now able to give a sufficient condition for two type schemes to be in the subsumption relation. This condition will then serve as the theoretical basis for the algorithm. Let us begin with an informal explanation of the underlying intuition.

Recall that the assertion

$$A \Rightarrow \tau \mid C \leq^{\forall} A' \Rightarrow \tau' \mid C'$$

is equivalent to

$$\forall \rho' \vdash C' \quad \exists \rho \vdash C \quad \rho(A \Rightarrow \tau) \leq \rho'(A' \Rightarrow \tau')$$

That is, to show that a subsumption assertion holds, we must be able, given any instance of the right-hand scheme, to produce a smaller instance of the left-hand one.

Consider the case where the right-hand side contains no type variables. Then, the problem becomes a simple solvability problem, since it is is equivalent to determining whether $C + (A \Rightarrow \tau \leq A' \Rightarrow \tau')$ admits a solution. This problem, as shown previously, is decidable:

it suffices to compute the closure of this set. If the computation fails, there is no solution; if it succeeds, we obtain a closed form, off which a solution can be read straightforwardly.

Now, let us come back to the case where the right-hand scheme's domain, V', is non-empty. (We shall assume that the two schemes have disjoint domains, which is not restrictive, since subsumption contains α-conversion.) Once again, we must determine whether the above set admits a solution, but this time, we have no control over the value of the variables in V'. So, it seems natural to once again try computing a closure, but this time, considering the variables of V' as unknown constants. We must then fail if any new constraints concerning them appear, since we are not allowed to choose their values. If the computation succeeds, then we have solved the problem. Informally speaking, the closed graph then describes the values that must be given to the left-hand variables, in terms of the values of the right-hand ones. Trifonov and Smith [46] term the latter *rigid*, while the former are called *flexible*, in keeping with the idea that only they are free to vary.

A slight correction must be made to the above discussion: instead of using the regular definition of closure, we shall replace it with the notion of weak closure introduced in section 9.2. The former, which is less precise, would require adding more constraints. In particular, it might lead us to add new constraints on rigid variables, which would cause the algorithm to fail. Weak closure is finer, so it eases this problem: in many cases, these new constraints are provably entailed by existing ones, and we do not have to add them, thus avoiding the failure. (Nevertheless, the problem still exists, since weak closure is based on an incomplete axiomatization of entailment, and our new algorithm shall be incomplete too.) Thus, rigid variables are considered as unknown constants, but they are known to satisfy the constraints expressed by the graph C', and we use this information through entailment.

Example. Consider the assertion

$$\alpha \to \alpha \leq^{\forall} \beta' \to \gamma' \mid C'$$

where $C' = \varnothing + (\beta' \leq \gamma')$. Let us try to compute a weakly closed extension of C' which entails

$$\alpha \to \alpha \leq \beta' \to \gamma'$$

We add to C' the constraints $\beta' \leq \alpha$ and $\alpha \leq \gamma'$. By transitivity, this requires $\beta' \leq \gamma'$ which is a constraint on variables of V'. However, it turns out that this constraint is already entailed by C', so we have a weakly closed extension, and we stop.

Now, suppose we are given a solution ρ' of C'. Define ρ by choosing $\rho(\alpha)$ as any ground type which is between $\rho'(\beta')$ and $\rho'(\gamma')$; then it is easy to verify that ρ is a suitable witness for the subsumption assertion. So, the weakly closed extension computed above is a solved form of the problem; it explains, given any instance of the right-hand scheme, how to build an instance of the left-hand one which is smaller.

We shall now formalize this discussion.

Proposition 9.4 *Let $A \Rightarrow \tau \mid C$ and $A' \Rightarrow \tau' \mid C'$ be two type schemes of disjoint domains. Assume there exists a constraint graph D such that*

- *D is a weakly closed extension of C' to $\mathrm{dom}(C) \cup \mathrm{dom}(C')$;*
- *$D \Vdash C + (A \Rightarrow \tau \leq A' \Rightarrow \tau')$.*

Then

$$A \Rightarrow \tau \mid C \leq^{\forall} A' \Rightarrow \tau' \mid C'$$

Proof. Let ρ' be a solution of C'. D is a weakly closed extension of C'; according to theorem 9.2, there exists a solution ρ of D which extends ρ'. Because $D \Vdash C + (A \Rightarrow \tau \leq A' \Rightarrow \tau')$, ρ is a solution of C, and it satisfies

$$\rho(A \Rightarrow \tau) \leq \rho(A' \Rightarrow \tau')$$

which can also be written

$$\rho(A \Rightarrow \tau) \leq \rho'(A' \Rightarrow \tau')$$

since ρ extends ρ'. This ends the proof. □

As mentioned above, the reciprocal of this proposition is false, because the notion of weakly closed extension relies on our incomplete axiomatization of entailment. Thus, any example which shows the incompleteness of entailment can be coded into an example which shows the incompleteness of the above proposition. In other words, the subsumption algorithm to be developed in the next section has (at least) the same incompleteness examples as the entailment algorithm.

Proposition 9.5 *The reciprocal of proposition 9.4 is false.*

Proof. Assume given an entailment assertion, of the form $C \Vdash \alpha \leq \beta$, which is true but not provable, i.e. $C \vdash \alpha \leq \beta$ is false. (Such assertions exist and have been discussed in the proof of proposition 8.4.)

This entailment assertion can be coded into a subsumption assertion, like this:

$$\gamma \rightarrow \gamma \leq^{\forall} \alpha \rightarrow \beta \mid C$$

where we choose $\gamma \notin \mathrm{dom}(C)$. It is clear that there exists no weakly closed extension of C to $\mathrm{dom}(C) \cup \{\gamma\}$ which entails $\gamma \rightarrow \gamma \leq \alpha \rightarrow \beta$; indeed, according to definition 9.3, that would require precisely $C \vdash \alpha \leq \beta$. □

9.4 Algorithm

We are now ready to give a formal definition of the scheme subsumption algorithm, based on the ideas presented in the previous section.

Definition 9.5 *Let $\sigma = A \Rightarrow \tau \mid C$ and $\sigma' = A' \Rightarrow \tau' \mid C'$ be type schemes of disjoint domains. C is assumed to be weakly closed, and $\leq_C$ is assumed to be transitive. (No closure requirements are put on C'; however, the algorithm shall obtain better results if C' is closed.) Furthermore, σ' is assumed to be simple.*

By convention, variables of σ shall be denoted by α, β, ... while variables of σ' shall be denoted by α', β', ...

A state *of the algorithm is a pair (D, Q), where D is a constraint graph of domain* $\mathrm{dom}(C) \cup \mathrm{dom}(C')$ *and Q is a set of constraints of the form $\alpha \leq \alpha'$ or $\alpha' \leq \alpha$.*

If $\mathrm{dom}(A) \neq \mathrm{dom}(A')$*, the algorithm fails immediately. Otherwise, its* initial state *is* $(C \cup C', A \Rightarrow \tau \leq A' \Rightarrow \tau')$.

The algorithm switches *from a state (D, Q) to a new state as follows. If the waiting queue Q is empty, report success and terminate the algorithm. Otherwise, pick a constraint c in Q and set $Q' = Q \setminus \{c\}$. Two cases are then to be considered, depending on the form of c.*

- *c is of the form $\alpha \leq \alpha'$. If $\alpha \leq_D \alpha'$, switch to the state (D, Q'). Otherwise, do the following:*

   - *Define E by* $\leq_E = \leq_D \cup \{(\beta, \alpha')\,;\, \beta \leq_D \alpha\}$, $E^\uparrow = D^\uparrow$ *and* $E^\downarrow = D^\downarrow$.
   - *Define R as* $Q' \cup \mathrm{subc}(D^\downarrow(\alpha) \leq D^\downarrow(\alpha'))$.
   - *Verify* $\forall \beta' \leq_D \alpha \quad D \vdash \beta' \leq \alpha'$. *If this succeeds, switch to the state* (E, R)*; otherwise, report failure and terminate the algorithm.*

- *c is of the form $\alpha' \leq \alpha$. If $\alpha' \leq_D \alpha$, switch to the state (D, Q'). Otherwise, do the following:*
   - *Define E by* $\leq_E = \leq_D \cup \{(\alpha', \beta)\,;\, \alpha \leq_D \beta\}$, $E^\uparrow = D^\uparrow$ *and* $E^\downarrow = D^\downarrow$.
   - *Define R as* $Q' \cup \mathrm{subc}(D^\uparrow(\alpha') \leq D^\uparrow(\alpha))$.
   - *Verify* $\forall \beta' \geq_D \alpha \quad D \vdash \alpha' \leq \beta'$. *If this succeeds, switch to the state* (E, R)*; otherwise, report failure and terminate the algorithm.*

Starting from the initial state, the algorithm keeps switching to new states until failure or success is reported.

Example. Let us illustrate the algorithm's functioning. Consider the type scheme $\sigma = \alpha \to \beta \to \gamma \times \delta \mid C$, where C is the constraint graph defined by $\alpha \leq_C \gamma$, $\beta \leq_C \gamma$, $\alpha \leq_C \delta$ and $\beta \leq_C \delta$. We wish to verify that this scheme is equivalent to $\sigma' = \epsilon \to \epsilon \to \epsilon \times \epsilon$. (Please ignore the fact that these schemes do not satisfy the small terms invariant; this does not affect the principle of the algorithm.)

The assertion $\sigma \leq^\forall \sigma'$ trivially holds, since σ' can be obtained from σ through a simple substitution. Let us see how the algorithm behaves. The variables $\alpha \ldots \gamma$ are considered here as flexible, while ϵ is rigid. The initial state of the algorithm is made up of the graph C (extended, to be precise, to the domain $\{\alpha \ldots \epsilon\}$), and of the waiting queue $\mathrm{subc}(\alpha \to \beta \to \gamma \times \delta \leq \epsilon \to \epsilon \to \epsilon \times \epsilon)$, which equals $\{\gamma \leq \epsilon, \delta \leq \epsilon, \epsilon \leq \alpha, \epsilon \leq \beta\}$. We shall not describe in detail the four steps necessary to remove these constraints from the waiting queue. In short, each of these constraints is added to the constraint graph. By transitivity, this creates constraints on rigid variables. For instance, since initially $\alpha \leq \gamma$ holds, adding $\epsilon \leq \alpha$ and $\gamma \leq \epsilon$ causes $\epsilon \leq \epsilon$ to appear. This constraint between rigid variables must be checked using the entailment algorithm, which is trivial. Thus, no error is detected, and the algorithm reports a success. Generally speaking, one could prove that the algorithm always succeeds when the right-hand scheme can be obtained from the left-hand one through a substitution.

Reciprocally, it might not be clear, at first sight, that $\sigma' \leq^\forall \sigma$. Let us observe the algorithm's behavior. This time, ϵ is the only flexible variable—all others are rigid. The initial constraint graph is unchanged, but the waiting queue is now $\{\epsilon \leq \gamma, \epsilon \leq \delta, \alpha \leq \epsilon, \beta \leq \epsilon\}$. Once again, these four constraints are added to the constraint graph. By transitivity on ϵ, constraints on rigid variables appear, which must be checked using the entailment algorithm. For instance, $\alpha \leq \epsilon$ and $\epsilon \leq \gamma$ yield $\alpha \leq \gamma$. However, this constraint is already present in C, so it is provably entailed by C. The same holds for all other constraints between rigid variables thus obtained, and the algorithm succeeds.

Let us comment on this result. The algorithm has proved the scheme subsumption assertion, i.e. that for all values of $\alpha \ldots \delta$ such that C, there exists a value of ϵ such that $\epsilon \to \epsilon \to \epsilon \times \epsilon \leq \alpha \to \beta \to \gamma \times \delta$. The computation performed by the algorithm consists, as explained when presenting the intuition behind the algorithm, in giving an explicit definition of ϵ in terms of $\alpha \ldots \delta$. Here, this definition is $\alpha \sqcup \beta \leq \epsilon \leq \gamma \sqcap \delta$. In other words, any value of ϵ which satisfies these conditions is suitable. There exists at least one, according to our hypotheses on $\alpha \ldots \delta$; we verified it using the entailment algorithm. If the computation had revealed a constraint on $\alpha \ldots \delta$ which could not be proved using our hypotheses, then ϵ's existence would not have been guaranteed, and the algorithm would have failed.

The equivalence between σ and σ' allows a simplification: if we replace σ with σ' in a type inference derivation, the derivation shall still yield a principal type scheme. A question remains: are we capable of detecting this opportunity, in practice? An affirmative answer shall be given in chapter 13 by the minimization algorithm. (This example shall be further investigated there, and the simplification referred to as *eliminating 2-crowns*.)

The algorithm is now defined. There remains to show that it terminates and that it is correct.

Proposition 9.6 *The subsumption algorithm terminates.*

Proof. If the algorithm does not terminate, then there exists an infinite switching sequence. Note that when a constraint is picked, if it is already in the relation $\leq_D$, then the size of the queue strictly decreases. On the other hand, if it is not in $\leq_D$, then it is immediately added to it (see the definition of $\leq_E$). As a consequence, if $(D_n, Q_n)_{n\in\mathbb{N}^+}$ is an infinite switching sequence, the sequence $(\leq_{D_n})_{n\in\mathbb{N}^+}$ is non-decreasing, and strictly increases an infinite number of times. However, $(\leq_{D_n})_{n\in\mathbb{N}^+}$ is bounded, because all D_n's have the same domain, namely $\mathrm{dom}(C) \cup \mathrm{dom}(C')$. Hence, no infinite switching sequence can exist, and the algorithm terminates. $\square$

Here are two lemmas which constitute the bulk of the correctness proof.

Lemma 9.7 *Assume that the algorithm reaches a state (D, Q). Then, D is an extension of C' to $\mathrm{dom}(C) \cup \mathrm{dom}(C')$, and it is an extension of C to $\mathrm{dom}(C) \cup \mathrm{dom}(C')$.*

Proof. This is true of the initial state: since σ and σ' have disjoint domains, $C \cup C'$ is an extension of C' (resp. C) to $\mathrm{dom}(C) \cup \mathrm{dom}(C')$.

Now, assume the algorithm switches from state (D, Q) to state (E, R) and D is an extension of C' (resp. C) to $\mathrm{dom}(C) \cup \mathrm{dom}(C')$. According to the definition of the algorithm, the only difference between D and E is that $\leq_E$ might contain additional constraints of the form $\alpha \leq_E \alpha'$ or $\alpha' \leq_E \alpha$. So, the restriction of E to $\mathrm{dom}(C')$ (resp. $\mathrm{dom}(C)$) coincides with the restriction of D to $\mathrm{dom}(C')$ (resp. $\mathrm{dom}(C)$). It follows that E is also an extension of C' (resp. C) to $\mathrm{dom}(C) \cup \mathrm{dom}(C')$. $\square$

Lemma 9.8 *Assume that the algorithm reaches a state (D, Q). Then*

1. $\alpha' \leq_D \beta$ *and* $\beta \leq_D \gamma'$ *imply* $D \vdash \alpha' \leq \gamma'$;
2. $\alpha' \leq_D \alpha$ *and* $\alpha \leq_D \beta$ *imply* $\alpha' \leq_D \beta$;
3. $\alpha \leq_D \beta$ *and* $\beta \leq_D \beta'$ *imply* $\alpha \leq_D \beta'$;
4. $\alpha \leq_D \alpha'$ *implies* $\exists \beta \geq_D \alpha \quad D, Q \vdash D^{\downarrow}(\beta) \leq D^{\downarrow}(\alpha')$;
5. $\alpha' \leq_D \alpha$ *implies* $\exists \beta \leq_D \alpha \quad D, Q \vdash D^{\uparrow}(\alpha') \leq D^{\uparrow}(\beta)$;
6. $D, Q \vdash A \Rightarrow \tau \leq A' \Rightarrow \tau'$.

(Note that in some of the entailment assertions above, the contents of Q are made into axioms.)

Proof. First, consider the initial state (D, Q), where $D = C \cup C'$. Because C and C' have disjoint domains, there are no constraints of the form $\alpha \leq \alpha'$ or $\alpha' \leq \alpha$ in $\leq_D$, so the five first conditions are trivially satisfied. Besides, the waiting queue is precisely $Q = A \Rightarrow \tau \leq A' \Rightarrow \tau'$; so the result holds for the initial state.

Next, assume that the result holds for a state (D, Q) (this is the induction hypothesis) and that the algorithm switches from this state to the state (E, R). Assume that the constraint c which has been picked out of Q is $\alpha \leq \alpha'$; the other case is symmetric. Set $Q' = Q \setminus \{c\}$.

If $\alpha \leq_D \alpha'$, then $(E, R) = (D, Q')$. It suffices to show that if a constraint is provably entailed by (D, Q), then it is also provably entailed by (D, Q'). The only difference is that the axiom $\alpha \leq \alpha'$ is missing in the second case. However, this axiom is derivable since $\alpha \leq_D \alpha'$. So, the result holds.

Now, assume $\alpha \not\leq_D \alpha'$. We have

$$\begin{aligned} \leq_E &= \leq_D \cup \{(\beta, \alpha')\,;\, \beta \leq_D \alpha\} \\ R &= Q' \cup \mathrm{subc}(D^{\downarrow}(\alpha) \leq D^{\downarrow}(\alpha')) \end{aligned}$$

Consider the first requirement. Assume $\gamma' \leq_E \beta$ and $\beta \leq_E \delta'$. Necessarily, $\gamma' \leq_D \beta$. If $\beta \leq_D \delta'$, then $D \vdash \gamma' \leq \delta'$ by induction hypothesis, and $E \vdash \gamma' \leq \delta'$ because E contains D. Otherwise, we have $\delta' = \alpha'$ and $\beta \leq_D \alpha$ by definition of $\leq_E$. We now have $\gamma' \leq_D \beta \leq_D \alpha$. By applying the induction hypothesis, we obtain $\gamma' \leq_D \alpha$. Now, according to the definition of the algorithm, the statement

$$\forall \beta' \leq_D \alpha \quad D \vdash \beta' \leq \alpha'$$

was verified prior to switching to the state (E, R). It follows that $D \vdash \gamma' \leq \alpha'$, and $E \vdash \gamma' \leq \alpha'$ because E contains D. The first requirement is met.

The second requirement is trivially satisfied, because it does not involve any of the newly added constraints.

Consider the third requirement. Assume $\gamma \leq_E \beta$ and $\beta \leq_E \delta'$. Necessarily, $\gamma \leq_D \beta$. If $\beta \leq_D \delta'$, we conclude easily, as above. Otherwise, we have $\delta' = \alpha'$ and $\beta \leq_D \alpha$ by definition of $\leq_E$. However, since each state switch adds "heterogeneous" constraints of the form $\delta \leq \delta'$ or $\delta' \leq \delta$, $\gamma \leq_D \beta$ implies $\gamma \leq_C \beta$. Similarly, $\beta \leq_D \alpha$ implies $\beta \leq_C \alpha$. Because $\leq_C$ is transitive, $\gamma \leq_C \alpha$, which implies $\gamma \leq_D \alpha$. By definition of E, it follows that $\gamma \leq_E \alpha'$. We have proved that $\leq_E$ verifies the third requirement.

Consider the fourth requirement. Assume $\beta \leq_E \delta'$. The case $\beta \leq_D \delta'$ is straightforward, so let us assume $\beta \not\leq_D \delta'$. Then, by definition of $\leq_E$, $\delta' = \alpha'$ and $\beta \leq_D \alpha$. Besides, according to the definition of R, it is immediate that

$$E, R \vdash E^{\downarrow}(\alpha) \leq E^{\downarrow}(\alpha')$$

Finally, we have $\beta \leq_E \alpha$, so the fourth requirement is satisfied.

The fifth requirement is trivially satisfied, because it does not involve any of the newly added constraints.

The sixth requirement is still satisfied, because the "missing axiom" $\alpha \leq \alpha'$ is derivable from E. □

We can finally state that

Theorem 9.3 *The subsumption algorithm is correct with respect to the subsumption relation.*

Proof. Assume that the algorithm reports a success. Then it has reached some state $(D, \varnothing)$. According to lemma 9.7, D is an extension of C', and of C, to $\mathrm{dom}(C) \cup \mathrm{dom}(C')$. According to lemma 9.8, we have

1. $\alpha' \leq_D \beta$ and $\beta \leq_D \gamma'$ imply $D \vdash \alpha' \leq \gamma'$;
2. $\alpha' \leq_D \alpha$ and $\alpha \leq_D \beta$ imply $\alpha' \leq_D \beta$;

3. $\alpha \leq_D \beta$ and $\beta \leq_D \beta'$ imply $\alpha \leq_D \beta'$;
4. $\alpha \leq_D \alpha'$ implies $\exists \beta \geq_D \alpha \quad D \vdash D^{\downarrow}(\beta) \leq D^{\downarrow}(\alpha')$;
5. $\alpha' \leq_D \alpha$ implies $\exists \beta \leq_D \alpha \quad D \vdash D^{\uparrow}(\alpha') \leq D^{\uparrow}(\beta)$;
6. $D \vdash A \Rightarrow \tau \leq A' \Rightarrow \tau'$.

Additionally, because C is a weakly closed constraint graph, we have, for all $\alpha, \beta \in \mathrm{dom}(C)$:

- $\alpha \leq_C \beta$ implies $C \vdash C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$ and $C \vdash C^{\uparrow}(\alpha) \leq C^{\uparrow}(\beta)$;
- $C \vdash C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$.

Because D is an extension of C, this can be rewritten

- whenever $\alpha \leq_D \beta$, $D \vdash D^{\downarrow}(\alpha) \leq D^{\downarrow}(\beta)$ and $D \vdash D^{\uparrow}(\alpha) \leq D^{\uparrow}(\beta)$;
- whenever $\alpha \in V$, $D \vdash D^{\downarrow}(\alpha) \leq D^{\uparrow}(\alpha)$.

The five first assertions, together with the last two above, tell precisely that D is a weakly closed extension of C' to $\mathrm{dom}(C) \cup \mathrm{dom}(C')$. Besides, we have $D \vdash A \Rightarrow \tau \leq A' \Rightarrow \tau'$, and $D \vdash C$ since D is an extension of C. According to proposition 9.4, we have $\sigma \leq^{\forall} \sigma'$. $\square$

Chapter 10

Polarities and garbage collection

A type scheme $A \Rightarrow \tau \mid C$ is but a description (albeit an approximate one) of an expression's data flow. The context A describes a set of data which are necessary for the expression to operate properly; thus, it represents a set of input points. The body τ, on the contrary, describes the value computed by the expression, so it represents an output point. They are linked together by the constraints in the graph C. Indeed, each function application, that is, each data transmission between a supplier and a consumer, generates a constraint. This corresponds to the idea that typing is a flow analysis.

This is an interesting view. In particular, it suggests classifying the type variables of a type scheme according to their functionality. More precisely, if σ is the type scheme associated to an expression e, it would be interesting to distinguish the type variables of σ which represent an input (i.e. some data expected by the expression e) from those which represent an output (i.e. some result supplied by e). We shall annotate each type variable with a $-$ sign in the former case, and a $+$ sign in the latter. Of course, it is possible for a variable to play both roles at once, and thus to carry both signs. Some variables, on the other hand, shall turn out to carry no sign at all. Thus, we shall associate a pair of boolean flags, which we call *polarity*, to each variable.

What is the point of this computation? It is the basis for the *garbage collection* process, which we shall also introduce in this chapter. This process consists in identifying, and then eliminating, certain superfluous constraints in the constraint graph. As we shall see, the polarity computation allows detecting a large class of superfluous constraints. Besides, polarities play an important role in the definitions of the mono-polarity invariant (see chapter 12) and of the minimization algorithm (see chapter 13). Finally, they are also used during the "external" simplification phase, which helps make a type scheme more readable prior to display (see section 15.2).

Rather than directly give the definition of polarity and of its direct application, garbage collection, we proceed in several steps. We begin with a coarser notion (section 10.1), which we shall then refine twice (sections 10.2 and 10.3) to obtain a definitive version. Finally, section 10.4 provides an interesting view of the garbage collection process, and proves its correctness.

10.1 A coarse definition

Rather than giving the definition of polarity right away, we first introduce a notion of *reachability*. It is computed by annotating each variable with a single flag, which indicates whether the variable plays a role in the data flow, without distinguishing input from output.

This coarser notion was proposed in [42]. Although it is no longer necessary to our theory, we use it as a starting point, and shall later show how it can be enhanced, in two independent ways, to obtain the notion of polarity.

Definition 10.1 *Let $\sigma = A \Rightarrow \tau \mid C$ be a closed type scheme. The set of* reachable *variables of σ, denoted by $\mathrm{R}(\sigma)$, is the smallest subset R of $\mathrm{dom}(\sigma)$ such that*

- $\mathrm{fv}(\tau) \subseteq R$
- $\forall (x : \tau_x) \in A \quad \mathrm{fv}(\tau_x) \subseteq R$
- $\forall \alpha \in R \quad (\alpha' \leq_C \alpha) \vee (\alpha \leq_C \alpha') \Rightarrow \alpha' \in R$
- $\forall \alpha \in R \quad \mathrm{fv}(C^{\downarrow}(\alpha)) \cup \mathrm{fv}(C^{\uparrow}(\alpha)) \subseteq R$

The intuition behind this definition is very simple. Our goal is to mark all variables which might play a role in σ's behavior. By σ's behavior, we mean the way it shall combine, during the typing derivation, with other schemes, eventually leading to a success or a failure of the derivation. By examining the typing rules, we notice that the constraints they explicitly create shall only involve σ's "interface", that is, its context A and its body τ. Thus, to begin with, we mark all variables of A and τ as reachable.

However, recall that the typing rules require that all type schemes have a non-empty denotation, that is, that their constraint graph admit a solution. To verify this condition, a closure computation shall be performed, which combines new constraints with the ones already present in C, using transitivity and structural decomposition. Thus, any variable of σ which might play a role in this computation must also be considered reachable. This translates to the last two conditions above, which express that any bound of a reachable variable is itself reachable.

To conclude this explanation, one could say that the reachability computation essentially simulates a future closure computation, and marks all variables which might be involved in it.

A simplification method, known as *removal of unreachable constraints* in [42], can be derived straightforwardly from this definition. If a variable is unreachable, then we are certain that it will not be involved in any future closure computation. So, any information about this variable is superfluous, and can be dropped without affecting the behavior of the type scheme.

We do not attempt to prove this result here, since it shall be generalized further on. Rather, let us give an example which shall be used throughout the forthcoming sections.

Example. Consider the type scheme $\sigma = \alpha \rightarrow \gamma \mid C$, where C is the constraint graph defined by

- $\alpha \leq_C \beta \leq_C \gamma$;
- $C^{\uparrow}(\alpha) = C^{\uparrow}(\beta) = C^{\uparrow}(\gamma) = \delta \rightarrow \epsilon$;
- $C^{\downarrow}(\lambda) = \alpha \rightarrow \gamma$;
- $C^{\downarrow}(\epsilon) = \top$.

To compute $\mathrm{R}(\sigma)$, we start by marking α and γ reachable. Then, we let the marks propagate (every bound of a reachable variable becomes reachable) until no more variables can be marked. We obtain

$$\mathrm{R}(\sigma) = \{\alpha, \beta, \gamma, \delta, \epsilon\}$$

So, λ is the only unreachable variable. If we remove unreachable constraints, we obtain a new type scheme $\alpha \rightarrow \gamma \mid D$, where D is defined by

- $\alpha \leq_D \beta \leq_D \gamma$;
- $D^\uparrow(\alpha) = D^\uparrow(\beta) = D^\uparrow(\gamma) = \delta \rightarrow \epsilon$;
- $D^\downarrow(\epsilon) = \top$.

The constraint concerning λ has been dropped. It is easy to see why this is correct. This constraint did not actually restrict the possible values of α or γ, so it had no effect on the denotation of the type scheme σ.

To sum up, the reachability analysis is a simulation of a hypothetical, future closure phase, and the removal of unreachable constraints is its natural application. In [46], Trifonov and Smith refine the notion of reachability, to obtain the notion of polarity. Its natural application is a more powerful simplification method, known as *garbage collection*. We will now explain the improvements they made. There are actually two independent enhancements. We shall discuss each of them in isolation.

10.2 First enhancement

The first enhancement consists in making the analysis "directional". As we have seen, the reachability analysis simulates a closure computation, and flags all variables which can potentially receive new constraints. However, the direction of these new constraints is not taken into account, i.e. the analysis does not draw a distinction between new lower bounds and new upper bounds. Yet, this distinction is fundamental.

So, we refine the analysis by annotating each variable with two flags. The first one indicates whether the variable can receive new lower bounds, and the second one indicates whether it can receive new upper bounds. By convention, the variable shall be called *negative* (resp. *positive*), and annotated with a $-$ (resp. $+$) sign, if the former (resp. latter) flag is set.

These marks are computed, as before, as a fix-point. We have mentioned before that when new constraints are grafted onto a type scheme σ, they necessarily concern its context A and its body τ. Let us examine the typing rules more closely. To bring new constraints onto the body τ, the expression e corresponding to σ must be passed to some function. But the constraint thus created always constitutes an *upper* bound for τ. Symmetrically, to bring new constraints onto an element x of the context A, the expression e must be λ-abstracted over x, and the function thus obtained must be applied to some argument. However, the constraint thus created always constitutes a *lower* bound for $A(x)$. Thus, at the beginning of the fix-point computation, it suffices to mark τ positive and A negative.

During the fix-point computation, marks propagate, as before, along constraints. However, the propagation rules are finer. If a variable is positive, then it might receive a new upper bound in the future; so, by transitivity, all of its lower bounds are in the same situation, and must also be marked positive. Thus, $+$ signs propagate downwards, and, symmetrically, $-$ signs propagate upwards.

Finally, note that if τ is a constructed term, adding a new upper bound to it actually leads to adding new upper bounds to the variables of $\mathrm{fv}^+(\tau)$ and new lower bounds to the variables of $\mathrm{fv}^-(\tau)$. (This is a consequence of the structural decomposition rules.) Thus, marking τ positive actually means marking $\mathrm{fv}^+(\tau)$ positive and $\mathrm{fv}^-(\tau)$ negative.

Let us now give a formal definition of the analysis. (Recall that this is not the definitive definition of polarity; another enhancement remains to be discussed in section 10.3.)

Definition 10.2 *Let $\sigma = A \Rightarrow \tau \mid C$ be a closed type scheme. The set of* positive *variables of σ, and the set of* negative *variables of σ, respectively denoted by $\mathrm{dom}^+(\sigma)$ and $\mathrm{dom}^-(\sigma)$, are the smallest subsets P and N of $\mathrm{dom}(\sigma)$ such that*

- $\mathrm{fv}^+(\tau) \subseteq P \wedge \mathrm{fv}^-(\tau) \subseteq N$
- $\forall (x : \tau_x) \in A \quad \mathrm{fv}^+(\tau_x) \subseteq N \wedge \mathrm{fv}^-(\tau_x) \subseteq P$
- $\forall \alpha \in P \quad \alpha' \leq_C \alpha \Rightarrow \alpha' \in P$
- $\forall \alpha \in N \quad \alpha \leq_C \alpha' \Rightarrow \alpha' \in N$
- $\forall \alpha \in P \quad \mathrm{fv}^+(C^\downarrow(\alpha)) \subseteq P \wedge \mathrm{fv}^-(C^\downarrow(\alpha)) \subseteq N$
- $\forall \alpha \in N \quad \mathrm{fv}^+(C^\uparrow(\alpha)) \subseteq N \wedge \mathrm{fv}^-(C^\uparrow(\alpha)) \subseteq P$

This analysis is finer than the previous one, since reachability marks propagate both ways, whereas $+$ signs propagate only downwards and $-$ signs upwards. Besides, as announced, the marks carried by a variable indicate its role as an input or an output. Indeed, if a variable α represents the type of some result output by the expression e, then it is possible to build a program P, containing e, in which this result it used. Recall that, in a system based on subtyping, any use of a piece of data generates a constraint $\tau_1 \leq \tau_2$, where τ_1 is the type of the data which is being supplied, and τ_2 is the type expected by the consumer. So, typing the expression P generates a new upper bound for α. Thus, α is necessarily positive. We have informally shown that polarities constitute a conservative approximation of each variable's role: if a variable does not carry the $+$ (resp. $-$) sign, then it plays no output (resp. input) role.

We can now apply this new analysis to obtain a simplification method. The principle is the same as before: a constraint must be kept only if it has a chance of playing a role at some point in the future. Consider, for instance, a non-negative variable α. It cannot receive any new lower bounds in the future. So, there is no point in keeping α's upper bound. Indeed, the only way the constraint $\alpha \leq C^\uparrow(\alpha)$ could be involved in a future closure computation is if it is combined, by transitivity, with a new lower bound of α. Symmetrically, we can drop the lower bound of any non-positive variable. Finally, a constraint between two variables $\alpha \leq \beta$ represents a lower bound for β and an upper bound for α, so it can be dropped only if β is non-positive and α is non-negative.

Thus, we obtain a simplification algorithm which is finer than the removal of unreachable constraints. Let us come back to the example introduced in the previous section.

Example. To compute $\mathrm{dom}^+(\sigma)$ and $\mathrm{dom}^-(\sigma)$, we start by marking $\alpha \to \gamma$ positive, that is, α negative and γ positive. Then, we let the marks propagate ($+$ signs downwards, $-$ signs upwards). We obtain

$$\mathrm{dom}^+(\sigma) = \{\alpha, \beta, \gamma, \delta\}$$
$$\mathrm{dom}^-(\sigma) = \{\alpha, \beta, \gamma, \epsilon\}$$

Thus, λ is the only variable which carries no marks at all. Once again, we shall thus dismiss all information about it. But the analysis also finds that ϵ is negative, so its lower bound does not have any meaning and can be dropped. Hence, the simplification yields a new type scheme $\alpha \to \gamma \mid E$, where E is defined by

- $\alpha \leq_E \beta \leq_E \gamma$;
- $E^\uparrow(\alpha) = E^\uparrow(\beta) = E^\uparrow(\gamma) = \delta \to \epsilon$.

10.3 Second enhancement and definitive version

We shall now introduce a second enhancement to the notion of reachability. It is entirely independent from the previous one, and somewhat more delicate, but of utmost importance. It consists in totally ignoring constraints between variables during the polarity computation.

Recall that the fundamental idea underlying the notion of reachability is a simulation of a future closure computation, which shall be actually performed when the current expression e becomes part of a larger program P. So far, we have tried to determine which variables could potentially be involved in this computation. However, ultimately, our only interest is in determining whether P is typable, i.e. whether the closure computation can fail. So, we can restrict our attention to those type variables which can potentially cause a failure. (Informally speaking, a type variable α is said to cause a failure when the constraint $C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$ cannot be decomposed because it is insolvable.)

Consider the variable β in the above example. It is both positive and negative. The only reason it is positive is because γ is positive, and the $+$ sign has propagated down the constraint $\beta \leq \gamma$. So, β might receive new upper bounds in the future; but every time β receives a new upper bound, we know that γ has just received the same bound. Assume β causes a failure after receiving a new bound τ. Then $C^{\downarrow}(\beta) \leq \tau$ is insolvable. But so is $C^{\downarrow}(\gamma) \leq \tau$, since $C^{\downarrow}(\gamma)$ contains $C^{\downarrow}(\beta)$. Hence, γ also causes a failure. Thus, we can disregard any failures caused by β, since they shall also be caused by γ.

One could say that the $+$ sign carried by β serves two purposes. The first is to signal that β might cause a failure. We have seen that this fact can be ignored. The second is to propagate to all of β's lower bounds. But, since $\beta \leq \gamma$, these are also part of γ's lower bounds, and γ is positive, so they already carry the $+$ sign for that reason. Thus, this second use is also redundant.

Let us conclude this informal discussion. We have verified that the $+$ sign carried by β actually brings no information. Consequently, it is possible not to mark β positive. We have thus discovered that it is not necessary for polarities to propagate from one variable to another, but only from a variable to its constructed bound.

As we shall see, this idea is fundamental, because the resulting simplification method shall be very powerful. Since β no longer carries any sign, all references to it shall be eliminated. This is excellent—β was indeed superfluous, since it was only an intermediate variable between α and γ, and carried no information of its own.

We can now integrate this second enhancement and give the definitive definition of polarity. This definition, as well as the *garbage collection* which stems from it, has been proposed by Trifonov and Smith [46]. Aiken and Fähndrich [4] also present this notion, with and without the second enhancement (which they refer to as "detecting truly intermediate variables").

Besides, we weaken our assumptions on the type scheme; plain closure is not necessary.

Definition 10.3 *Let $\sigma = A \Rightarrow \tau \mid C$ be a weakly closed type scheme. The set of* positive *variables of σ, and the set of* negative *variables of σ, respectively denoted by $\mathrm{dom}^+(\sigma)$ and $\mathrm{dom}^-(\sigma)$, are the smallest subsets P and N of $\mathrm{dom}(\sigma)$ such that*

- $\mathrm{fv}^+(\tau) \subseteq P \wedge \mathrm{fv}^-(\tau) \subseteq N$
- $\forall (x : \tau_x) \in A \quad \mathrm{fv}^+(\tau_x) \subseteq N \wedge \mathrm{fv}^-(\tau_x) \subseteq P$
- $\forall \alpha \in P \quad \mathrm{fv}^+(C^{\downarrow}(\alpha)) \subseteq P \wedge \mathrm{fv}^-(C^{\downarrow}(\alpha)) \subseteq N$
- $\forall \alpha \in N \quad \mathrm{fv}^+(C^{\uparrow}(\alpha)) \subseteq N \wedge \mathrm{fv}^-(C^{\uparrow}(\alpha)) \subseteq P$

A variable α is said to be positive *if $\alpha \in \mathrm{dom}^+(\sigma)$, and* negative *if $\alpha \in \mathrm{dom}^-(\sigma)$. It is said to be* bipolar *if it is positive and negative, and* neutral *if it is neither.*

Furthermore, the function $\mathrm{polarity}_\sigma$*, which maps any variable* $\alpha \in \mathrm{dom}(\sigma)$ *to a subset of* $\{+,-\}$*, is defined by*

$$\epsilon \in \mathrm{polarity}_\sigma(\alpha) \iff \alpha \in \mathrm{dom}^\epsilon(\sigma)$$

Example. Let us continue to study the example introduced in the previous sections. To compute $\mathrm{dom}^+(\sigma)$ and $\mathrm{dom}^-(\sigma)$, we begin by marking α negative and γ positive. Then, we perform the fix-point computation. This time, β receives no marks, because marks do not follow links between variables any more. For the same reason, γ does not become negative, and α does not become positive. We obtain

$$\mathrm{dom}^+(\sigma) = \{\gamma, \delta\}$$
$$\mathrm{dom}^-(\sigma) = \{\alpha, \epsilon\}$$

which is a much finer result than before. We shall now see how to perform garbage collection, given this data.

In section 10.2, we explained that the constructed upper (resp. lower) bound of a non-negative (resp. non-positive) variable can be dropped, because its presence cannot cause any failure during future closure computations. We have also indicated under which conditions a constraint of the form $\alpha \leq \beta$ could be dropped; however, our second enhancement allows us to refine this criterion.

Assume, for instance, that β is non-positive. Then, if β receives a new upper bound in the future, then there exists some γ such that $\beta \leq \gamma$ and γ receives the same upper bound. Then, by transitivity, we have $\alpha \leq \gamma$; so we can derive that α also receives this new bound, without using the constraint $\alpha \leq \beta$. Thus, this constraint plays no essential role in the closure computation, and it can be eliminated. In the symmetric case where α is non-negative, we obtain the same conclusion. Consequently, a constraint $\alpha \leq \beta$ must be kept only if α is negative and β is positive—a rather strong result.

Example. Let us now conclude the analysis of the example used throughout this discussion. As explained above, β now carries no signs, so the constraints $\alpha^- \leq \beta$ and $\beta \leq \gamma^+$ can be dropped. Actually, all references to β disappear. The link between α and γ is not broken, though; because $\leq_C$ is closed by transitivity, the constraint $\alpha^- \leq \gamma^+$ was present and remains.

As before, λ disappears entirely, and ϵ's lower bound is dropped. Additionally, this time, γ is non-negative, so its upper bound is also dropped. We are left with the type scheme $\alpha \to \gamma \mid F$, where F is defined by

- $\alpha \leq_F \gamma$;
- $F^\uparrow(\alpha) = \delta \to \epsilon$.

This type scheme is significantly simpler than σ. Garbage collection has thus eliminated a large quantity of superfluous information, at a low cost. In particular, it removes "truly intermediate variables", i.e. variables such as β, which only serve to form a link between other variables. This is fundamental, because such links are generated in large quantity by the type inference rules and by the canonization algorithm (introduced in chapter 11).

Let us now move on to the formal definition of garbage collection. For the sake of clarity, we first define the conditions that a type scheme must fulfill before the process can be applied to it.

Definition 10.4 *A type scheme* $\sigma = A \Rightarrow \tau \mid C$ *is* collectable *iff*

- *C is weakly closed, and $\leq_C$ is transitive;*
- *σ is simple.*

Actually, the second requirement above appears for a purely technical reason: the proof of the garbage collection algorithm relies on the subsumption algorithm, which has been developed for simple type schemes only. It would be possible to remove this requirement, but that would require more proof machinery. In practice, this restriction requires that canonization be performed before garbage collection.

That said, here is the definition proper:

Definition 10.5 *Let σ be a collectable type scheme. The image of σ through* garbage collection*, denoted by* $\mathrm{GC}(\sigma)$*, is the type scheme $A \Rightarrow \tau \mid D$, of domain* $\mathrm{dom}^+(\sigma) \cup \mathrm{dom}^-(\sigma)$*, where the constraint graph D is defined as follows:*

- *$\alpha \leq_D \beta$ iff $\alpha \leq_C \beta$, $\alpha \in \mathrm{dom}^-(\sigma)$ and $\beta \in \mathrm{dom}^+(\sigma)$;*
- *$D^\downarrow(\alpha)$ equals $C^\downarrow(\alpha)$ if $\alpha \in \mathrm{dom}^+(\sigma)$, and $\bot$ otherwise;*
- *$D^\uparrow(\alpha)$ equals $C^\uparrow(\alpha)$ if $\alpha \in \mathrm{dom}^-(\sigma)$, and $\top$ otherwise.*

10.4 Correctness

There remains to prove that this transformation is correct. Our goal is to show that the type schemes σ and $\mathrm{GC}(\sigma)$ have the same denotation, that is, $\sigma =^\forall \mathrm{GC}(\sigma)$. To this end, we shall use the algorithm developed in chapter 9.

About this proof, a very interesting remark can be formulated, which yields both a better understanding of the proof and an alternative definition of garbage collection, which is short and elegant. Imagine using the algorithm to determine whether $\sigma \leq^\forall \sigma$. (*A priori*, it is not obvious that the algorithm yields a positive answer, since it is incomplete.) By examining the behavior of the algorithm in such a situation, one finds out that the answer is always positive; but the most interesting, and unexpected, remark is that the algorithm is able to give a positive answer without making use of all of the constraints present in the right-hand member. A simplification method can be immediately derived from this remark: let σ' be a scheme identical to σ, except that all constraints unused by the algorithm in the previous experiment have been dropped. Then, we still have $\sigma \leq^\forall \sigma'$. Besides, $\sigma' \leq^\forall \sigma$ is immediate, since σ' contains fewer constraints. Thus, the two schemes are equivalent, and we have built a correct simplification process, which is none other than garbage collection.

This remark is interesting, because it gives a very succinct definition of garbage collection, while exposing its relationship with the subsumption algorithm. It justifies garbage collection in a *semantic* way, by proving that it does not affect a type scheme's denotation. On the other hand, the explanations given in the previous sections justify it in a more *operational* way, by proving that dropping these constraints shall not modify the type scheme's observable behavior.

Theorem 10.1 *Let σ be a collectable type scheme. Then*

$$\sigma =^\forall \mathrm{GC}(\sigma)$$

Proof. Set $\sigma' = \mathrm{GC}(\sigma) = A \Rightarrow \tau \mid D$ as in definition 10.5. Then, because D contains fewer constraints than C, it is clear that $C \Vdash D$. It follows that $\sigma' \leq^\forall \sigma$.

Reciprocally, we wish to prove $\sigma \leq^\forall \sigma'$. To do this, we shall simulate a run of the subsumption algorithm defined in section 9.4 and verify that the algorithm reports a success.

Because the algorithm works with two type schemes of disjoint domains, we first perform a step of α-conversion on σ'. For each $\alpha \in \mathrm{dom}(\sigma)$, we pick a fresh variable $\phi(\alpha)$.

We must verify that σ and $\phi(\sigma')$ are suitable inputs for the subsumption algorithm. They have disjoint domains; C is weakly closed, and $\leq_C$ is transitive; $\phi(\sigma')$ is simple.

We shall now show that whenever the algorithm reaches a state (D_n, Q_n), then

$$\begin{aligned}\leq_{D_n} \subseteq \leq_C \cup \leq_{\phi(D)} &\cup \{\beta \leq \phi(\alpha)\,;\, \alpha \in \mathrm{dom}^+(\sigma) \wedge \beta \leq_C \alpha\} \\ &\cup \{\phi(\alpha) \leq \beta\,;\, \alpha \in \mathrm{dom}^-(\sigma) \wedge \alpha \leq_C \beta\}\end{aligned}$$

and

$$\begin{aligned}Q_n \subseteq &\{\alpha \leq \phi(\alpha)\,;\, \alpha \in \mathrm{dom}^+(\sigma)\} \\ &\cup \{\phi(\alpha) \leq \alpha\,;\, \alpha \in \mathrm{dom}^-(\sigma)\}\end{aligned}$$

This is true of the initial state (D_0, Q_0), because $\leq_{D_0} = \leq_C \cup \leq_{\phi(D)}$ and $Q_0 = (A \Rightarrow \tau) \leq (\phi(A) \Rightarrow \phi(\tau))$. Assume it is true of a state (D_n, Q_n). Assume the algorithm reaches the next state (D_{n+1}, Q_{n+1}) by picking a constraint c. According to the induction hypothesis on Q_n, c is of the form $\alpha \leq \phi(\alpha)$, where $\alpha \in \mathrm{dom}^+(\sigma)$. (The other case is symmetric, so we do not treat it explicitly.) If this constraint is already in $\leq_{D_n}$, then $(D_{n+1}, Q_{n+1}) = (D_n, Q_n \setminus \{c\})$ and the result holds. Otherwise, we have $\leq_{D_{n+1}} = \leq_{D_n} \cup \{(\beta, \phi(\alpha))\,;\, \beta \leq_{D_n} \alpha\}$. $\beta \leq_{D_n} \alpha$ implies $\beta \leq_C \alpha$; since $\alpha \in \mathrm{dom}^+(\sigma)$, $\leq_{D_{n+1}}$ is of the expected form. Besides, we have $Q_{n+1} \subseteq Q_n \cup \mathrm{subc}(C^{\downarrow}(\alpha) \leq (\phi(D))^{\downarrow}(\phi(\alpha)))$. Because $\alpha \in \mathrm{dom}^+(\sigma)$, α has the same lower bound in D as in C, so $(\phi(D))^{\downarrow}(\phi(\alpha)) = \phi(C^{\downarrow}(\alpha))$. So, the constraints added to the queue are the subconstraints of $C^{\downarrow}(\alpha) \leq \phi(C^{\downarrow}(\alpha))$. According to the definition of dom^+ and dom^-, they are of the form $\beta \leq \phi(\beta)$ (resp. $\phi(\beta) \leq \beta$) where $\beta \in \mathrm{dom}^+(\sigma)$ (resp. $\beta \in \mathrm{dom}^-(\sigma)$). So, the result stated at the beginning of this paragraph holds.

We can now verify that the algorithm does not fail. The only way it can fail is if one of the entailment assertions that must be verified when switching states is not derivable. Consider an attempt to switch from state (D_n, Q_n) to a new state. Assume that the constraint c which has been picked out of Q_n is of the form $\alpha \leq \phi(\alpha)$, where $\alpha \in \mathrm{dom}^+(\sigma)$. (The other case is symmetric, so we do not treat it explicitly.) The assertions that must be verified are of the form $D_n \vdash \phi(\beta) \leq \phi(\alpha)$, where $\phi(\beta) \leq_{D_n} \alpha$. According to the result proved in the previous paragraph, $\phi(\beta) \leq_{D_n} \alpha$ implies $\beta \in \mathrm{dom}^-(\sigma)$ and $\beta \leq_C \alpha$. By definition of garbage collection, $\beta \in \mathrm{dom}^-(\sigma) \wedge \alpha \in \mathrm{dom}^+(\sigma) \wedge \beta \leq_C \alpha$ implies $\beta \leq_D \alpha$. It follows that $\phi(\beta) \leq_{\phi(D)} \phi(\alpha)$. Finally, since $\leq_{D_n}$ contains $\leq_{\phi(D)}$, the assertion $D_n \vdash \phi(\beta) \leq \phi(\alpha)$ is provable with a single use of (TAUTO).

We have verified that the algorithm does not fail. Hence, it must report a success, and the subsumption assertion holds. □

Since garbage collection does nothing but throw constraints away, the type scheme produced by it is simple and still verifies the small terms invariant. Furthermore, garbage collection preserves polarities, as shown by the following proposition.

Proposition 10.1 *Let σ be a collectable type scheme. Then*

$$\mathrm{dom}^+(\mathrm{GC}(\sigma)) = \mathrm{dom}^+(\sigma)$$
$$\mathrm{dom}^-(\mathrm{GC}(\sigma)) = \mathrm{dom}^-(\sigma)$$

Proof. Recall that definition 10.3 defines the sets of positive and negative variables of a type scheme as the smallest sets which satisfy a certain conjunction of conditions. By comparing the definitions of $\mathrm{dom}^+(\sigma)$ and $\mathrm{dom}^-(\sigma)$, on the one hand, and of $\mathrm{dom}^+(\mathrm{GC}(\sigma))$ and $\mathrm{dom}^-(\mathrm{GC}(\sigma))$, on the other hand, we find that these conditions are identical. The result follows. □

Lastly, it would be interesting to know whether the type scheme $\mathrm{GC}(\sigma)$ verifies some kind of closure property. This issue shall be discussed in chapter 12.

Chapter 11

Canonization

Until now, we have worked with type schemes where the constructors $\sqcup$ and $\sqcap$ were allowed to appear. As explained in section 2.1, they are a way of encoding conjunctions of constraints. However, several advanced algorithms, such as the minimization algorithm, have difficulty in dealing with them. Thus, it is desirable to eliminate them. Besides, doing so offers new opportunities to the garbage collection process, and thus constitutes a simplification in itself.

The goal of this chapter is to show that they can indeed be eliminated, without loss of expressivity, and to define an algorithm that performs the elimination. This process is called *canonization*, following Trifonov and Smith [46]. Their description of the algorithm is not entirely satisfactory, because it does not preserve the closure property. In practice, it is then necessary to have canonization followed with a new closure phase. To avoid this loss of efficiency, we hereby give a more precise description of the algorithm, and show that it preserves *simple closure*. Next, we show that most of the constraints produced to achieve closure shall actually be done away with by garbage collection. We can then define an algorithm which combines canonization and garbage collection, which avoids creating these superfluous constraints.

Besides, we shall see that the canonization algorithm can also be used to eliminate bipolar variables from a type scheme, so as to make it compliant with the *mono-polarity invariant* described in chapter 12. Thus, both tasks can be carried out in one pass by the algorithm.

This chapter begins with a short presentation of the principle of canonization (section 11.1). Next comes a section containing technical preliminaries (section 11.2), which describes a new notion of closure, well suited to the proof of the canonization process. This allows us to define a first notion, called *raw canonization* (section 11.3). We then simulate a run of the garbage collection algorithm on the type scheme produced by raw canonization, and define *canonization* as the combination of these two phases (section 11.4). Finally, section 11.5 discusses a hypothetical incremental canonization algorithm.

11.1 Principle

The principle of canonization is very simple; so, we shall describe it only briefly. The idea is to replace the expression $\alpha \sqcap \beta$ with a fresh variable γ, together with the constraints $\gamma \leq \alpha$ and $\gamma \leq \beta$. Thus, all upper bounds of α and β shall become, by transitivity, upper bounds of γ. So, γ shall have the same behavior as $\alpha \sqcap \beta$, meaning that the constraints $\tau \leq \gamma$ and $\tau \leq \alpha \sqcap \beta$ shall have the same consequences with respect to closure. (Constraints of the form $\alpha \sqcap \beta \leq \tau$ don't concern us, since $\sqcap$ cannot appear in such a position.) Symmetrically,

$\alpha \sqcup \beta$ shall be replaced with a fresh variable γ together with the constraints $\alpha \leq \gamma$ and $\alpha \leq \beta$.

The canonization algorithm must preserve a closure property, in order to be composed with the following simplification phase, namely garbage collection. Thus, we must add not only the constraints mentioned above, but also their consequences through transitivity and structural decomposition.

If we only introduced introduced the above constraints, then computed their consequences using the usual closure rules, we might risk introducing $\sqcup$ or $\sqcap$ again. For instance, assume F is a unary, covariant type constructor. Assume that α's (resp. β's) upper bound is $F\,\alpha$ (resp. $F\,\beta$). Then, when we eliminate $\alpha \sqcap \beta$, we introduce the constraints $\gamma \leq \alpha$ and $\gamma \leq \beta$. By transitivity, γ's constructed upper bound is $F\,(\alpha \sqcap \beta)$. The $\sqcap$ constructor has not been entirely eliminated. Thus, we cannot entrust this closure computation to the usual algorithm; we have to specify, in the definition of canonization, that γ's constructed upper bound is $F\,\gamma$.

This built-in closure computation is why the initial specification of the algorithm, given by definition 11.4, seems complex. However, we shall later see that parts of the constraints it introduces can be immediately eliminated by garbage collection, which leads us to a simpler specification (definition 11.5).

11.2 Simple closure

From a practical point of view, the canonization phase is to be inserted between the closure and the garbage collection phases. Thus, the canonization algorithm may assume that the type scheme under study is closed, while the type scheme it builds must be weakly closed.

Because it is more pleasant to work with an invariant, we choose to show that the algorithm preserves *simple closure*, a notion intermediate between closure and weak closure. (Neither closure, nor weak closure are preserved, because the former is too strong a goal, and the latter too weak a hypothesis.) The object of this preliminary section is to define this notion and to establish some of its properties.

Let us explain, tersely, what it is about. All notions of closure are based on the same idea: they require that the consequences (by transitivity and structural decomposition) of the constraints present in the graph be entailed, in a certain way, by the graph. However, it is possible to choose a more or less powerful notion of entailment. Plain closure is based on type containment (see definition 2.11), whereas weak closure uses our axiomatization of entailment, which yields much more flexibility. Here, we adopt an intermediate position, by using a rather naive notion of entailment, based on a straightforward extension of the relation $\leq_C$ to small terms. The notion of closure thus obtained remains very simple, while being sufficiently flexible to suit our purpose.

Definition 11.1 *Let C be a constraint graph. The relation $\leq_C$ is extended to leaf terms by setting*

$$\begin{array}{rcl} \alpha \leq_C \sqcup V & \iff & \exists \beta \in V \quad \alpha \leq_C \beta \\ \sqcap V \leq_C \alpha & \iff & \exists \beta \in V \quad \beta \leq_C \alpha \\ \sqcup V \leq_C \tau & \iff & \forall \beta \in V \quad \beta \leq_C \tau \\ \tau \leq_C \sqcap V & \iff & \forall \beta \in V \quad \tau \leq_C \beta \end{array}$$

Then, $\leq_C$ is extended to small terms straightforwardly, by structural decomposition.

Note that $\tau \leq_C \tau'$ is thus defined when the following conditions are met:

- τ and τ' are either two leaf terms or two small terms;

- $(\tau, \tau') \in (\mathcal{T}^+ \times \mathcal{T}^-) \cup (\mathcal{T}^+ \times \mathcal{T}^+) \cup (\mathcal{T}^- \times \mathcal{T}^-)$.

In particular, it remains undefined when $(\tau, \tau') \in \mathcal{T}^- \times \mathcal{T}^+$.

Proposition 11.1 *If $\leq_C$ is transitive on type variables, then its extension to leaf terms and small terms is transitive.*

Proof. For leaf terms, there are four cases to handle, depending on the form of the three types involved; all cases are immediate. The property then carries over to small terms by structural decomposition. □

Proposition 11.2 *The following properties, which, according to definition 11.1, hold when S and T are sets of type variables, carry over to leaf and small terms.*

$$\begin{array}{lcl} \sqcup S \leq_C \sqcap T & \iff & \forall \tau \in S \quad \forall \tau' \in T \quad \tau \leq_C \tau' \\ \sqcup S \leq_C \sqcup T & \iff & \forall \tau \in S \quad \exists \tau' \in T \quad \tau \leq_C \tau' \\ \sqcap S \leq_C \sqcap T & \iff & \forall \tau' \in T \quad \exists \tau \in S \quad \tau \leq_C \tau' \end{array}$$

Proof. Note that for the assertions to make sense, S and T must be sets of pos-types (resp. neg-types) when used as argument of $\sqcup$ (resp. $\sqcap$). The verification is left to the reader. □

Definition 11.2 *A constraint graph C, of domain V, is* simply closed *iff*

- *$\leq_C$ is transitive;*
- *for all $\alpha, \beta \in V$ such that $\alpha \leq_C \beta$, $C^\downarrow(\alpha) \leq_C C^\downarrow(\beta)$ and $C^\uparrow(\alpha) \leq_C C^\uparrow(\beta)$;*
- *for all $\alpha \in V$, $C^\downarrow(\alpha) \leq_C C^\uparrow(\alpha)$.*

A type scheme $\sigma = A \Rightarrow \tau \mid C$ is simply closed iff C is.

Note that the second and third conditions above can be checked prior to closing $\leq_C$ by transitivity. That is, formally speaking:

Proposition 11.3 *Let C be a constraint graph, of domain V, such that*

- *for all $\alpha, \beta \in V$ such that $\alpha \leq_C \beta$, $C^\downarrow(\alpha) \leq_C C^\downarrow(\beta)$ and $C^\uparrow(\alpha) \leq_C C^\uparrow(\beta)$;*
- *for all $\alpha \in V$, $C^\downarrow(\alpha) \leq_C C^\uparrow(\alpha)$.*

Then, the constraint graph D defined by $\leq_D = \leq_C^*$, $D^\uparrow = C^\uparrow$ and $D^\downarrow = C^\downarrow$ is simply closed.

Proof. $\leq_D$ is transitive by construction, and the third condition holds by hypothesis, so it suffices to verify the second condition. Assume $\alpha \leq_D \beta$. Since $\leq_D$ is the transitive closure of $\leq_C$, there exists a chain $\alpha = \gamma_1 \leq_C \ldots \leq_C \gamma_n = \beta$. According to our hypothesis, this implies $C^\downarrow(\gamma_1) \leq_C \ldots \leq_C C^\downarrow(\gamma_n)$. Since $\leq_D$ contains $\leq_C$, this implies $C^\downarrow(\gamma_1) \leq_D \ldots \leq_D C^\downarrow(\gamma_n)$. Finally, since $\leq_D$ is transitive, proposition 11.1 implies $C^\downarrow(\gamma_1) \leq_D C^\downarrow(\gamma_n)$, which can also be written $D^\downarrow(\alpha) \leq_D D^\downarrow(\beta)$. Similarly, $D^\uparrow(\alpha) \leq_D D^\uparrow(\beta)$. □

The following proposition formalizes the fact that simple closure is intermediate between plain closure and weak closure. It also indicates that garbage collection can be performed on the type scheme produced by the canonization phase.

Proposition 11.4 *Any closed constraint graph is simply closed. Any simply closed constraint graph is weakly closed. Any simple and simply closed type scheme is collectable.*

Proof. Let C be a closed graph. Then $\leq_C$ is transitive. Additionally, it is easy to verify that if a leaf pos-term τ' contains a leaf pos-term τ, then $\tau \leq_C \tau'$. A symmetric result is shown for neg-terms; both results then carry over to small terms. The second condition required for simple closure follows from this property. As for the third condition, it follows from a similar property, this time relating a pos-term to a neg-term.

Let C be a simply closed graph. It is easy to verify that $\tau \leq_C \tau'$ implies $C \vdash \tau \leq \tau'$ (once again, it is verified for leaf terms first, then extended to small terms). It follows that C is also weakly closed.

Finally, let $\sigma = A \Rightarrow \tau \mid C$ be a simple, simply closed type scheme. Then C is weakly closed and $\leq_C$ is transitive; thus, according to definition 10.4, σ is collectable. □

11.3 Raw canonization

We shall now describe a first version of the canonization algorithm. It can be used to eliminate the constructors $\sqcup$ and $\sqcap$, but also, optionally, to eliminate bipolar variables. In order to obtain a unified presentation, we parameterize it by a *filter*.

Definition 11.3 *Let V be a set of type variables. A* filter *of domain V is a subset of 2^V (i.e. a set of parts of V) which contains all parts of cardinality greater than 1, and does not contain the empty set.*

A filter $\mathcal{F}$ tells the algorithm which transformations should be performed. Any leaf term can be written $\sqcup S$ or $\sqcap S$, where S is a set (possibly containing only one element) of variables. If $S \in \mathcal{F}$, such a term shall be eliminated, and replaced with a fresh variable; it shall be left alone otherwise.

Any set S of cardinality (strictly) greater than 1 must be a member of $\mathcal{F}$, which means that all occurrences of $\sqcup$ and $\sqcap$ shall be eliminated. A variable α can be written, according to its position in the term, $\sqcup\{\alpha\}$ or $\sqcap\{\alpha\}$; if $\{\alpha\} \in \mathcal{F}$, each of these two terms shall be replaced with a different fresh variable. We shall show that the transformation introduces no bipolar variables; thus, to eliminate all bipolar variables, it suffices to include all of the corresponding singletons in the filter. If, on the contrary, only canonization is desired, then the filter does not have to contain any singleton.

Let us now define the canonization process. Recall that it shall be later combined with a partial garbage collection phase (section 11.4), which shall yield a much simpler definition.

$$
\begin{array}{rlrl}
r^+(\alpha) = \alpha & \text{when } \{\alpha\} \notin \mathcal{F} & r^-(\alpha) = \alpha & \text{when } \{\alpha\} \notin \mathcal{F} \\
r^+(\sqcup S) = \lambda_S & \text{when } S \in \mathcal{F} & r^-(\sqcap S) = \gamma_S & \text{when } S \in \mathcal{F} \\
r^+(\bot) = \bot & & r^-(\bot) = \bot & \\
r^+(\top) = \top & & r^-(\top) = \top & \\
r^+(\tau_0 \to \tau_1) = r^-(\tau_0) \to r^+(\tau_1) & & r^-(\tau_0 \to \tau_1) = r^+(\tau_0) \to r^-(\tau_1) &
\end{array}
$$

Figure 11.1: Definition of the rewriting functions

Definition 11.4 *Let $\sigma = A \Rightarrow \tau \mid C$ be a simply closed type scheme, of domain V. Let $\mathcal{F}$ be a filter of domain V.*

For each $S \in \mathcal{F}$, pick two fresh variables λ_S and γ_S. (By fresh variables, we mean that these variables are pairwise distinct, and distinct from σ's variables.) The rewriting

$$D^{\downarrow}(\alpha) = r^{+}(C^{\downarrow}(\alpha)) \qquad D^{\uparrow}(\alpha) = r^{-}(C^{\uparrow}(\alpha))$$
$$D^{\downarrow}(\gamma_S) = r^{+}(\bigsqcup_{\alpha \leq_C \sqcap S} C^{\downarrow}(\alpha)) \qquad D^{\uparrow}(\gamma_S) = r^{-}(\bigsqcap_{\alpha \in S} C^{\uparrow}(\alpha))$$
$$D^{\downarrow}(\lambda_S) = r^{+}(\bigsqcup_{\alpha \in S} C^{\downarrow}(\alpha)) \qquad D^{\uparrow}(\lambda_S) = r^{-}(\bigsqcap_{\sqcup S \leq_C \alpha} C^{\uparrow}(\alpha))$$

Figure 11.2: Definition of $D^{\downarrow}$ and $D^{\uparrow}$

$$\begin{array}{rcll}
\alpha & \leq_D & \beta & \text{when } \alpha \leq_C \beta \\
\gamma_S & \leq_D & \alpha & \text{when } \alpha \in S \\
\alpha & \leq_D & \lambda_S & \text{when } \alpha \in S \\
r^{-}(\sqcap S) & \leq_D & \gamma_T & \text{when } |\, S \,| \geq 1 \text{ and } \sqcap S \leq_C \sqcap T \\
\lambda_S & \leq_D & r^{+}(\sqcup T) & \text{when } |\, T \,| \geq 1 \text{ and } \sqcup S \leq_C \sqcup T \\
r^{+}(\sqcup S) & \leq_D & r^{-}(\sqcap T) & \text{when } |\, S \,| \geq 1, |\, T \,| \geq 1 \text{ and } \sqcup S \leq_C \sqcap T
\end{array}$$

Figure 11.3: Definition of $\leq_D$, modulo transitive closure

functions r^{+} *and* r^{-} *are defined according to figure 11.1 on the preceding page. The former (resp. latter) is defined on leaf or small pos-terms (resp. neg-terms).*

The result graph D *is defined by its components* $D^{\downarrow}$ *and* $D^{\uparrow}$*, given by figure 11.2, and by the relation* $\leq_D$*, which is the* transitive closure *of the relation given in figure 11.3.*

The type scheme produced by the raw canonization *process, denoted by* $\mathrm{Can}^0_{\mathcal{F}}(\sigma)$*, is* $r^{-}(A) \Rightarrow r^{+}(\tau) \mid D$*.*

The remainder of this section is dedicated to proving two results. First, this transformation is indeed canonization, i.e. it produces a type scheme which is equivalent to the original one, but free of any occurrences of $\sqcup$ or $\sqcap$. Second, it preserves simple closure, as announced.

Lemma 11.5 *If τ is a leaf pos-term or a small pos-term, then $\tau \leq_D r^{+}(\tau)$. If τ is a leaf neg-term or a small neg-term, then $r^{-}(\tau) \leq_D \tau$.*

Proof. Consider a leaf pos-term $\sqcup S$. If $S \notin \mathcal{F}$, then $S = \{\alpha\}$, so $\tau = \alpha = r^{+}(\tau)$. Otherwise, $r^{+}(\tau) = \lambda_S$. We then have to verify $\sqcup S \leq_D \lambda_S$. Pick $\alpha \in S$. Then, by definition of $\leq_D$, $\alpha \leq_D \lambda_S$, so the result holds for leaf pos-terms. The case of leaf neg-terms is symmetric, and the extension to small terms poses no difficulty. □

Here comes the first of the expected results.

Theorem 11.1 *Let σ be a simply closed type scheme. Let $\mathcal{F}$ be a filter of domain* $\mathrm{dom}(\sigma)$*. Set $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$. Then σ' is simple, and $\sigma =^{\forall} \sigma'$.*

Proof. Because the rewriting functions r^{+} and r^{-} never produce $\sqcup$ or $\sqcap$ constructs, it is clear than σ' is simple.

Because $\leq_D$ contains $\leq_C$, and because of lemma 11.5, it is clear than $D \Vdash C$, and that $D \Vdash (A \Rightarrow \tau) \leq (r^{-}(A) \Rightarrow r^{+}(\tau))$. It follows that $\sigma \leq^{\forall} \sigma'$.

Reciprocally, let ρ be a solution of C. Define ρ' by

$$\rho'(\alpha) = \rho(\alpha) \qquad \rho'(\gamma_S) = \bigsqcap_{\alpha \in S} \rho(\alpha) \qquad \rho'(\lambda_S) = \bigsqcup_{\alpha \in S} \rho(\alpha)$$

Then, one easily verifies that for any leaf pos-term τ, $\rho'(r^+(\tau)) = \rho(\tau)$. Similarly, for any leaf neg-term τ, $\rho'(r^-(\tau)) = \rho(\tau)$. This result is easily extended to small terms. Using it, it is straightforward to verify that ρ' is a solution of D, and that

$$\rho'(r^-(A) \Rightarrow r^+(\tau)) \leq \rho(A \Rightarrow \tau)$$

So, $\sigma' \leq^\forall \sigma$. □

Lemma 11.6 *If τ and τ' are two leaf pos-terms, or two small pos-terms, such that $\tau \leq_C \tau'$, then $r^+(\tau) \leq_D r^+(\tau')$. If τ and τ' are two leaf neg-terms, or two small neg-terms, such that $\tau \leq_C \tau'$, then $r^-(\tau) \leq_D r^-(\tau')$.*

Proof. Let $\sqcup S$ and $\sqcup T$ be two leaf pos-terms such that $\sqcup S \leq_C \sqcup T$. We distinguish two cases, depending on whether $S \in \mathcal{F}$.

If $S \notin \mathcal{F}$, then S must be of the form $\{\alpha\}$, and $r^+(\sqcup S) = \alpha$. The hypothesis $\sqcup S \leq_C \sqcup T$ translates to $\alpha \leq_C \sqcup T$. Besides, according to lemma 11.5, $\sqcup T \leq_D r^+(\sqcup T)$. Since $\leq_D$ contains $\leq_C$, and $\leq_D$ is transitive, it follows that $\alpha \leq_D r^+(\sqcup T)$, which was our goal.

If $S \in \mathcal{F}$, then $r^+(\sqcup S) = \lambda_S$, so the goal becomes $\lambda_S \leq_D r^+(\sqcup T)$. Recall that $\sqcup S \leq_C \sqcup T$; the result follows by definition of $\leq_D$.

We have handled the case of leaf pos-terms; the case of leaf neg-terms is symmetric. The extension to small terms is straightforward. □

Lemma 11.7 *Let τ be a leaf pos-term and τ' be a leaf neg-term such that $\tau \leq_C \tau'$. Then $r^+(\tau) \leq_D r^-(\tau')$. This result carries over to small terms.*

Proof. Write $\tau = \sqcup S$ and $\tau' = \sqcap T$. Then $r^+(\sqcup S) \leq_D r^-(\sqcap T)$ follows by definition of $\leq_D$. The extension to small terms is straightforward. □

Now, here is the second promised result.

Theorem 11.2 *Let σ be a simply closed type scheme. Let $\mathcal{F}$ be a filter of domain* $\mathrm{dom}(\sigma)$. *Set $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$. Then σ' is simply closed.*

Proof. $\leq_D$ is closed by construction, so there remains to check the two other conditions. Let us start with the last one, which states that each variable's lower and upper bounds must be in the relation $\leq_D$.

- Let $\alpha \in V$. Because C is simply closed, we have $C^\downarrow(\alpha) \leq_C C^\uparrow(\alpha)$. By lemma 11.7, we have $r^+(C^\downarrow(\alpha)) \leq_D r^-(C^\uparrow(\alpha))$.
- Let $S \in \mathcal{F}$. We wish to show

  $$r^+(\bigsqcup_{\alpha \leq_C \sqcap S} C^\downarrow(\alpha)) \leq_D r^-(\bigsqcap_{\alpha \in S} C^\uparrow(\alpha))$$

  Thanks to lemma 11.7, it suffices to show

  $$\bigsqcup_{\alpha \leq_C \sqcap S} C^\downarrow(\alpha) \leq_C \bigsqcap_{\alpha \in S} C^\uparrow(\alpha)$$

  which, according to proposition 11.2, is equivalent to

  $$\forall \alpha \leq_C \sqcap S \quad \forall \beta \in S \quad C^\downarrow(\alpha) \leq_C C^\uparrow(\beta)$$

  Pick $\alpha \leq_C \sqcap S$ and $\beta \in S$. This implies $\alpha \leq_C \beta$. Since C is simply closed, this implies $C^\downarrow(\alpha) \leq_C C^\downarrow(\beta)$. Besides, we have $C^\downarrow(\beta) \leq_C C^\uparrow(\beta)$. By transitivity of $\leq_C$, it follows that $C^\downarrow(\alpha) \leq_C C^\uparrow(\beta)$.

- The case of λ_S is symmetric.

Let us now deal with the second condition, which states that whenever two variables are in the relation $\leq_D$, their lower (resp. upper) bounds should be too. Thanks to proposition 11.3, it suffices to verify that this condition holds for the constraints explicitly given in figure 11.3.

- Consider $\alpha \leq_D \beta$ when $\alpha \leq_C \beta$. Since C is simply closed, we have $C^{\downarrow}(\alpha) \leq_C C^{\downarrow}(\beta)$ and $C^{\uparrow}(\alpha) \leq_C C^{\uparrow}(\beta)$. By lemma 11.6, the result follows.

- Consider $\gamma_S \leq_D \alpha$ when $\alpha \in S$. We have to prove

$$r^+(\bigsqcup_{\beta \leq_C \sqcap S} C^{\downarrow}(\beta)) \leq_D r^+(C^{\downarrow}(\alpha)) \quad \text{and} \quad r^-(\bigsqcap_{\beta \in S} C^{\uparrow}(\beta)) \leq_D r^-(C^{\uparrow}(\alpha))$$

According to lemma 11.6, it suffices to show

$$\bigsqcup_{\beta \leq_C \sqcap S} C^{\downarrow}(\beta) \leq_C C^{\downarrow}(\alpha) \quad \text{and} \quad \bigsqcap_{\beta \in S} C^{\uparrow}(\beta) \leq_C C^{\uparrow}(\alpha)$$

The second of these assertions is a direct consequence of proposition 11.2. Still according to proposition 11.2, the first one is equivalent to

$$\forall \beta \leq_C \sqcap S \quad C^{\downarrow}(\beta) \leq_C C^{\downarrow}(\alpha)$$

Pick $\beta \leq_C \sqcap S$. Then $\beta \leq_C \alpha$, because $\alpha \in S$. It follows that $C^{\downarrow}(\beta) \leq_C C^{\downarrow}(\alpha)$.

- Consider $\alpha \leq_D \lambda_S$ when $\alpha \in S$. This case is symmetric to the previous one.

- Consider $r^-(\sqcap S) \leq_D \gamma_T$ when $|\,S\,| \geq 1$ and $\sqcap S \leq_C \sqcap T$. First, assume $S \in \mathcal{F}$. Then, we have to show (modulo lemma 11.6, as usual)

$$\bigsqcup_{\alpha \leq_C \sqcap S} C^{\downarrow}(\alpha) \leq_C \bigsqcup_{\beta \leq_C \sqcap T} C^{\downarrow}(\beta) \quad \text{and} \quad \bigsqcap_{\alpha \in S} C^{\uparrow}(\alpha) \leq_C \bigsqcap_{\beta \in T} C^{\uparrow}(\beta)$$

Consider the first of these assertions. By transitivity of $\leq_C$, $\alpha \leq_C \sqcap S$ implies $\alpha \leq_C \sqcap T$, so the left-hand $\sqcup$ expression has fewer operands than the right-hand one. The assertion follows by proposition 11.2. Consider the second assertion. We have $\sqcap S \leq_C \sqcap T$, that is,

$$\forall \beta \in T \quad \exists \alpha \in S \quad \alpha \leq_C \beta$$

Since C is simply closed, this implies

$$\forall \beta \in T \quad \exists \alpha \in S \quad C^{\downarrow}(\alpha) \leq_C C^{\downarrow}(\beta)$$

which in turn, according to proposition 11.2, yields exactly the second assertion.

There remains to deal with the case $S \notin \mathcal{F}$. Then, S is of the form $\{\alpha\}$ and we have to show

$$C^{\downarrow}(\alpha) \leq_C \bigsqcup_{\beta \leq_C \sqcap T} C^{\downarrow}(\beta) \quad \text{and} \quad C^{\uparrow}(\alpha) \leq_C \bigsqcap_{\beta \in T} C^{\uparrow}(\beta)$$

We notice that the second of these assertions is identical to the second one above. As for the first one, it is a consequence of the first assertion above, because

$$C^{\downarrow}(\alpha) \leq_C \bigsqcup_{\alpha \leq_C \sqcap S} C^{\downarrow}(\alpha)$$

- Consider $\lambda_S \leq_D r^+(\sqcup T)$ when $|T| \geq 1$ and $\sqcup S \leq_C \sqcup T$. This case is symmetric to the previous one.
- Consider $r^+(\sqcup S) \leq_D r^-(\sqcap T)$ when $|S| \geq 1$, $|T| \geq 1$ and $\sqcup S \leq_C \sqcap T$. First, assume $S \in \mathcal{F}$ and $T \in \mathcal{F}$. We have to show (modulo lemma 11.6)

$$\bigsqcup_{\alpha \in S} C^{\downarrow}(\alpha) \leq_C \bigsqcup_{\beta \leq_C \sqcap T} C^{\downarrow}(\beta)$$

(We omit the other goal, which is symmetric.) This is equivalent to

$$\forall \alpha \in S \quad C^{\downarrow}(\alpha) \leq_C \bigsqcup_{\beta \leq_C \sqcap T} C^{\downarrow}(\beta)$$

Pick $\alpha \in S$. Because $\sqcup S \leq_C \sqcap T$, we have $\alpha \leq_C \sqcap T$. It follows that α is one of the β's above, and the inequality holds.

When $S \notin \mathcal{F}$, S is of the form $\{\alpha\}$ and the left-hand term is $C^{\downarrow}(\alpha)$; so the assertion is unchanged.

When $T \notin \mathcal{F}$, T is of the form $\{\alpha\}$ and the right-hand term is $C^{\downarrow}(\alpha)$. It suffices to verify

$$\bigsqcup_{\beta \leq_C \sqcap T} C^{\downarrow}(\beta) \leq_C C^{\downarrow}(\alpha)$$

Pick $\beta \leq_C \sqcap T$. Then $\beta \leq_C \alpha$, so $C^{\downarrow}(\beta) \leq_C C^{\downarrow}(\alpha)$. All operands of the left-hand expression are less (for $\leq_C$) than the right-hand expression, hence the result. □

11.4 Canonization

Out of a simply closed type scheme σ, the raw canonization process described in the previous section makes a collectable type scheme σ'. Thus, we can compute polarities, then perform garbage collection on σ'. We shall now simulate this computation.

In general, we cannot exactly predict each variable's polarity in σ', because it partially depends on σ, which is unknown. However, we can compute an approximation of it. It shall turn out to be precise enough to allow safely eliminating a large class of constraints. We can then combine the raw canonization process introduced above with this approximate garbage collection phase. In practice, this amounts to avoiding creating these superfluous constraints, rather than building them and throwing them away immediately afterwards. Thus, we obtain a much leaner specification of canonization.

This approximate polarity computation on σ' reveals that the newly introduced λ_S's are at most positive, while the γ_S's are at most negative. The polarity of an existing variable α must decrease (in a non-strict sense). In particular, if $\{\alpha\} \in \mathcal{F}$, then α becomes neutral. These results are formalized below.

Lemma 11.8 *Let σ be a simply closed type scheme, $\mathcal{F}$ a filter of domain* $\mathrm{dom}(\sigma)$ *and* $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$. *Then*

$$\mathrm{dom}^+(\sigma') \subseteq \{\lambda_S\,;\, S \in \mathcal{F} \wedge S \subseteq \mathrm{dom}^+(\sigma)\} \cup \{\alpha\,;\, \{\alpha\} \notin \mathcal{F} \wedge \alpha \in \mathrm{dom}^+(\sigma)\}$$
$$\mathrm{dom}^-(\sigma') \subseteq \{\gamma_S\,;\, S \in \mathcal{F} \wedge S \subseteq \mathrm{dom}^-(\sigma)\} \cup \{\alpha\,;\, \{\alpha\} \notin \mathcal{F} \wedge \alpha \in \mathrm{dom}^-(\sigma)\}$$

Proof. Let P_0 (resp. N_0) be the right-hand side of the first (resp. second) assertion above.

Let us begin with a remark concerning the rewriting functions. If $S \subseteq \mathrm{dom}^+(\sigma)$, then $r^+(\sqcup S) \in P_0$. That is, r^+ maps any positive leaf term to an element of P_0. The function r^- enjoys a symmetric property.

According to definition 10.3, $\mathrm{dom}^+(\sigma')$ and $\mathrm{dom}^-(\sigma')$ are the smallest sets P and N such that

- $r^+(\tau) \in P$
- $\forall (x : \tau_x) \in A \quad r^-(\tau_x) \in N$
- $\forall \delta \in P \quad (\mathrm{fv}^+(D^\downarrow(\delta)) \subseteq P) \wedge (\mathrm{fv}^-(D^\downarrow(\delta)) \subseteq N)$
- $\forall \delta \in N \quad (\mathrm{fv}^+(D^\uparrow(\delta)) \subseteq N) \wedge (\mathrm{fv}^-(D^\uparrow(\delta)) \subseteq P)$

We claim that P_0 and N_0 also satisfy these four conditions. The result shall then follow immediately.

τ is a leaf term, so it can be written $\sqcup S$, where $S \subseteq \mathrm{dom}^+(\sigma)$ by definition of the polarities in σ. According to the above remark, this implies $r^+(\tau) \in P_0$. Similarly, for all $(x : \tau_x) \in A$, we have $r^-(\tau_x) \in N_0$.

Now, consider the third condition. Let $\delta \in P_0$. Regardless of whether δ is of the form α, γ_S or λ_S, we have

$$D^\downarrow(\delta) = r^+(\bigsqcup_{\alpha \in V} C^\downarrow(\alpha))$$

for some $V \subseteq \mathrm{dom}(\sigma)$ (see figure 11.2). The value of V depends on the form of δ:

- If δ is of the form $\alpha \in \mathrm{dom}(\sigma)$, then $V = \{\alpha\}$. Since $\delta \in P_0$, we have $\alpha \in \mathrm{dom}^+(\sigma)$.
- δ cannot be of the form γ_S, because $\delta \in P_0$.
- If δ is of the form λ_S, then $V = S$. Since $\delta \in P_0$, we have $S \subseteq \mathrm{dom}^+(\sigma)$.

So, regardless of the form of δ, we have $V \subseteq \mathrm{dom}^+(\sigma)$. That is, each $\alpha \in V$ is positive in σ. By definition of the polarities in σ, this implies that each $C^\downarrow(\alpha)$ is also a positive term. Hence, so is the least upper bound of these terms, $\bigsqcup_{\alpha \in V} C^\downarrow(\alpha)$, which we shall call τ_δ. More precisely, $\mathrm{fv}^+(\tau_\delta) \subseteq \mathrm{dom}^+(\sigma)$ and $\mathrm{fv}^-(\tau_\delta) \subseteq \mathrm{dom}^-(\sigma)$. According to our opening remark, this implies $\mathrm{fv}^+(r^+(\tau_\delta)) \subseteq P_0$ and $\mathrm{fv}^-(r^+(\tau_\delta)) \subseteq N_0$. So, the third condition is satisfied.

The fourth condition is symmetric to the third one. □

From this lemma, several interesting properties can be deduced. The first one states that polarities decrease (in a non-strict sense) through canonization.

Proposition 11.9 *Let σ be a simply closed type scheme, $\mathcal{F}$ a filter of domain* $\mathrm{dom}(\sigma)$ *and* $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$. *Then*

$$\mathrm{dom}^+(\sigma') \cap \mathrm{dom}(\sigma) \subseteq \mathrm{dom}^+(\sigma)$$
$$\mathrm{dom}^-(\sigma') \cap \mathrm{dom}(\sigma) \subseteq \mathrm{dom}^-(\sigma)$$

Proof. Let $\alpha \in \mathrm{dom}^+(\sigma') \cap \mathrm{dom}(\sigma)$. According to lemma 11.8, $\alpha \in \mathrm{dom}^+(\sigma')$ implies $\alpha \in \{\lambda_S\,;\, S \in \mathcal{F} \wedge S \subseteq \mathrm{dom}^+(\sigma)\} \cup \{\alpha\,;\, \{\alpha\} \notin \mathcal{F} \wedge \alpha \in \mathrm{dom}^+(\sigma)\}$. Taking into account the fact that $\alpha \in \mathrm{dom}(\sigma)$, this assertion can be refined to $\alpha \in \{\alpha\,;\, \{\alpha\} \notin \mathcal{F} \wedge \alpha \in \mathrm{dom}^+(\sigma)\}$, which implies $\alpha \in \mathrm{dom}^+(\sigma)$, hence the first expected result. The second one is symmetric. □

The following property shows that the algorithm can be used to eliminate bipolar variables, and thus to enforce the *mono-polarity invariant* introduced in chapter 12. At the same time, it indicates that this invariant is preserved by the transformation.

Proposition 11.10 *Let σ be a simply closed type scheme, and $\mathcal{F}$ a filter of domain* $\mathrm{dom}(\sigma)$. *All bipolar variables of σ are assumed to be "filtered", i.e.*

$$\forall \alpha \in \mathrm{dom}^+(\sigma) \cap \mathrm{dom}^-(\sigma) \quad \{\alpha\} \in \mathcal{F}$$

Then, $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$ has no bipolar variables, that is,

$$\mathrm{dom}^+(\sigma') \cap \mathrm{dom}^-(\sigma') = \varnothing$$

Proof. According to lemma 11.8,

$$\mathrm{dom}^+(\sigma') \cap \mathrm{dom}^-(\sigma') \subseteq \{\alpha\,;\, \{\alpha\} \notin \mathcal{F} \wedge \alpha \in \mathrm{dom}^+(\sigma) \cap \mathrm{dom}^-(\sigma)\}$$

The right-hand side of this assertion is precisely the empty set, because of our hypothesis. □

Finally, our approximation of the polarities allows us to perform a step of garbage collection. Note that it is only partial, because polarities are not fully known. In other words, the type scheme σ'' which we shall compute is not $\mathrm{GC}(\sigma')$; it contains more constraints than $\mathrm{GC}(\sigma')$, but fewer than σ'. In practice, this does not pose any problems, since the canonization phase is followed by an actual garbage collection phase. The machine shall thus compute $\mathrm{GC}(\sigma'')$, which is $\mathrm{GC}(\sigma')$, as if we had directly fed σ' to the garbage collection algorithm. Thus, from a theoretical point of view, we have added one step, but in practice, we gain performance. Indeed, the machine shall not compute σ'; it shall directly produce σ'', thus avoiding the generation of useless constraints.

We now define canonisation as the combination of raw canonization and of this partial garbage collection step.

$$\begin{array}{ll}
E^{\downarrow}(\alpha) = r^+(C^{\downarrow}(\alpha)) & E^{\uparrow}(\alpha) = r^-(C^{\uparrow}(\alpha)) \\
E^{\downarrow}(\gamma_S) = \bot & E^{\uparrow}(\gamma_S) = r^-(\bigsqcap_{\alpha \in S} C^{\uparrow}(\alpha)) \\
E^{\downarrow}(\lambda_S) = r^+(\bigsqcup_{\alpha \in S} C^{\downarrow}(\alpha)) & E^{\uparrow}(\lambda_S) = \top
\end{array}$$

Figure 11.4: Definition of $E^{\downarrow}$ et $E^{\uparrow}$

$$\begin{array}{rcll}
\alpha & \leq_E & \beta & \text{when } \alpha \leq_C \beta \\
\gamma_S & \leq_E & \alpha & \text{when } \sqcap S \leq_C \alpha \\
\alpha & \leq_E & \lambda_S & \text{when } \alpha \leq_C \sqcup S \\
\gamma_S & \leq_E & \lambda_T & \text{when } \exists \alpha \in S \quad \exists \beta \in T \quad \alpha \leq_C \beta
\end{array}$$

Figure 11.5: Definition of $\leq_E$

Definition 11.5 *Let $\sigma = A \Rightarrow \tau \mid C$ be a simply closed type scheme, and $\mathcal{F}$ be a filter of domain* $\mathrm{dom}(\sigma)$. *Let E be the constraint graph defined by figures 11.4 and 11.5. The type scheme output by the* canonization *process, denoted by* $\mathrm{Can}_{\mathcal{F}}(\sigma)$, *is $r^-(A) \Rightarrow r^+(\tau) \mid E$.*

There remains to prove the correctness of this definition:

Proposition 11.11 *Let σ be a simply closed type scheme, and $\mathcal{F}$ be a filter of domain* $\mathrm{dom}(\sigma)$. *Set* $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$ *and* $\sigma'' = \mathrm{Can}_{\mathcal{F}}(\sigma)$. *Then*

$$\mathrm{GC}(\sigma'') = \mathrm{GC}(\sigma')$$

Proof. Before we can compute the effects of garbage collection on σ', we must compute $\leq_D$ explicitly. Recall that, until now, $\leq_D$ has been defined modulo transitive closure; we now need to know explicitly what kind of constraints exist in $\leq_D$. We claim that any constraint in $\leq_D$ must be of one of the following forms:

1.	$\alpha \leq \beta$	where $\alpha \leq_C \beta$
2.	$\gamma_S \leq \alpha$	where $\sqcap S \leq_C \alpha$
3.	$\alpha \leq \lambda_S$	where $\alpha \leq_C \sqcup S$
4.	$\alpha \leq \gamma_S$	where $\alpha \leq_C \sqcap S$
5.	$\gamma_S \leq \gamma_T$	where $\sqcap S \leq_C \sqcap T$
6.	$\lambda_S \leq \alpha$	where $\sqcup S \leq_C \alpha$
7.	$\lambda_S \leq \lambda_T$	where $\sqcup S \leq_C \sqcup T$
8.	$\lambda_S \leq \gamma_T$	where $\sqcup S \leq \sqcap T$
9.	$\gamma_S \leq \lambda_T$	where $\exists \alpha \in S \quad \exists \beta \in T \quad \alpha \leq_C \beta$

The proof of this claim is simple. First, one easily verifies that any of the constraints explicitly given in figure 11.3 is of one of these forms. Then, one verifies that if two arbitrary constraints from the above list are combined by transitivity, the resulting constraint is also of one of these forms. There are 81 cases, all of them trivial, and left to the reader.

We can now use lemma 11.8, which gives a conservative approximation of the polarities in σ'. Garbage collection keeps only constraints linking a negative variable to a positive one. It follows that constraints of the forms 4–8 necessarily disappear. The remaining ones are precisely those given in figure 11.5.

Finally, lemma 11.8 allows us to drop γ_S's lower bound and λ_S's upper bound, yielding the definition of figure 11.4. □

Finally, there remains a simple optimization which our theoretical presentation doesn't account for. Our definition of canonization creates fresh variables γ_S and λ_S for each $S \in \mathcal{F}$, which represents an exponential number of new variables. In practice, one easily verifies that if there is no occurrence of $\sqcup S$ (resp. $\sqcap S$) in σ, then λ_S (resp. γ_S) is neutral in σ'. Thus, in practice, one generates λ_S (resp. γ_S) only when encountering an occurrence of $\sqcup S$ (resp. $\sqcap S$) in σ.

To conclude this section, here a few simple examples. (For the sake of brevity, constraint graphs are sometimes represented by constraint sets.)

Example. First, here is a short example which illustrates the elimination of bipolar variables. The identity function $\lambda x.x$ can be given the type scheme

$$\delta^+ \mid \{\epsilon^\pm \rightarrow \epsilon^\pm \leq \delta^+\}$$

(The scheme $\epsilon^\pm \rightarrow \epsilon^\pm$ is also correct, but does not satisfy the small terms invariant.) This scheme does not satisfy the mono-polarity invariant, which we shall introduce in chapter 12, because it contains a bipolar variable ϵ. To eliminate all bipolar variables, we apply the canonization algorithm, with a filter $\mathcal{F}$ which contains the singleton $\{\epsilon\}$. The algorithm introduces two fresh variables γ and λ, linked by the constraint $\gamma \leq \lambda$. It replaces all negative (resp. positive) occurrences of ϵ with the former (resp. latter). Thus, we obtain

$$\delta^+ \mid \{\gamma^- \rightarrow \lambda^+ \leq \delta^+, \gamma^- \leq \lambda^+\}$$

Example. As explained above, when the mono-polarity invariant is enforced, the type scheme inferred for the identity function $\lambda x.x$ is

$$\delta \mid \{\alpha \rightarrow \beta \leq \delta,\, \alpha \leq \beta\}$$

Now, consider the type scheme inferred for the expression

$$\texttt{if true then } \lambda x.x \texttt{ else } \lambda x.x$$

We obtain two copies of the above type scheme, linked with a $\sqcup$ constructor:

$$\delta_1 \sqcup \delta_2 \mid \{\alpha_1 \rightarrow \beta_1 \leq \delta_1,\, \alpha_2 \rightarrow \beta_2 \leq \delta_2,\, \alpha_1 \leq \beta_1,\, \alpha_2 \leq \beta_2\}$$

Let us apply to the canonization algorithm to this scheme. A fresh variable δ appears, in order to replace $\delta_1 \sqcup \delta_2$. Furthermore, while computing δ's constructed lower bound, the expressions $\alpha_1 \sqcap \alpha_2$ and $\beta_1 \sqcup \beta_2$ appear; two fresh variables α and β are created to replace them. We obtain $\delta \mid C$, where C is the constraint graph defined by

- $\delta_1 \leq_C \delta$, $\delta_2 \leq_C \delta$;
- $\alpha \leq_C \alpha_1 \leq_C \beta_1 \leq_C \beta$, $\alpha \leq_C \alpha_2 \leq_C \beta_2 \leq_C \beta$ and $\alpha \leq_C \beta$;
- $C^{\downarrow}(\delta_1) = \alpha_1 \rightarrow \beta_1$, $C^{\downarrow}(\delta_2) = \alpha_2 \rightarrow \beta_2$ and $C^{\downarrow}(\delta) = \alpha \rightarrow \beta$.

So, as expected, we have eliminated the $\sqcup$ and $\sqcap$ constructors. The most interesting point is that, at the same time, we have opened new opportunities for garbage collection. Indeed, before canonization, no variables were neutral, and garbage collection would have had no effect. At present, on the contrary, computing polarities shows that δ and β are positive, α is negative, and all other variables have become neutral. Thus, garbage collection produces the type scheme

$$\delta \mid \{\alpha \rightarrow \beta \leq \delta,\, \alpha \leq \beta\}$$

which is identical to the identity's type scheme. Thus, canonization, with the help of garbage collection, has been able to identify the two sets of identical constraints produced by the two branches of the conditional. So, in general, canonization is a simplification, not only because it eliminates the $\sqcup$ and $\sqcap$ constructors, which are unwanted in some later phases, but also because in doing so, it creates new opportunities for the garbage collection process.

11.5 Incremental canonization

We have presented canonization as a stand-alone algorithm, which requires a dedicated phase during the type scheme simplification process. This choice can be questioned. Rather than allowing the closure phase to introduce $\sqcup$ and $\sqcap$ constructs (or, equivalently, multiple bounds; see section 2.1), and then having to perform a canonization pass to eliminate them, could we not perform canonization incrementally? That is, modify the closure algorithm so that these constructs are immediately replaced with fresh variables, together with appropriate constraints?

At first glance, this idea seems interesting, because if correctly carried out, it should allow saving part of the cost of canonization. Indeed, the canonization algorithm has to walk the whole constraint graph to check for occurrences of $\sqcup$ or $\sqcap$, while an incremental algorithm would only perform computations as needed.

Let us first recap the principle of canonization. Roughly speaking, it consists in replacing the expression $\alpha \sqcap \beta$ with a fresh variable γ, while adding the constraints $\gamma \leq \alpha$ and $\gamma \leq \beta$. So, the idea is to perform this replacement during the closure computation, whenever combining two bounds causes such an expression to appear. However, a few issues should be mentioned.

The first issue is the risk of non-termination. For instance, assume our graph contains the constraints $\alpha \leq F\,\alpha$ and $\beta \leq F\,\beta$, where F is a covariant type constructor. Assume the expression $\alpha \sqcap \beta$ appears while closing the graph. So, we introduce a variable γ, as shown above. The closure computation continues; in particular, it must compute γ's constructed upper bound. Since $\gamma \leq \alpha$ and $\gamma \leq \beta$, this bound is $F(\alpha \sqcap \beta)$. Thus, by adding these new constraints, we have caused a new $\sqcap$ construct to appear. If we associate a new variable to it, the same problem shall arise again and the process shall not terminate. So, we must be able to memorize the association between $\alpha \sqcap \beta$ and γ. This requires keeping track of which variables have been created and which expressions they stand for—in fact, exactly what the stand-alone algorithm has to do, since it maintains the mapping $S \mapsto (\gamma_S, \lambda_S)$. So, switching to an incremental version might allow saving the (linear) cost of the graph traversal, but the computations remain the same.

The second issue is the fact that the closure algorithm has to be reformulated. The current algorithm is based on the notion of plain closure, which, if $\sqcup$ and $\sqcap$ constructs are forbidden, becomes too restrictive. A more flexible notion, such as simple closure, must be adopted. This should not pose any problem; the algorithm remains fundamentally identical, with a slightly modified termination criterion, based on the relation $\leq_C$ rather than on type containment.

Such a closure algorithm, equipped with a built-in incremental canonization mechanism, seems conceivable. However, the details of its design and of its proof, especially as far as termination is concerned, seemed rather delicate to us. This is why we kept the non-incremental algorithm, which is clearly understood. The performance loss should be minimal, since the canonization phase is performed only rarely (namely, at `let` nodes), and the time lost during each phase roughly corresponds to a single traversal of the constraint graph. Still, this problem remains a research subject, with a possible efficiency gain at stake.

Chapter 12

The mono-polarity invariant

When garbage collection was defined (see chapter 10), a question was left unanswered: does garbage collection produce a closed type scheme, according to one of the notions of closure we developed? This property is desirable, since all the algorithms we designed require (weakly, simply or plainly) closed type schemes. The various simplification phases shall thus combine more easily, and more efficiently, if it is satisfied. However, in the general case, the answer is negative.

The object of this chapter is to show that, provided we adopt a certain non-restrictive invariant on type schemes, this question can be answered positively. This invariant shall be named *mono-polarity invariant*.

We first study the shape of type schemes produced by garbage collection, so as to understand the cause of the problem (section 12.1); this naturally leads us to the invariant's definition and to its effect on garbage collection (section 12.2). We then take interest in the means of enforcing this invariant (section 12.3), and conclude with miscellaneous remarks about its secondary advantages (section 12.4).

12.1 Motivation

To analyze the problem, let us look at the result produced by garbage collection in a particular case. Let σ be the type scheme

$$\alpha \to \alpha \mid \{F\,\beta \leq \alpha \leq F\,\gamma,\ \beta \leq \gamma\}$$

where F is a covariant type constructor. (For the sake of simplicity, we use a constraint set here, rather than a graph; besides, the small terms invariant isn't respected. This has no influence on the problem at hand.)

σ is closed, because linking α's lower bound to its upper bound yields the constraint $F\,\beta \leq F\,\gamma$, whose consequence through structural decomposition, namely $\beta \leq \gamma$, is part of the constraint graph. Thus, it is correct to apply garbage collection to σ. Let us first compute polarities:

$$\alpha^{\pm} \to \alpha^{\pm} \mid \{F\,\beta^{+} \leq \alpha^{\pm} \leq F\,\gamma^{-},\ \beta^{+} \leq \gamma^{-}\}$$

We can then compute $\mathrm{GC}(\sigma)$. It is identical to σ, except that the constraint $\beta^{+} \leq \gamma^{-}$ is eliminated. Indeed, garbage collection keeps a constraint between two variables only when the former is negative and the latter is positive. So, $\mathrm{GC}(\sigma)$ is

$$\alpha^{\pm} \to \alpha^{\pm} \mid \{F\,\beta^{+} \leq \alpha^{\pm} \leq F\,\gamma^{-}\}$$

However, this type scheme is not closed, regardless of which notion of closure is used. Indeed, for it to be weakly closed, the assertion $\{F\,\beta^+ \leq \alpha^\pm \leq F\,\gamma^-\} \vdash \beta \leq \gamma$ would have to hold—but it doesn't.

This situation is embarrassing. From a practical point of view, it means that the garbage collection phase does not fit naturally into the type inference and simplification process. Indeed, a closure property is required, primarily to ensure that the program at hand is well-typed, but also as a prerequisite for the simplification algorithms, including garbage collection itself.

Without this property, the closure of the graph output by garbage collection would have to be explicitly re-computed. Thus, the closure computation would no longer be incremental. Furthermore, it would seem particularly unnatural, since this computation would then add back constraints which were explicitly dismissed by garbage collection!

Happily, this problem has a simple solution. In the above example, the closure property is violated because the variable α retains non-trivial lower and upper bounds after garbage collection. However, this is only possible because α is bipolar. If it were only positive (resp. negative), then its upper (resp. lower) bound after garbage collection would be $\top$ (resp. $\perp$) and the problem wouldn't arise. Hence, we suggest to disallow bipolar variables altogether.

12.2 Definition

Definition 12.1 *Let σ be a weakly closed type scheme. σ verifies the* mono-polarity invariant *iff*

$$\mathrm{dom}^+(\sigma) \cap \mathrm{dom}^-(\sigma) = \varnothing$$

i.e. iff σ has no bipolar type variables.

(In the above definition, the weak closure property is required only so that the polarity computation has a meaning.) It is easy to verify that $\mathrm{GC}(\sigma)$ is (plainly) closed when σ verifies the mono-polarity invariant; actually, the closure conditions hold vacuously, as shown by the following proposition.

Proposition 12.1 *Let σ be a collectable type scheme, which verifies the mono-polarity invariant. Then $\mathrm{GC}(\sigma)$ is closed.*

Proof. Write $\sigma = A \Rightarrow \tau \mid C$ and $\mathrm{GC}(\sigma) = A \Rightarrow \tau \mid D$, as in definition 10.5. We must first verify that $\leq_D$ is transitive. Assume $\alpha \leq_D \beta$ and $\beta \leq_D \gamma$. Then, by definition of $\leq_D$, $\beta \in \mathrm{dom}^+(\sigma)$ and $\beta \in \mathrm{dom}^-(\sigma)$. This is impossible, because σ verifies the mono-polarity invariant. Hence, the transitivity of $\leq_D$ vacuously holds.

Next, assume $\alpha \leq_D \beta$. We want to prove that $D^\uparrow(\alpha)$ contains $D^\uparrow(\beta)$. By definition of $\leq_D$, $\alpha \leq_D \beta$ implies $\beta \in \mathrm{dom}^+(\sigma)$. Because σ verifies the mono-polarity invariant, it follows that $\beta \notin \mathrm{dom}^-(\sigma)$. According to definition 10.5, this implies $D^\uparrow(\beta) = \top$. It follows that the containment assertion is trivially true. Symmetrically, one verifies that $D^\downarrow(\beta)$ contains $D^\downarrow(\alpha)$.

Finally, pick $\alpha \in \mathrm{dom}(\mathrm{GC}(\sigma))$. We wish to prove that D contains the constraint $D^\downarrow(\alpha) \leq D^\uparrow(\alpha)$. Since α cannot be both positive and negative in σ, either $D^\downarrow(\alpha)$ is equal to $\perp$, or $D^\uparrow(\alpha)$ is equal to $\top$. In both cases, that constraint is trivially contained in D.

We have verified that all three conditions of definition 3.10 trivially hold. Thus, D is closed, and so is $\mathrm{GC}(\sigma)$. □

Thanks to the mono-polarity invariant, garbage collection produces type schemes which are in a particularly simple form, which we shall call *perfect.* We shall now describe it briefly. Let us first define it:

Definition 12.2 *A type scheme σ is* perfect *iff it is equal to* $\mathrm{GC}(\sigma_0)$*, for a certain type scheme σ_0 which is collectable and satisfies the mono-polarity invariant.*

It is clear that any type scheme is equivalent to a perfect scheme. To produce the latter, it suffices to apply successively the closure, canonization and garbage collection algorithms. Perfect schemes are extremely simple, as shown by the following proposition:

Proposition 12.2 *Let $\sigma = A \Rightarrow \tau \mid C$ be a perfect type scheme. Then*

- *any variable of σ is either negative, or positive, but neither neutral nor bipolar;*
- *for all $\alpha \in \mathrm{dom}^+(\sigma)$, $C^\downarrow(\alpha)$ is simple and $C^\uparrow(\alpha)$ is equal to $\top$;*
- *for all $\alpha \in \mathrm{dom}^-(\sigma)$, $C^\downarrow(\alpha)$ is equal to $\bot$ and $C^\uparrow(\alpha)$ is simple;*
- *if $\alpha \leq_C \beta$, then $\alpha \in \mathrm{dom}^-(\sigma)$ and $\beta \in \mathrm{dom}^+(\sigma)$.*

Proof. Immediate, by taking the mono-polarity invariant into account. □

The minimization algorithm, which shall be described in chapter 13, expects a perfect type scheme and preserves this property. So, the fully simplified type schemes produced by our system shall be perfect schemes.

This representation is particularly simple. Besides, a rather interesting remark can be made about it: a type scheme's "polymorphism" comes solely from the constraints between variables. This statement is formalized and explained below.

Definition 12.3 *Let $\sigma = A \Rightarrow \tau \mid C$ be a perfect type scheme. σ is* trivial *iff it contains no constraints between variables, i.e. iff the relation $\leq_C$ is the diagonal.*

Proposition 12.3 *Let σ be a trivial type scheme. Then, its denotation is a cone.*

Proof. Let σ' be the type scheme obtained from σ by adding an equality constraint between each positive (resp. negative) variable and its constructed lower (resp. upper) bound. Since σ' contains no constraints between variables, it is clearly closed. Hence, one can apply garbage collection to it. One immediately finds that $\mathrm{GC}(\sigma') = \sigma$. As a result, σ and σ' have the same denotation.

However, σ''s constraint graph is made up solely of equations relating each variable to its constructed bound. In other words, it is a contractive system of equations. Thus, it admits a unique solution. So, the type scheme σ' has a unique ground instance; its denotation is the cone generated by this instance, that is, the set of points which are above this instance with respect to the ground subtyping relation. □

In less technical terms, the above proposition states that if σ is a trivial type scheme, then it admits a smallest ground instance, which characterizes it fully. So, σ essentially offers no polymorphism, even though it might contain universally quantified variables, since one of its instances is more precise than all others. This property shall be used in section 16.1, when typing expansive toplevel `let` constructs.

The ML analog of the notion of trivial type scheme is the notion of type scheme without variables. The latter would not be satisfactory here; in our system, some type schemes cannot be expressed without using variables, because our model contains recursive ground types, and because we chose to use the small terms invariant. Thus, the above proposition expresses the fact that, in a trivial type scheme, variables do not serve to provide polymorphism, but only to label the structure's nodes.

Hence, in the absence of any constraint between variables, a type scheme is characterized by a single point in the space of ground typings. (If surprised by this result, recall that it

$$\frac{\alpha, \beta, \gamma \notin F \qquad A = \text{single}_{(x,\alpha,\beta)}(\Gamma)}{[F]\ \Gamma \vdash_{\mathrm{I}} x : [F \cup \{\alpha, \beta, \gamma\}]\ A \Rightarrow \gamma \mid \varnothing + (\alpha \leq \gamma)} \quad (\text{VAR}_{\mathrm{I}})$$

$$\frac{[F]\ \text{lift}_x(\Gamma); x \vdash_{\mathrm{I}} e : [F']\ (A; x : \tau) \Rightarrow \tau' \mid C \qquad \alpha \notin F'}{[F]\ \Gamma \vdash_{\mathrm{I}} \lambda x.e : [F' \cup \{\alpha\}]\ A \Rightarrow \alpha \mid C + (\tau \to \tau' \leq \alpha)} \quad (\text{ABS}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}[F]\ \Gamma \vdash_{\mathrm{I}} e_1 : [F']\ A_1 \Rightarrow \tau_1 \mid C_1 \\ [F']\ \Gamma \vdash_{\mathrm{I}} e_2 : [F'']\ A_2 \Rightarrow \tau_2 \mid C_2 \qquad \alpha, \gamma \notin F''\end{array}}{\begin{array}{c}[F]\ \Gamma \vdash_{\mathrm{I}} e_1\, e_2 : [F'' \cup \{\alpha, \gamma\}]\ (A_1 \sqcap A_2) \Rightarrow \gamma \mid C \\ \text{where } C = (C_1 \cup C_2) + (\alpha \leq \gamma) + (\tau_1 \leq \tau_2 \to \alpha)\end{array}} \quad (\text{APP}_{\mathrm{I}})$$

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ renaming of } \sigma \qquad \text{rng}(\rho) \cap F = \varnothing}{[F]\ \Gamma \vdash_{\mathrm{I}} X : [F \cup \text{rng}(\rho)]\ \rho(\sigma)} \quad (\text{LETVAR}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}[F]\ \Gamma \vdash_{\mathrm{I}} e_1 : [F']\ A_1 \Rightarrow \tau_1 \mid C_1 \\ [F']\ \Gamma; X : A_1 \Rightarrow \tau_1 \mid C_1 \vdash_{\mathrm{I}} e_2 : [F'']\ A_2 \Rightarrow \tau_2 \mid C_2\end{array}}{[F]\ \Gamma \vdash_{\mathrm{I}} \texttt{let}\, X = e_1 \,\texttt{in}\, e_2 : [F'']\ (A_1 \sqcap A_2) \Rightarrow \tau_2 \mid C_1 \cup C_2} \quad (\text{LET}_{\mathrm{I}})$$

Figure 12.1: Type inference rules, compliant with the mono-polarity invariant

assumes the mono-polarity invariant. For instance, the type scheme $\alpha \to \alpha$ contains no constraint between variables, but α is bipolar in it. Eliminating this bipolar variable yields $\alpha \to \beta \mid \{\alpha \leq \beta\}$, which is not trivial.) Thus, the complexity of the scheme subsumption relation (whose decidability is currently an open problem) stems entirely from the presence of constraints between variables.

12.3 Enforcing the invariant

Is it possible to enforce the mono-polarity invariant at each step of the type inference process? Indeed, and in fact, very easily. A few slight modifications to the type inference rules are enough to guarantee that any type scheme which appears in a type inference judgement complies with the mono-polarity invariant. Besides, each simplification phase (canonization, garbage collection, minimization) preserves the invariant. Finally, we have seen in chapter 11 that the canonization algorithm is able to eliminate the bipolar variables of a given type scheme; this enables us to transform user-supplied type schemes, such as those found in module signatures, to make them compliant with the invariant.

The modified type inference rules are given in figure 12.1. The only rules that were actually modified for compliance with the mono-polarity invariant are (VAR_{I}) and (APP_{I}). The other rules are unchanged, but are displayed again here because this is the definitive formulation of our type inference system.

Both changes have the same cause. In the previous versions of (VAR_{I}) and (APP_{I}), a variable α was created and was used in a way which could potentially make it bipolar. Indeed, in the type scheme $\text{single}_{(x,\alpha,\beta)}(\Gamma) \Rightarrow \alpha \mid \varnothing$, created by the previous rule (VAR_{I}), α is bipolar, since it appears negatively (in the context) and positively (in the scheme's body). In $(A_1 \sqcap A_2) \Rightarrow \alpha \mid C$, where $C = (C_1 \cup C_2) + (\tau_1 \leq \tau_2 \to \alpha)$, created by the previous rule

(App_I), α is positive (since it appears in the body), and it might be negative, because it appears positively in τ_1's upper bound.

The new rules avoid this problem by creating two fresh variables, α and γ, instead of one, and linking them with an additional constraint $\alpha \leq \gamma$. They shall be used in such a way that α is (at most) negative and γ positive. None shall be bipolar. (Recall that signs do not propagate from one variable to another, so the constraint $\alpha \leq \gamma$ does not make these variables bipolar.) It is easy to see that each of the new rules produces a type scheme which is equivalent to the one produced by its previous version. Thus, the modified type inference system is equivalent to the previous one.

Thus, we have eliminated the two obvious places where bipolar variables could be created. There remains to show that this is sufficient to ensure that no bipolar variables can ever appear. The essential explanation of this property is the fact that a variable's polarity decreases throughout its lifetime. In other words, if a variable is not positive (resp. negative) when it is first created, then it cannot become so during the type inference derivation. This fact is formalized by the following lemma.

Lemma 12.4 *Let $[F]\,\Gamma \vdash_\text{I} e : [F']\,\sigma$ be a type inference judgement (derived using the rules of figure 12.1). Its premises contain zero or more type inference judgements; let $(\sigma_i)_{i\in I}$ be the type schemes which appear in them. Let $\alpha \in \text{dom}(\sigma)$. If $\alpha \in \text{dom}(\sigma_i)$ for some $i \in I$, then $\alpha \in \text{dom}^+(\sigma)$ implies $\alpha \in \text{dom}^+(\sigma_i)$, and $\alpha \in \text{dom}^-(\sigma)$ implies $\alpha \in \text{dom}^-(\sigma_i)$.*

Proof. It would be elegant to formulate a proof based on the above intuition. Here is a very rough sketch of it. First, we show that a variable is positive if and only if it is liable to receive a new upper bound in the future. (This corresponds to the intuition given in chapter 10.) Then, we consider a positive variable α. It is possible to find a context where α receives a new bound. Since composing two contexts yields a new context, the same was true of α when it was created. So, α was created positive.

However, even though it is very intuitive, the above speech contains inaccuracies, and it is not clear whether a rigorous version of it can be given. So, we shall simply prove the above statement, by case analysis on the last rule used in the derivation of $[F]\,\Gamma \vdash_\text{I} e : [F']\,\sigma$. There is one case per type inference rule. All are easy, except (App_I) and (Let_I). Furthermore, as explained in sections 5.3 and 5.5, rule (Let_I) can be considered as a combination of a simpler rule and of an application rule. Thus, we shall only detail the case of (App_I). We use exactly the notations of figure 12.1, which allows us not to write the hypotheses explicitly.

The type scheme produced by the rule is $\sigma = (A_1 \sqcap A_2) \Rightarrow \beta \mid C$, where C equals $(C_1 \cup C_2) + (\alpha \leq \gamma) + (\tau_1 \leq \tau_2 \to \alpha)$. We now wish to perform an (approximate) computation of the polarities in σ, so as to compare them with the polarities in σ_1 and σ_2.

However, recall that polarities are defined only if the type scheme is weakly closed (see definition 10.3). However, C, as defined above, isn't. To determine σ's polarities, a closure computation must be performed first. The bulk of the proof consists in simulating this computation.

This simulation is made significantly more complex by the use of constraint graphs, where multiple bounds must be combined using $\sqcup$ or $\sqcap$. This makes the proof cumbersome and interferes with the intuition. So, exceptionally, we shall write the proof in terms of constraint sets, rather than graphs. This is a lack of rigor, but we hope it should help to understand the proof. So, in the following, we assume that C is a constraint set, and we wish to compute its closure C^*, by transitivity on variables and by structural decomposition.

Define $B = C_1 \cup C_2$ and $B' = B \cup \{\alpha \leq \gamma\}$. We assume C_1 and C_2 to be closed constraint sets, since they are produced by type inference judgements. Moreover, $\text{dom}(C_1)$, $\text{dom}(C_2)$ and $\{\alpha, \gamma\}$ are pairwise disjoint, so B' is closed. There remains to add the constraint $\tau_1 \leq \tau_2 \to \alpha$ and to perform the closure computation.

By convention, if β is a variable belonging to $\mathrm{dom}(\sigma_i)$ for a certain $i \in \{1, 2\}$, we shall simply write β^+ (resp. β^-) to indicate that $\beta \in \mathrm{dom}^+(\sigma_i)$ (resp. $\mathrm{dom}^-(\sigma_i)$). A similar notation is adopted for small terms.

Let us attempt to give an intuition for this proof. A function application generates a constraint between a producer and a consumer, that is, between a positive term and a negative one. The closure phenomenon causes additional constraints to appear, which are of the same form. Thus, the new constraints do not cause polarities to increase. Indeed, a constraint can possibly make its left side positive, and its right side negative; however, here, they already are.

To compute C^*, we consider all constraints in C, and we combine them using transitivity on variables and structural decomposition. We claim that the resulting constraints are of one of the following forms:

1. $c \in B'$;
2. $\beta \leq \tau_2^+ \rightarrow \alpha$, where $\beta \leq_B \delta^+$;
3. $\tau^+ \leq \phi$, where τ is a small term, or $\beta \leq \phi$, where $\beta \leq_B \delta^+$, and $\phi \in \{\alpha, \gamma\}$;
4. $\beta \leq \lambda$, where $\beta \leq_B \delta^+$ and $\epsilon^- \leq_B \lambda$;
5. $\tau^+ \leq \lambda$, where τ is a small term, and $\epsilon^- \leq_B \lambda$;
6. $\beta \leq \tau^-$, where τ is a small term, and $\beta \leq_B \delta^+$.

To prove this assertion, we first verify that all constraints present at the beginning of the computation are of one of these forms. It is true of constraints in B', since they are of the form 1; as for the constraint $\tau_1^+ \leq \tau_2^+ \rightarrow \alpha$, it is of the form 2.

Then, there remains to verify that the set described above is stable by transitivity and by structural decomposition. For instance, imagine that we combine a constraint of the form 1 with a constraint of the form 2. The latter can be written $\beta \leq \tau_2^+ \rightarrow \alpha$, where $\beta \leq_B \delta^+$. The former can be of the form $\epsilon \leq \beta$, where $\epsilon \leq_B \beta$, or of the form $\tau \leq \beta$, where τ is a small term. In the first case, the resulting constraint is $\epsilon \leq \tau_2^+ \rightarrow \alpha$, which is of the form 2, because $\epsilon \leq_B \delta^+$. (This last assertion is obtained by transitivity of $\leq_B$.) In the second case, the resulting constraints are obtained by structural decomposition of the constraint $\tau \leq \tau_2^+ \rightarrow \alpha$. However, still by transitivity, B contains the constraint $\tau \leq \delta^+$, so τ is positive. Consequently, the decomposition yields constraints of the form 3, for the range, and 4, for the domain. The 35 other cases are no harder, and left to the reader.

Once this result is established, we can verify that polarities have decreased, that is,

$$\mathrm{dom}^+(\sigma) \subseteq \mathrm{dom}^+(\sigma_1) \cup \mathrm{dom}^+(\sigma_2) \cup \{\gamma\}$$
$$\mathrm{dom}^-(\sigma) \subseteq \mathrm{dom}^-(\sigma_1) \cup \mathrm{dom}^-(\sigma_2) \cup \{\alpha\}$$

It suffices to verify that the sets $P = \mathrm{dom}^+(\sigma_1) \cup \mathrm{dom}^+(\sigma_2) \cup \{\gamma\}$ and $N = \mathrm{dom}^-(\sigma_1) \cup \mathrm{dom}^-(\sigma_2) \cup \{\alpha\}$ satisfy the conditions of definition 10.3. The first one requires $\gamma \in P$, and the second one $\mathrm{fv}(A_1 \sqcap A_2) \subseteq N$; they hold. The third one requests that for any constraint of the form $\tau \leq \beta$, where τ is a small term and $\beta \in P$, we have $\mathrm{fv}^+(\tau) \in P$ and $\mathrm{fv}^-(\tau) \in N$. So, let us consider such a constraint, and reason by case on its form, using the classification given above:

1. $c \in B'$. Then c is present in C_i, for a certain $i \in \{1, 2\}$, so the condition holds by definition of $\mathrm{dom}^+(\sigma_i)$ and $\mathrm{dom}^-(\sigma_i)$.
2. This case cannot occur, since the two forms being considered are incompatible.

3. $\tau^+ \leq \phi$, where τ is a small term and $\phi \in \{\alpha, \gamma\}$; then the condition holds, because τ is positive.

4. This case cannot occur.

5. $\tau^+ \leq \lambda$, where τ is a small term and $\epsilon^- \leq_B \lambda$; then the condition holds, because τ is positive.

6. This case cannot occur.

The fourth condition is handled similarly. □

As a corollary, we obtain the expected result:

Proposition 12.5 *If* $[F]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ \sigma$ *is derivable in the inference system given by figure 12.1, then* σ *verifies the mono-polarity invariant.*

Proof. It is easy to verify that no variable is bipolar when it is first created. Additionally, according to lemma 12.4, polarities decrease throughout a variable's lifetime. The result follows. □

Besides, one easily verifies that all simplification methods introduced so far (canonization, garbage collection) preserve the mono-polarity invariant. So shall the minimization algorithm. We shall thus assume, from now on, that all type schemes under study satisfy this invariant.

12.4 Remarks

The main reason why we introduce the mono-polarity invariant is its guarantee that the garbage collection algorithm creates closed type schemes. This is a very nice property, which significantly simplifies the theory. Indeed, as explained before, in the absence of this property, we would need to either have the type inference engine re-compute the closure after garbage collection, or develop a more complex theory, allowing us to work with non-closed constraint graphs. We have investigated both possibilities before discovering the effect of the mono-polarity invariant. The former is simple, but inelegant and inefficient. The latter seems workable and does not cause runtime inefficiencies, but it involves rather tricky proofs. In conclusion, the mono-polarity invariant seems by far the most simple and elegant solution to this problem.

Besides, this invariant has other interesting effects. For instance, it slightly simplifies the definition of the minimization algorithm (see chapter 13). This algorithm could be formulated without the mono-polarity invariant; it would suffice to specify that a bipolar variable cannot be merged with any other variable (i.e. its equivalence class is a singleton). When the mono-polarity invariant is enforced, the algorithm becomes slightly simpler, thus more efficient, since bipolar variables do not have to be special-cased. Moreover, it becomes slightly more effective. Indeed, a bipolar variable cannot be identified with any other variable; whereas, if we split it into a pair of a negative variable and a positive one, each of these variables can possibly be merged with other variables. So, the mono-polarity invariant should create some more opportunities for the minimization algorithm. In practice, experiments show that they are not numerous; however, this was historically the first reason why we imagined the invariant.

Early implementations of our system contained a specific pass to eliminate cycles of variables. Whenever such a cycle, of the form $\alpha_1 \leq \ldots \leq \alpha_n \leq \alpha_1$, was detected, all α_i's would be merged together. This simplification tactic is also found in other works [14, 4].

Today, the combination of garbage collection and of the mono-polarity invariant makes this pass superfluous. Indeed, assume a graph contains a chain of two constraints $\alpha \leq \beta \leq \gamma$. If this chain survives garbage collection, then we have $\alpha^- \leq \beta^+$ and $\beta^- \leq \alpha^+$, so β is bipolar. This is impossible; hence, there are no chains after garbage collection, and *a fortiori*, no cycles. Actually, when a cycle appears, it is first completed by closure; then, garbage collection destroys it by keeping only constraints which link a negative variable to a positive one; finally, minimization merges all positive (resp. negative) variables of the cycles together. The result thus obtained is satisfactory. However, one might wish to detect and eliminate the cycle as soon as it appears, so as to avoid needless closure computations. This suggestion has been made by Fähndrich *et al.* [16]; we intend to study it.

Another consequence of the mono-polarity invariant is the fact that it is no longer legal to replace a positive (resp. negative) variable with its unique lower (resp. upper) bound. Replacing a variable with its bound has been proposed as a simplification tactic in numerous papers [14, 4, 8, 42]. Why is it illegal here? We have mentioned in chapter 6 that replacing a variable with a constructed term breaks the small terms invariant. Until now, it was still legal when the bound also was a variable. Assume, for instance, that α is a positive variable whose unique lower bound is a type variable β. The constraint $\beta \leq \alpha$ has survived garbage collection, so β is negative. Hence, if we merge α and β, we create a bipolar variable and violate the mono-polarity invariant.

Thus, replacing a type variable with its unique bound is now entirely disallowed inside the type inference engine. One should stress, however, that this is not a disadvantage of our system. First, the space loss is at most a factor of two, and there is no efficiency loss, since on the contrary, we claim that the invariant has beneficial effects. Second, although this substitution strategy is prohibited within the type inference engine, it remains valid from a theoretical point of view. So, it can still be used, after type inference is over, to make type schemes more readable. To conclude, we have shown that the two goals of simplification, namely readability and efficiency, conflict with one another, since efficiency requires certain invariants (small terms and mono-polarity) which decrease readability. Thus, efficiency should be favored throughout the inference process, and readability should be taken care of in a final pass, immediately before display.

Chapter 13

Minimization

Up to now, we have presented few simplification methods. First, canonization, which is a simplification, insofar as it subsequently allows certain algorithms, which cannot function in the presence of $\sqcup$ or $\sqcap$ constructs, to be run. Then, garbage collection, which simply detects superfluous constraints and eliminates them. However, by browsing through the literature about constraint simplification, one finds many methods based on the idea of *substitution*, which consists in replacing a variable with a term without modifying the denotation of the type scheme of interest.

This chapter is devoted to the description of a simplification method, called *minimization*, which generalizes all known substitution methods, while being remarkably efficient. Its principle was inspired by the so-called "Hopcroft" method in Felleisen and Flanagan [18, 19].

Section 13.1 describes the problem and sketches the minimization algorithm. We then give a formal specification of minimization (section 13.2), and explain how to implement it (section 13.3). Next, we give some examples of applications (section 13.4). Lastly, section 13.5 discusses the completeness of this simplification method.

13.1 Introduction

Among the known substitution methods, one finds specialized algorithms, which are efficient but only handle a precise particular case; and heuristics, which are *a priori* rather powerful, but often very inefficient.

Among the non-heuristic algorithms, one can mention eliminating cycles of variables, and replacing a positive (resp. negative) variable with its unique lower (resp. upper) bound. However, we have noticed that, if the mono-polarity invariant is enforced, the former is automatically performed as a side-effect of garbage collection (see section 12.4). As for the latter, we have shown that it breaks the small terms invariant or the mono-polarity invariant (see sections 6.4 and 12.4), and is thus illegal, at least as part of the type inference process—it remains valid if used for display purposes only.

All heuristics have a common pattern, in two stages. First, come up, in one way or another, with a substitution ρ. Then, determine whether it is legal to apply ρ to the type scheme σ under study, that is, whether $\rho(\sigma) =^\forall \sigma$. This test can be performed using the algorithm developed in chapter 9, or possibly using less powerful algorithms, such as the axiomatization of entailment given in chapter 8. We have proposed several heuristics based on the latter in [42]. However, the number of possible substitutions is huge. (One can, of course, restrict one's attention to particular cases, but they are often rather *ad hoc*.) Furthermore, the entailment algorithm used during the second step is rather costly. The combination of these two factors yields very inefficient simplification methods.

Thus, it seems desirable to design a systematic algorithm, capable of directly building as powerful a substitution as possible. Let us first comment on what we mean by "powerful". In order to respect the small terms invariant, a variable can only be replaced with another variable. Thus, a substitution can be viewed as a partition of the set of variables. In this view, two variables are to be merged iff they belong to the same block (i.e. class). We can then say that a substitution is less powerful than another one iff its associated partition is finer. Taking into account the fact that the mono-polarity invariant must be respected, the most powerful substitution is the partition with only two classes, namely $\text{dom}^+(\sigma)$ et $\text{dom}^-(\sigma)$; in other words, the substitution which merges all positive (resp. negative) variables together.

Thus, we are looking for a substitution ρ, which is finer than the one described above (so as to comply with the mono-polarity invariant), correct (i.e. such that $\rho(\sigma) =^\forall \sigma$), and as powerful as possible given these criteria. However, because the relation $=^\forall$ is complex, it is not clear whether such a substitution exists in the general case. Thus, we will replace the correctness criterion with a slightly more restrictive one, which is simpler, because it is directly related to the form of constraints.

To imagine this criterion, we can sketch, in a very rough way, an algorithm capable of building the substitution. Start with the most powerful substitution. Run the subsumption algorithm to determine whether this substitution is correct. It if reports a failure, determine its cause, and weaken the substitution by splitting an appropriate class, so as to eliminate the problem. Repeat this process until no more errors are found.

Still informally, one could say that such a failure happens when attempting to merge two variables which play different roles in the type scheme. The "role" of a variable is defined by the way it is linked, through constraints, to other variables. So, one could say that two variables can be identified if each of them carries the same kind of links as the other, and if these links lead to variables which can in turn be merged.

These are familiar notions. Refining a partition, as described above, resembles the process used to minimize finite state automata. The informal criterion given in the previous paragraph is a kind of bisimulation, a concept also found in automata theory. Hence, the idea to *minimize* a constraint graph, somewhat like an automaton. As we shall now see, this idea makes sense, and even allows reusing the efficient algorithms developed to minimize finite state automata.

Applying minimization techniques to constraint systems was proposed by Felleisen and Flanagan [18, 19]. Their definition of compatibility (see definition 13.3) is somewhat different, and in particular less symmetrical, because their constraint language departs rather significantly from ours, both from a syntactic and from a semantic point of view. Nevertheless, the principle is identical. It is independent from the use of subtyping, and should be applicable to numerous recursive-constraint-based systems, regardless of their semantics.

13.2 Formalization

We shall now give a formal specification of the algorithm, and prove its correctness. First, here are a few preliminary definitions.

Definition 13.1 *Given a set of variables V, the equivalence relations on V are isomorphic to the partitions of V. Any relation $\equiv$ is extended to small terms with variables in V by setting*

$$\begin{aligned} \bot &\equiv \bot \\ \top &\equiv \top \\ \alpha_0 \to \alpha_1 \equiv \beta_0 \to \beta_1 &\iff (\alpha_0 \equiv \beta_0) \wedge (\alpha_1 \equiv \beta_1) \end{aligned}$$

Definition 13.2 *Let C be a constraint graph. For $\alpha \in \mathrm{dom}(C)$, define*

$$\mathrm{pred}_C(\alpha) = \{\beta\,;\,\beta \leq_C \alpha\}$$
$$\mathrm{succ}_C(\alpha) = \{\beta\,;\,\alpha \leq_C \beta\}$$

We have mentioned, in section 13.1, that a substitution's correctness is too complex a criterion—possibly even an undecidable one—because it involves the relation $=^\forall$. So, we introduce a notion of *compatibility*, which is weaker, but whose definition is purely syntactic and directly leads to an algorithm. All subsequent developments shall be based on this notion.

Definition 13.3 *Let $\sigma = A \Rightarrow \tau \mid C$ be a perfect type scheme. An equivalence relation $\equiv$, of domain* $\mathrm{dom}(\sigma)$*, is* compatible *with σ iff*

- $\alpha \equiv \beta$ *implies* $\mathrm{pred}_C(\alpha) = \mathrm{pred}_C(\beta)$ *and* $\mathrm{succ}_C(\alpha) = \mathrm{succ}_C(\beta)$*;*
- $\alpha \equiv \beta$ *implies* $C^\downarrow(\alpha) \equiv C^\downarrow(\beta)$ *and* $C^\uparrow(\alpha) \equiv C^\uparrow(\beta)$*;*
- $\alpha \equiv \beta$ *implies* $\mathrm{polarity}_\sigma(\alpha) = \mathrm{polarity}_\sigma(\beta)$.

Lemma 13.1 *The first condition above can also be formulated as follows:*

- $\alpha \leq_C \beta$ *implies* $\forall \alpha' \equiv \alpha \quad \forall \beta' \equiv \beta \quad \alpha' \leq_C \beta'$.

Proof. Assume the first requirement holds. Assume $\alpha \leq_C \beta$, $\alpha' \equiv \alpha$ and $\beta' \equiv \beta$. We have $\beta \in \mathrm{succ}_C(\alpha)$; by applying the hypothesis, we obtain $\beta \in \mathrm{succ}_C(\alpha')$, that is, $\alpha' \leq_C \beta$. By applying the hypothesis once again to this result, we obtain $\alpha' \leq_C \beta'$, as desired.

Reciprocally, assume the variant of the first condition is satisfied. Assume $\alpha \equiv \beta$, and $\alpha' \in \mathrm{pred}_C(\alpha)$. Then $\alpha' \leq_C \alpha$. By applying the hypothesis, we obtain $\alpha' \leq_C \beta$. We have shown $\mathrm{pred}_C(\alpha) \subseteq \mathrm{pred}_C(\beta)$. By symmetry, the two are equal. Similarly, one shows that $\mathrm{succ}_C(\alpha) = \mathrm{succ}_C(\beta)$. □

It is easy to verify that there exists a unique partition compatible with a given scheme, and of maximum power for this criterion. It is this partition that shall be computed by the minimization algorithm.

Proposition 13.2 *Let σ be a perfect type scheme. The set of all partitions compatible with σ admits a greatest element, denoted by $\equiv_\sigma$.*

Proof. First, it is clear that the trivial partition, where all points are isolated, is compatible. Next, let us verify that the fusion $\equiv$ of two compatible partitions $\equiv_1$ and $\equiv_2$ is itself compatible. Recall that $\equiv$ is defined as the transitive closure of the relation $\equiv_{12} = \equiv_1 \cup \equiv_2$.

Let us consider the first condition of definition 13.3. It is satisfied by $\equiv_1$ and $\equiv_2$ (since these are compatible), hence also by their union $\equiv_{12}$. Thus, $\equiv_{12}$ is included in the relation "to have same successors and predecessors". However, the latter is transitive; hence, it also contains the transitive closure of $\equiv_{12}$, namely $\equiv$. The first condition thus holds. The second and third conditions are handled similarly.

The set of compatible partitions is non-empty and closed by fusion; thus, it admits a greatest element. □

To establish a link between partitions and substitutions, let us define the *quotient* of a type scheme by a partition, that is, the operation which collapses classes down to a single variable.

Definition 13.4 *Let $\sigma = A \Rightarrow \tau \mid C$ be a perfect type scheme; let $\equiv$ be a partition compatible with σ. Out of each class, pick a representative; that is, let π be a function of* $\mathrm{dom}(\sigma)$ *into itself such that*

- $\forall\alpha \in \mathrm{dom}(\sigma) \quad \pi(\alpha) \equiv \alpha$;
- $\forall\alpha, \beta \in \mathrm{dom}(\sigma) \quad \alpha \equiv \beta \iff \pi(\alpha) = \pi(\beta)$.

Let $\sigma/_{\equiv}$ be the type scheme $\pi(\sigma)$. More precisely, $\sigma/_{\equiv} = \pi(A) \Rightarrow \pi(\tau) \mid \pi(C)$ where $\pi(C)$ is the constraint graph of domain $\pi(\mathrm{dom}(\sigma))$ defined by

- $\pi(\alpha) \leq_{\pi(C)} \pi(\beta)$ *iff* $\alpha \leq_C \beta$;
- $(\pi(C))^{\downarrow}(\pi(\alpha)) = \pi(C^{\downarrow}(\alpha))$ *and* $(\pi(C))^{\uparrow}(\pi(\alpha)) = \pi(C^{\uparrow}(\alpha))$.

Here, the graph $\pi(C)$ is defined by two characteristic properties. Hence, we must verify that there exists a unique object which satisfies them.

Proof. When α and β range over all of $\mathrm{dom}(\sigma)$, $\pi(\alpha)$ and $\pi(\beta)$ range over all of $\pi(\mathrm{dom}(\sigma))$. Thus, these two properties specify a unique object. However, each point of $\pi(\mathrm{dom}(\sigma))$ may be visited several times; to prove the object's existence, one must verify that there is no contradiction between these visits.

Let us consider the first property. Assume $\pi(\alpha) = \pi(\alpha')$ and $\pi(\beta) = \pi(\beta')$. Then, according to the definition of π, $\alpha \equiv \alpha'$ and $\beta \equiv \beta'$. Since $\equiv$ is compatible with σ, this implies $(\alpha \leq_C \beta) \iff (\alpha' \leq_C \beta')$.

Now, consider the second one. Assume $\pi(\alpha) = \pi(\alpha')$. Then $\alpha \equiv \alpha'$. Since $\equiv$ is compatible, this implies $C^{\downarrow}(\alpha) \equiv C^{\downarrow}(\alpha')$, which in turn implies $\pi(C^{\downarrow}(\alpha)) = \pi(C^{\downarrow}(\alpha'))$. □

Note that the definition of $\sigma/_{\equiv}$ depends on the choice of π. (Recall that π is defined by picking a representative out of each congruence class.) However, different choices of π yield α-equivalent type schemes. Thus, $\sigma/_{\equiv}$ is defined up to α-conversion. This shall not be a problem; the properties we are interested in are independent of the choice of π.

We can now proceed to the correctness proof, which states that any compatible partition leads to a quotient scheme equivalent to the original scheme.

Theorem 13.1 *Let σ be a perfect type scheme, and $\equiv$ a partition compatible with σ. Then*

$$\sigma/_{\equiv} =^{\forall} \sigma$$

Proof. The assertion $\sigma \leq^{\forall} \sigma/_{\equiv}$ is straightforward, because the latter is the image of the former through the substitution π. More precisely, if ρ is a solution of $\pi(C)$, then $\rho \circ \pi$ is a solution of C and a witness to this assertion.

Reciprocally, we wish to show that $\sigma/_{\equiv} \leq^{\forall} \sigma$. Let ρ be a solution of C. Define ρ' by

$$\rho'(\pi(\alpha)) = \bigsqcap_{\alpha' \equiv \alpha} \rho(\alpha') \text{ if } \pi(\alpha) \in \mathrm{dom}^{+}(\sigma);$$
$$\rho'(\pi(\alpha)) = \bigsqcup_{\alpha' \equiv \alpha} \rho(\alpha') \text{ if } \pi(\alpha) \in \mathrm{dom}^{-}(\sigma).$$

We must verify that this definition makes sense. First, note that each case is dealt with once and only once, since σ has neither neutral nor bipolar variables. Next, if $\pi(\alpha) = \pi(\beta)$, then $\alpha \equiv \beta$, so $\{\alpha' ; \alpha' \equiv \alpha\} = \{\beta' ; \beta' \equiv \beta\}$. It follows that ρ' is well-defined.

Let us now verify that ρ' is a solution of $\pi(C)$. First, pick a constraint in $\leq_{\pi(C)}$, which we can write $\pi(\alpha) \leq_{\pi(C)} \pi(\beta)$. According to definition 13.4, we have

$$\forall\alpha' \equiv \alpha \quad \forall\beta' \equiv \beta \quad \alpha' \leq_C \beta'$$

Since ρ is a solution of C, this implies

$$\forall \alpha' \equiv \alpha \quad \forall \beta' \equiv \beta \quad \rho(\alpha') \leq \rho(\beta')$$

which be can be rewritten

$$\bigsqcup_{\alpha' \equiv \alpha} \rho(\alpha') \leq \bigsqcap_{\beta' \equiv \beta} \rho(\beta')$$

Now, $\alpha' \leq_C \beta'$ implies $\alpha' \in \mathrm{dom}^-(\sigma)$ and $\beta' \in \mathrm{dom}^+(\sigma)$, because σ is perfect, and thus only contains constraints between a negative and a positive variable. Since $\equiv$ is compatible with σ, and since $\alpha' \equiv \pi(\alpha)$, α' has the same polarity as $\pi(\alpha)$. This implies $\pi(\alpha) \in \mathrm{dom}^-(\sigma)$. Similarly, $\pi(\beta) \in \mathrm{dom}^+(\sigma)$. Given these facts, the above assertion can be read as

$$\rho'(\pi(\alpha)) \leq \rho'(\pi(\beta))$$

which means that ρ' satisfies the chosen constraint.

Next, we must verify that $\rho'((\pi(C))^\downarrow(\pi(\alpha))) \leq \rho'(\pi(\alpha))$ holds for any α. By definition of $\pi(C)$, this is equivalent to $\rho'(\pi(C^\downarrow(\alpha))) \leq \rho'(\pi(\alpha))$. If $\alpha \in \mathrm{dom}^-(\sigma)$, then $C^\downarrow(\alpha) = \bot$ (because σ is perfect) and the result is immediate. So, assume $\alpha \in \mathrm{dom}^+(\sigma)$. The goal then becomes

$$\rho'(\pi(C^\downarrow(\alpha))) \leq \bigsqcap_{\alpha' \equiv \alpha} \rho(\alpha')$$

which can be rewritten

$$\forall \alpha' \equiv \alpha \quad \rho'(\pi(C^\downarrow(\alpha))) \leq \rho(\alpha')$$

Note that $\alpha' \equiv \alpha$ implies $C^\downarrow(\alpha') \equiv C^\downarrow(\alpha)$, because $\equiv$ is compatible with σ. Thanks to this remark, it suffices to show that for any $\alpha' \equiv \alpha$,

$$\rho'(\pi(C^\downarrow(\alpha'))) \leq \rho(\alpha')$$

Since ρ is a solution of C, we have $\rho(C^\downarrow(\alpha')) \leq \rho(\alpha')$. So, it suffices to show

$$\rho'(\pi(C^\downarrow(\alpha'))) \leq \rho(C^\downarrow(\alpha'))$$

We now perform a case analysis on $C^\downarrow(\alpha')$. If it equals $\bot$ or $\top$, the result is immediate. So, assume $C^\downarrow(\alpha') = \alpha'_0 \rightarrow \alpha'_1$. The goal is split into two sub-goals, one of which is

$$\rho'(\pi(\alpha'_1)) \leq \rho(\alpha'_1)$$

(The other sub-goal is symmetric, so we do not treat it explicitly.) Now, because $\alpha \in \mathrm{dom}^+(\sigma)$ and $\alpha' \equiv \alpha$, we have $\alpha' \in \mathrm{dom}^+(\sigma)$. According to definition 10.3, this implies $\alpha'_1 \in \mathrm{dom}^+(\sigma)$. Thus, $\pi(\alpha'_1) \in \mathrm{dom}^+(\sigma)$. The goal can be rewritten

$$\bigsqcap_{\beta \equiv \alpha'_1} \rho(\beta) \leq \rho(\alpha'_1)$$

which is immediate, since the right-hand expression appears as one of the operands of the left-hand expression.

Symmetrically, one verifies that $\rho'(\pi(\alpha)) \leq \rho'((\pi(C))^\uparrow(\pi(\alpha)))$ holds for any α. Hence, ρ' is a solution of $\pi(C)$. There remains to verify that $\rho'(\pi(A \Rightarrow \tau)) \leq \rho(A \Rightarrow \tau)$, which is straightforward, using the same techniques as in the previous paragraphs. To conclude, ρ' is a witness for $\sigma/_\equiv \leq^\forall \sigma$. The theorem follows. $\square$

13.3 Algorithm

We have specified a simplification method, which consists in computing the quotient of a perfect type scheme σ by the most powerful compatible partition, thus replacing σ with $\sigma/_{\equiv_\sigma}$. There remains to determine whether this computation can be efficiently performed. The answer is positive—we can compute $\equiv_\sigma$ in time $O(dn \log n)$, where n is the number of variables in σ, and d is the degree of the graph $\leq_C$ (i.e. the maximum number of branches coming out of or into a single node).

To define the algorithm, we shall slightly rephrase the definition of compatibility, so as to make apparent the various steps required to compute $\equiv_\sigma$. This definition states that, if $\equiv$ is compatible, then two equivalent variables have the same predecessor and successor sets, equivalent constructed bounds, and the same polarity. The second condition is the trickiest, because it is "recursive", i.e. the relation $\equiv$ appears on both sides of the implication. Furthermore, it uses the relation $\equiv$ to compare small terms; to make things simpler, we first rephrase it so that it only compares variables.

Lemma 13.3 *Let $\sigma = A \Rightarrow \tau \mid C$ be a perfect type scheme. A partition $\equiv$ is compatible with σ iff*

- $\alpha \equiv \beta$ *implies* $\mathrm{pred}_C(\alpha) = \mathrm{pred}_C(\beta)$ *and* $\mathrm{succ}_C(\alpha) = \mathrm{succ}_C(\beta)$*;*
- $\alpha \equiv \beta$ *implies* $\mathrm{polarity}_\sigma(\alpha) = \mathrm{polarity}_\sigma(\beta)$*;*
- $\alpha \equiv \beta$ *implies* $\mathrm{head}(C^\downarrow(\alpha)) = \mathrm{head}(C^\downarrow(\beta))$ *and* $\mathrm{head}(C^\uparrow(\alpha)) = \mathrm{head}(C^\uparrow(\beta))$*;*
- *if $\alpha \equiv \beta$, $C^\downarrow(\alpha) = \alpha_0 \to \alpha_1$ and $C^\downarrow(\beta) = \beta_0 \to \beta_1$, then $\alpha_0 \equiv \beta_0$ and $\alpha_1 \equiv \beta_1$;*
- *if $\alpha \equiv \beta$, $C^\uparrow(\alpha) = \alpha_0 \to \alpha_1$ and $C^\uparrow(\beta) = \beta_0 \to \beta_1$, then $\alpha_0 \equiv \beta_0$ and $\alpha_1 \equiv \beta_1$.*

The first three conditions above pose no problem, since they actually define an initial partition, which we must then refine. The last two conditions resemble the definition of a bisimulation, as mentioned in section 13.1. We are now ready to define the algorithm.

Theorem 13.2 *Let $\sigma = A \Rightarrow \tau \mid C$ be a perfect type scheme. Then $\equiv_\sigma$ can be computed in time $O(dn \log n)$, where $n = |\,\mathrm{dom}(\sigma)\,|$, and d is the degree of the graph $\leq_C$.*

Proof. The computation is done in two stages: first, compute an initial partition, then refine it until a fix-point is reached.

Let $\equiv_0$ be the coarsest partition which verifies the first three conditions of lemma 13.3. We compute it as follows. First, build a list of σ's variables, sorted according to the keys

$$\alpha \mapsto (\mathrm{pred}_C(\alpha), \mathrm{succ}_C(\alpha), \mathrm{polarity}_\sigma(\alpha), \mathrm{head}(C^\downarrow(\alpha)), \mathrm{head}(C^\uparrow(\alpha)))$$

(Any ordering on these quintuples will do; lexicographic ordering is the most natural.) Building the list takes time $O(n)$. Sorting requires $O(n \log n)$ comparisons. The above data are assumed to be available in the constraint graph, so they can be accessed in constant time by the comparison function. The sets $\mathrm{pred}_C(\alpha)$ and $\mathrm{succ}_C(\alpha)$ have cardinality less than (or equal to) d, by definition of d. Hence, comparing two keys takes time $O(d)$, and sorting takes time $O(dn \log n)$. Once the sorted list is built, a single pass is enough to create $\equiv_0$, since elements of the same class must be adjacent in the list. This pass requires $O(n)$ comparisons, so it takes time $O(dn)$.

Note that $\equiv_\sigma$ is the coarsest partition which is finer than $\equiv_0$ and verifies the last two conditions of lemma 13.3. For $i \in \{0, 1\}$, let Low_i and Upp_i be the partial functions on $\mathrm{dom}(\sigma)$ defined by

$$\mathrm{Low}_i(\alpha) = \alpha_i \text{ if } C^\downarrow(\alpha) = \alpha_0 \to \alpha_1$$
$$\mathrm{Upp}_i(\alpha) = \alpha_i \text{ if } C^\uparrow(\alpha) = \alpha_0 \to \alpha_1$$

We can now rephrase the problem: $\equiv_\sigma$ is the coarsest partition which is finer than $\equiv_0$ and stable with respect to these four functions. (A partition $\equiv$ is *stable* with respect to a function f iff for every block (i.e. class) B of $\equiv$, either f is undefined on all of B, or f is defined on all of B and $f(B)$ lies entirely within some block B'.)

So, the problem is now to find the coarsest refinement of a given partition which is stable with respect to a finite number of given functions. This is a well-studied problem. It appears, in particular, when minimizing finite state automata; Hopcroft [29] gives an $O(n \log n)$ algorithm to solve it. Another interesting paper on this topic is [37], although it deals with a more general problem.

To conclude, the initial sorting step takes time $O(dn \log n)$, while the refining step takes time $O(n \log n)$. The result follows. □

It would be nice to have a complexity bound in terms of N, the size of the type scheme σ, rather than of d and n. However, in the worst case, both d and n can be of the order of N; which gives us a bound of $O(N^2 \log N)$.

This analysis is crude, though, because if both d and n are comparable to N, then few nodes in the graph can have degree d. In that case, the *mean* degree δ is much lower than the maximum degree. This seems to indicate that we should base our analysis on δ, rather than on d. Indeed, informally speaking, if the mean degree is δ, then a comparison between two successor or predecessor sets takes time $O(\delta)$ on average. So, the first step of the algorithm takes time $O(\delta n \log n)$. By definition of the mean degree, δn is the number of arcs in the graph $\leq_C$, so this bound is less than $O(N \log N)$. Besides, the second step of the algorithm takes time $O(n \log n)$, which is also less than $O(N \log N)$. So, we informally conclude that the computation takes time $O(N \log N)$ on average. This result is confirmed by a few practical tests, on examples of arbitrary sizes, which suggest that the algorithm's behavior is approximately linear in the size of its input.

From a practical point of view, note that the complexity of Hopcroft's algorithm, when refining with respect to k functions, is $O(k^2 n \log n)$. Thus, it is important to ensure that k is as small as possible. The proof of theorem 13.2 uses $k = 4$ functions; in fact, it is possible to use only two. Indeed, the function Low_i (for $i \in \{0, 1\}$) is defined on positive variables only, whereas Upp_i is defined on negative ones. Hence, their union $\mathrm{Low}_i \cup \mathrm{Upp}_i$ is a function. Furthermore, the initial partition $\equiv_0$ separates positive variables from negative ones. Thus, refining $\equiv_0$ with respect to Low_i and Upp_i yields the same result as refining it with respect to their union. The same idea applies again when the type system is extended. For instance, if we introduce product types, we can keep $k = 2$ functions. Indeed, the initial partition separates variables according to their bound's head constructor. Hence, during the refinement step, one can use the index 0 (resp. 1) to represent both the domain (resp. range) of function types and the first (resp. second) component of product types. Generally speaking, if the type constructors present in the graph have arity k at most, then k functions suffice.

This remark is particularly important when adding record (or variant) types. Indeed, the maximum arity k is then no longer bounded, which worsens, in theory, the algorithm's complexity. However, thanks to our remark, k equals the maximum number of labels which appear inside a single term of the type scheme at hand, rather than the total number of labels which appear in the scheme. Thus, in practice, k can be considered bounded; even though programs often use a very large number of labels, they seldom use very large records.

13.4 Examples

The reduction algorithm is quite versatile, and successfully simplifies many problems. Here are a few typical examples. (For the sake of readability, the small terms invariant isn't fully respected, and constraint sets shall be used instead of constraint graphs.)

Example (Fix-point folding). We assume here that F is a covariant type operator, distinct from the identity. (For instance, take $F : \alpha \mapsto \top \to \alpha$.) Define σ as

$$\alpha^- \to \bot \mid \{\alpha^- \leq F\,\beta^-,\, \beta^- \leq F\,\beta^-\}$$

Then, it is easy to verify that $\alpha \equiv_\sigma \beta$ holds, because both of these variables are negative, and they have equivalent bounds. So, σ can be replaced with its simplified version $\sigma/_{\equiv_\sigma}$, which is

$$\alpha^- \to \bot \mid \{\alpha^- \leq F\,\alpha^-\}$$

In essence, the algorithm just simplified a partially unfolded recursive type. To see this, recall that garbage collection throws away the lower bounds of negative variables; so, σ is equivalent to

$$\alpha \to \bot \mid \{\alpha = F\,\beta,\, \beta = F\,\beta\}$$

Here, β actually stands for the fix-point of F, and α equals $F\,\beta$, so we can—informally—rewrite σ as

$$\alpha \to \bot \mid \{\alpha = F(\mu t.F\,t)\}$$

Here, the simplification opportunity is clearly apparent. The type $F(\mu t.F\,t)$ is equal to $\mu t.F\,t$, by folding the μ binder. If we perform the simplification, we obtain

$$\alpha \to \bot \mid \{\alpha = \mu t.F\,t\}$$

and by coming back, we find that this type scheme is equivalent to

$$\alpha \to \bot \mid \{\alpha \leq F\,\alpha\}$$

which is precisely the output of the minimization algorithm.

We have shown that the effect of minimization can be seen as a one-step fix-point folding, i.e. replacing $F(\mu t.F\,t)$ with $\mu t.F\,t$. Of course, the algorithm is also able to perform a many-step folding in a single run. This is quite useful in practice. Imagine, for instance, a function which accepts a binary tree as input, looks at its first two levels (perhaps to perform some balancing operation) and then uses recursive calls to deal with the sub-trees at depth 2. The domain of the type inferred for this function shall be a recursive type (the type of binary trees), but unfolded twice, because making explicit use of the upper nodes causes a fresh type variable to be generated for each of them. So, such situations do occur quite frequently in practice, and the minimization algorithm deals with them satisfactorily.

Example. This example is similar, in principle, to the previous one, but is worth mentioning because it also corresponds to a typical practical situation. We assume here that F is a covariant, binary type operator. (For instance, if we have product and variant types, we can choose $F : (\alpha, \beta) \mapsto$ `[ Nil | Cons of` $\alpha \times \beta$ `]`.) Define σ as

$$\gamma_1^- \to \gamma_2^- \to \lambda^+ \mid \{\gamma_1^- \leq F(\alpha^-, \gamma_1^-),\, \gamma_2^- \leq F(\alpha^-, \gamma_2^-),\, F(\beta^+, \lambda^+) \leq \lambda^+,\, \alpha^- \leq \beta^+\}$$

Then, the greatest partition compatible with σ is

$$\{\{\alpha\}, \{\beta\}, \{\gamma_1, \gamma_2\}, \{\lambda\}\}$$

so the simplified type scheme $\sigma/_{\equiv_\sigma}$ is

$$\gamma^- \to \gamma^- \to \lambda^+ \mid \{\gamma^- \leq F(\alpha^-, \gamma^-),\, F(\beta^+, \lambda^+) \leq \lambda^+,\, \alpha^- \leq \beta^+\}$$

The minimization algorithm has been able to identify γ_1 and γ_2. Using an argument similar to the one developed in the previous example, we can say that both of these variables stand for the fix-point $\mu t.F(\alpha, t)$, which is why they can be identified.

Situations like this occur frequently in practice. For instance, consider a function which accepts two sorted lists and merges them. Its inferred type scheme could be σ. The minimization algorithm finds that the two arguments of the function are used in the same way, since they carry similar constraints, and thus, that they can be given the same type.

Example (Eliminating 2-crowns). Define σ as

$$\alpha^- \to \beta^- \to \gamma^+ \times \delta^+ \mid \{\alpha^- \leq \gamma^+, \alpha^- \leq \delta^+, \beta^- \leq \gamma^+, \beta^- \leq \delta^+\}$$

This situation is sometimes called a 2-*crown* in the literature, because the constraint graph looks like this:

$$\begin{tikzcd}
\gamma^+ & \delta^+ \\
\alpha^- \arrow[u] \arrow[ur] & \beta^- \arrow[u] \arrow[ul]
\end{tikzcd}$$

Similar situations are very common in practice. The scheme σ corresponds to the code

```
fun x y -> if true then (x, y) else (y, x)
```

The computation of $\equiv_\sigma$ yields

$$\{\{\alpha, \beta\}, \{\gamma, \delta\}\}$$

So, $\sigma/_{\equiv_\sigma}$ is

$$\alpha^- \to \alpha^- \to \gamma^+ \times \gamma^+ \mid \{\alpha^- \leq \gamma^+\}$$

If, furthermore, we replace each variable with its unique bound, we can go as far as

$$\alpha^\pm \to \alpha^\pm \to \alpha^\pm \times \alpha^\pm$$

But this violates the mono-polarity invariant, so it shall be done only for display purposes, not internally (see section 15.2).

This example is interesting on several grounds. First, the classic method which consists in replacing each variable with its unique bound fails here, since each variable has two bounds. (Of course, it is possible to argue that α's unique upper bound is $\gamma \sqcap \delta$, and to perform the substitution; and similarly for β. But this is not a simplification; the result is actually more complex than σ.)

Second, the fact that 2-crowns (or n-crowns, for that matter) can be efficiently eliminated is excellent news for the canonization algorithm. Recall that canonization is the process of eliminating any occurrences of $\sqcap$ and $\sqcup$ introduced by the closure process. This algorithm is run in a separate phase, after the closure phase. If the algorithm discovers the expression $\gamma \sqcap \delta$, it replaces it with a fresh variable α, and creates the constraints $\alpha \leq \gamma$ and $\alpha \leq \delta$. If a later run discovers the same expression, it will create a new variable β, and add the constraints $\beta \leq \gamma$ and $\beta \leq \delta$. A 2-crown has thus been created. Thanks to the minimization algorithm, this is not a problem: the crown shall be efficiently done away with. Repeated use of the canonization algorithm might otherwise have led to an accumulation of "crowns".

Example. Our last example clears up a detail of definition 13.3. The first condition for a partition to be compatible with σ is

- $\alpha \leq_C \beta$ implies $\forall \alpha' \equiv \alpha \quad \forall \beta' \equiv \beta \quad \alpha' \leq_C \beta'$.

However, in our informal introduction, we talked about bisimulations, and the definition of a bisimulation should rather look like this:

- $\alpha \leq_C \beta$ implies $\forall \alpha' \equiv \alpha \quad \exists \beta' \equiv \beta \quad \alpha' \leq_C \beta'$.

Here is an example to show that the latter condition is not correct. Let σ be the type scheme

$$\alpha \to \beta \to \gamma \times \delta \mid \{\alpha \leq \gamma,\, \beta \leq \delta\}$$

If we adopt this modified definition of compatibility, then the greatest compatible partition is

$$\{\{\alpha, \beta\}, \{\gamma, \delta\}\}$$

and the quotient is

$$\alpha \to \alpha \to \gamma \times \gamma \mid \{\alpha \leq \gamma\}$$

However, this scheme is clearly not equivalent to σ. The type scheme σ actually describes two separate flows of data, and although they have the same form, it is not correct to identify them.

So, the "$\forall\exists$" condition isn't correct; the "$\forall\forall$" condition is the appropriate one. Note that in Felleisen and Flanagan [18, 19], a "$\forall\exists$" appears in the case of constraints between variables; but, on the contrary, a "$\forall\forall$" condition is used in the case of constraints involving the dom destructor. Thus, the principle remains the same, but technical differences appear, because of the differences between our two formalisms.

13.5 Completeness

The above examples show that the minimization algorithm is very powerful. So, a natural question arises: is it complete, that is, is it capable of performing all correct substitutions? (Recall what has been said in section 13.1; that is, a correct substitution is a partition of the variables which respects polarities and leads to a quotient scheme equivalent to the original type scheme.)

The answer is negative. That was to be expected, since we voluntarily replaced the correctness criterion with the compatibility criterion, which is simpler and allows more efficient computations, but is also weaker. So, the algorithm is complete with respect to its specification, but the latter does not authorize all correct substitutions. Here is an example.

Example. Let F be a covariant type operator, distinct from the identity. (For instance, take $F : \alpha \mapsto \top \to \alpha$.) Let σ be the type scheme

$$\alpha^- \to \beta^- \to \gamma^+ \mid \{\alpha^- \leq F\,\alpha^-,\, \beta^- \leq F\,\beta^-,\, F\,\gamma^+ \leq \gamma^+,\, \alpha^- \leq \gamma^+\}$$

Here, α and β cannot be in the same class. If they were, then the presence of the constraint $\alpha \leq \gamma$ would require $\beta \leq \gamma$ to also be present, which is not the case.

However, the constraint $\alpha \leq \gamma$ is superfluous; that is, it is implied by the other constraints. (One can use the axiomatization of entailment, given in figure 8.2 on page 84, to verify that $\alpha \leq F\,\alpha$ and $F\,\gamma \leq \gamma$ imply $\alpha \leq \gamma$.) Consequently, σ is equivalent to

$$\alpha^- \to \beta^- \to \gamma^+ \mid \{\alpha^- \leq F\,\alpha^-,\, \beta^- \leq F\,\beta^-,\, F\,\gamma^+ \leq \gamma^+\}$$

In this new type scheme, α and β can be put into the same class; thus, it is equivalent to

$$\alpha^- \to \alpha^- \to \gamma^+ \mid \{\alpha^- \leq F\,\alpha^-,\, F\,\gamma^+ \leq \gamma^+\}$$

We can now add the constraint $\alpha \leq \gamma$ back in if we so wish; thus, we have proved that σ is equivalent to $[\beta \leftarrow \alpha]\,\sigma$.

Thus, not every correct substitution is compatible. The problem, in the above example, comes from the fact that the constraint $\beta \leq \gamma$ is not explicitly present in the graph, even though it can be derived from other constraints.

We could give a more powerful definition of compatibility by modifying the first condition of definition 13.3. It would suffice to redefine the predecessor and successor sets using entailment:

$$\begin{aligned} \mathrm{pred}_C(\alpha) &= \{\beta\,;\, C \Vdash \beta \leq \alpha\} \\ \mathrm{succ}_C(\alpha) &= \{\beta\,;\, C \Vdash \alpha \leq \beta\} \end{aligned}$$

With this modification, the notion of compatibility might coincide with that of correctness. However, this would have little practical interest, for two reasons. First, we could only compute an approximation of predecessor and successor sets, since no complete entailment algorithm is known. The minimization algorithm would thus become incomplete with respect to its specification. Second, the incomplete entailment algorithm is rather costly, so performance would probably suffer significantly, for a minimal gain of power.

In practice, one could dedicate, every once in a while (for instance, after each toplevel phrase), a phase to eliminating superfluous constraints, such as $\alpha \leq \gamma$ in the previous example. Such constraints are detected using the entailment algorithm. We have implemented this phase, yielding a gain of a few percent, both in efficiency and in result.

Part III

Discussion

Chapter 14

Extensions

One might rightfully think that our set of ground types, built using only $\bot$, $\top$ and $\rightarrow$, is remarkably poor. One shall even notice that, in such a restricted model, any constraint graph admits a solution: the substitution which maps any variable to the regular tree $\mu t.t \rightarrow t$. (This property is in accordance with the fact, mentioned in the proof of proposition 5.10, that there can be no execution errors in pure λ-calculus.)

So, several of our statements have trivial proofs, if one takes advantage of the fact that $\rightarrow$ is the only type constructor beside $\bot$ and $\top$. However, our proofs don't use this property, and thus are easily extensible to much richer type systems. We could have chosen such a system and explicitly dealt with it, or tried to parameterize the whole theory by an arbitrary ground lattice, satisfying a few requirements. However, it seemed to us that doing so would significantly increase the apparent complexity of many proofs, even though no intrinsic complexity was to be added. So, we chose to present our theory in the simplest possible setting.

Besides, one can easily conceive other extensions, which we refer to as *orthogonal*, because their interaction with subtyping is limited or void. These extensions are mostly independent from our theory, which is why we did not mention them up to here. They do considerably augment our system's power, though, so they deserve to be studied here.

During this chapter, we first give a rather formal idea of how we could have parameterized our theory (section 14.1). In the following sections, we delve into a few particular cases of it, which correspond to classic notions: base types (section 14.2), isolated n-ary type constructors (section 14.3), then polymorphic record and variant types (section 14.4). Next, we present two orthogonal extensions: the addition of row variables (section 14.5) and of an exception analysis (section 14.6). Row variables might seem, at first sight, expressive enough to make subtyping itself superfluous; we discuss this topic in section 14.7.

14.1 Parameterizing the theory

As mentioned above, we could have parameterized our theory by the choice of the language of ground types. To that end, we would have defined the ground lattice in a more abstract way; instead of fully specifying it, we would only have required a series of conditions sufficient to build our theory. We avoided this approach, so as not to hide the fundamental ideas of this work behind another abstraction layer. However, it is interesting to know what these sufficient conditions would be, so as to measure the generality of our approach. So, this section defines a notion of *ground signature*, which partially specifies the ground lattice. It then revisits the results of chapter 1 and parameterizes them by an arbitrary ground signature.

Definition 14.1 *Let $\mathcal{L}$ be a denumerable set of* labels. *An* arity *is a finite set of labels. A* ground signature Σ_g *consists of:*

- *a set of* constructors c*, each equipped with an arity* $a(c)$*;*
- *a* variance *function* v*, which maps any label to an element of* $\{-, +\}$*;*
- *an ordering relation* $\leq_g$ *on constructors, such that*
  - *the set of constructors, equipped with this order, forms a lattice;*
  - *if* $c_1 \leq_g c_2 \leq_g c_3$*, and if* $l \in a(c_1) \cap a(c_3)$*, then* $l \in a(c_2)$ *(one might call this condition "convexity of arity");*
  - *the constructors* $\bot$ *and* $\top$ *are constant, i.e. of arity* $\varnothing$*.*

We then define the set of ground trees, given a signature Σ_g, as follows.

Definition 14.2 *A* path *is an element of* $\mathcal{L}^*$*. The* parity *of a path is the number of elements of negative variance it contains, modulo 2. A* ground tree τ *is a partial function from paths into* Σ_g*, whose domain is non-empty and prefix-closed, and such that for all* $l \in \mathcal{L}$*,* $\tau(pl)$ *is defined iff* $l \in a(\tau(p))$*.*

The definition of subtyping is unchanged (see definition 1.4). It is easy to verify that it is an ordering; reflexivity is immediate, antisymmetry is proved as in proposition 1.4, and transitivity is easily obtained by using the convexity of arity property defined above.

Note that this convexity property is necessary. Indeed, assume it is violated for some c_1, c_2, c_3 and l. Then, we have $c_1(\top) \leq c_2$ and $c_2 \leq c_3(\bot)$, but not $c_1(\top) \leq c_3(\bot)$, because that would entail $\top \leq \bot$. (The latter might hold if the lattice contains only one point, but then c_1, c_2 and c_3 are identical and the property cannot be broken.) Thus, the relation $\leq$ isn't transitive. (We have made a slight abuse of language here, since we have used the constructors c_1, c_2 and c_3 without specifying which value was associated to labels other than l; any fixed value will do.)

Then, one verifies that the subtyping relation defines a lattice. The definition of the $\sqcup$ and $\sqcap$ operations is similar to the one given in the proof of proposition 1.4, and the proof itself is unchanged. $\sqcup$ and $\sqcap$ are characterized by the following equations:

$$\begin{array}{rl} & (\tau_1 \sqcup \tau_2)(\epsilon) = \tau_1(\epsilon) \sqcup_g \tau_2(\epsilon) \\ \forall l \in a((\tau_1 \sqcup \tau_2)(\epsilon)) & (\tau_1 \sqcup \tau_2)(l) = \tau_1(l) \sqcup^{v(l)} \tau_2(l) \end{array}$$

(These two equations deal with $\sqcup$; they have to be accompanied by two symmetrical ones concerning $\sqcap$.) In the second equation above, $\sqcup^+$ stands for $\sqcup$ and $\sqcup^-$ stands for $\sqcap$. Besides, $\tau_1(l)$ and $\tau_2(l)$ are possibly undefined; they must then be read as the neutral element of $\sqcup^{v(l)}$.

These equations state that $\sqcup$ and $\sqcap$ are distributive operations. They are computed by first performing an elementary operation (defined by the ground signature) on the head constructors, and then distributing the operations (while respecting variance) onto the subterms.

The whole theory can be recreated on this new basis. (We haven't specified how to extend the typing rules, so chapter 5 isn't involved; but most of the theory is built purely within the universe of types and type schemes, and thus can be extended.) The only reason why we do not prove this claim is, we favored simplicity throughout our theoretical presentation. Anyway, it is backed by practical experience, since our prototype implements a very rich type language, making liberal use of the possibilities offered by definition 14.1.

By the way, notice that since the bulk of the theory can be parameterized by a ground signature, so can the implementation. (Except the constraint generator, though, since it implements the type inference rules, which explicitly refer to a known signature.) The algorithms which handle and simplify constraints can easily be written as functors, parameterized by this signature. As a result, the code is made significantly simpler, since the different flavors of types (function types, product types, record types, etc.) are now dealt with in a uniform way. However, there is a performance penalty, since the various particular cases can no longer be optimized. In practice, we have adopted this abstract implementation; we have measured an increase of roughly 10% in the execution time. (Record and variant types are still dealt with separately, though, because of the presence of row variables—see section 14.5.) This penalty is acceptable, considering the increase in readability and generality of the code.

Finally, we haven't yet explained why $\bot$ and $\top$ are required to be constant constructors. If this were not true, then the smallest, or the greatest, element of the ground lattice would be an infinite term. (For instance, if $\top$ is unary, the greatest ground type is $\mu t.\top(t)$.) This would create rather tricky problems, both theoretically and practically, because infinite ground types cannot be expressed in our type language. This condition does not seem very restrictive, so we adopt it.

14.2 Base types

Most programming languages supply elementary data types such as integers, reals, booleans, etc. Introducing these base types is a straightforward application of the previously described extension mechanism. In the simplest case, we can define some base types as isolated, constant constructors:

`bool` `unit`

However, we can also define subtyping relationships between base types, as shown by these classic examples:

$$\begin{tikzcd} \texttt{float} & & \texttt{string} \\ \texttt{int} \arrow[u] & & \texttt{char} \arrow[u] \end{tikzcd}$$

Generally speaking, it is possible to define an arbitrary complex partial order on base types, as long as the ground signature Σ_g remains a lattice. For instance, one might imagine the following situation:

$$\begin{tikzcd} & \texttt{rat} & \\ \texttt{int} \arrow[ur] & & \texttt{posrat} \arrow[ul] \\ & \texttt{posint} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

In this slightly unusual example, `posint` represents non-negative integers, `int` represents integers, `posrat` represents non-negative rationals, and `rat` represents rationals.

14.3 Isolated n-ary constructors

A second possible extension, still within the framework defined by section 14.1, consists in adding an isolated n-ary constructor to the language. Such a constructor is said to be isolated because it is incompatible, subtyping-wise, with any other constructor (except of course $\bot$ and $\top$). The pair constructor $*$, and the $\rightarrow$ constructor itself, are examples of isolated binary constructors.

We are limited, when defining such a constructor c, by a condition coming from definition 14.1: each of its n arguments must be either *covariant*, or *contravariant*; it must not be *invariant*. Let us quickly recall the meaning of these terms.

Formally speaking, a label $l \in \mathcal{L}$ is covariant if it is mapped to $+$ by the arity function $a(c)$, and it is contravariant if it is mapped to $-$. More intuitively, an argument is contravariant if it reverses the subtyping relation during structural decomposition, and covariant otherwise. For instance, the first argument of the $\rightarrow$ constructor is contravariant because $\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'$ implies $\tau_1' \leq \tau_1$, rather than $\tau_1 \leq \tau_1'$; the second argument is covariant.

An argument is said to be invariant if it is both covariant and contravariant, that is, if an inequation between constructed types implies equality between their sons. Notice that definition 14.1 does not allow invariant arguments. Indeed, the invariance phenomenon causes problems, for several reasons. First, the lattice operations are not distributive over an invariant argument. This prevents us from pushing the $\sqcup$ and $\sqcap$ constructors down to variables, and thus from doing canonization. Besides, enforcing the mono-polarity invariant becomes impossible, because if a variable appears as an invariant argument, then it shall be either neutral, or bipolar.

For instance, consider the case of reference types. Many functional languages offer a unary type constructor called `ref`: τ `ref` is the type of reference cells whose content is of type τ. Since these cells can be read and written, τ `ref` $\leq \tau'$ `ref` must imply $\tau = \tau'$, if the language is to be type-safe. As a consequence, (τ `ref`) $\sqcup$ (τ' `ref`) equals (τ `ref`) if $\tau = \tau'$, and $\top$ otherwise. Hence, $\sqcup$ is not distributive over `ref`.

To work around this problem, we are led to make `ref` a binary type constructor: (τ_0, τ_1) `ref` shall be the type of reference cells into which data of type τ_0 can be written and out of which data of type τ_1 can be read. (This idea was suggested to us by Luca Cardelli, and independently discovered by Smith and Trifonov.) With this convention, it is safe to define (τ_0, τ_1) `ref` $\leq (\tau_0', \tau_1')$ `ref` as equivalent to $\tau_0' \leq \tau_0 \wedge \tau_1 \leq \tau_1'$. That is, `ref` is contravariant in its first argument and covariant in its second argument. Thus, it behaves exactly as the $\rightarrow$ constructor. The three primitives used to create, read and write ref cells have the following type schemes:

$$\begin{aligned}
\texttt{ref} &: \alpha \rightarrow (\alpha, \alpha)\ \texttt{ref} \\
\texttt{!} &: (\alpha, \beta)\ \texttt{ref} \rightarrow \beta \\
\texttt{:=} &: (\alpha, \beta)\ \texttt{ref} \rightarrow \alpha \rightarrow \texttt{unit}
\end{aligned}$$

Thus, historically, the idea of making the `ref` constructor binary appeared to overcome a technical problem. However, it is fundamentally in agreement with the philosophy of subtyping. Indeed, subtyping essentially allows keeping track of the direction of data flow, while using equality leads to a non-directed, and thus coarser, flow graph. Using a unary `ref` constructor amounts to confusing reading and writing operations; hence, it necessarily implies a loss of precision. For instance, consider the program

```
(fun x -> x := No; !x) (ref Yes)
```

(`Yes` and `No` are data constructors; they shall be introduced, along with polymorphic variant types, in section 14.4.) If a unary `ref` constructor is used, then this program is ill-typed, even in the presence of subtyping. Indeed, `ref Yes` receives type `[ Yes ] ref`, and the instruction `x := No` creates the insolvable constraint `[ No ]` $\leq$ `[ Yes ]`. On the contrary, if a binary `ref` constructor is chosen, then read and write operations are correctly distinguished, and the expression's type is `[ Yes ]` $\sqcup$ `[ No ]`, i.e. `[ Yes | No ]`. In conclusion, using a binary `ref` constructor offers more precision, and appears natural in a subtyping framework.

14.4 Polymorphic records and variants

The extensions mentioned so far don't allow a very advanced usage of subtyping. The presence of $\bot$ allows proving, through the typing system, that some expressions do not terminate, and using them in any context. The presence of $\top$ allows handling objects of unknown type. From a practical point of view, these are of little use. Subtyping between base types can be useful, but it alone does not warrant the development of such a powerful theory. Things become more interesting when polymorphic records or variants are added. This extension is also a particular case of the theory sketched in section 14.1.

Assume given a denumerable set of labels $l \in \mathcal{L}$. A record type is of the form

$$\texttt{\{}\ l\texttt{:}\ \tau_l\ \texttt{\}}_{l \in L}$$

where L is a finite subset of $\mathcal{L}$. The subtyping relation between record types is defined by

$$\texttt{\{}\ l\texttt{:}\ \tau_l\ \texttt{\}}_{l \in L} \leq \texttt{\{}\ l\texttt{:}\ \tau'_l\ \texttt{\}}_{l \in L'} \iff \forall l \in L' \quad (l \in L \wedge \tau_l \leq \tau'_l)$$

Thus, if we hide some of the fields of a given record type, we obtain a super-type of it. In other words, subtyping allows forgetting the presence of some fields; wherever a certain record type is expected, a record containing more information than necessary can be supplied. This is why these types are called *polymorphic*: a given record value has many different types, depending on whether each field is shown or hidden.

For instance, we have

$$\texttt{\{ a: }\alpha\texttt{; b: }\beta\texttt{ \}} \leq \texttt{\{ a: }\alpha\texttt{ \}}$$

This makes the following expression valid, even though the supplied argument has more fields than the function expects:

```
(fun x -> x.a) { a = 0; b = true }
```

Polymorphic record types can be used as a basis for various encodings of object-oriented programming into a functional language. Sending a given message to an object shall typically involve accessing a given field in a record. Since the latter operation can be applied to records containing arbitrary other fields, a message defined by a certain class can then be sent to objects belonging to its sub-classes.

This extension is an application of the parameterization presented at the beginning of this chapter. The constructors are the $\texttt{\{\}}_L$, where L varies among finite subsets of $\mathcal{L}$. The constructor $\texttt{\{\}}_L$ has arity L, and all labels $l \in \mathcal{L}$ are covariant. The order on constructors is the reverse of the inclusion order on arities: $\texttt{\{\}}_L \leq_g \texttt{\{\}}_M$ holds if and only if $L \supset M$. Hence, the ground signature is indeed a lattice. Lastly, the convexity property is satisfied; actually, arity is, in this case, a non-increasing function.

Polymorphic variant types are entirely symmetrical. They are of the form

$$\texttt{[}\ l\texttt{:}\ \tau_l\ \texttt{]}_{l \in L}$$

The subtyping relation between them is given by

$$\texttt{[}\ l\texttt{:}\ \tau_l\ \texttt{]}_{l \in L} \leq \texttt{[}\ l\texttt{:}\ \tau'_l\ \texttt{]}_{l \in L'} \iff \forall l \in L \quad (l \in L' \wedge \tau_l \leq \tau'_l)$$

This states that adding new cases to a given variant type yields a super-type of it. Thus, a constructed value can be passed to a function which handles several cases. For instance,

$$\texttt{[None: unit]} \leq \texttt{[None: unit | Some: }\alpha\texttt{]}$$

This makes the following expression valid, even though the function's domain mentions several cases, while the type of its actual argument only specifies one:

```
(fun f None     -> ()
   | f (Some x) -> f x) print_int (Some 3)
```

Note that all this works without requiring the user to supply any type information. A single label can thus appear in several different record types, and a single data constructor can be part of several different variant types. This lends great flexibility to the language.

14.5 Extensible records and variants

Polymorphic record types, as presented in section 14.4, have relatively limited power. They allow ignoring the presence of certain fields in a record, but not referring to these unknown fields, even if only to copy them as a whole.

Because of this, no satisfactory type can be given to the extension functions. (The extension function for a field l accepts a record r and a value v, and returns a record identical to r, except that the field l has been created if necessary, and has taken value v.) It is impossible to express the fact that the types of the fields other than l are unchanged. If one considers the particular case where the field l is assumed to exist already and to have the same type, then one might naively give the extension function the type scheme

$$\alpha \rightarrow \beta \rightarrow \alpha \mid \{\alpha \leq \texttt{\{}\ l\texttt{:}\ \ \beta\ \texttt{\}}\}$$

However, this is incorrect! Indeed, since β is negative, it can be replaced with its upper bound without affecting the scheme's denotation:

$$\alpha \rightarrow \top \rightarrow \alpha \mid \{\alpha \leq \texttt{\{}\ l\texttt{:}\ \ \top\ \texttt{\}}\}$$

Here, the function's second argument can have any type, so it is incorrect to return α as a result. The problem stems from the fact that any two values have a common type, namely $\top$. Hence, requiring two quantities to have "the same type" has no effect.

14.5.1 Description

To solve this problem, Rémy [43] suggests introducing row variables. We shall describe them only informally, since they form an independent theory, which interacts very simply with subtyping. It is detailed in [43]. Thus, this is not a novelty, and the notion of row variable does not depend at all on subtyping; but it is interesting to remark that these notions integrate without difficulty.

First, one gives a more elaborate definition of record types, which involves *row terms* θ:

$$\begin{aligned}\tau &::= \ldots \mid \texttt{\{}\ l_1\texttt{:}\ \theta_1\texttt{;}\ \ \ldots\texttt{;}\ \ l_n\texttt{:}\ \theta_n\texttt{;}\ \theta\ \texttt{\}} \\ \theta &::= \rho \mid \texttt{Abs} \mid \texttt{Pre}\ \tau\end{aligned}$$

A row term provides a piece of information about a field. Its value can be `Abs`, to indicate the absence of this field, or `Pre` τ to indicate that the field is present and contains a value of type τ. It can also be a *row variable* ρ, when this information is unknown. In a record type, a row term is associated to each field. Furthermore, at the last position comes a row term without any label, which stands for the (infinite) set of all fields which are not explicitly mentioned.

So, any record type has, in a way, an infinite number of fields, but only a finite number of them are explicitly named. The subtyping relation on record types is then defined field-wise. To allow ignoring certain fields, one sets $\texttt{Pre}\ \tau \leq \texttt{Abs}$ for any τ. Besides, the `Pre` constructor is of course covariant. The assertion

$$\texttt{\{ a: Pre}\ \alpha\texttt{; b: Pre}\ \beta\texttt{;}\ \rho_1\ \texttt{\}} \leq \texttt{\{ a: Abs; b: Pre}\ \gamma\texttt{;}\ \rho_2\ \texttt{\}}$$

is thus equivalent to $\beta \leq \gamma \wedge \rho_1 \leq \rho_2$. Note that row variables, like regular ones, can receive constraints.

It is then possible to give the extension function on field l a satisfactory type scheme, namely

$$\texttt{\{}\ l\texttt{: Abs;}\ \rho\ \texttt{\}} \to \alpha \to \texttt{\{}\ l\texttt{: Pre}\ \alpha\texttt{;}\ \rho\ \texttt{\}}$$

This type indicates that the field l can be absent from the initial record r. Thanks to the subtyping mechanism, the field l can *a fortiori* be present, with any type. In the record returned by the function, the field l appears with type α, which is the type of the newly supplied value, and all other fields, represented by the row variable ρ, are unchanged.

The most delicate feature of row variables is *demand-driven expansion*. Imagine, for instance, that the following constraint appears during the inference process:

$$\texttt{\{ a: Pre}\ \alpha\texttt{; b: Pre}\ \beta\texttt{;}\ \rho\ \texttt{\}} \leq \texttt{\{ a: Pre}\ \alpha'\texttt{;}\ \rho'\ \texttt{\}}$$

The closure algorithm must decompose this constraint. As far as the field `a` is concerned, one obtains $\texttt{Pre}\ \alpha \leq \texttt{Pre}\ \alpha'$, then $\alpha \leq \alpha'$. But how to deal with `b`? In the right-hand record type, the information about this field is contained in the variable ρ', which describes all fields not named explicitly. To "extract" this information, one *expands* the variable ρ', that is, one replaces all occurrences of ρ' with $\langle\texttt{b:}\ \rho'_1\texttt{;}\ \rho'_2\rangle$, where ρ'_1 and ρ'_2 are fresh variables. The above constraint thus becomes

$$\texttt{\{ a: Pre}\ \alpha\texttt{; b: Pre}\ \beta\texttt{;}\ \rho\ \texttt{\}} \leq \texttt{\{ a: Pre}\ \alpha'\texttt{; b:}\ \rho'_1\texttt{;}\ \rho'_2\ \texttt{\}}$$

which can be decomposed into $\alpha \leq \alpha' \wedge \texttt{Pre}\ \beta \leq \rho'_1 \wedge \rho \leq \rho'_2$. Note that the variable ρ' could appear not only in record types, as above, but also in constraints, e.g. $\rho' \leq \rho''$. ρ''s expansion then causes ρ'' to be expanded as well; all of ρ'''s occurrences shall be replaced with $\langle\texttt{b:}\ \rho''_1\texttt{;}\ \rho''_2\rangle$. The constraint $\rho' \leq \rho''$ can then be decomposed, and replaced with $\rho'_1 \leq \rho''_1 \wedge \rho'_2 \leq \rho''_2$. So, the expansion mechanism must be integrated with the closure algorithm, which poses no practical problem.

It is possible to build ill-formed terms, such as $\texttt{\{ a: Pre}\ \alpha\texttt{;}\ \rho\ \texttt{\}} \to \texttt{\{}\ \rho\ \texttt{\}}$, where the right-hand record type contains no information about the field `a`. To prohibit them, one sets up a very simple system of *kinds*. User-supplied types are rejected if ill-kinded. The types built by the typing rules are necessarily well-kinded. Note that all of the simplifications we described naturally preserve this property, except minimization; so, we have to explicitly require that two variables have the same kind in order to be merged.

14.5.2 Row variables and variant types

We have described the use of row variables in record types. They are also introduced, symmetrically, in variant types. Row terms are made up of row variables and of two constructors, which we shall also denote by `Pre` et `Abs`, by abuse of language. They behave similarly to their record counterparts, but the subtyping relationship is $\texttt{Abs} \leq \texttt{Pre}\ \tau$. This allows adding new cases to a variant type:

$$\texttt{[Nil: Pre unit; Abs]} \leq \texttt{[Nil: Pre unit; Cons: Pre}\ \alpha \times \beta\texttt{; Abs]}$$

The counterpart, in variants, of the record extension function is the elementary matching function on a data constructor A, named $m_{\texttt{A}}$, whose semantics is informally given by the following code:

```
fun yes no v -> match v with
  A x -> yes x
| other -> no other
```

It can be given type

$$(\alpha \to \beta) \to (\texttt{[A: Abs; } \rho \texttt{]} \to \beta) \to \texttt{[A: Pre } \alpha\texttt{; } \rho \texttt{]} \to \beta$$

which is reminiscent of the record extension function's type. The function `no` is invoked if the value `v` is not built using the constructor `A`. Thus, it does not have to handle this case, and it is only required to have type `[ A: Abs;` ρ `]` $\to \beta$. The cases which it is able to handle are represented by the row variable ρ, which is found again in the type allowed for `v`. Thus, just as row variables allow typing record extension, they allow typing pattern matching extension.

Let us give a few indications about the way pattern matchings are typed. The elementary pattern matching functions are primitive, that is, they are added to the language as constants. The type of these constants is provided by δ-rules. That is, the constant m_K, which represents the elementary matching function for the constructor K, shall admit, by definition, the type scheme

$$(\alpha \to \beta) \to (\texttt{[} K\texttt{: Abs; } \rho \texttt{]} \to \beta) \to \texttt{[} K\texttt{: Pre } \alpha\texttt{; } \rho \texttt{]} \to \beta$$

One also provides a primitive `reject`, of type

$$\texttt{[Abs]} \to \bot$$

Then, any simple pattern matching, i.e. any matching corresponding to a single n-ary switch, can easily be compiled using these primitives. For instance, the expression

```
fun (A x) -> x
```

can be encoded as

$$m_{\texttt{A}}\,(\lambda x.x)\,\texttt{reject}$$

which has the expected type, namely

$$\texttt{[A: Pre } \alpha\texttt{; Abs]} \to \alpha$$

The `reject` primitive is a function which accepts no argument; it is used to "close" the matching. The row term `Abs` in the above type comes from its type. Its use is not necessary when the matching ends with an irrefutable clause, as in

```
fun (A x) -> x
  | (B y) -> y
  | z -> 0
```

Indeed, this matching is encoded by

$$m_{\texttt{A}}\,(\lambda x.x)\,(m_{\texttt{B}}\,(\lambda y.y)\,(\lambda z.z))$$

which has type

$$\texttt{[A: Pre } \alpha\texttt{; B: Pre } \alpha\texttt{; Pre } \top \texttt{]} \to \alpha \mid \{\texttt{int} \leq \alpha\}$$

Complex matchings, where constructors can appear at an arbitrary depth, can also be compiled using these elementary constants. So, to type these matchings, one first compiles them into an expression which is then typed. This method has the advantage of simplicity: the theory is unchanged, and it is clear that the obtained type is correct. In particular, the notion of non-exhaustive pattern matching disappears; any function is guaranteed to be able to handle all arguments specified by its type. (Compare with the case of ML, where a separate analysis is necessary.) The drawback of this method is the fact that the obtained type depends on the way pattern matchings are compiled, which makes it *a priori* unpredictable. For instance, in our implementation, the function

```
fun (A, _) -> 0
  | (_, B) -> 0
```

receives type

$$\texttt{[A: Pre unit; Pre } \top \texttt{]} \times \texttt{[B: Pre unit; Abs]} \rightarrow \texttt{int}$$

This type requires the second component of the argument to be `B`, thus lessening the function's power and breaking symmetry. However, one can argue that this problem is unavoidable, since a precise description of this function would require preserving the correlation between the two components, that is, expressing the fact that either the former is `A`, or the latter is `B`, which is beyond the abilities of our type system. If the function is rewritten in a more explicit and more restrictive way, such as

```
fun (A, (A|B)) -> 0
  | ((A|B), B) -> 0
```

then we obtain the expected type $\tau \times \tau \rightarrow \texttt{int}$, where τ equals

```
[ A: Pre unit; B: Pre unit; Abs ]
```

This method of typing pattern matchings works, in practice, in many cases. However, it is not entirely satisfactory, since it reflects the way pattern matchings are compiled, which makes it dissymmetric and unpredictable. It should be interesting to give explicit typing rules for complex matchings, rather than use an implicit compilation phase, but that appears to be very delicate.

14.6 Exception analysis

In ML, an expression's evaluation can end in two ways: either the expression returns a result, or it raises an *exception*. From a logical point of view, results and exceptions could be presented at the same level: they are simply two distinct ways for a function to transmit a value to its caller. However, in ML's type system, they are handled very differently: the full power of the system is used to analyze the types of the results, while all exceptions receive type `exc` and are handled in a purely monomorphic way. However, with a sufficiently expressive type system, it is possible to restore symmetry, and to analyze results and exceptions with the same precision.

14.6.1 Description

Let us first recall the functioning of exceptions. The expression language is extended with two new constructs:

$$e ::= \ldots \mid \texttt{raise} \mid \texttt{try } e \texttt{ with } e$$

The constant `raise` allows raising an exception; it is used as a function of one argument, which is the value of the exception to be raised. The construct $\texttt{try}\ e_1\ \texttt{with}\ e_2$ evaluates the expression e_1; if an exception occurs, then it is passed as argument to the handler e_2. This construct is more concise, and more powerful, than ML's notation. The ML expression

$$\texttt{try}\ e_1\ \texttt{with A}\ x\ \texttt{->}\ e_2$$

would here be written

$$\texttt{try}\ e_1\ \texttt{with}\ m_{\texttt{A}}\,(\lambda x.e_2)\,(\lambda e.\texttt{raise}\ e)$$

The operational semantics is also extended straightforwardly to account for exceptions. Rather than giving the details of these modifications, we prefer to consider the following encoding:

$$\begin{aligned}
[\![x]\!] &= \texttt{Val}\ x \\
[\![\lambda x.a]\!] &= \texttt{Val}\ (\lambda x.[\![a]\!]) \\
[\![a_1\,a_2]\!] &= m_{\texttt{Val}}\,(\lambda v_1.m_{\texttt{Val}}\,(\lambda v_2.v_1\,v_2)\,(\lambda e.e)\,[\![a_2]\!])\,(\lambda e.e)\,[\![a_1]\!] \\
[\![X]\!] &= \texttt{Val}\ X \\
[\![\texttt{let}\,X = a_1\ \texttt{in}\ a_2]\!] &= m_{\texttt{Val}}\,(\lambda v_1.\texttt{let}\,X = v_1\ \texttt{in}\ [\![a_2]\!])\,(\lambda e.e)\,[\![a_1]\!] \\
[\![\texttt{raise}]\!] &= \texttt{Val}\ (\lambda x.\texttt{Exc}\ x) \\
[\![\texttt{try}\ a_1\ \texttt{with}\ a_2]\!] &= m_{\texttt{Exc}}\,(\lambda e_1.m_{\texttt{Val}}\,(\lambda v_2.v_2\,e_1)\,(\lambda e.e)\,[\![a_2]\!])\,(\lambda v.v)\,[\![a_1]\!]
\end{aligned}$$

This encoding, defined by induction on the structure of expressions, eliminates the newly introduced constructs `raise` and $\texttt{try}\ e_1\ \texttt{with}\ e_2$. The expression $[\![a]\!]$ can thus be evaluated using the initial semantics. One verifies that its result is $\texttt{Val}\ v$ iff the expression a returns the result v, and $\texttt{Exc}\ e$ iff the expression a raises the exception e. This encoding is thus a way of defining the semantics of exceptions.

The point of this encoding is to restore the symmetry between results and exceptions. In both cases, they are values computed by the expression; the only distinction stems from the constructor used, that is, `Val` or `Exc`. The existence of this encoding shows that typing exceptions is no more difficult than typing results. Indeed, it suffices to type $[\![a]\!]$ to obtain not only a's type, but also the type of the exceptions potentially raised by a.

However, in practice, rather than explicitly using this encoding, we prefer to introduce a set of modified typing rules, given by figure 14.1 on the next page. The principle remains the same; the use of dedicated rules prevents mixing types generated by the encoding with types coming from the user program, and thus enhances readability.

The $\rightarrow$ constructor is now ternary; it is written $\alpha \rightarrow \beta\ \texttt{raises}\ \gamma$, where the first argument is contravariant and the last two are covariant, and where γ represents the type of exceptions potentially raised by the function. (In the encoding, the function would have had type $\alpha \rightarrow \texttt{[Val: Pre}\ \beta\texttt{; Exc: Pre}\ \gamma\texttt{; Abs]}$.) In keeping with this idea, type schemes are now of the form $A \Rightarrow \tau\ \texttt{raises}\ \tau_e \mid C$, where τ_e is the type of the exceptions possibly raised by the expression. In both cases, an annotation $\texttt{raises}\ \bot$ means that the function or the expression at hand raises no exceptions, and can be omitted for the sake of brevity.

That said, the modifications made to the rules are simple, and reflect the constraints that would be obtained by typing the expression produced by the encoding. Note the appearance of a new rule (Try). As for the `raise` construct, no dedicated rule is necessary; it suffices to make it a primitive, whose type scheme is $(\alpha \rightarrow \bot\ \texttt{raises}\ \alpha)\ \texttt{raises}\ \bot$.

For the sake of clarity, we show only the rewritten typing rules. The type inference rules can also be modified without difficulty; one shall be careful to preserve the mono-polarity invariant by introducing, wherever necessary, two fresh variables linked by a constraint, rather than a single variable (see section 12.3).

$$\frac{\mathrm{dom}(A) = \mathrm{dom}_\lambda(\Gamma)}{\Gamma \vdash x : A \Rightarrow A(x)\ \texttt{raises}\ \bot \mid C} \quad (\textsc{Var})$$

$$\frac{\mathrm{lift}_x(\Gamma); x \vdash e : (A; x : \tau) \Rightarrow \tau'\ \texttt{raises}\ \tau_e \mid C}{\Gamma \vdash \lambda x.e : A \Rightarrow (\tau \to \tau'\ \texttt{raises}\ \tau_e)\ \texttt{raises}\ \bot \mid C} \quad (\textsc{Abs})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A \Rightarrow (\tau_2 \to \tau\ \texttt{raises}\ \tau_e)\ \texttt{raises}\ \tau_e \mid C \\ \Gamma \vdash e_2 : A \Rightarrow \tau_2\ \texttt{raises}\ \tau_e \mid C\end{array}}{\Gamma \vdash e_1\, e_2 : A \Rightarrow \tau\ \texttt{raises}\ \tau_e \mid C} \quad (\textsc{App})$$

$$\frac{\Gamma(X) = \sigma}{\Gamma \vdash X : \sigma} \quad (\textsc{LetVar})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma_1 \\ \sigma_1 = A_1 \Rightarrow \tau_1\ \texttt{raises}\ \tau_e' \mid C_1 \\ \sigma_1' = A_1 \Rightarrow \tau_1\ \texttt{raises}\ \bot \mid C_1 \\ \Gamma; X : \sigma_1' \vdash e_2 : A_2 \Rightarrow \tau_2\ \texttt{raises}\ \tau_e \mid C_2 \\ \sigma_1 \leq^\forall A_2 \Rightarrow \top\ \texttt{raises}\ \tau_e \mid C_2\end{array}}{\Gamma \vdash \texttt{let}\, X = e_1\, \texttt{in}\ e_2 : A_2 \Rightarrow \tau_2\ \texttt{raises}\ \tau_e \mid C_2} \quad (\textsc{Let})$$

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \leq^\forall \sigma'}{\Gamma \vdash e : \sigma'} \quad (\textsc{Sub})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A \Rightarrow \tau\ \texttt{raises}\ \tau_e \mid C \\ \Gamma \vdash e_2 : A \Rightarrow (\tau_e \to \tau\ \texttt{raises}\ \tau_e')\ \texttt{raises}\ \tau_e' \mid C\end{array}}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ e_2 : A \Rightarrow \tau\ \texttt{raises}\ \tau_e' \mid C} \quad (\textsc{Try})$$

Figure 14.1: Typing rules, extended for the exception analysis

14.6.2 Application

Thanks to the row variables mechanism (see section 14.5) in variant types, pattern matchings receive precise types. But exception handlers are often—actually, always, in ML—based on pattern matchings. So, the exception analysis shall be sufficiently fine. For instance, the handler

```
fun (A x) -> x
  | e -> raise e
```

receives type

$$\texttt{[A: Pre } \alpha\texttt{; } \rho \texttt{]} \rightarrow \alpha \texttt{ raises [A: Abs; } \rho \texttt{]}$$

This type is expressive enough to indicate that the exception e thrown by this function cannot be of the form `A`.

In fact, the above type is a way of encoding *subtraction* of the element `A` from a set of constructors. Indeed, let τ be a variant type. τ can be considered as a way of encoding a set of constructors. Then, consider the constraint $\tau \leq \texttt{[A: Pre } \alpha\texttt{; } \rho \texttt{]}$. If this constraint is satisfied, then $\texttt{[A: Abs; } \rho \texttt{]}$ is a variant type identical to τ (provided ρ is chosen minimal), but deprived of the constructor `A`. Hence, it represents the same set of constructors as τ, minus the constructor `A`. Thus, in a way, we have encoded into types an operation which is expressed in terms of sets in other analyses [24].

This explains why the following function:

```
function x ->
  try
    raise (if true then (A x) else (B x))
  with A x ->
    x
```

has type

$$\alpha \rightarrow \alpha \texttt{ raises [B: Pre } \alpha\texttt{; Abs]}$$

The exception handler used here is precisely the one described above. The subtraction operation indeed occurs: even though the heart of the function raises exceptions `A` and `B`, the type correctly tells that only `B` can be observed from the outside. Besides, note the use of polymorphism: the result type of this function is equal to its argument type, even though the data flows internally through an exception. This flexibility would not be possible in ML, where arguments of exceptions are necessarily monomorphic.

14.7 Subtyping vs. row variables

The reader might feel that the above exception analysis, or more generally, the way we type pattern matchings, owe their precision mainly to the presence of row variables, and not to the use of subtyping. This impression is not entirely unjustified: row variables are a very interesting refinement, which produces precise results, even when only unification constraints are used. However, this precision depends on a sufficient degree of polymorphism, which isn't always available in a language where polymorphism is restricted to first order. Subtyping, on the contrary, works perfectly in the absence of any polymorphism. Let us detail this comparison.

Recall that the main novelty of subtyping consists in using the *direction* of the data flow, thus creating a *directed* constraint graph; while a classic analysis, based on unification, leads to an undirected graph. The latter is *a priori* coarser, since it creates needless communication edges. However, in the presence of polymorphism, the generalization and instantiation mechanism allows renaming certain parts of the graph, thus breaking these undesirable communication edges.

Let us consider an example. Suppose we are using plain polymorphic variant types, i.e. without row variables. Let x be a λ-bound variable, of type α. If x is passed to two functions, of which the first one handles cases `A` and `B`, while the other one handles cases `A` and `C`, then the following constraints are created:

$$\alpha \leq \texttt{[A: unit | B: unit]}$$
$$\alpha \leq \texttt{[A: unit | C: unit]}$$

One immediately remarks that this constraint set is closed, hence solvable, and this double application is well-typed. (Actually, these two constraints can be reduced to $\alpha \leq$ `[ A: unit ]`, thus telling that x must be equal to `A`.)

If, on the contrary, we have row variables, but no subtyping, we shall obtain the constraints

$$\alpha = \texttt{[A: Pre unit; B: Pre unit; Abs]}$$
$$\alpha = \texttt{[A: Pre unit; C: Pre unit; Abs]}$$

By transitivity on α, we have to unify $\langle$`B: Pre unit; Abs`$\rangle$ with $\langle$`C: Pre unit; Abs`$\rangle$, which is inconsistent. Thus, the constraints have no solution. By replacing inequations with equations, we have created a superfluous communication between `B` and `C`, which leads to too coarse an approximation.

However, in some cases, polymorphism allows eliminating this problem. If, rather than being λ-bound, x was bound by a construct of the form `let` x = `A in` ..., then the type scheme associated to x in the environment would be

$$\forall\rho.\texttt{[A: Pre unit; }\rho\texttt{]}$$

where ρ is a row variable. Then, each of the two applications would use a different instance of x, and the constraints thus created would be

$$\texttt{[A: Pre unit; }\rho_1\texttt{]} = \texttt{[A: Pre unit; B: Pre unit; Abs]}$$
$$\texttt{[A: Pre unit; }\rho_2\texttt{]} = \texttt{[A: Pre unit; C: Pre unit; Abs]}$$

Here, no transitivity takes place, and the constraints are satisfiable. One notices that, thanks to polymorphism, the two parts of the graph have been separated, thus avoiding the problems caused by the coarseness of equality.

So, row variables and subtyping complement one another. The former are indispensable when typing extensible records or matchings. Furthermore, provided a sufficient amount of polymorphism is available, they lessen the need for subtyping in numerous situations. Still, subtyping remains interesting in cases where no polymorphism is available; for instance, when dealing with a λ-bound variable. These cases do arise in practice: the above example, where an unknown (i.e. λ-bound) datum x is passed to two functions with different domains, seems natural. Thus, the presence of subtyping is desirable when dealing with variant types. This example can be translated, symmetrically, into records. The problem shows up when an unknown message m—here, a "message" is an access function of the form $\lambda x.(x.\texttt{field})$—is applied to two records with different fields. The need for subtyping hence arises when dealing with messages as first-class values.

Chapter 15

Implementation

Our theoretical study is now complete—we now have all the tools required to create an efficient type inference and simplification engine. There only remains to assemble the various parts, and to complete the whole with a display module.

Section 15.1 recalls the various components which make up the engine, and describes the way they fit together. The "external" simplification phase, performed immediately before displaying a type scheme, is described by section 15.2. The functioning of the whole is then summed up using an example (section 15.3). Lastly, section 15.4 gives a measure of our implementation's performance.

15.1 Engine

The so-called *engine* constitutes the central part of the system. The first part of its work is to build a type inference derivation for the supplied program, using the rules given by figure 12.1 on page 132. According to section 5.6, this can be done in a linear pass over the λ-term. This pass creates a principal type scheme, provided the constraints thus obtained have a solution.

To verify this condition, the engine must compute the closure of the constraint graph built by this first pass. This is easily done using the incremental closure algorithm described by definition 7.1. If the algorithm reports a failure, then the program is ill-typed. Otherwise, it is well typed, and the algorithm produces a closed constraint graph.

Finally, the engine simplifies the inferred type scheme. To this end, it applies the three simplification algorithms described in this thesis: canonization, garbage collection, and minimization. Prior to canonization, the type scheme is closed, and possibly contains occurrences of the $\sqcup$ and $\sqcap$ constructors. After canonization, any occurrence of $\sqcup$ or $\sqcap$ is gone, and the type scheme is, according to proposition 11.11, a partially garbage collected version of a simply closed scheme. Thus, it is legal to compute its polarities and to perform garbage collection. The garbage collection phase produces a perfect scheme, which can be minimized, finally yielding another perfect scheme. Thus, the three simplification phases combine very easily.

We have just described the engine's operation in three distinct phases: constraint creation, closure and simplification. In theory, these three phases can be performed in succession. However, in practice, it is fundamental, for efficiency reasons, that simplification be performed regularly as constraints appear. Indeed, the `let` construct allows moving the type scheme associated with a certain sub-expression into the environment; each use of the variable X bound by this construct shall generate a new copy of that scheme. Thus, it is important that all type schemes placed into the environment be fully simplified; otherwise, some closure and simplification work would be duplicated.

This does not pose any problem. It suffices, upon encountering a `let` node, to interrupt the constraint gathering phase, and to perform closure and simplification of the type scheme, before entering it into the environment. The type scheme produced by the simplification phase is perfect, so in particular, it is closed; thus, simplification does not interfere with the incremental closure computations to come.

One can go further and perform some computations not only at `let` nodes, but at all nodes. The closure algorithm, for instance, is incremental, so there is no reason not to update the closure immediately whenever a new constraint appears.

As for the three simplification algorithms, none is incremental; thus, experiments are needed to determine whether it is worthwhile to execute them at all nodes. Each algorithm is *a priori* costly, since it analyzes the whole constraint graph, not just the constraints that were recently added. However, if it is powerful enough, it might perform a drastic simplification and thus accelerate the following phases. For instance, garbage collection is cheap, because it only destroys constraints and thus requires (almost) no allocation, and very efficient, because it eliminates a large number of intermediate variables. So, it is worthwhile to execute it at each node, because this shall speed up subsequent closure computations. For instance, in a typical case, performing garbage collection at all nodes increases the time devoted to it by a factor of 2, but decreases the time devoted to closure by a factor of 5. As for canonization, it does not help much with future closure computations; on the contrary, it cancels the slight gain of efficiency due to the use of the $\sqcup$ and $\sqcap$ constructors. Finally, minimization, though quasi-linear, is rather costly. In practice, garbage collection shall be the only simplification performed at all nodes. (Note that to perform garbage collection before canonization, we have to prove that the former is correct in the presence of $\sqcup$ and $\sqcap$ constructors, which is straightforward, considering that polarities decrease during canonization.)

The inference engine constitutes the upper part of the diagram shown in figure 15.1 on the next page.

Let us say a few words about the machine representation of constraint graphs. Recall that in a type inference derivation, no variable is shared between two distinct branches (see lemma 5.2). Consequently, at any point of the engine's operation, each variable belongs to at most one constraint graph. Thus, it is possible to store the constraints related to it inside the variable itself, which makes them accessible and writable in constant time. From the machine's point of view, a type scheme simply consists of a context A and a body τ; the associated constraint graph is obtained implicitly, by following the constraints from these entry points. This representation is easy to use and efficient. It even allows unreachable variables to be automatically eliminated by the garbage collector, without requiring our garbage collection algorithm to be run!

15.2 Display

We have explained, after introducing the mono-polarity invariant, that some "simplification" methods attempt to augment the inference engine's efficiency, while others actually tend to improve the readability of the result (see section 12.4). We have remarked that the latter necessarily conflict with the former, since they violate fundamental invariants such as the small terms invariant and the mono-polarity invariant. For this reason, they must be used only ultimately before display, and not inside the inference engine. For this reason, the former can be referred to as *internal*, since they are part of the engine, while the latter shall be termed *external*, because they only constitute a display mechanism.

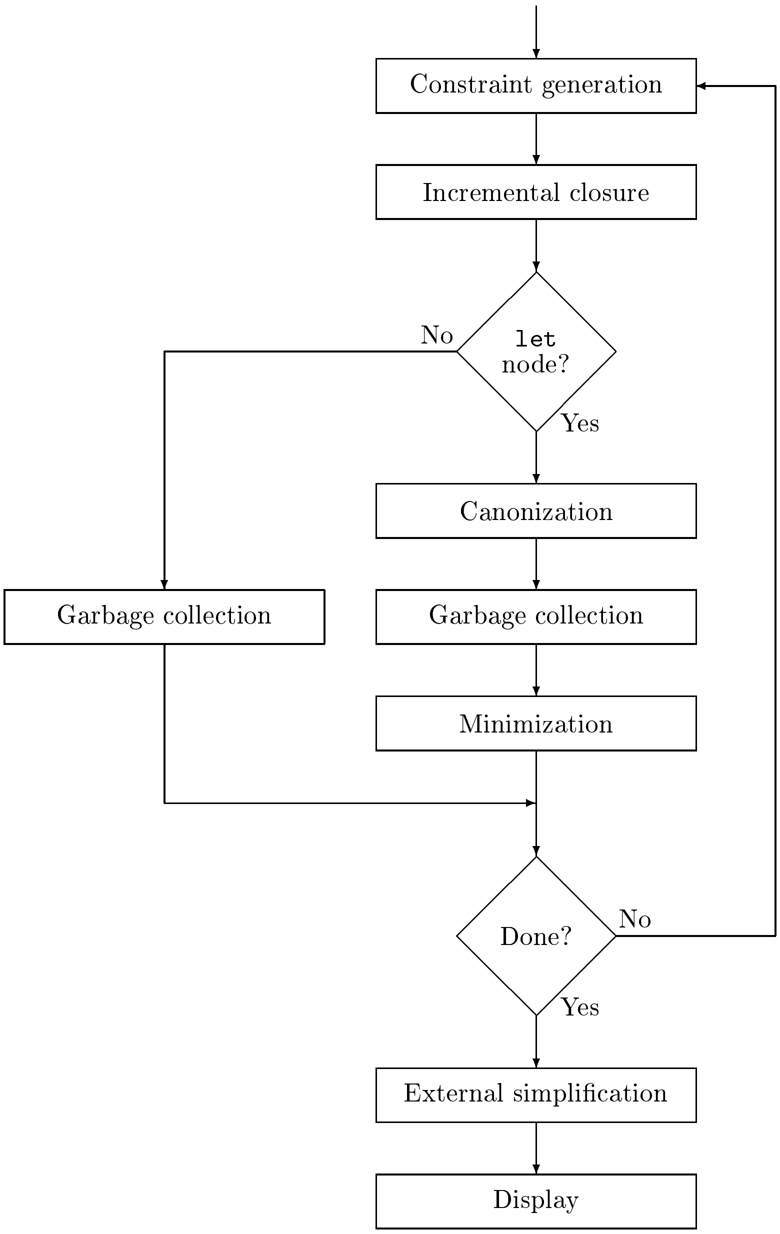

Figure 15.1: Inference and simplification process

This distinction is worth insisting upon, for it is fundamental. Trying to improve efficiency and readability at the same time would be a serious design error—a trap into which we fell, historically, and which lead us to inextricable problems, since some "simplifications" would build terms which others would destroy, leading to a very random behavior.

Besides, a similar distinction is found in ML, where types are presented to the user as trees, but are actually represented by graphs internally. If trees were used directly, certain nodes would no longer be shared, leading to an exponential efficiency loss, as suffered by the first versions of certain ML typecheckers. Similarly, in our system, not preserving the small terms invariant amounts, as explained in section 6.4, to a loss of sharing.

The classic simplification which consists in replacing a variable with its unique bound [14, 4, 8, 42] must thus be considered external, and performed only before display. It forms the lower part of the diagram shown on figure 15.1.

We haven't formally described this simplification yet. We shall do so now; it is extremely simple.

Definition 15.1 *Let $\sigma = A \Rightarrow \tau \mid C$ be a perfect type scheme. Let $\alpha \in \mathrm{dom}^+(\sigma)$. Then*

- *if $\mathrm{pred}_C(\alpha) = \varnothing$, then α's unique bound is $C^{\downarrow}(\alpha)$;*
- *if $\mathrm{pred}_C(\alpha) = \{\beta\}$ and $C^{\downarrow}(\alpha) = \bot$, then α's unique bound is β.*

A symmetric definition applies to negative variables.

Proposition 15.1 *Let σ be a perfect type scheme. Let σ' be the type scheme obtained by adding to σ the constraint $\alpha = \tau$ whenever a variable α has a unique bound τ. Then $\sigma =^{\forall} \sigma'$.*

Proof. Let α be a positive variable whose unique bound is a constructed term τ. Then, we add the constraint $\alpha \leq \tau$. Since α's only lower bound is τ, the only transitive consequence of this addition is $\tau \leq \tau$, which is trivially contained in the graph. So, the graph remains closed.

Now, let α be a positive variable whose unique bound is a variable β. Then, we add the constraint $\alpha \leq \beta$. For the graph to remain closed, we must also add the constraint $\alpha \leq \gamma$ for any γ such that $\beta \leq_C \gamma$, as well as the constraint $\alpha \leq C^{\uparrow}(\beta)$. Since α has no lower bounds other than β, the closure computation stops there.

The case of negative variables is symmetric. So, we have computed a closed form of σ'. It is then legal to perform a garbage collection computation. One immediately remarks that polarities are unchanged, since the constructed lower (resp. upper) bounds of positive (resp. negative) variables haven't been affected. As a result, all of the newly added constraints are eliminated by garbage collection. This proves the equivalence $\sigma =^{\forall} \sigma'$. □

So, it is possible to impose an equality between a variable and its unique bound, provided the latter exists, without affecting the type scheme's denotation. Thus, one can also substitute the bound for the variable, provided the variable does not appear free in its bound. This is our external simplification method, meant to enhance a type scheme's readability before presenting it to the user.

Example. Here is a classic function, which computes the length of a list:

```
let rec list_length = function
  Nil -> 0
| Cons (_, rest) -> succ (list_length rest);;
```

The type scheme produced by the inference engine is α_1^+ `raises` $\alpha_2^+ \mid C$, where C is the following constraint graph:

$$\begin{aligned}
\alpha_8^- \to \alpha_3^+ \texttt{ raises } \alpha_2^+ &\leq \alpha_1^+ \\
\bot &\leq \alpha_2^+ \\
\texttt{int} &\leq \alpha_3^+ \\
\alpha_4^- &\leq \texttt{Abs} \\
\alpha_5^- &\leq \texttt{unit} \\
\alpha_6^- &\leq \texttt{Pre } \alpha_5^- \\
\alpha_7^- &\leq \top \\
\alpha_8^- &\leq \texttt{[Nil: } \alpha_6^- \texttt{; Cons: } \alpha_{10}^- \texttt{; } \alpha_4^- \texttt{]} \\
\alpha_9^- &\leq \alpha_7^- \times \alpha_8^- \\
\alpha_{10}^- &\leq \texttt{Pre } \alpha_9^-
\end{aligned}$$

Here, all variables have a unique bound. By performing as many substitutions as possible, we obtain the scheme $\alpha_8^- \to \texttt{int} \mid D$, where D is given by

$$\alpha_8^- \leq \texttt{[Nil: Pre unit; Cons: Pre } \top \times \alpha_8^- \texttt{; Abs]}$$

(The `raises` annotations carried by the arrow and by the scheme have disappeared, because we have agreed to omit them, when they are of the form `raises` $\bot$.) Although α_8 has a unique upper bound, the substitution could not be performed, because α_8 appears in its own bound. Informally, one could introduce a μ binder, yielding

$$\mu\alpha_8.\texttt{[Nil: Pre unit; Cons: Pre } \top \times \alpha_8 \texttt{; Abs]} \to \texttt{int}$$

Note that, in the presence of recursive constraints, the order in which substitutions are performed is significant. For instance, if we had first replaced α_8 with its bound, one might have obtained `[ Nil: Pre unit; Cons:` α_{10}^-`; Abs ]` $\to$ `int` $\mid D'$, where D' is given by

$$\alpha_{10}^- \leq \texttt{Pre } \top \times \texttt{[Nil: Pre unit; Cons: } \alpha_{10}^- \texttt{; Abs]}$$

Of course, this scheme is equivalent, but less readable, because a recursive type has been partially unfolded. It would thus be interesting to be able to choose which variables shall be replaced and which ones shall remain visible, so as to lead to an optimal textual representation. Let us draw a graph linking each variable to the free variables of its unique constructed bound, if the latter exists:

$$\begin{tikzcd}
\alpha_3 & \alpha_1 \arrow[l] \arrow[r] \arrow[d] & \alpha_2 & \\
\alpha_6 \arrow[d] & \alpha_8 \arrow[l] \arrow[r] \arrow[d] & \alpha_4 & \\
\alpha_5 & \alpha_{10} \arrow[r] & \alpha_9 \arrow[r] \arrow[ul] & \alpha_7
\end{tikzcd}$$

The fact that a variable cannot be replaced with a term in which it appears translates to the fact that, in this graph, any cycle must contain at least one visible variable. In the above example, there is a cycle between α_8, α_{10} and α_9; at least one of these variables must remain visible.

On could decide to choose, within each cycle, a variable as close to the graph's entry points as possible. Here, the entry points are α_1 and α_2, so α_8 is closest to them. Indeed, if we choose to leave α_8 visible and to replace α_9 and α_{10}, we obtain an optimal representation.

However, in general, in a cycle, the variable closest to the entry points is not necessarily unique. Besides, other criteria also seem acceptable. For instance, if a given variable belongs to several cycles, it might be interesting to leave it visible. The problem thus seems difficult.

To conclude, we have not studied this optimality problem any further. First, it seems delicate. Second, it seems of rather minor importance, since in practice, a random choice yields acceptable results in most cases.

15.3 A full example

This section shows the type inference engine and the display module at work on a small, but typical example, namely the classic `map` function on lists. For the sake of simplicity, we use polymorphic variant types (without row variables), and we do not perform the exception analysis.

```
rec map in function f -> function
    Nil -> Nil
  | Cons (x, rest) -> Cons (f x, map f rest)
```

To make things simpler, we assume that the simplification algorithms are run only after all constraints have been gathered.

The inference rules indicate that the expression has type v_{22}, together with the following constraints, in no particular order:

$$\begin{array}{llll}
v_3 \leq v_4 & v_5 \leq v_6 & v_7 \leq v_8 & v_4 \leq v_6 \to v_7 \\
v_{11} \leq v_{12} & v_{13} \leq v_{14} & v_1 \leq v_{20} & v_{12} \leq v_{14} \to v_{15} \\
v_8 * v_{16} \leq v_{17} & v_{19} \to v_{20} \leq v_{21} & v_2 \leq v_5 * v_{13} & \texttt{[Cons of } v_{17} \texttt{]} \leq v_{18} \\
v_{18} \leq v_{20} & v_{22} \leq v_9 & v_9 \leq v_{10} & v_{19} \leq \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil]} \leq v_1 & v_3 \to v_{21} \leq v_{22} & v_{15} \leq v_{16} & v_{10} \leq v_4 \to v_{11}
\end{array}$$

Let us compute the closure of these constraints. It is given below, as a constraint graph.

$$\begin{array}{rl}
\texttt{[Nil]} \leq v_1 & \leq v_{15}, v_{16}, v_{20} \\
v_2 & \leq v_5 * v_{13} \\
v_4 \leq v_3 & \leq v_4, v_6 \to v_7 \\
v_3 \leq v_4 & \leq v_3, v_6 \to v_7 \\
v_5 & \leq v_6 \\
v_5 \leq v_6 & \\
v_7 & \leq v_8 \\
v_7 \leq v_8 & \\
v_3 \to v_{21}, v_{22} \leq v_9 & \leq v_{10}, v_4 \to v_{11} \\
v_3 \to v_{21}, v_9, v_{22} \leq v_{10} & \leq v_4 \to v_{11} \\
v_{19} \to v_{20}, v_{21} \leq v_{11} & \leq v_{12}, v_{14} \to v_{15} \\
v_{19} \to v_{20}, v_{11}, v_{21} \leq v_{12} & \leq v_{14} \to v_{15} \\
v_{13} & \leq v_{14}, v_{19}, \texttt{[Nil | Cons of } v_2 \texttt{]} \\
v_{13} \leq v_{14} & \leq v_{19}, \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]}, v_1, v_{18}, v_{20} \leq v_{15} & \leq v_{16} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]}, v_1, v_{18}, v_{15}, v_{20} \leq v_{16} & \\
v_8 * v_{16} \leq v_{17} & \\
\texttt{[Cons of } v_{17} \texttt{]} \leq v_{18} & \leq v_{15}, v_{16}, v_{20} \\
v_{13}, v_{14} \leq v_{19} & \leq \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]}, v_1, v_{18} \leq v_{20} & \leq v_{15}, v_{16} \\
v_{19} \to v_{20} \leq v_{21} & \leq v_{11}, v_{12}, v_{14} \to v_{15} \\
v_3 \to v_{21} \leq v_{22} & \leq v_9, v_{10}, v_4 \to v_{11}
\end{array}$$

Here, no $\sqcup$ or $\sqcap$ constructors were created by the closure computation. Actually, a few appeared in order to combine multiple bounds into a single bound, but they were immediately eliminated by the reduction rules of definition 2.2. For instance, `[ Nil ]` $\sqcup$ `[ Cons of` v_{17} `]` = `[ Nil | Cons of` v_{17} `]`. Thus, the canonization algorithm has no effect.

Let us move on to the polarity computation. We first mark the only entry point, namely v_{22}, positive; then, we let marks propagate through the graph until a fix-point is reached. We thus discover that $v_6, v_8, v_{16}, v_{17}, v_{20}, v_{21}, v_{22}$ are positive, while $v_2, v_3, v_5, v_7, v_{13}, v_{19}$ are negative.

Thanks to this information, we can perform garbage collection. Positive (resp. negative) variables lose all of their upper (resp. lower) bounds; furthermore, constraints involving two variables are kept only if the left-hand one is negative and the right-hand one is positive. This yields

$$\begin{aligned}
v_2 &\leq v_5 * v_{13} \\
v_3 &\leq v_6 \rightarrow v_7 \\
v_5 &\leq v_6 \\
v_5 &\leq v_6 \\
v_7 &\leq v_8 \\
v_7 &\leq v_8 \\
v_{13} &\leq \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]} &\leq v_{16} \\
v_8 * v_{16} &\leq v_{17} \\
v_{19} &\leq \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]} &\leq v_{20} \\
v_{19} \rightarrow v_{20} &\leq v_{21} \\
v_3 \rightarrow v_{21} &\leq v_{22}
\end{aligned}$$

We can now run the minimization algorithm. We compute the largest partition such that two equivalent variables have the same polarity, the same successors and predecessors, and equivalent constructed bounds. In this case, it is easy to see that the only non-trivial equivalence classes are $\{v_{13}, v_{19}\}$ and $\{v_{16}, v_{20}\}$.

Note that replacing v_{19} with v_{13} amounts to recognizing a partially unrolled fix-point. Indeed, it essentially consists in replacing $F(\mu t.F(t))$ with $\mu t.F(t)$, where F is the type operator

$$t \mapsto \texttt{[Nil | Cons of } v_5 * t \texttt{]}$$

A similar remark can be made about v_{16} and v_{20}. After collapsing the classes, the graph becomes

$$\begin{aligned}
v_2 &\leq v_5 * v_{13} \\
v_3 &\leq v_6 \rightarrow v_7 \\
v_5 &\leq v_6 \\
v_5 &\leq v_6 \\
v_7 &\leq v_8 \\
v_7 &\leq v_8 \\
v_{13} &\leq \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]} &\leq v_{16} \\
v_8 * v_{16} &\leq v_{17} \\
v_{13} \rightarrow v_{16} &\leq v_{21} \\
v_3 \rightarrow v_{21} &\leq v_{22}
\end{aligned}$$

Simplification is over; the engine is done. There remains to apply "external" simplifications, so as to make the type scheme more readable. We replace each variable with its unique

	LOC	Sec.	/	Clo.	Can.	G.C.	Min.	Caml-Light
Graphs lib.	900	2s	450	45%	10%	30%	15%	1s
CL standard lib.	4300	5s	860	35%	20%	25%	15%	6s
Format lib.	1100	7s	160	25%	10%	25%	40%	1s
MLgraph lib.	9900	27s	370	35%	10%	25%	30%	13s

Figure 15.2: Implementation's performance

bound (except where disallowed by the occur check). Finally, `map` receives the type scheme $(v_6 \rightarrow v_8) \rightarrow v_{13} \rightarrow v_{16} \mid C$, where C is the constraint graph

$$\begin{aligned} v_{13} &\leq \texttt{[Nil | Cons of } v_6 * v_{13} \texttt{]} \\ \texttt{[Nil | Cons of } v_8 * v_{16} \texttt{]} &\leq v_{16} \end{aligned}$$

This result is optimal, insofar as we work only with concrete types, i.e. we haven't defined the "list" type. If one wishes to go a little further, one can introduce μ binders as in section 15.2. The type scheme then becomes

$$(v_6 \rightarrow v_8) \rightarrow \mu t.\texttt{[Nil | Cons of } v_6 * t \texttt{]} \rightarrow \mu t.\texttt{[Nil | Cons of } v_8 * t \texttt{]}$$

15.4 Performance

The algorithms described in this thesis have been implemented in a prototype, whose source code should be made available electronically in the near future. This prototype typechecker deals with an enriched λ-calculus, whose syntax resembles Caml's, including its imperative aspects. The type language contains all of the extensions presented in chapter 14.

Figure 15.2 presents the results of a few performance measurements. The prototype, compiled into machine language using Objective Caml 1.07, was tested on a 150MHz Pentium Pro processor.

The sample programs used in this test have been written by various authors in Caml-Light, and adapted to our language using a summary translator. Thus, they do not take advantage of subtyping; nevertheless, they allow a relevant measurement of our inference engine's efficiency. Since our prototype does not support separate compilation, each sample was turned by the translator into a single module, containing no type information. So, none of the type information—in particular, abstract type declarations—contained in the interface of the Caml-Light modules was used. Therefore, the types thus obtained are usually much more precise than those inferred by Caml-Light, and the typechecker's task is much heavier.

The Graphs library (written by Laurent Chéno) contains a few graph handling primitives. The next two entries concern Caml-Light's standard library, whose Format module we isolated. It is significantly more difficult to handle than the other modules, because it involves much more complex types. Finally, the MLgraph library (written by Emmanuel Chailloux and Guy Cousineau) provides a set of primitives to deal with graphic objects.

In each case, we first give the size of the example at hand, as a number of lines of code. Then, we measure the time consumed by the whole typing process, in seconds of user time; hence the number of lines handled per second. The next four columns show how the time is split up between the various phases of the inference engine: constraint creation and incremental closure; canonization; garbage collection; and minimization. The last column gives, as a reference point, the time required by the Caml-Light compiler to handle this code.

The Graphs and MLgraph libraries are made up of medium-size functions. In these cases, which can be probably be regarded as typical, our typechecker's performance is of about 400 lines per second. This is only a factor of 2 behind the Caml-Light compiler, which is known for its speed. Thus, this is a satisfactory result.

The standard library contains functions of lesser size; the speed then doubles. Lastly, the Format library contains rather complex functions, which receive large concrete types; performance is then halved. Thus, some difficulties still arise when dealing with very large type schemes. Recall that the reason why such schemes appear is mainly our exclusive use of concrete types; in a realistic language, modularity would allow introducing abstract types, leading to much smaller schemes. So, the decision to deal solely with concrete types allows a good test of our implementation; the performance problems encountered here would not necessarily occur when typing a more realistic, modular language.

How to solve the remaining problems? First, one can wish to design incremental simplification algorithms. Indeed, a non-incremental algorithm, even if it is linear, leads to a quadratic behavior, since it is run at each node of the syntax tree (or at each `let` node). A hypothetical incremental canonization algorithm was discussed in section 11.5. Besides, it seems possible to use the minimization algorithm in a more incremental way, with a certain loss of power. Indeed, if we can draw a distinction between "old" and "recent" parts in the constraint graph, then we can choose an initial partition where all old variables are isolated. Thus, the algorithm can only discover sharing between recent variables. On the other hand, all links between old variables, and all old variables not linked to a recent one, can be ignored during the refinement phase, whence a computation time which depends only on the size of the recent part. There remains to see whether such a compromise is worthwhile.

However, note that these efficiency problems are not due only to our simplification algorithms. Indeed, in all of the cases considered above, constraint creation and closure requires a constant part of the time, namely about one third. Thus, one can say that the simplification phases are not a limiting factor: even in the ideal case where simplification is performed at no cost, the complexity of the inference process remains unchanged. This is a very satisfactory result, which was far from granted at the start of this work. It implies that, to improve performance, we should now also try to speed up the closure phase. One can think of performing a topological analysis of the constraints, before or during the closure phase. This analysis could allow, for instance, speeding up the closure algorithm's convergence by feeding it the constraints in an optimal order; or lightening its task by performing on-the-fly simplification, such as cycle elimination [16]. These ideas currently remain to be studied. Besides, a different formulation of our typing system might allow generating fewer constraints; this idea shall be detailed in chapter 16.

To conclude, the efficiency obtained so far is very satisfactory for programs of small size. The time dedicated to simplification seems to be of the same order than that devoted to closure, which tells that our simplification algorithms are not a limiting factor, and is thus a testimony in their favor. On the other hand, our implementation is not efficient enough to easily deal with large programs. The issue deserves further study; to this end, the following chapter offers some research leads.

Chapter 16

Difficulties and directions

The system studied throughout this thesis enjoys a rather simple and well understood formalization. Nevertheless, it is not perfect, and several points are unsatisfactory. This results, in practice, in expressiveness or efficiency problems. Here, we review a few of these problems, then envision various ways of solving them. Of course, these are only leads for future research; these matters require a more thorough study.

The first problem considered here is the addition of imperative constructs to the language (section 16.1). The thing is easy, as long as one considers a single expression, that is, a non-modular program; however, limitations arise when one wishes to obtain separate compilation. The second shortcoming, evidenced by section 16.2, is an inefficiency which seems inherent to our use of λ-lifting. Indeed, the latter causes part of the typing information to be duplicated, thus needlessly increasing the load on the simplification algorithms. To answer these problems, we propose a few possible solutions: automatic creation of type abbreviations (section 16.3), or adoption of an "ML-like" type system (section 16.4).

16.1 Typing imperative programs

The language we studied so far is purely functional, that is, its semantics is expressed simply using a rewriting relation between terms, without involving the notion of memory state. However, some algorithms can be expressed more naturally, or more efficiently, in an imperative form. Thus, it is desirable, as in ML, to provide the user with the ability to manipulate *mutable memory cells*, or *references*.

We extend the language with three primitive operations: `ref`, `!` and `:=`, which respectively allow creating a cell, reading its content and modifying it. As indicated in section 14.3, these primitives accept the following type schemes:

$$\begin{aligned} \texttt{ref} &: \alpha \to (\alpha, \alpha)\ \texttt{ref} \\ \texttt{!} &: (\alpha, \beta)\ \texttt{ref} \to \beta \\ \texttt{:=} &: (\alpha, \beta)\ \texttt{ref} \to \alpha \to \texttt{unit} \end{aligned}$$

where the `ref` constructor is binary, contravariant with respect to its first argument and covariant with respect to the second one.

We then reformulate the language's semantics, so as to give formal significance to these three operations. This is entirely classic [49, 33], so we shall not give the extended semantics explicitly. There remains to verify that the type system is still correct with respect to the semantics. However, it is well known that this result is false, because of the interaction between references and polymorphism. For instance, the program

```
let x = ref (fun x -> x) in
x := succ;
!x true
```

is well-typed, because the `let` construct creates a "polymorphic reference" `x`, which can then be used indifferently with integers or booleans; yet, this program leads to an execution error, since it computes `succ true`.

In order to avoid this problem, polymorphism must be restricted. Various possibilities have been studied, in the case of ML, by Leroy [33]. However, the simplest solution is probably the one proposed by Wright [47, 48]; it is the one we choose here. It consists, using an extremely simple syntactic criterion, in detecting the `let` constructs whose body is an *expansive* expression (i.e. one whose evaluation might create new cells), and in preventing the body's type from being generalized. However, recall that our system does not support the notion of monomorphic, or ungeneralized, type variable; hence, this formulation of Wright's restriction does not suit us. Fortunately, this first problem can easily be worked around; it suffices to adopt an equivalent point of view, which consists in rewriting expansive `let` constructs into plain β-redexes. Thus, the above program becomes

```
(fun x -> x := succ; !x true) (ref (fun x -> x))
```

This time, the expressions `x := succ` and `!x true` are typed in a context where `x` is λ-bound, and thus give birth to the constraint $\texttt{bool} \leq \texttt{int}$. So, the program is correctly rejected.

There remains to deal with toplevel `let` constructs, that is, the ones which define the various components of a module. Concretely, a module is a series of declarations:

$$\begin{array}{l}\texttt{let}\, X_1 = e_1\,\texttt{;;} \\ \dots \\ \texttt{let}\, X_n = e_n\,\texttt{;;}\end{array}$$

To obtain separate compilation, this group of declarations must analyzed in isolation. Hence, it is impossible, when e_i is expansive, to apply the above method, which would consist in abstracting the rest of the program over the variable X_i.

Since our system does not support the notion of ungeneralized variable, and since we can no longer use a rewriting technique, we typecheck these `let` declarations in the usual way. So, one uses the inference algorithm to give each expression e_i a type scheme σ_i, which shall then be associated to the variable X_i. However, we then have to impose a sufficient condition to ensure correctness with respect to the semantics. A simple condition consists in requiring σ_i to be trivial whenever e_i is expansive. Indeed, as shown by proposition 12.3, a trivial scheme offers no polymorphism. So, generalizing it brings no additional power, and is not dangerous.

This approach is correct, but more restrictive than the one used e.g. by Objective Caml. Firstly, ML's type system allows a toplevel `let` construct to produce a partially generalized type scheme. For instance, if a module is made up of the declaration

```
let x = ref (fun x -> x);;
```

then, the type scheme associated to `x` is $(\alpha \rightarrow \alpha)$ `ref`, where α is not universally quantified. Secondly, in the case of ML, the subsumption relation, used to compare the inferred scheme to the declared one, can easily be extended to the case where the left-hand scheme contains unquantified variables. Thus, one can declare, in the interface:

```
val x: (int -> int) ref
```

During the comparison, the monomorphic variable α shall be instantiated to the value `int`, and the module shall be accepted with this signature. Our system is less flexible; any instantiation must be performed as soon as the value is defined, not during the comparison between module and interface. Thus, the module must be written, for instance,

```
let x = [ ref (fun x -> x) : (int -> int, int -> int) ref ];;
```

(The construct `[` e `:` τ `]` simulates a use of the expression e with type τ. If the type scheme associated to e is $A \Rightarrow \tau_e \mid C$, this construct creates the extra constraint $\tau_e \leq \tau$. Here, this constraint allows associating a trivial type scheme to `x`.) One can then indicate, in the signature:

```
val x : (int -> int, int -> int) ref
```

Any program acceptable in Objective Caml is also acceptable in our system, provided enough explicit annotations are added to make any expansive declaration monomorphic. In practice, this typing information could be automatically extracted from the signature and inserted into the program source.

This restriction is imposed only by the objective of separate compilation. In the case of an interactive loop, where the user supplies, one at a time, a series of toplevel `let` declarations, the problem is less difficult, and the technique of rewriting expansive `let` constructs into β-redexes can again be used. However, its interest is rather limited; it does not seem desirable to offer more flexibility in the interactive loop than in the modular programming system.

To solve this problem, it seems desirable to come back to a system where all variables are not necessarily quantified. It is rather easy to formulate such a system, by abandoning λ-lifting and coming closer to ML. However, this yields a rather complex subtyping rule. We present this system in section 16.4.

16.2 Drawbacks of λ-lifting

The system studied throughout this thesis uses a λ-lifting technique. Its main advantage lies in the fact that all type schemes are then closed. This simplifies the subtyping rule, and therefore, the bulk of our theory. However, it does have drawbacks.

The most immediate one lies in the fact that λ-lifting is a rather exotic mechanism, often counter-intuitive for the user, even though it is based on a rather simple idea. Let us briefly recall its functioning, by considering the way the expression $\lambda x.(x, x)$ is handled by the inference algorithm.

When the algorithm encounters the λx binder, it adds the name x to the environment, but does not associate a type variable to it. The two occurrences of x are thus typechecked entirely independently, and the type schemes they produce share no variable. Let us denote these schemes by $(\varnothing; x : \alpha_i) \Rightarrow \beta_i \mid \{\alpha_i \leq \beta_i\}$, for $i \in \{1, 2\}$. To type the pair (x, x), we must, as for an application, compute the intersection of the contexts coming from the two branches. So, the scheme associated to this pair shall be

$$(\varnothing; x : \alpha_1 \sqcap \alpha_2) \Rightarrow \gamma \mid \{\alpha_1 \leq \beta_1,\ \alpha_2 \leq \beta_2,\ \beta_1 \times \beta_2 \leq \gamma\}$$

and the whole expression receives the scheme

$$(\alpha_1 \sqcap \alpha_2) \rightarrow \gamma \mid \{\alpha_1 \leq \beta_1,\ \alpha_2 \leq \beta_2,\ \beta_1 \times \beta_2 \leq \gamma\}$$

Now, if we apply the canonization algorithm, we obtain

$$\alpha \rightarrow \gamma \mid \{\alpha \leq \alpha_1,\ \alpha \leq \beta_1,\ \alpha \leq \alpha_2,\ \alpha \leq \beta_2,\ \alpha_1 \leq \beta_1,\ \alpha_2 \leq \beta_2,\ \beta_1 \times \beta_2 \leq \gamma\}$$

A garbage collection pass yields

$$\alpha \to \gamma \mid \{\alpha \leq \beta_1,\, \alpha \leq \beta_2,\, \beta_1 \times \beta_2 \leq \gamma\}$$

Finally, by running the minimization algorithm, we obtain an optimal result, namely

$$\alpha \to \gamma \mid \{\alpha \leq \beta,\, \beta \times \beta \leq \gamma\}$$

Thus, the type system does not use the environment to share information between the various branches of the derivation. Sharing is first voluntarily lost, thus leading each branch to be typed in isolation, then explicitly restored by the context intersection operation. In a system without λ-lifting, as ML, a fresh variable α would have been associated to x when the λx binder is entered, and both occurrences of x would have received this same variable as type. The optimal type $\alpha \to \alpha \times \alpha$ would thus have been obtained without requiring any simplification.

It is clear that the use of λ-lifting can hinder a non-specialist's understanding of the system, and thus poses a pedagogic problem. However, the above example further suggests that it also causes a certain efficiency loss. Indeed, it shows that the canonization, garbage collection and minimization algorithms spend part of their time restoring sharing properties which were voluntarily lost because of the typing rules' formulation. Generally speaking, consider an expression of the form

$$\lambda x.\mathtt{let}\, X = e_1 \,\mathtt{in}\; e_2$$

The expression e_1 can use the variable x in a complex way, e.g. by feeding it to a deep pattern matching. Hence, the type scheme associated to e_1 shall be of the form $(\ldots; x : \alpha \ldots) \Rightarrow \ldots \mid (C \cup \ldots)$, where C contains a number of constraints concerning α, thus defining x's expected type. During e_2's analysis, each use of the variable X causes these constraints to be duplicated; then, the context intersection operation restores the sharing between the various copies. The scheme inferred for e_2 shall thus be of the form $(\ldots; x : \alpha_1 \sqcap \cdots \sqcap \alpha_n \ldots) \Rightarrow \ldots \mid (C_1 \cup \cdots \cup C_n \cup \ldots)$, where n is the number of occurrences of X in e_2, and where $\alpha_i \mid C_i$ are copies of $\alpha \mid C$. Once again, simplification computations are necessary to get rid of the redundancy.

Which solutions can be brought to this problem? In the following, we consider two possibilities. The first one, discussed in section 16.3, consists in reducing the impact of these needless duplications by devising an automatic abbreviation creation mechanism. The idea is to represent a couple $\alpha \mid C$ by an abbreviation, that is, a mere name, possibly parameterized. Only the abbreviation shall then be duplicated, hence, hopefully, a gain of efficiency. The second possibility, described by section 16.4, consists in abandoning λ-lifting and adopting a more classic formulation of the type system, already mentioned in the previous section. The variable α shall then be monomorphic, so it shall not be duplicated; provided a special-purpose analysis is performed, the same applies to the constraints concerning α contained in the graph C. Besides, note that these two ideas are not mutually exclusive; the interest of the notion of abbreviation is independent of the underlying system.

16.3 Automatic abbreviations

Automatic creation of abbreviations, used by Rémy and Vouillon [44], consists in representing a piece of structure by a mere name. For instance, the pair $(0, 0)$ admits the type scheme

$$\beta \mid \{\mathtt{int} \leq \alpha,\, \alpha \times \alpha \leq \beta\}$$

This scheme is trivial, in the sense of definition 12.3, i.e. its variables introduce no polymorphism. They only serve, as explained when the small terms invariant was introduced, to label each node. Consequently, duplicating this scheme is useless. For instance, when typing the expression

$$\texttt{let}\, X = (0,0)\, \texttt{in}\ (X,X)$$

each occurrence of X is given a copy of the above scheme, and minimization is necessary to restore the sharing of the two occurrences. So, one can imagine setting up the following mechanism: before X is introduced in the environment, one defines an abbreviation

$$\texttt{intpair} = \forall\alpha \mid \{\texttt{int} \leq \alpha\}.\, \alpha \times \alpha$$

X is then given the scheme $\beta \mid \{\texttt{intpair} \leq \beta\}$ in the environment. This scheme is duplicated at each occurrence of X; thus, the variable β is duplicated, but not the structure represented by the `intpair` abbreviation. Thus, incrementality has been improved.

The above example is particularly simple, since the type scheme at hand is trivial. Because it offers no polymorphism, it can be entirely represented by a single name, `intpair`. More generally, a type scheme contains a certain degree of polymorphism, and it shall be necessary to create *parameterized* abbreviations. For instance, consider the scheme $(\varnothing; l : \alpha_1) \Rightarrow \alpha_9 \mid C$, where C is given by

$$\begin{aligned}
\alpha_1 &\leq \texttt{[Nil: } \alpha_2\texttt{; Cons: } \alpha_3\texttt{; } \alpha_4 \texttt{]} \\
\alpha_2 &\leq \texttt{Pre } \alpha_5 \\
\alpha_3 &\leq \texttt{Pre } \alpha_6 \\
\alpha_4 &\leq \texttt{Abs} \\
\alpha_5 &\leq \texttt{unit} \\
\alpha_6 &\leq \alpha_7 \times \alpha_1 \\
\alpha_7 &\leq \alpha_8 \\
\alpha_7 &\leq \alpha_8 \\
\alpha_8 \times \alpha_8 &\leq \alpha_9
\end{aligned}$$

(This scheme could belong to an expression which reads a list from variable l, and builds a pair using two if its elements.) This scheme is not trivial, because it contains a constraint which links two variables, $\alpha_7 \leq \alpha_8$. Other variables only serve to label nodes, and can be hidden by abbreviations. α_7 and α_8, on the contrary, must remain visible, because duplicating them is necessary to avoid a loss of generality; they shall thus appear as parameters of these abbreviations. Concretely, we shall create two abbreviations here, that is, one for each of the scheme's entry points. Set

$$\begin{aligned}
&\alpha_7\ \texttt{list} = \exists\alpha_1 \ldots \alpha_6 \mid C_{1-6}.\, \texttt{[Nil: } \alpha_2\texttt{; Cons: } \alpha_3\texttt{; } \alpha_4 \texttt{]} \\
&\alpha_8\ \texttt{pair} = \alpha_8 \times \alpha_8
\end{aligned}$$

(C_{1-6} denotes the constraint graph made up of the six first lines of the graph C.) The original type scheme can then be written

$$(\varnothing; l : \alpha_1) \Rightarrow \alpha_9 \mid \{\alpha_1 \leq \alpha_7\ \texttt{list}, \alpha_7 \leq \alpha_8, \alpha_8\ \texttt{pair} \leq \alpha_9\}$$

(α_1 and α_9 still appear in this scheme, but only to preserve the small terms invariant. In principle, they could also have been hidden.) Once again, this scheme can now be duplicated while preserving sharing of the internal structure. Note that the `list` abbreviation is meant to be used in negative position, hence the use of existential quantifiers in its definition. The `pair` abbreviation is meant to be used in positive position; so, as `intpair`, it uses universal quantifiers. In fact, in the case of `pair`, no quantifier appears, because no variable is hidden by `pair`; in practice, one can dispense with introducing this abbreviation, since no profit is to be reaped from it.

In each case, the only variables free in the body of the definition are the parameters of the abbreviation. To expand an abbreviation, one replaces it with its body, while substituting the actual parameters for the formal ones, and a fresh variable for each variable bound in the definition. Numerous operations can be performed without requiring abbreviations to be expanded: scheme duplication, garbage collection, minimization... The minimization algorithm can no longer achieve optimal sharing, but it is used in a more incremental fashion. Indeed, when the algorithm is invoked (that is, at each `let` node, before abbreviations are created), the recent part of the constraint graph is made up of concrete types, newly introduced by the typing rules, while older parts are hidden by abbreviations, considered abstract by the algorithm. So, certain sharing opportunities between old parts cannot be discovered; on the other hand, the time consumed by minimization should be approximately proportional to the size of the recent parts, hence a better incrementality. Besides, expansion of abbreviations remains of course necessary during the closure computation. For instance, if a constraint of the form α `list` $\leq$ `[ Nil:` β`; Cons:` γ`;` ρ `]` appears, then the left-hand term must be expanded, so as to allow structural decomposition. The closure algorithm is incremental, so only abbreviations reached by recent constraints shall be expanded.

We shall not formalize this automatic abbreviation mechanism here. To sum up, it is made up of two components: an abbreviation generator, invoked e.g. at each `let` node, and the expansion mechanism, called on demand by the closure and canonization algorithms. The former creates several abbreviations per type scheme, delimited by constraints between variables, such as the constraint $\alpha_7 \leq \alpha_8$ above. Its formalization should be rather simple. The latter is in principle very simple, but its introduction breaks the invariants of the closure and canonization algorithms; for this reason, a formal description of it should be more difficult.

We have implemented this system of abbreviations and obtained, in most cases, no benefits. How to explain that? It is possible that the abbreviations often carry a large number of parameters. Indeed, the λ-lifting technique leads us to abstract each expression with respect to the whole environment. The expression thus appears to be very polymorphic, and its associated abbreviation carries numerous parameters. In the setting of the "ML-like" system which we shall describe below, on the contrary, variables which appear in the environment are monomorphic and can be hidden inside the abbreviation. Thus, it should be interesting to implement the abbreviation mechanism in this system.

16.4 An "ML-like" type system

We shall now introduce a type system whose formulation is closer to that of ML, and prove its correctness. However, its subtyping rule is complex, so it is not easy to derive judgement comparison and simplification algorithms from it, as we did in our system. This is why we chose, during this thesis, to work with a less flexible, but theoretically simpler, system.

16.4.1 Presentation

The new system is given by figure 16.1 on the facing page. Typing judgements are of the form $C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e : \tau$. The constraint graph C concerns the variables free in Γ, A and τ; for the judgement to be considered valid, C must be solvable. The environment Γ maps each `let`-bound variable to a type scheme of the form $\forall\bar{\beta} \mid C.\tau$, while the environment A maps each λ-bound variable to a type τ. (These two environments are distinguished here, so as to better insist on their different roles, in particular in the subtyping rule; however,

$$C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} x : A(x) \qquad (\textsc{Var}_{\mathrm{ML}})$$

$$\frac{C \Rightarrow \Gamma, (A; x : \tau) \vdash_{\mathrm{ML}} e : \tau'}{C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} \lambda x.e : \tau \to \tau'} \qquad (\textsc{Abs}_{\mathrm{ML}})$$

$$\frac{C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e_1 : \tau_2 \to \tau \qquad C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e_2 : \tau_2}{C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e_1\, e_2 : \tau} \qquad (\textsc{App}_{\mathrm{ML}})$$

$$\frac{\Gamma(X) = \forall\bar{\beta} \mid C.\tau \qquad \rho \text{ substitution of domain } \bar{\beta}}{\rho(C) \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} X : \rho(\tau)} \qquad (\textsc{LetVar}_{\mathrm{ML}})$$

$$\frac{\begin{array}{c} C_1 \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e_1 : \tau_1 \qquad \bar{\beta} \cap \mathrm{fv}(A) = \varnothing \\ C_2 \Rightarrow (\Gamma; X : \forall\bar{\beta} \mid C_1.\tau_1), A \vdash_{\mathrm{ML}} e_2 : \tau_2 \end{array}}{C_2 \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} \texttt{let}\, X = e_1 \;\texttt{in}\; e_2 : \tau_2} \qquad (\textsc{Let}_{\mathrm{ML}})$$

$$\frac{\begin{array}{c} C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e : \tau \\ \forall \rho' \vdash C' \quad \exists \rho \vdash C \quad \rho'(\Gamma') \leq^{\forall} \rho(\Gamma) \wedge \rho'(A') \leq \rho(A) \wedge \rho(\tau) \leq \rho'(\tau') \end{array}}{C' \Rightarrow \Gamma', A' \vdash_{\mathrm{ML}} e : \tau'} \qquad (\textsc{Sub}_{\mathrm{ML}})$$

Figure 16.1: "ML-like" typing rules

this distinction is not indispensable.) It is important to notice that type schemes are not necessarily closed any longer.

Which advantages do we expect from this system? First, its presentation is simpler and more classic, which is interesting, if only from a pedagogic point of view. Next, it makes the interaction between polymorphism and imperative constructs significantly easier to deal with. Third, the type scheme associated to a given expression is less polymorphic, since it is no longer abstracted over the environment. Thus, in the setting of automatic abbreviation creation, one can hope to be able to create abbreviations with fewer parameters. Lastly, the `let` construct does not generalize variables which appear free in the environment. Thus, one gets rid of the context duplication/intersection phenomenon evidenced in section 16.2.

About this last point, note that a very simple analysis is necessary, at each `let` node, to avoid needless duplication. If rule ($\textsc{Let}_{\mathrm{ML}}$) is applied blindly, one generalizes all variables which are not free in the environment. However, as said before, some of these variables serve only to label nodes, not to allow polymorphism, so it is useless to duplicate them. For instance, consider the expression

$$\lambda x.\texttt{let}\, X = (x, x)\;\texttt{in}\;(X, X)$$

In the environment $(\varnothing; x : \alpha)$, the expression (x, x) has type β, with the constraint $\{\alpha \times \alpha \leq \beta\}$. When X is introduced into the environment, it is legal to generalize β. X is then given the type scheme $\forall\beta \mid \{\alpha \times \alpha \leq \beta\}.\beta$. Each occurrence of X shall then receive a different instance of β. However, this is not needed; since α is monomorphic, these two instances shall have the same lower bound, namely $\alpha \times \alpha$, and they shall be identified by the minimization algorithm. Thus, it is more efficient not to generalize β. Generally speaking, the set of variables to be generalized is computed as a fix-point. First, any "non-trivial" variable, i.e. carrying a link to another variable, must be generalized. Second, if the bound

$$A \vdash_{\mathrm{G}} x : A(x) \qquad (\textsc{Var}_{\mathrm{G}})$$

$$\frac{A; x : \tau \vdash_{\mathrm{G}} e : \tau'}{A \vdash_{\mathrm{G}} \lambda x.e : \tau \to \tau'} \qquad (\textsc{Abs}_{\mathrm{G}})$$

$$\frac{A \vdash_{\mathrm{G}} e_1 : \tau_2 \to \tau \qquad A \vdash_{\mathrm{G}} e_2 : \tau_2}{A \vdash_{\mathrm{G}} e_1\, e_2 : \tau} \qquad (\textsc{App}_{\mathrm{G}})$$

$$\frac{A \vdash_{\mathrm{G}} e : \tau \quad A' \leq A \quad \tau \leq \tau'}{A' \vdash_{\mathrm{G}} e : \tau'} \qquad (\textsc{Sub}_{\mathrm{G}})$$

Figure 16.2: "Ground" typing rules

of a variable α contains a variable which must be generalized, then so must α. This analysis is interesting, since it allows avoiding some duplication, at a small cost. However, note that it is less powerful than the abbreviation mechanism described previously. Indeed, here, if a term contains a polymorphic leaf, then the whole path which leads from the term's root to that leaf shall be duplicated. On the contrary, if an abbreviation is introduced, the leaf becomes a parameter of it, and the path remains internal to the abbreviation; thus, only the leaf shall be duplicated. Hence, the introduction of abbreviations still makes sense in this new system.

16.4.2 Correctness

Before delving into the problems posed by system $\vdash_{\mathrm{ML}}$, one should verify that it is correct with respect to the semantics. This is necessary to ensure that the rules of figure 16.1 make sense, and that their study is worthwhile.

We shall only give an outline of the proof of correctness. The method proposed here differs from the one adopted in chapter 5. We could have proceeded in a similar way, by first weakening rule ($\textsc{Sub}_{\mathrm{ML}}$) to obtain a system of greater simplicity, but still complete with respect to the initial one, then by proving that typings are preserved by reduction in this second system. The approach used here provides less satisfactory results: in particular, it only proves the system's correctness, not the preservation of typings through reduction. However, it is based on an interesting intuition, which seems to justify its adoption here.

The principle of the proof consists in showing that if a typing judgement holds in system $\vdash_{\mathrm{ML}}$, then any ground instance of it also holds. By ground instance, we mean a typing judgement obtained by applying a substitution to the initial judgement, and stated in a system which involves ground types only. Since this second system, which is extremely simple, is correct, it then suffices to verify that any judgement has at least one ground instance to derive the correctness of system $\vdash_{\mathrm{ML}}$.

Figure 16.2 defines this auxiliary type system, which we shall call "ground". The types involved by these rules are exclusively ground types, containing no type variables; so, rule ($\textsc{Sub}_{\mathrm{G}}$) directly uses the subtyping relation $\leq$. It is easy to verify that typing is preserved by reduction in this system. Of course, it has no notion of polymorphism, and thus does not support the `let` construct, which shall be explicitly expanded by our proof.

Definition 16.1 *The* `let`-expansion *operation is defined by*

$$\begin{aligned} \mathrm{LE}(x) &= x \\ \mathrm{LE}(\lambda x.e) &= \lambda x.\mathrm{LE}(e) \\ \mathrm{LE}(e_1\, e_2) &= \mathrm{LE}(e_1)\,\mathrm{LE}(e_2) \\ \mathrm{LE}(\texttt{let}\, X = e_1 \;\texttt{in}\; e_2) &= [\mathrm{LE}(e_1)/X]\,\mathrm{LE}(e_2) \end{aligned}$$

We can now state the main lemma.

Lemma 16.1 *Assume $C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e : \tau$. Let ρ be a solution of C. Suppose given an environment A_0 and a substitution g, of domain $\mathrm{dom}(\Gamma)$, such that:*

- $A_0 \leq \rho(A)$;
- $\forall X \in \mathrm{dom}(\Gamma) \quad \forall \theta \quad (\mathrm{dom}(\theta) = \bar{\beta}_X \wedge \rho\theta \vdash C_X) \Rightarrow A_0 \vdash_{\mathrm{G}} g(X) : \rho\theta(\tau_X)$.

(Here, we note $\Gamma(X) = \forall \bar{\beta}_X \mid C_X . \tau_X$, and θ ranges over ground substitutions.) Then

$$A_0 \vdash_{\mathrm{G}} g(\mathrm{LE}(e)) : \rho(\tau)$$

Proof. By induction on the derivation of $C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e : \tau$. The proof presents no particular difficulty, so we shall omit it for the sake of brevity. The second condition above expresses the fact that the substitution g constitutes a satisfactory implementation of the environment Γ; indeed, each expression $g(X)$ admits all of the ground types obtained by instantiating the type scheme $\rho(\Gamma(X))$. Introducing the parameter A_0, together with the first condition, is made necessary by the presence of the subtyping rule. If one imposes $A_0 = \rho(A)$, the statement remains of course correct, since we just selected a particular case of it, but a direct induction proof is no longer possible. □

As a corollary, we obtain the following theorem:

Theorem 16.1 *Assume $C \Rightarrow \varnothing, \varnothing \vdash_{\mathrm{ML}} e : \tau$. Let ρ be a solution of C. Then*

$$\varnothing \vdash_{\mathrm{G}} \mathrm{LE}(e) : \rho(\tau)$$

Proof. Immediate, by setting A_0 to the empty environment and g to the substitution of empty domain. □

Since system $\vdash_{\mathrm{G}}$ is correct, this result entails that, if $C \Rightarrow \varnothing, \varnothing \vdash_{\mathrm{ML}} e : \tau$ can be proved and if C has a solution, then the expression $\mathrm{LE}(e)$ causes no execution errors. In other words, provided e and $\mathrm{LE}(e)$ have the same semantics, system $\vdash_{\mathrm{ML}}$ is correct.

Note that, given our definition of `let`-expansion, one must choose a call-by-name semantics for e and $\mathrm{LE}(e)$ to have the same semantics. If one prefers to use a call-by-value semantics, one can define

$$\mathrm{LE}(\texttt{let}\, X = e_1 \;\texttt{in}\; e_2) = [\mathrm{LE}(e_1)/X]\,((\lambda\_.\mathrm{LE}(e_2))\,X)$$

To preserve lemma 16.1, one must then add to rule ($\mathrm{L{\small ET}_{ML}}$) the premise $C_2 \Vdash C_1$. Thus, one accounts for the effect on the environment of the constraints coming from the expression e_1. (The same choice arises in our system, and has been mentioned in section 5.3.)

To conclude, the result established here shows that our formulation of system $\vdash_{\mathrm{ML}}$ is satisfactory. The proof method used here has drawbacks: in particular, we haven't proved that typings are preserved through reduction, and the use of `let`-expansion forbids the addition of imperative features. Nevertheless, it seemed interesting to emphasize the relationship with a "ground" type system, rather than to repeat the proof process used in chapter 5.

Note, by the way, that a third proof method is conceivable. One might attempt to prove the equivalence between the judgements $\varnothing \vdash e : \varnothing \Rightarrow \tau \mid C$ and $C \Rightarrow \varnothing, \varnothing \vdash_{\mathrm{ML}} e : \tau$. This result implies that both systems accept the same programs, and allows transferring the subject reduction theorem from the original system to system $\vdash_{\mathrm{ML}}$. We haven't studied this possibility closely, but it also seems viable.

16.4.3 Comments

All rules of figure 16.1 are classic, except rule (SUB$_{\mathrm{ML}}$), which deserves an in-depth study. Generally speaking, a subtyping rule allows weakening a typing judgement, by replacing it with a less precise one. Thus, it directly stems from a judgement comparison relation. In the system presented in chapter 5, judgements are of the form $\Gamma \vdash e : \sigma$. Rule (SUB) allows no modifications to the environment Γ; so, it directly stems from the type scheme comparison relation. It can be written

$$\frac{\begin{array}{c}\Gamma \vdash e : A \Rightarrow \tau \mid C \\ \forall \rho' \vdash C' \quad \exists \rho \vdash C \quad \rho'(A') \leq \rho(A) \wedge \rho(\tau) \leq \rho'(\tau')\end{array}}{\Gamma \vdash e : A' \Rightarrow \tau' \mid C'} \ (\textsc{Sub})$$

So, the rule (SUB$_{\mathrm{ML}}$) presented here is a generalization to the case where the environment contains open type schemes. The difference with rule (SUB) is the appearance of the condition $\rho'(\Gamma') \leq^{\forall} \rho(\Gamma)$. It is a point-wise comparison of the two environments. Since they contain type schemes, a scheme comparison relation has to be used, rather than the plain subtyping relation. Thus, if one notes $\Gamma(X) = \forall \bar{\beta}_X \mid C_X . \tau_X$ and $\Gamma'(X) = \forall \bar{\beta}'_X \mid C'_X . \tau'_X$, then the condition $\rho'(\Gamma') \leq^{\forall} \rho(\Gamma)$ can be written

$$\begin{array}{lll}\forall X \in \mathrm{dom}(\Gamma) & \forall \theta & (\mathrm{dom}(\theta) = \bar{\beta}_X \wedge \rho\theta \vdash C_X) \quad \Rightarrow \\ & \exists \theta' & (\mathrm{dom}(\theta') = \bar{\beta}'_X \wedge \rho'\theta' \vdash C'_X \wedge \rho'\theta'(\tau'_X) \leq \rho\theta(\tau_X))\end{array}$$

The second premise of rule (SUB$_{\mathrm{ML}}$) is thus a logical assertion quantified by $\forall\exists\forall\exists$, whereas our rule (SUB) only required $\forall\exists$. Thus, we have reached a new degree of complexity.

One might object that rule (SUB$_{\mathrm{ML}}$) allows replacing the environment Γ with a new environment Γ', which was not authorized by rule (SUB); that this is the cause of the problem, and that it suffices to remove this feature to solve it. We answer this objection with three remarks.

First, if rule (SUB) does not allow modifying the environment Γ, it is because such modification is not at all necessary in practice. Indeed, when rule (LET) is applied to introduce a new type scheme into the environment, one can simplify it by applying rule (SUB). Thus, the environment shall contain, at any time, fully simplified type schemes, and it shall not be necessary to modify it. In the case of system $\vdash_{\mathrm{ML}}$, one can also simplify each scheme upon entry into the environment. However, type schemes are now *open*; hence, the creation of new constraints can open new simplification opportunities in the environment. Thus, one might want the subtyping rule to allow modifying the environment.

Second, even if one accepts a subtyping rule incapable of modifying the environment, restricting rule (SUB$_{\mathrm{ML}}$) to the case where $\Gamma = \Gamma'$ does not suffice to solve the problem, because this does not make the condition $\rho'(\Gamma') \leq^{\forall} \rho(\Gamma)$ trivial. Indeed, the condition then becomes

$$\forall X \in \mathrm{dom}(\Gamma) \quad \rho'(\Gamma(X)) \leq^{\forall} \rho(\Gamma(X))$$

If the scheme $\Gamma(X)$ contains free variables, then $\rho'(\Gamma(X))$ is not necessarily equal to $\rho(\Gamma(X))$. To guarantee the equality, one can add, for each variable $\alpha \in \mathrm{fv}(\Gamma)$, the condition $\rho(\alpha) = \rho'(\alpha)$. The subtyping rule thus obtained uses an assertion of the same kind as the one which appears in rule (SUB). Polarity computation and garbage collection are thus performed as in our system, except that any variable $\alpha \in \mathrm{fv}(\Gamma)$ receives polarity $\pm$. Thus, this restriction leads to simplification algorithms close to those presented in this thesis, and easy to implement. However, the fact that all variables free in the environment are bipolar is a problem. First, it breaks the mono-polarity invariant, whose benefits we lose. Besides, it makes garbage collection coarser, since the direction of data flow is not fully accounted for; one then moves closer to a plain elimination of unreachable variables. So, though this solution is worth mentioning in passing for its simplicity, it is not satisfactory.

Third, the complexity of rule ($\mathrm{SUB_{ML}}$) can be found again, in part, in the problem of comparison between modules and interfaces. Recall that the issue is to compare an inferred type scheme with the one supplied by the user in the signature. In the system studied throughout this thesis, all schemes are closed; so, the problem is identical to the one posed by rule (SUB). In system $\vdash_{\mathrm{ML}}$, the scheme which appears in the signature must be closed, to guarantee the safety of the separate compilation system; however, the inferred scheme isn't. So, the problem presents itself as an $\exists\forall\exists$ assertion, *a priori* less complex than the $\forall\exists\forall\exists$ assertion of rule ($\mathrm{SUB_{ML}}$), but already more difficult than the $\forall\exists$ problem studied in chapter 9. This is one more argument in favor of studying the general problem. In case of success, the algorithm thus designed shall be useful not only from a theoretical point of view, to help understand rule ($\mathrm{SUB_{ML}}$) and design simplification algorithms, but also from a practical point of view, to perform the comparison between modules and interfaces.

So, though system $\vdash_{\mathrm{ML}}$ has a very natural formulation, which in particular avoids the use of λ-lifting and sports a generalization/instantiation mechanism identical to that of ML, it requires studying more complex assertions, which is why we did not adopt it. Recall that in a system based on subtyping, simplification algorithms directly stem from the form of the subtyping rule. Thus, in chapter 9, we studied an (incomplete) algorithm to compare typing judgements. From this algorithm, we then deduced, in a very natural way, the notions of polarity and garbage collection. However, it is not easy to extend this algorithm to deal with the judgements of system $\vdash_{\mathrm{ML}}$.

Recall that the problem is to decide an assertion of the following form (using, for the sake of clarity, a slightly informal notation):

$$\forall\bar{\alpha}' \quad \exists\bar{\alpha} \quad \forall\bar{\beta}' \vdash C' \quad \exists\bar{\beta} \vdash C + (\beta \leq \beta')$$

(In the case of a module-interface comparison, one has $\bar{\alpha}' = \varnothing$, and the problem is slightly simpler.) Here is a sketch of an algorithm. First, one weakens the assertion by exchanging the central quantifiers:

$$\forall\bar{\alpha}'\bar{\beta}' \vdash C' \quad \exists\bar{\alpha}\bar{\beta} \vdash C + (\beta \leq \beta')$$

One can apply to this assertion the algorithm developed in chapter 9. If it succeeds, then it yields a constraint graph which describes the values of $\bar{\alpha}\bar{\beta}$ as a function of the values of $\bar{\alpha}'\bar{\beta}'$. One then tries, by possibly reinforcing the constraints present in this graph, to eliminate the dependencies between $\bar{\alpha}$ and $\bar{\beta}'$, i.e. to express $\bar{\alpha}$ as a function of the $\bar{\alpha}'$ only. To do so, a possible approach consists in computing the extremal values of each β', as a function of $\bar{\alpha}'$, when $\bar{\beta}'$ vary. An undesirable dependence on a variable β' can then be eliminated by replacing the latter with its extremal value.

Of course, this description is informal and incomplete. Everything remains to be done: clear up the technical details, prove correctness, determine whether the algorithm thus obtained has sufficient power in practice, lastly deduce the associated notions of polarity and garbage collection. We haven't studied this problem deeply enough to say anything more here, but it is probably an interesting research topic.

Conclusion

The time has come to put an end to this work. Let us first recall the path followed during these last few years, while mentioning a few historical detours which do not appear in this study.

Our starting point is the type inference algorithm published by Eifrig, Smith and Trifonov [15]. It contains the principle of constraint-based analysis, as well as the closure algorithm. This system already accepts the same set of programs as the one presented in this thesis; however, its subtyping rule is extremely rudimentary, because it is based on plain inclusion between constraint sets; thus, it does not allow any form of simplification.

Our first goal was to increase the power of the subtyping rule, which we did using the entailment relation. In [15], the correctness of the type system is proved in a fully syntactic way, based solely on the definition of closure, without using the notion of solution of a constraint set. We followed this path for a while, and our initial definition of entailment was also purely syntactic. Instead of considering all solutions of the hypotheses set, it would envision all possible contexts, where a context is itself a constraint set. One would thus obtain a more "observational" definition of entailment. However, this method lead to very heavy proofs; so, we preferred to introduce the lattice of ground types, and the notion of solution, to simplify the theory. Both approaches are presented in [42]; only the second one appears in this thesis.

Equipped with the notion of entailment, we improved the power of the subtyping rule, and used it to justify certain substitution-based simplifications. A substitution was said to be valid if one could come back to the initial judgement, from the simplified one, by applying the subtyping rule. In practice, we would then use a heuristic to propose plausible substitutions, and our axiomatization of entailment to check their validity. A considerable effort has been dedicated to designing a complete entailment algorithm, by us and by several other teams, without success to date. From a theoretical point of view, this remains an interesting open problem.

At the same time, we proposed an algorithm to eliminate unreachable constraints, of very simple design. However, from a theoretical point of view, its correctness could not be proved by a mere use of the subtyping rule; a meta-theorem, based on a rewriting of typing derivations, was necessary. Trifonov and Smith [46] solve this problem by proposing a more general version of the subtyping rule, based on polymorphic scheme subsumption, which is the one found in this thesis. At the same time, they enhance our algorithm by introducing the notion of polarity, thus obtaining the garbage collection algorithm.

However, the simplification problem is still not solved. First, the use of heuristics to choose substitutions is extremely costly. Designing a heuristic requires a difficult compromise between expected cost and benefit; furthermore, because the distinction between efficiency and readability goals is not clearly understood yet, some heuristics meant to enhance readability cause an efficiency loss. Second, the various components of the system are not well integrated yet; in particular, canonization and garbage collection produce constraint sets which are not closed, thus requiring extra closure phases.

Here, we solve the first of these problems by introducing the minimization algorithm, together with the small terms invariant. All heuristics are thus eliminated, in favor of an efficient algorithm, capable of performing at once a quasi-optimal substitution. Adopting the small terms invariant, which is necessary for the algorithm to function properly, favors the efficiency goal throughout the inference process. The readability goal is recognized as secondary, and the "external" simplification phase shall be performed only immediately before display.

The second problem, namely the integration between the various algorithms, has also been eliminated. First, we give a precise description of the canonization algorithm, which produces a (simply) closed constraint graph, and we optimize it by combining it with a partial garbage collection phase. Next, we modify the type inference rules to respect the mono-polarity invariant, which is probably the simplest way of ensuring that garbage collection yields a closed graph.

We think that the inference and simplification system thus obtained forms a homogeneous and well understood whole. In practice, the system is simple. It is made up of a small number of components: constraint generation engine (based on the inference rules), incremental closure algorithm, canonization, garbage collection and minimization algorithms. The implementation of these algorithms corresponds exactly to the descriptions—and to the proofs—supplied in this theory, and their integration is perfect, since each one produces data acceptable for its successor. In the setting of separate compilation, comparing modules with interfaces requires the scheme subsumption algorithm proposed by Trifonov and Smith, of which we supplied a detailed proof.

However, all is not satisfactory. First, imperative constructs are typed in a slightly indirect way, especially in the case of separate compilation. Second, our implementation's performance, although largely superior to what was possible at the onset of this work, is not quite sufficient to deal with large programs. An in-depth study of an "ML-like" type system could contribute to solve both of these problems; furthermore, it seems to pose interesting theoretical problems. Hence, we hope to continue researching this topic. Besides, one can imagine other solutions to the efficiency problem, among which the use of abbreviations, or the design of incremental versions of our algorithms. Thus, this domain still offers plenty of paths to explore.

Bibliography

URLs have been provided where possible. They are current as of late 1997. An electronic version of this bibliography should be available online, at `http://pauillac.inria.fr/~fpottier/biblio/these-en.html`.

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects — untyped and first-order systems. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, April 1994. URL: `http://research.microsoft.com/research/cambridge/luca/Papers/PrimObj1stOrder.A4.ps`.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects — second-order systems. In D. Sannella, editor, *Proc. of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 1–25, New York, N.Y., 1994. Springer Verlag. URL: `http://research.microsoft.com/research/cambridge/luca/Papers/PrimObj2ndOrder.A4.ps`.

[3] Alexander S. Aiken. The Illyria system. URL: `http://http.cs.berkeley.edu:80/~aiken/Illyria-demo.html`.

[4] Alexander S. Aiken and Manuel Fähndrich. Making set-constraint based program analyses scale. Technical Report CSD-96-917, University of California, Berkeley, September 1996. URL: `http://http.cs.berkeley.edu/~manuel/papers/scw96.ps.gz`.

[5] Alexander S. Aiken and Edward L. Wimmers. Solving systems of set constraints. In Andre Scedrov, editor, *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, Santa Cruz, CA, June 1992. IEEE Computer Society Press. URL: `http://http.cs.berkeley.edu/~aiken/ftp/lics92.ps`.

[6] Alexander S. Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming & Computer Architecture*, pages 31–41. ACM Press, June 1993. URL: `http://http.cs.berkeley.edu/~aiken/ftp/fpca93.ps`.

[7] Alexander S. Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Principles of Programming Languages*, pages 163–173, January 1994. URL: `http://http.cs.berkeley.edu/~aiken/ftp/popl94.ps`.

[8] Alexander S. Aiken, Edward L. Wimmers, and Jens Palsberg. Optimal representations of polymorphic types with subtyping. Technical Report CSD-96-909, University of California, Berkeley, July 1996. URL: `http://http.cs.berkeley.edu/~aiken/ftp/quant.ps`.

[9] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993. URL: `http://research.microsoft.com/research/cambridge/luca/Papers/SRT.A4.ps`.

[10] André Arnold and Maurice Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticæ*, 3(4):181–205, 1980.

[11] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66. URL: `http://research.microsoft.com/research/cambridge/luca/Papers/Inheritance.A4.ps`.

[12] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 13–27. ACM, ACM, August 1986.

[13] Bruno Courcelle. Fundamental properties of infinite trees. *Theoret. Comput. Sci.*, 25(2):95–169, March 1983.

[14] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95 Conference Proceedings*, volume 30(10) of *ACM SIGPLAN Notices*, pages 169–184, 1995. URL: `http://www.cs.jhu.edu/~trifonov/papers/sptio.ps.gz`.

[15] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics, New Orleans*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. URL: `http://www.elsevier.nl/locate/entcs/volume1.html`.

[16] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander S. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 Conference on Programming Languages Design and Implementation*, Montréal, June 1998. To appear. URL: `http://www.cs.berkeley.edu/~manuel/papers/pldi98.ps`.

[17] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, May 1997. URL: `http://www.cs.rice.edu/CS/PLT/Publications/thesis-flanagan.ps.gz`.

[18] Cormac Flanagan and Matthias Felleisen. Modular and polymorphic set-based analysis: Theory and practice. Technical Report TR96-266, Rice University, November 1996. URL: `http://www.cs.rice.edu/CS/PLT/Publications/tr96-266.ps.gz`.

[19] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 235–248, Las Vegas, Nevada, June 1997. URL: `http://www.cs.rice.edu/CS/PLT/Publications/pldi97-ff.ps.gz`.

[20] Alexandre Frey. Satisfying subtype inequalities in polynomial space. In Pascal Van Hentenryck, editor, *Proceedings of the Forth International Symposium on Static Analysis (SAS'97)*, number 1302 in Lecture Notes in Computer Science, pages 265–277, Paris, France, September 1997. Springer Verlag. URL: `http://www.ensmp.fr/~frey/Publications/SAS97.ps.gz`.

[21] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

[22] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 2*, volume 352 of *LNCS*, pages 167–183, Berlin, March 1989. Springer.

[23] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, June 1972.

[24] Juan Carlos Guzmán and Ascánder Suárez. An extended type system for exceptions. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, number 2265 in INRIA Research Reports, pages 127–135. INRIA, BP 105, 78153 Le Chesnay Cedex, France, June 1994. URL: `http://www.ldc.usb.ve/~suarez/PAPERS/except.ps`.

[25] Nevin Heintze. Set based analysis of ML programs. Technical Report CMU-CS-93-193, Carnegie Mellon University, School of Computer Science, July 1993. URL: `ftp://reports.adm.cs.cmu.edu/usr/anon/1993/CMU-CS-93-193.ps`.

[26] Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. In Benjamin Pierce, editor, *Proc. 4th Int'l Workshop on Foundations of Object-Oriented Languages (FOOL), Paris, France*, January 1997. URL: `http://www.cs.indiana.edu/hyplan/pierce/fool/henglein.ps.gz`.

[27] Fritz Henglein and Jakob Rehof. Constraint automata and the complexity of recursive subtype entailment. In *25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, July 1998. To appear.

[28] My Hoang and John C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of the 22nd Symposium on Principles of Programming Languages (POPL'95)*, pages 176–185, New York, NY, USA, January 1995. ACM Press.

[29] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, NY, 1971.

[30] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[31] Deepak Kapur and Hantao Zhang. An overview of Rewrite Rule Laboratory (RRL). *J. Comput. Appl. Math.*, 29(2):91–114, 1995. URL: `ftp://ftp.cs.albany.edu/pub/ipl/papers/overview.rrl.ps.gz`.

[32] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proceedings POPL '93*, pages 419–428, 1993. URL: `ftp://ftp.daimi.aau.dk/pub/palsberg/papers/popl93.ps.Z`.

[33] Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris VII, June 1992. URL: `http://pauillac.inria.fr/~xleroy/publi/these-doctorat.ps.gz`.

[34] David B. MacQueen, Gordon D. Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1–2):95–130, October–November 1986.

[35] John C. Mitchell. Coercion and type inference. In *11th Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[36] Joachim Niehren, Martin Müller, and Andreas Podelski. Inclusion constraints over non-empty sets of trees. In Max Dauchet, editor, *Theory and Practice of Software Development, International Joint Conference CAAP/FASE/TOOLS*, volume 1214 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1997. URL: `ftp://ftp.ps.uni-sb.de/pub/papers/ProgrammingSysLab/ines97.ps.Z`.

[37] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, December 1987.

[38] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. URL: `http://www.cs.purdue.edu/homes/palsberg/paper/ic95-p.ps.gz`.

[39] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.* ACM, January 1995. URL: `ftp://ftp.daimi.aau.dk/pub/palsberg/papers/popl95.ps.Z`.

[40] Jens Palsberg and Scott Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems*, 18(5):519–527, September 1996. URL: `http://www.cs.purdue.edu/homes/palsberg/paper/toplas96-ps.ps.gz`.

[41] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. URL: `http://www.cs.indiana.edu/hyplan/pierce/ftp/fsubpopl.ps.gz`.

[42] François Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133, January 1996. URL: `http://pauillac.inria.fr/~fpottier/publis/ICFP96.ps.gz`.

[43] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* MIT Press, 1993. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz`.

[44] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, pages 40–53, Paris, France, January 1997. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/objective-ml!popl97.ps.gz`.

[45] Jerzy Tiuryn. Subtype inequalities. In Andre Scedrov, editor, *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 308–317, Santa Cruz, CA, June 1992. IEEE Computer Society Press.

[46] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. SV, September 1996. URL: `http://www.cs.jhu.edu/~trifonov/papers/subcon.ps.gz`.

[47] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report 93-200, Rice University, February 1993.

[48] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

[49] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994. URL: `http://www.cs.rice.edu/CS/PLT/Publications/ic94-wf.ps.gz`.

Index