Principal typings and type inference

by

Trevor Jim

M.Sc. Computer Science
Massachusetts Institute of Technology, 1991

B.S.E. Electrical Engineering and Computer Science
Princeton University, 1987

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

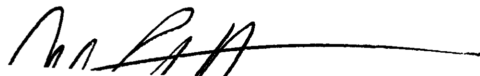
Doctor of Philosophy

at the

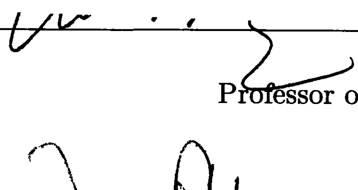
Massachusetts Institute of Technology

February 1996

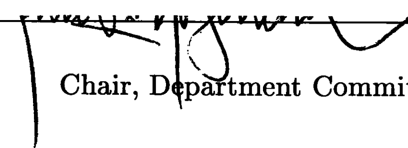
© 1996 Massachusetts Institute of Technology. All rights reserved.



Signature of author: ______________________________
Department of Electrical Engineering and Computer Science
January 12, 1996



Certified by: ______________________________
Albert R. Meyer
Professor of Computer Science and Engineering
Thesis Supervisor



Accepted by: ______________________________
F.R. Morgenthaler
Chair, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

APR 11 1996

LIBRARIES

Principal typings and type inference

by

Trevor Jim

Submitted to the
Department of Electrical Engineering and Computer Science
on January 12, 1996
in Partial Fulfillment of the Requirements
of the Degree of Doctor of Philosophy in Computer Science

ABSTRACT

Notions of principal typing are central to the design of type inference algorithms. We identify a fundamental distinction that divides notions of principal typing into two styles. We show that one style of principal typing provides natural support for such practical applications as the typing of recursive definitions, incremental type inference, and the separate compilation of programs. The other style, exemplified by ML, does not.

Many type systems have no notion of principal typings, making type inference difficult. We develop two techniques for type inference in such systems. First, we give two examples in which a type system without principal typings is equivalent, in a certain sense, to another type system having principal typings. A type inference algorithm for the first system can then be derived from the algorithm for the second system. Second, we give two examples of how type inference in a system lacking principal typings in general can be reduced to type inference for a restricted class of programs having principal typings.

Thesis Supervisor: Albert R. Meyer
Title: Professor of Computer Science and Engineering

Acknowledgements

I was fortunate to have Albert Meyer as my advisor at M.I.T. Albert is a dynamic and elegant mathematician, and I thank him for his insights and his encouragement in my time here.

For both his technical expertise and personal kindness I thank Assaf Kfoury, who graciously served as an outside reader on this thesis. Assaf's comments and suggestions have improved this thesis in many ways.

Dave Gifford suggested many improvements as a thesis reader, and I gained invaluable experience by twice teaching the graduate programming languages course at M.I.T. with him.

Before arriving at M.I.T., I worked with Andrew Appel and Dave MacQueen compiling Standard ML. That work started me on the research that led to this thesis.

All of the people mentioned above provided financial support at some time during my studies at M.I.T., and Arvind funded my final term here. This work was supported by NSF grants CCR–9113196 and CCR–9417382, and ONR Contract N00014–92–J–1310.

I've learned a great deal at M.I.T. through teaching and watching others teach, including Franklyn Turbak, Brian Riestad, John Leo, Nancy Lynch, Joanna Bryson, and Ian Eslick.

Among the many friends and colleagues I've had over the years, I thank Margrit Betke, Mojdeh Mohtashemi, Jane Ko, Cliff Stein, Phill Swagel, Shimi Anisfeld, Francis Lee, Javed Aslam, Bobby Blumofe, Charles Isbell, Alex Ishii, Bruce Maggs, Mark Smith, Mic Grigni, Esther Jesurum, Joe Kilian, Dina Kravets, Isaac Saias, Donna Slonim, Stan Chung, Rich Halberstein, Bard Bloom, Mike Ernst, Jakov Kučan, Arthur Lent, Jens Palsberg, and Jon Riecke. I would like to thank Tom Cormen for taking longer than I to graduate.

No single person has done more to make the Theory Group at M.I.T. a pleasant place than Be Hubbard. Thanks are also due David Jones, Bruce Dale, William Ang, and Scott Blomquist for their help.

I don't think I could have completed this thesis without the support of Mona Singh.

Finally, I thank my parents, grandparents, and my brother and sister, for their long wait, and for their love and support over many years.

Contents

1 Introduction **9**

2 Principal typings **13**

2.1 Preliminaries 16
2.2 The type system $\mathbf{P}_2$ 17
2.3 Subtype satisfaction 21
2.4 Type inference 25
2.5 Recursive definitions 27
2.6 Mutual recursion 28
2.7 Separate compilation 31
2.7.1 Incremental type inference 31
2.7.2 Smartest recompilation 33
2.8 Error messages 35
2.9 Does ML have principal typings? 36
2.10 Living without principal typings 38
2.11 Related work 39

3 Rank 2 type systems **41**

3.1 The rank 2 intersection type system 45
3.2 Rank 2 of System F 46
3.3 ML 47
3.4 Relationship of Λ_2^{s} and $\mathbf{I}_2^{\mathrm{s}}$ 48
3.5 Type inference for $\mathbf{I}_2^{\mathrm{s}}$ 56
3.6 Type inference for Λ_2^{s} 61
3.7 Other systems of rank 2 intersection types 62
3.7.1 A restriction of $\mathbf{I}_2^{\mathrm{s}}$ 62
3.7.2 An extension of $\mathbf{I}_2^{\mathrm{s}}$ 63
3.8 Recursive definitions 64
3.9 The systems $\mathbf{P}_2$ and $\mathbf{P}_2^{\mathrm{R}}$ 65
3.9.1 Preliminaries 66
3.9.2 Soundness 67

3.9.3 Completeness 69
3.9.4 Comparison with the other rank 2 systems 73

4 Type inference with subtyping **77**
4.1 Recursive types with subtyping 78
4.2 Lambda calculi with subtyping 80
4.3 Constraint-based type systems 86
4.4 Relationship of $\Lambda_{\leq}$ and $\Lambda_{\Vdash}$ 90
4.5 Entailment and Principal Typings 91
4.6 An example 92

5 Type inference without principal typings **95**
5.1 System F_{η} 97
5.2 Principal typings for a restricted set of terms 99
5.3 Principal typings for abstractions 103
5.4 From type inference to subtype satisfaction 106
5.5 From subtype satisfaction to subtyping 109
5.6 From type checking closed terms to subtyping 112
5.7 Type inference for Λ_{μ}, revisited 112

Bibliography **115**

Index **121**

Chapter 1

Introduction

Modern programming languages provide for the definition and manipulation of complex data structures: trees, sets, lists, and so on. In these languages the concept of *type* takes on a central role. For the language implementor, the type of a value determines its machine representation. For the programmer, types provide valuable documentation, and automated type checking may eliminate a large class of bugs.

An implementor needs to know the type of every value computed by a program in order to translate the program into machine code. At the same time, it would be an excessive burden to require the programmer to annotate every program phrase with its type. *Type inference* is the automated construction of a *typing*—a fully type-annotated program—from a program with few or no type annotations. It permits concise programs while providing all of the safety and efficiency of fully typed programs. The process of type inference is the subject of this thesis.

Most type inference algorithms use a *compositional*, or "divide-and-conquer" strategy: the typing of the whole program is constructed by finding the typings of the parts, and combining the results; and the typings of the parts are found by the same method, until the basic components of the program are reached. We are particularly interested in how to carry out this strategy in programming languages with a minimal degree of *polymorphism*. Polymorphism is a word with many technical meanings in type theory, but we use it here in its literal sense, to indicate that there are programs in the language with many possible typings.

This kind of polymorphism poses a problem for the compositional strategy. If there are many ways of typing the parts of a program, which should be chosen? A wrong choice could mean that the typings of the parts cannot be combined into a typing for the whole.

What is needed is for there to be a "best" choice, a typing from among all of the various typings of each program part that can be used successfully, if any can. We will call such a typing a *principal typing*. In the ideal case, every possible program fragment has a principal typing, and the compositional algorithm simply choses the principal typing at each step. Whether or not such a typing exists for every program

fragment depends on the type system of the language, as does the sense in which it is better than other typings.

All of this is well-known in the type inference community, where this strategy has been successfully applied to many different type systems. What is not generally recognized is that there are certain fundamental distinctions in the various resulting notions of what it means to be the "best" typing, distinctions arising from more than just the obvious differences in type systems: the presence or absence of typing features such as polymorphism, overloading, effects, subtyping, and so on. In particular, consider the following two properties of type systems.

Property A

Given: a term M typable in type environment A.
There exists: a type σ representing all possible types for M in A.

Property B

Given: a typable term M.
There exists: a typing $A \vdash M : \sigma$ representing all possible typings of M.

The properties are more alike than not. Each spells out a notion of best typing and asserts that every program fragment has such a typing—exactly what is needed to apply the compositional strategy. But they are distinct properties. Some type systems have Property A but not B, some have Property B but not A, some have both, and some have neither.

There are two reasons why the distinction is worth pointing out. First, historically the two have been confused. Partly this is due to a lack of standard terminology. Both properties have been called by many names—sometimes the same name is used for both. More seriously, some authors have mistaken one property for the other. For example, there have been claims that ML possesses Property B, and we know of no way in which this is true.

The second and more important reason for distinguishing the properties is that they lead to two different type inference algorithms. Property A leads to an algorithm taking two inputs, a term M and a type environment A. This is exactly the strategy of Milner's type inference algorithm for ML, and indeed, ML's *principal type property* is a version of Property A.[1]

In contrast, Property B leads to an algorithm that takes a single input, a term M, and produces two outputs, the A and σ of the typing $A \vdash M : \sigma$. The environment A specifies the types required of external variables referenced by M; but A is a byproduct of type inference, not a necessary input. This gives the algorithm of Property B an advantage in certain practical applications. Perhaps the most important example is *separate compilation*, in which a large program is broken up into

[1]However, the principal type property is not the basis of Milner's algorithm; a stronger property of similar character is needed, see §2.9.

modules, each of which is to be compiled and typed independently. The modules may refer to each other, that is, they may reference external variables. Thus the type and even the compiled machine code of one module may depend on the type of another module.

In separate compilation the algorithm of Property B enjoys two advantages. First, the algorithm of Property A requires the types of all external variables as input; in practice, the task of specifying these types is left to the programmer. With Property B this is unnecessary; the algorithm itself derives the types required of the external variables. The second advantage is in *recompilation*, arising from the edit/test cycle of program development. Certainly, when a module is edited it must be compiled again. But because the machine code of a module can depend on the types of external variables, it may also be necessary to recompile other, *unchanged* modules that reference the changed module. We will show that Property B avoids this problem: it allows us to achieve *smartest recompilation*, guaranteeing that only changed modules need be recompiled.

In addition to smartest recompilation, we will demonstrate the advantages of Property B in areas such as the typing of *recursive definitions*, the problem of *incremental type inference*, and the reporting of *accurate type error messages.*

All of this is well and good for type systems that have Property B, but what of type systems without it? There are many such systems, including ML. We will show that, in some cases, such a system can still benefit from our observations, by establishing an equivalence with a second type system that does have Property B. Essentially, we show that the first system has a weak form of Property B in which the typings of a program in the first system are represented by a principal typing in the second. Such an equivalence can be established for ML.

Furthermore, we will establish such equivalences for type systems with *neither* Property A nor Property B, and indeed, no notion of principal typing whatsoever. It is difficult to design type inference algorithms for such systems, much less implement the other applications we have mentioned. Our method results in type inference algorithms for two such systems, the *recursive type system* of Amadio and Cardelli, and a restriction of System F based on a notion of *rank.*

We are also interested in designing type inference algorithms without first discovering an equivalent system with principal typings. There may be type systems for which such an equivalence is not immediately evident, or even possible. A case in point is the extension of System F by Mitchell's subtyping relation; to the best of our knowledge, this system does not have a principal typing property like Property B or even Property A, and is not equivalent to a system with principal typings. Nevertheless, we can show that although not every program has a principal typing in the system, some programs do. We use principal typings for this subclass of programs to transform the type inference problem for arbitrary programs into a subtype satisfaction problem. By a recent result of Wells, this subtype satisfaction problem is undecidable. However, we are proposing a *methodology*, not an ad hoc algorithm;

the same methods will apply to other type systems. As a proof of concept, we use the methodology to design a second type inference algorithm for the Amadio-Cardelli system of recursive types mentioned above.

Overview of the thesis

In Chapter 2 we discuss the difference between the Properties A and B. We extend the system of rank 2 intersection types to define a type system, $\mathbf{P}_2$, that satisfies Property B, and show in detail how this provides better support for separate compilation, accurate type error messages, and typing recursive definitions.

In Chapter 3 we define a number of rank 2 type systems, including the system of rank 2 intersection types, and rank 2 of System F. We show that all of the rank 2 systems type the same set of terms. This equivalence allows us to derive a type inference algorithm for rank 2 of System F, which has no known notion of principal typing.

In Chapter 4, we present a general class of type systems with subtyping, and a single, compositional algorithm that reduces type inference in any such system to a subtype satisfaction problem. When subtype satisfaction is decidable, this gives a type inference algorithm, even for systems with no known notion of principal typing. We then define a class of type systems based on constraints, and show that our reduction gives principal typings in the constraint-based system. We then prove an equivalence between systems in the first class and systems in the second, constraint-based class.

In Chapter 5, we give a non-compositional algorithm for reducing the type inference problem into a subtype satisfaction problem. The type system we use is System F extended with Mitchell's subtyping relation. This system has no known notion of principal typing, and no known equivalent system with principal typings. As we have mentioned, subtype satisfaction is undecidable for this system. However, we show that our method applies to decidable systems by using it to construct a new type inference algorithm for the recursive type system of Chapter 4.

Chapter 2

Principal typings

In the introduction, we argued that a careful distinction should be made between the following two properties of type systems.

> **Property A**
> Given: a term M typable in type environment A.
> There exists: a type σ representing all possible types for M in A.

> **Property B**
> Given: a typable term M.
> There exists: a typing $A \vdash M : \sigma$ representing all possible typings of M.

To resolve the problem of terminology, we will call Property A the *principal type property*, as it is sometimes known in ML. By analogy, we will call Property B the *principal typing property.* This is not an ideal solution, as the names are close, and indeed, some authors have used "principal typings" in reference to Property A. But "principal typings" is also the name traditionally applied to Property B, and we do not wish to invent new terminology.

Why do we care to make such a distinction? Property A—principal types—is certainly useful. But Property B—principal typings—is more useful still. We believe this has been overlooked because ML and its extensions completely dominate current research on type inference; and we know of no sense in which ML has principal typings. This was already noted by Damas in his dissertation [13], but there have been subsequent claims that ML has the principal typing property, indicating that the distinction between principal types and principal typings is not widely appreciated.

In this chapter, we demonstrate the usefulness of the principal typing property by studying a type system that has it. We emphasize that our results are motivated entirely by the general principal typing property, and not by the technical details of this particular case study. Any system with principal typings can benefit from our observations.

Nevertheless, we take some care in choosing our case study, so that its relevance to current practice will be immediately evident. Therefore, we seek a type system closely related to ML: it should be able to type all ML programs, it should have decidable type inference, and the complexity of type inference should be approximately the same as in ML.

The type system that satisfies all of these requirements is the system of *rank 2 intersection types.* This system is closely related to the more well-known rank 2 of System F—we will show that they type exactly the same terms—but it possesses the additional property of principal typings. We use a variant of the intersection system, called $\mathbf{P}_2$, as our case study.

We illustrate the benefits of principal typings in three areas: *recursive definitions*, *separate compilation*, and *accurate type error messages.*

Recursive definitions. Two rules that have been used to type recursive definitions in ML are given below.

$$(\text{REC-SIMPLE}) \quad \frac{A \cup \{x : \tau\} \vdash M : \tau}{A \vdash (\mu x M) : \tau} \quad \tau \text{ is a simple type}$$

$$(\text{REC-POLY}) \quad \frac{A \cup \{x : \sigma\} \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \quad \sigma \text{ is an ML type scheme}$$

The rule (REC-SIMPLE) requires the body M of the recursive definition $(\mu x M)$ to be typed under the assumption that x has a simple type. This restriction is relaxed in (REC-POLY), the rule of *polymorphic recursion* [50, 31], which permits M to be typed under the assumption that x has a polymorphic type.

More terms are typable under (REC-POLY) than (REC-SIMPLE), and practical examples of programs requiring polymorphic recursion are a recurring topic on the ML mailing list. But (REC-SIMPLE) is used in practice, because type inference for (REC-POLY) is undecidable [32, 22]. To understand why, consider type inference using Milner's algorithm: in order to infer a type, σ, for the definition M, we need to know the type to use for the free variable x, that is, σ. In the case of (REC-POLY), this "chicken and egg" problem cannot be solved.

The principal typing property suggests a new rule for typing recursive definitions:

$$\frac{A \cup \{x : \tau\} \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \quad (\text{where } \sigma \leq \tau)$$

In this rule, the type τ assumed for the recursive variable x need not be the same as the type σ derived for its definition M. The type τ expresses the requirements on x needed to give M the type σ; as long as σ meets these requirements ($\sigma \leq \tau$), it is safe to assume it as the type of the definition.

Now the strategy for type inference becomes clear: infer the principal typing $A \vdash M : \sigma$ for M, producing *both* σ and $\tau = A(x)$. It only remains to ensure

$\sigma \leq \tau$, and this can be accomplished by *subtype satisfaction*, a procedure similar to unification.

When we use this strategy to type recursive definitions in $\mathbf{P}_2$, we obtain an interesting typing rule, lying between (REC-SIMPLE) and (REC-POLY): it is able to type some, but not all, examples of polymorphic recursion.

Separate compilation. In separate compilation, a large program is divided into smaller modules, each of which is type checked and compiled in isolation. The program as a whole is closed, but modules have free variables—a module may refer to other modules. Types play an important role in compilation; for instance, the data representations and calling conventions of a module may depend on its type. Thus the compiled machine code of a module may depend on the types of external variables that it references.

Consequently, most compilers require the user to specify the types of external variables referenced in each module. In $\mathbf{P}_2$, our ability to perform type inference on program fragments with free variables means that the user need not write these specifications: the compiler can infer them itself. More significantly, principal typings will enable us to achieve *smartest recompilation* [57], which guarantees that a module need not be recompiled unless its own definition changes. We also show that principal typings enable an elegant and efficient solution to a related problem, *incremental type inference* [1].

Error messages. Most compilers for strongly typed languages do not do a good job of pinpointing the location of type errors in programs; see Wand [66] for a discussion. As a final example of the utility of principal typings, we show that principal typings help to produce error messages that accurately identify the source of type errors.

Organization of the chapter. After some preliminary definitions (§2.1), we introduce the type system $\mathbf{P}_2$ in §2.2, and state its equivalence with rank 2 of System F. We describe how to solve subtype satisfaction for $\mathbf{P}_2$ types in §2.3, and we define the $\mathbf{P}_2$ type inference algorithm in §2.4. We describe how we type recursive definitions in §2.5 and §2.6, and we show how principal typings support separate compilation in §2.7. We describe how principal typings produce more accurate type error messages in §2.8. In §2.9, we address the question of whether principal typings exist for ML. We describe alternatives to principal typings in §2.10, and we discuss related work in §2.11.

For the sake of exposition we will delay most proofs until Chapter 3, where all of the rank 2 systems will be discussed in detail.

2.1 Preliminaries

We will be defining a number of type systems; here we develop machinery that will be useful in all of them.

We use $x, y, \ldots$ to range over a countable set of (term) variables, and $s, t, \ldots$ to range over a countable set, **Tv**, of type variables. The terms and types of the systems will vary, but in all cases we use $\sigma, \tau, \ldots$ to range over types, and M, N, ... to range over terms.

The *terms of the (pure) lambda calculus* are defined by the following grammar:

$$M ::= x \mid (M_1 M_2) \mid (\lambda x M)$$

Unless stated otherwise, terms are considered syntactically equal modulo renaming of bound variables. We adopt the usual conventions that allow us to omit parentheses: application associates to the left, and the scope of an abstraction 'λx' extends to the right as far as possible. We write $\lambda x_1 \cdots x_n.M$ for $(\lambda x_1(\cdots(\lambda x_n M)\cdots))$.

The types of our systems will all be subsets of the types with quantification and intersection:

$$\sigma ::= t \mid (\sigma_1 \to \sigma_2) \mid (\forall t \sigma) \mid (\sigma_1 \wedge \sigma_2)$$

By convention, '$\to$' associates to the right, so that, e.g., $(t \to (t \to t))$ may be written more compactly as $t \to t \to t$, and '$\wedge$' binds more tightly than '$\to$', e.g., $\sigma \wedge \tau \to t$ means $(\sigma \wedge \tau) \to t$. The scope of a quantifier '$\forall t$' extends as far to the right as possible. We write $(\forall \vec{t} \sigma)$ for the type

$$(\forall t_1(\forall t_2(\ldots(\forall t_n \sigma)\ldots))),$$

where $\vec{t} = t_1, t_2, \ldots, t_n$ and $n \geq 0$.

A *type environment* is a finite set $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ of (variable, type) pairs, where the variables $x_1, \ldots, x_n$ are distinct. We use A, B to range over type environments. We write $A(x)$ for the type paired with x in A, and $\mathbf{dom}(A)$ for the set $\{x \mid \exists \tau.(x : \tau) \in A\}$. We use $A \backslash x$ to denote the type environment A with any pair for the variable x removed, and if $X = \{x_1, \ldots, x_n\}$, then $A \backslash X = (\cdots(A \backslash x_1)\cdots)\backslash x_n$. We write $A_1 \cup A_2$ for the union of two type environments; by convention we assume that $\mathbf{dom}(A_1)$ and $\mathbf{dom}(A_2)$ are disjoint. For any set $\mathbf{T}$ of types, we say A is a $\mathbf{T}$ *type environment* if $A(x) \in \mathbf{T}$ for all $x \in \mathbf{dom}(A)$.

A preorder $\leq$ on types is extended to a preorder on type environments as follows: $A \leq B$ iff $\mathbf{dom}(B) \subseteq \mathbf{dom}(A)$ and $A(x) \leq B(x)$ for all $x \in \mathbf{dom}(B)$. Note that $A \leq A \backslash x$ for any A and x, and $A \backslash x \leq B \backslash x$ if $A \leq B$.

The notion of *free type variable* is defined as usual. We write $\mathrm{FTV}(\sigma)$ for the free type variables of a type σ, and $\mathrm{FTV}(A)$ for the free type variables of all types appearing in the type environment A. We write $\mathrm{Gen}(A, \sigma)$ for the $\forall$-closure of σ by the type variables free in σ but not A.

A *judgment* is a triple of a type environment, term, and type, written $A \vdash M : \sigma$. The *derived type* in a judgment $A \vdash M : \sigma$ is the type σ. We write $\mathscr{T} \triangleright A \vdash M : \sigma$ if the judgment $A \vdash M : \sigma$ is derivable in the type system $\mathscr{T}$, and say a term M is *typable* in $\mathscr{T}$ if $\mathscr{T} \triangleright A \vdash M : \sigma$ for some A and σ. A pair $\langle A, \sigma \rangle$ of a type environment and a type is called simply a *pair*. Two pairs $\langle A_1, \sigma_1 \rangle$ and $\langle A_2, \sigma_2 \rangle$ are *disjoint* if their free type variables are disjoint. An *acceptable pair of a term* M *in a type system* $\mathscr{T}$ is a pair $\langle A, \sigma \rangle$ such that $\mathscr{T} \triangleright A \vdash M : \sigma$. We write $\mathrm{AP}_{\mathscr{T}}(M)$ for the set of acceptable pairs of M in $\mathscr{T}$.

A *substitution* is a mapping from type variables to types that is the identity on all but a finite number of type variables. We use S, R, Q, U to range over substitutions. The *domain* and *range* of a substitution S are defined

$$\begin{aligned} \mathbf{dom}(S) &= \{t \mid St \neq t\}, \\ \mathbf{rng}(S) &= \bigcup_{t \in \mathbf{dom}(S)} \mathrm{FTV}(St). \end{aligned}$$

In particular note that $\mathbf{rng}(S)$ is always a set of type variables; this is standard in the unification community. If $\mathbf{dom}(S) = \{t_1, t_2, \ldots, t_n\}$ and $St_i = \tau_i$ for all i, then S can be written in the form $\{t_1 := \tau_1, \ldots, t_n := \tau_n\}$.

The application of substitutions is extended to types, type environments, and pairs in the usual way. The composition of substitutions is denoted by juxtaposition, so that $SRt = (SR)t = S(R(t))$. We say S_1 and S_2 are *disjoint* if $\mathbf{dom}(S_1)$ and $\mathbf{dom}(S_2)$ are disjoint sets. If S_1 and S_2 are disjoint, then the substitution $S_1 \cup S_2$ is defined as follows:

$$(S_1 \cup S_2)(t) = \begin{cases} S_1(t) & \text{if } t \in \mathbf{dom}(S_1), \\ S_2(t) & \text{if } t \in \mathbf{dom}(S_2), \\ t & \text{otherwise.} \end{cases}$$

For any set $\mathbf{T}$ of types, we say S is a $\mathbf{T}$ *substitution* if $S(t) \in \mathbf{T}$ for all $t \in \mathbf{dom}(S)$.

2.2 The type system $\mathbf{P}_2$

We now present our type system, in an expository manner. For the most part, the system relies on familiar rules of subtyping and type assignment. However, the system is based on a notion of rank, and there are some complications due to the need to stay within rank. These complications are characteristic of all ranked systems (see Chapter 3).

Our programs are just the terms of the lambda calculus. In particular, we do not use ML's let-expressions. In $\mathbf{P}_2$, ($\mathbf{let}\ x = M\ \mathbf{in}\ N$) can be considered an abbreviation for $(\lambda x N)M$.

We will be defining several classes of types, each of which is a restriction of the types with quantification and intersection. For those unfamiliar with intersection

types, we present a brief example. A term of type $(\sigma \wedge \tau)$ is thought of as having *both* the type σ and the type τ. For example, the identity function has both type $(t \to t)$ and $(s \to s) \to (s \to s)$, so

$$(\lambda y.y) : (t \to t) \wedge ((s \to s) \to (s \to s)).$$

By this intuition, a quantified type stands for the *infinite* intersection of its instances:

$$(\lambda y.y) : (\forall u.u \to u).$$

The types $(t \to t)$ and $(s \to s) \to (s \to s)$ are instances of $(\forall u.u \to u)$, so in some sense this typing is "more general" than the first.

Intersections can be used to express a form of "polymorphic abstraction" not possible in ML. For example, we will be able to derive the following type in our system:

$$(\lambda x.xx) : \forall s,t.(s \wedge (s \to t)) \to t.$$

This says that as long as the argument of the function $(\lambda x.xx)$ has *both* the types s and $s \to t$, for some s and t, the result will be of type t. Hence the argument is required to have more than one type—it must be polymorphic. The "rank 2" limitation is that polymorphism can only be required of arguments, not arguments of arguments: '$\wedge$' may appear to the left of a single arrow, no deeper (see Chapter 3 for a further discussion of rank). Still, this goes beyond ML, where $(\lambda x.xx)$ is not typable.

An appropriate argument for $(\lambda x.xx)$ is the identity function:

$$(\lambda x.xx)(\lambda y.y) : (\forall u.u \to u).$$

Again, we will be able to derive this type in our system. This example is typable in ML, *provided* it is translated into a let-expression:

$$(\textbf{let } x = (\lambda y.y) \textbf{ in } xx) : (\forall u.u \to u).$$

We now give the details of the rank 2 system $\mathbf{P}_2$. The sets $\mathbf{T}_0$, $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_{\forall 2}$ of types are defined inductively by the equations below.

$$\begin{aligned}
\mathbf{T}_0 &= \{\, t \mid t \text{ is a type variable} \,\} \cup \{\, (\sigma \to \tau) \mid \sigma, \tau \in \mathbf{T}_0 \,\}, \\
\mathbf{T}_1 &= \mathbf{T}_0 \cup \{(\sigma \wedge \tau) \mid \sigma, \tau \in \mathbf{T}_1\}, \\
\mathbf{T}_2 &= \mathbf{T}_0 \cup \{(\sigma \to \tau) \mid \sigma \in \mathbf{T}_1, \tau \in \mathbf{T}_2\}, \\
\mathbf{T}_{\forall 2} &= \mathbf{T}_2 \cup \{(\forall t \sigma) \mid \sigma \in \mathbf{T}_{\forall 2}\}.
\end{aligned}$$

The set $\mathbf{T}_0$ is the set of simple types, and $\mathbf{T}_1$ is the set of finite, nonempty intersections of simple types. $\mathbf{T}_2$ is the set of rank 2 intersection types: these are types possibly containing intersections, but only to the left of a single arrow. Finally, $\mathbf{T}_{\forall 2}$

adds top-level quantification of type variables to $\mathbf{T}_2$. Note that $\mathbf{T}_0 = \mathbf{T}_1 \cap \mathbf{T}_2$, and for $i \in \{0, 1, 2, \forall 2\}$, if $\tau \in \mathbf{T}_i$, then $S\tau \in \mathbf{T}_i$ for any $\mathbf{T}_0$ substitution S.

Just as we have several classes of types, we have several subtyping relations (these could be combined into a single subtyping relation, but it is technically convenient to keep them separate). Their definition is simplified by observing the following conventions: we consider types to be syntactically equal modulo renaming of bound type variables, reordering of adjacent quantifiers, and elimination of unnecessary quantifiers; and we consider '$\wedge$' to be an associative, commutative, and idempotent operator, so that any $\mathbf{T}_1$ type may be considered a finite, nonempty set of simple types, written in the form $(\bigwedge_{i\in I} \sigma_i)$, where each $\sigma_i \in \mathbf{T}_0$. When a $\mathbf{T}_{\forall 2}$ type is written in the form $\forall\vec{s}\sigma$, we assume $\sigma \in \mathbf{T}_2$.

Definition 2.2.1 For $i \in \{1, 2, \forall 2\}$, we define the relation $\leq_i$ as the least partial order on $\mathbf{T}_i$ closed under the following rules:

- If $\{\tau_j \mid j \in J\} \subseteq \{\sigma_i \mid i \in I\}$, then $(\bigwedge_{i\in I} \sigma_i) \leq_1 (\bigwedge_{j\in J} \tau_j)$.
- If $\sigma_1 \leq_1 \tau_1$ and $\tau_2 \leq_2 \sigma_2$, then $(\tau_1 \to \tau_2) \leq_2 (\sigma_1 \to \sigma_2)$.
- If $\sigma \leq_2 \tau$, then $\sigma \leq_{\forall 2} \tau$.
- If $\tau \in \mathbf{T}_0$, then $(\forall t\sigma) \leq_{\forall 2} \{t := \tau\}\sigma$.
- If $\sigma \leq_{\forall 2} \tau$ and t is not free in σ, then $\sigma \leq_{\forall 2} (\forall t\tau)$.

The first rule says that $\leq_1$ expresses the natural ordering on intersection types. The second rule says that $\leq_2$ obeys the usual antimonotonic ordering on function types, restricted to rank 2. The rules for $\leq_{\forall 2}$ express the intuition that a type is a subtype of its instances (recall that $\{t := \tau\}\sigma$ is the type σ with τ substituted for t). They are equivalent to the following rule, similar to ML's notion of *generic instance*:

- If $\{\vec{s} := \vec{\rho}\}\sigma \leq_2 \tau$, where $\vec{\rho}$ is a vector of simple types, and the type variables $\vec{t}$ are not free in $(\forall\vec{s}\sigma)$, then $\forall\vec{s}\sigma \leq_{\forall 2} \forall\vec{t}\tau$.

Note that we only allow instantiation of simple types. This ensures that instantiation does not take us beyond rank 2. It also has less desirable implications, e.g., $(\forall t.t)$ is not a least type in the ordering $\leq_{\forall 2}$: $(\forall t.t) \not\leq_{\forall 2} (s \wedge (s \to u)) \to u$.

A fourth subtyping relation will play an important role in the type system. The relation $\leq_{\forall 2,1}$ between $\mathbf{T}_{\forall 2}$ and $\mathbf{T}_1$ is the smallest relation satisfying the rule:

- If $\sigma \leq_{\forall 2} \tau_i$ for all $i \in I$, then $\sigma \leq_{\forall 2,1} (\bigwedge_{i\in I} \tau_i)$.

The relation $\leq_{\forall 2,1}$ is not a partial order; it is not even reflexive. This is because it relates types "across rank." Note that in a comparison

$$(\forall t\sigma) \leq_{\forall 2,1} (\bigwedge_{i\in I} \tau_i),$$

(VAR) $\{x : (\bigwedge_{i\in I} \tau_i)\} \vdash x : \tau_{i_0}$ (where $i_0 \in I$)

(ABS) $$\frac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x M) : \sigma \to \tau}$$

(APP) $$\frac{A \vdash M : (\bigwedge_{i\in I} \tau_i) \to \sigma, \quad (\forall i \in I)\ A \vdash N : \tau_i}{A \vdash (MN) : \sigma}$$

(GEN) $$\frac{A \vdash M : \sigma}{A \vdash M : \forall t \sigma} \quad \text{(where } t \notin \mathrm{FTV}(A))$$

(SUB) $$\frac{A \vdash M : \tau}{A \vdash M : \sigma} \quad \text{(where } \tau \leq_{\forall 2} \sigma)$$

(ADD-HYP) $$\frac{A \vdash M : \sigma}{A \cup \{x : \tau\} \vdash M : \sigma}$$

Figure 2.1: Typing rules of $\mathbf{P}_2$. Types in type environments are in $\mathbf{T}_1$, and derived types are in $\mathbf{T}_{\forall 2}$.

the type variable t may be instantiated differently for each τ_i.

The typing judgments are of the form $A \vdash M : \sigma$, where σ is a $\mathbf{T}_{\forall 2}$ type, and all of the types in A are $\mathbf{T}_1$ types. The typing rules are given in Figure 2.1. Note that the only subtyping relation used in these rules is $\leq_{\forall 2}$, which is used in (SUB), the rule of *subsumption*. Later, it will be necessary to distinguish $\mathbf{P}_2$ typings from typing judgments that hold in other systems. In that case we will write $\mathbf{P}_2 \triangleright A \vdash M : \sigma$ for the $\mathbf{P}_2$ judgment $A \vdash M : \sigma$.

Example 2.2.2 Recall that the typings

$$\begin{aligned} (\lambda x.xx) &: \forall s,t.(s \wedge (s \to t)) \to t, \\ (\lambda y.y) &: (\forall u.u \to u), \end{aligned}$$

hold in our system. Then by rule (SUB),

$$(\lambda x.xx) : ((s \to s) \wedge ((s \to s) \to (s \to s))) \to (s \to s).$$

And $(\forall u.u \to u) \leq_{\forall 2} (s \to s)$ and $(\forall u.u \to u) \leq_{\forall 2} ((s \to s) \to (s \to s))$, so by rules (SUB) and (APP),

$$(\lambda x.xx)(\lambda y.y) : (s \to s).$$

Finally, by rule (GEN),

$$(\lambda x.xx)(\lambda y.y) : \forall s.s \to s.$$

We now give the definition of principal typings appropriate to our system.

Definition 2.2.3 (Principal typings)

- A pair $\langle B, \tau \rangle$ is a $\mathbf{P}_2$ *instance* of a pair $\langle A, \sigma \rangle$ if there is a substitution S such that $S\sigma \leq_{\forall 2} \tau$ and $B \leq_1 SA$.
- A typing $B \vdash M : \tau$ is a $\mathbf{P}_2$ instance of a typing $A \vdash M : \sigma$ if $\langle B, \tau \rangle$ is an $\mathbf{P}_2$ instance of $\langle A, \sigma \rangle$.
- A *principal typing* for a term M is a $\mathbf{P}_2$ typing $A \vdash M : \sigma$ of which any other $\mathbf{P}_2$ typing of M is an instance.

This definition is standard, cf. [49]. Note in particular that the notion of instance is monotonic in the derived type, but antimonotonic in the type environment. The intuition is, a principal typing EXPECTS LESS of its free variables, and PROVIDES MORE than any other typing judgment.

The following results (proved in Chapter 3) show that $\mathbf{P}_2$ is closely connected to Λ_2, the restriction of System F to rank 2 types.

Theorem 2.2.4 *A term M is typable in $\mathbf{P}_2$ iff M is typable in Λ_2 iff M is typable in the rank 2 intersection type system.*

Corollary 2.2.5 *Typability in $\mathbf{P}_2$ is DEXPTIME-complete.*

Thus the $\mathbf{P}_2$ programs are exactly the Λ_2 programs, and the complexity of type inference is exactly the same as for ML and Λ_2. As we will see, however, $\mathbf{P}_2$ has the principal typing property, while no notion of principal typing is known for ML or Λ_2 [13, 35].

2.3 Subtype satisfaction

In order to perform type inference, we must solve *subtype satisfaction problems.* Solving subtype satisfaction also gives a decision procedure for subtyping. We will focus on the relation $\leq_{\forall 2,1}$, as it is the most important for type inference; all of the other relations can be handled in a similar manner.

Up until now, we have relied on some syntactic conventions to simplify our presentation, namely, that '$\wedge$' is an associative, commutative, and idempotent operator. Part of the problem we are addressing here is how to decide whether two types are equivalent under these assumptions. Therefore, in this section, we do not rely on the syntactic conventions in any way.

Subtype satisfaction is a generalization of the well-known problem of *unification*, and the techniques we use here are based on those used to solve unification. For more details, consult a survey on unification [37, 40, 58, 27, 17, 60, 5]. One difference between unification and our satisfaction problems is that we work with types that

go beyond simple types, but we will only consider solutions (substitutions) involving simple types. This is not the typical case with unification, and it makes our problem easier to solve.

If S_1, S_2 are substitutions and V is a set of type variables, we say S_1 *and* S_2 *are equivalent on* V, written $S_1 =_V S_2$, if $S_1 t = S_2 t$ for every $t \in V$. We say S_1 *is more general than* S_2 *on* V, written $S_1 \leq_V S_2$, if there is a substitution S_3 such that $S_2 =_V S_3 S_1$. The relation $\leq_V$ is a partial order modulo $=_V$. We omit V when $V = \mathbf{Tv}$. A substitution S is *idempotent* if $S = SS$, or, equivalently, if $\mathbf{dom}(S) \cap \mathbf{rng}(S) = \varnothing$.

A $\leq_{\forall 2,1}$*-satisfaction problem* is a pair $\exists\vec{s}.P$, where P is a set whose every element is either: 1) an equality between simple types; or 2) an inequality between a $\mathbf{T}_{\forall 2}$ type and a $\mathbf{T}_1$ type. When $\vec{s}$ is empty $\exists\vec{s}$ may be omitted. We use π to range over $\leq_{\forall 2,1}$-satisfaction problems.

A $\mathbf{T}_0$ substitution S is a *solution* to $\exists\vec{s}.P$ if there is a substitution S' such that $S(t) = S'(t)$ for all $t \notin \vec{s}$, $S'\sigma \leq_{2,1} S'\tau$ for all inequalities $(\sigma \leq \tau) \in P$, and $S'\sigma = S'\tau$ for all equalities $(\sigma = \tau) \in P$. The (possibly empty) set of solutions to a problem π is written $\mathbf{Solutions}(\pi)$. Two problems π_1 and π_2 are *equivalent* if $\mathbf{Solutions}(\pi_1) = \mathbf{Solutions}(\pi_2)$.

Definition 2.3.1 A substitution U is a *most general solution to* π if it satisfies the following conditions.

- $U \in \mathbf{Solutions}(\pi)$.
- If $S \in \mathbf{Solutions}(\pi)$ then $U \leq_{\mathrm{FTV}(\pi)} S$.
- U is idempotent.
- $\mathbf{dom}(U) \subseteq \mathrm{FTV}(\pi)$.

We write $\mathbf{MGS}(\pi)$ for the (possibly empty) set of most general solutions to a $\leq_{\forall 2,1}$-satisfaction problem π.

We require the last two conditions on most general solutions for technical convenience only. We could relax the definition by eliminating those conditions; but any π has a solution under the relaxed definition if and only if it has a solution under our definition.

Sometimes it is useful to ensure that a most general solution does not interfere with a set of "protected" variables. For any set W of type variables, we say U is a *most general solution to* π *away from* W if $U \in \mathbf{MGS}(\pi)$ and $W \cap \mathbf{rng}(U) = \varnothing$, and we write $\mathbf{MGS}(\pi)[W]$ for the (possibly empty) set of most general solutions to π away from W.

Lemma 2.3.2 *If* $U \in \mathbf{MGS}(\pi)[W]$ *and* $S \in \mathbf{Solutions}(\pi)$*, then* $U \leq_{W \cup \mathrm{FTV}(\pi)} S$.

Proof: Since $U \leq_{\mathrm{FTV}(\pi)} S$, there is some R such that $RU =_{\mathrm{FTV}(\pi)} S$. Define

$$R'(t) = \begin{cases} R(t) & \text{if } t \in \mathbf{rng}(U), \\ S(t) & \text{otherwise.} \end{cases}$$

If $t \in \mathrm{FTV}(\pi)$, then $R'(U(t)) = R(U(t)) = S(t)$. And if $t \in W - \mathrm{FTV}(\pi)$, then $t \notin (\mathbf{dom}(\pi) \cup \mathbf{rng}(\pi))$, so $R'(U(t)) = R'(t) = S(t)$. □

A *unification problem* is a subtype satisfaction problem involving only equalities. Algorithms for solving unification problems are well known; in particular, we have the following result.

Lemma 2.3.3 *Let π be a unification problem and W be a finite set of type variables.*

- $\mathbf{Solutions}(\pi) = \varnothing$ *iff* $\mathbf{MGS}(\pi) = \varnothing$ *iff* $\mathbf{MGS}(\pi)[W] = \varnothing$.
- *There is an algorithm that decides whether π has a solution, and, if so, returns an element of* $\mathbf{MGS}(\pi)[W]$.

Proof: See for example Snyder [60], Lemma 3.3.11. □

Theorem 2.3.4 *Every $\leq_{\forall 2,1}$-satisfaction problem is equivalent to a unification problem, and moreover, there is an algorithm that transforms every $\leq_{\forall 2,1}$-satisfaction problem into an equivalent unification problem.*

Corollary 2.3.5 *Let π be a $\leq_{\forall 2,1}$-satisfaction problem and W be a finite set of type variables.*

- $\mathbf{Solutions}(\pi) = \varnothing$ *iff* $\mathbf{MGS}(\pi) = \varnothing$ *iff* $\mathbf{MGS}(\pi)[W] = \varnothing$.
- *There is an algorithm that decides whether π has a solution, and, if so, returns an element of* $\mathbf{MGS}(\pi)[W]$.

We will prove Theorem 2.3.4 by giving an algorithm that transforms any $\leq_{\forall 2,1}$-satisfaction problem into an equivalent unification problem. Corollary 2.3.5 follows by combining the transformation with any unification algorithm.

Our transformation is defined by rules of the form

$$\sigma \leq \tau \quad \Rightarrow \quad \exists \vec{s}.P.$$

The rules may need to introduce fresh type variables, that is, type variables that do not appear on the left-hand side. These variables will appear in the variables $\vec{s}$ of the right-hand side (but they are not the only source of variables in $\vec{s}$).

The rules are used to define a rewrite relation on problems:

$$\frac{\sigma \leq \tau \quad \Rightarrow \quad \exists \vec{t}.P}{\exists \vec{s}.P' \uplus \{\sigma \leq \tau\} \quad \Rightarrow \quad \exists \vec{s} \uplus \vec{t}.P' \cup P}$$

$$(\sigma_1 \to \sigma_2) \leq t \quad\Rightarrow\quad \exists t_1, t_2.\{t_1 \leq \sigma_1, \sigma_2 \leq t_2, t = t_1 \to t_2\}$$
if t_1, t_2 are fresh

$$(\sigma_1 \to \sigma_2) \leq (\tau_1 \to \tau_2)$$
$$\Rightarrow\quad \{\tau_1 \leq \sigma_1, \sigma_2 \leq \tau_2\}$$

$$\sigma \leq (\tau_1 \wedge \tau_2) \quad\Rightarrow\quad \{\sigma \leq \tau_1, \sigma \leq \tau_2\}$$

$$t \leq \tau \quad\Rightarrow\quad \{t = \tau\}$$
if τ is a simple type

$$(\forall t \sigma) \leq \tau \quad\Rightarrow\quad \exists t\{\sigma \leq \tau\}$$
if τ is not a $\wedge$-type, and t is not free in τ

Figure 2.2: Transformational rules for $\leq_{\forall 2,1}$-satisfaction problems

The operator '$\uplus$' is disjoint union; on the right of the consequent, it means that the variables $\vec{t}$ must be fresh (this can always be achieved by renaming).

The rules for transforming a $\leq_{\forall 2,1}$-unification problem into a unification problem are given in Figure 2.2.

Proof of Theorem 2.3.4: We show that the rules of Figure 2.2 constitute an algorithm for converting any $\leq_{\forall 2,1}$-satisfaction problem into an equivalent unification problem.

First, note that every rule transforms a $\leq_{\forall 2,1}$-satisfaction problem into another $\leq_{\forall 2,1}$-satisfaction problem (equalities are between simple types, inequalities are between $\mathbf{T}_{\forall 2}$ and $\mathbf{T}_1$ types).

Second, note that each rule preserves the set of solutions, so that each application of a rule transforms a problem into an equivalent problem.

Third, note that repeated application of these rules must halt: every rule reduces the number of type constructors ('$\to$' or '$\wedge$', or '$\forall$') in inequalities or reduces the number of inequalities.

Finally, note that a normal form contains no inequalities, and is therefore a unification problem. □

Lemma 2.3.6 (Subtyping is decidable) *The relation $\leq_{\forall 2,1}$ is decidable.*

Proof: To see whether $\sigma \leq_{\forall 2,1} \tau$, compute $U \in \mathbf{MGS}(\{\sigma \leq \tau\})$ and check to see whether U is the identity substitution. □

Decision procedures for the other subtyping relations can be obtained in a similar way. One complication is that the subtype satisfaction problems for the other subtyping relations are not *unitary*; that is, they do not always have a *single* most general solution. Instead, they may have a finite number of most general solutions. Thus they belong to the (well-understood) class of *finitary* problems.

Because we so often want to ensure that $U \in \mathbf{MGS}(\pi)$ is chosen "away" from a set of type variables, we adopt the following important convention.

Convention 2.3.7 (Non-interference) Whenever $U \in \mathbf{MGS}(\pi)$ occurs in any mathematical context, we assume that U is chosen so that it does not interfere with "current" type variables, that is, $U \in \mathbf{MGS}(\pi)[W]$ where $W \cup \mathrm{FTV}(\pi)$ is the set of type variables present in the context.

2.4 Type inference

The type inference algorithm is presented in the style favored by the intersection type community: for any M, we define a set, $\mathrm{PP}(M)$, called the ***principal pairs of*** M. Every element of $\mathrm{PP}(M)$ is a pair $\langle A, \sigma \rangle$ such that $A \vdash M : \sigma$ is a principal typing of M.

We first introduce the following notation. For type environments A_1 and A_2, define a type environment $A_1 + A_2$ as follows: for each $x \in \mathbf{dom}(A_1) \cup \mathbf{dom}(A_2)$,

$$(A_1 + A_2)(x) = \begin{cases} A_1(x) & \text{if } x \notin \mathbf{dom}(A_2), \\ A_2(x) & \text{if } x \notin \mathbf{dom}(A_1), \\ A_1(x) \wedge A_2(x) & \text{otherwise.} \end{cases}$$

Note that if A_1 and A_2 are $\mathbf{T}_1$ type environments, then $A_1 + A_2$ is a $\mathbf{T}_1$ type environment, and if $A \leq_1 A_1$ and $A \leq_1 A_2$, then $A \leq_1 A_1 + A_2$.

Definition 2.4.1 (Type inference) For any term M, the set $\mathrm{PP}(M)$ is defined by the following cases.

- If $M = x$, then $\langle \{x : t\}, t \rangle \in \mathrm{PP}(x)$ for any type variable t.
- If $M = \lambda x N$, and $\langle A, \forall \vec{s} \sigma \rangle \in \mathrm{PP}(N)$, where the type variables $\vec{s}$ are distinct from all other type variables, then:
  - If $x \notin \mathbf{dom}(A)$, and t is a fresh type variable, then $\langle A, \forall t \vec{s} (t \to \sigma) \rangle \in \mathrm{PP}(\lambda x N)$.
  - If $x \in \mathbf{dom}(A)$, then $\langle A \backslash x, \mathrm{Gen}(A \backslash x, A(x) \to \sigma) \rangle \in \mathrm{PP}(\lambda x N)$.
- If $M = M_1 M_2$, and $\langle A_1, \forall \vec{s} \sigma_1 \rangle \in \mathrm{PP}(M_1)$, then:

- If $\sigma_1 = t$ (a type variable), t_1 and t_2 are fresh type variables, the type variables of $\langle A_2, \sigma_2 \rangle \in \mathrm{PP}(M_2)$ are fresh, $U \in \mathbf{MGS}(\{\sigma_2 \leq t_1, t = t_1 \to t_2\})$, and $A = U(A_1 + A_2)$, then

$$\langle A, \mathrm{Gen}(A, Ut_2) \rangle \in \mathrm{PP}(M).$$

- If $\sigma_1 = (\bigwedge_{i \in I} \tau_i) \to \tau$, $(\forall i \in I)$ the type variables of $\langle A_i, \sigma_i \rangle \in \mathrm{PP}(M_2)$ are fresh, $U \in \mathbf{MGS}(\{\sigma_i \leq \tau_i \mid i \in I\})$, and $A = U(A_1 + \sum_{i \in I} A_i)$, then

$$\langle A, \mathrm{Gen}(A, U\tau) \rangle \in \mathrm{PP}(M).$$

When it becomes necessary to distinguish this type inference algorithm from others, we will write it as $\mathrm{PP}_{\mathbf{P}_2}$.

The following technical property is used to show that $\mathrm{PP}(M)$ indeed specifies a type inference algorithm: the set $\mathrm{PP}(M)$ is an equivalence class of pairs under permutations, i.e., $\langle A_1, \sigma_1 \rangle, \langle A_2, \sigma_2 \rangle \in \mathrm{PP}(M)$ iff $\langle A_1, \sigma_1 \rangle = S\langle A_2, \sigma_2 \rangle$ for some bijection S of type variables. Therefore, in choosing $\langle A, \sigma \rangle \in \mathrm{PP}(M)$ it is always possible to guarantee that the type variables of $\langle A, \sigma \rangle$ are "fresh."

To perform type inference, simply follow the definition of $\mathrm{PP}(M)$, choosing "fresh" type variables and using the **MGS** algorithm as necessary.

Example 2.4.2 We show how the algorithm finds a principal typing for $(\lambda x.xx)$.

- $\mathrm{PP}(x)$ produces a pair $\langle \{x : t_1\}, t_1 \rangle$.
- $\mathrm{PP}(x)$ (again) produces a pair $\langle \{x : t_2\}, t_2 \rangle$.
- To calculate $\mathrm{PP}(xx)$, we find a most general solution to

$$\{t_2 \leq t_3, t_1 = t_3 \to t_4\},$$

  such as $\{t_2 := t_3, t_1 := t_3 \to t_4\}$. Then

$$\langle \{x : t_3 \wedge (t_3 \to t_4)\}, t_4 \rangle \in \mathrm{PP}(xx).$$

- Finally, $\mathrm{PP}(\lambda x.xx)$ produces

$$\langle \varnothing, \forall t_3, t_4.(t_3 \wedge (t_3 \to t_4)) \to t_4 \rangle.$$

Theorem 2.4.3 (Principal typings) *If M is typable in $\mathbf{P}_2$, then there is a pair $\langle A, \sigma \rangle \in \mathrm{PP}(M)$ such that $A \vdash M : \sigma$ is a principal typing for M.*

2.5 Recursive definitions

We now add recursive definitions to our language: a term of the form $(\mu x M)$ represents the program x such that $x = M$, where M may contain occurrences of x.

As we remarked in the beginning of this chapter, the principal typing property suggests that we type recursive definitions by a rule of the following form.

$$(\text{REC}) \quad \frac{A \cup \{x : \tau\} \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \quad \sigma \leq_{\forall 2,1} \tau$$

The rule (REC) can type strictly more terms than the rule (REC-SIMPLE) of ML. For example, the following term is typable in $\mathbf{P}_2 +$ (REC), but not in $\mathbf{P}_2 +$ (REC-SIMPLE):

$$(\mu x.(\lambda yz.z)(xx)) : \forall t.t \to t.$$

The self-application xx cannot be typed if x is assigned just a simple type.

However, (REC) cannot type as many terms as (REC-POLY). For example, the term $(\mu x.xx)$ has type $(\forall t.t)$ in ML + (REC-POLY), but it is not typable with our rules.

It is interesting to compare (REC) with a rule, (FIX'), that Mycroft [50] suggested in the context of ML:

$$(\text{FIX'}) \quad \frac{A \vdash \lambda x_1 \cdots x_n.M' : \tau_1 \to \cdots \to \tau_n \to \tau}{A \vdash (\mu x M) : \tau}$$

Here M is a term with n occurrences of x, M' is M with each occurrence of x renamed to a fresh variable x_i, $\tau_1, \ldots, \tau_n, \tau$ are simple types, and $\text{Gen}(A, \tau) \leq \tau_i$ for all $i \leq n$.

The idea behind Mycroft's rule is that each of the finite occurrences of x in M may have a different simple type (so long as M can be shown to satisfy those types). The same idea explains the typing power of (REC). Note, however, that *this idea was not the motivation for* (REC). Instead, (REC) arose as an instance of a general rule motivated by the principal typing property. Other interesting typing rules may arise as instances of the general rule, in type systems other than $\mathbf{P}_2$.

Mycroft's rule (FIX') is actually more powerful than (REC). Its side condition, $\text{Gen}(A, \tau) \leq \tau_i$, permits τ to be generalized by any type variable not appearing in A, *including type variables appearing in the* τ_i. This is not allowed by (REC). The term $(\mu x.xx)$ is one place where this makes a difference: it is typable with (FIX') but not (REC). For a more practical example, consider the following ML code. It comes from the ML mailing list, and has arisen in practice.

```
datatype 'a T = EMPTY
              | NODE of 'a * ('a T) T
```

$$(\text{LETREC-SIMPLE}) \quad \frac{\begin{array}{c}(\forall j \in I) \quad A \cup \{x_i : \tau_i \mid i \in I\} \vdash M_j : \tau_j \quad (\tau_j \in \mathbf{T}_0) \\ A \cup \{x_i : \mathrm{Gen}(A, \tau_i) \mid i \in I\} \vdash M : \sigma\end{array}}{A \vdash (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ M) : \sigma}$$

$$(\text{LETREC-VAR}) \quad \frac{\forall j \in I \quad A \cup \{x_i : \tau_i \mid i \in I\} \vdash M_j : \sigma_j \quad (\sigma_j \leq_{\forall 2,1} \tau_j)}{A \vdash (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ x_{i_0}) : \sigma_{i_0}} \quad i_0 \in I$$

$$(\text{LETREC}) \quad \frac{\begin{array}{c}A \cup \{x_i : \tau_i \mid i \in I\} \vdash N : \sigma \\ \forall j \in I \quad A \vdash_\wedge (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ x_j) : \tau_j\end{array}}{A \vdash (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ N) : \sigma} \quad N \notin \{x_i \mid i \in I\}$$

Figure 2.3: Rules for typing mutually recursive definitions. The rule (LETREC-SIMPLE) is used by ML, while $\mathbf{P}_2$ uses (LETREC-VAR) and (LETREC).

```
fun collect EMPTY = nil
  | collect (NODE(n,t)) =
      n :: flatmap collect (collect t)
```

Here `'a T` is a polymorphic tree type, and `flatmap` is the mapping function of type `('a -> 'b list) -> 'a list -> 'b list`. The function `collect`, which collects all the labels of an `'a T` and returns them in an `'a list`, is typable with (REC-POLY) and with (FIX'), but not with (REC). Of course, we could generalize our rule along the lines of (FIX'):

$$(\text{REC'}) \quad \frac{A \cup \{x : \tau\} \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \quad \mathrm{Gen}(A, \sigma) \leq_{\forall 2,1} \tau$$

The system would retain principal typings and decidable type inference, but for simplicity, we stay with (REC).

2.6 Mutual recursion

In order to support the applications of principal typings in the next section, we add mutually recursive definitions to the language. Such definitions are written

$$(\mathbf{letrec}\ x_1 = M_1, \ldots, x_n = M_n\ \mathbf{in}\ N)$$

or

$$(\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ N),$$

where all of the x_i are distinct.

The typing rules for **letrec** are given in Figure 2.3. ML uses the rule (LETREC-SIMPLE) to type mutual recursion. In (LETREC-SIMPLE), the recursive definitions

must be typed under the assumption that the recursive variables have simple type. In typing the body of the **letrec**, however, the types of the recursive variables can be generalized, so that they can be used polymorphically.

We cannot use (LETREC-SIMPLE) with $\mathbf{P}_2$, because $\mathbf{P}_2$ does not permit quantified types to appear in type environments. And, it is not easily adapted to $\mathbf{P}_2$. In ML, the polymorphic type $\mathrm{Gen}(A, \tau_i)$ of x_i is easily obtained from the simple type τ_i used in typing the recursive definitions. The equivalent of $\mathrm{Gen}(A, \tau_i)$ in $\mathbf{P}_2$ is some intersection $(\bigwedge_{j\in J} \tau_j)$, where each τ_j is an instance of $\mathrm{Gen}(A, \tau_i)$. It is not immediately clear how to get directly from τ_i to $(\bigwedge_{j\in J} \tau_j)$.

Instead, our rules for $\mathbf{P}_2$ are based on the following observation: the typings of any term (**letrec** B **in** N) can be expressed in terms of the typings for terms (**letrec** B **in** x), where x is a variable defined by B. Formally, for any $B = \{x_1 = M_1, x_2 = M_2, \ldots, x_n = M_n\}$ and $M =$ (**letrec** B **in** N), we define $\langle\!\langle M \rangle\!\rangle$ to be the term

$$
\begin{aligned}
\langle\!\langle M \rangle\!\rangle = (\mathbf{let}\ & x_1 = (\mathbf{letrec}\ B\ \mathbf{in}\ x_1) \\
& \vdots \\
& x_n = (\mathbf{letrec}\ B\ \mathbf{in}\ x_n) \\
\mathbf{in}\ & N).
\end{aligned}
$$

The following lemma is easily proved.

Lemma 2.6.1 *In* ML + (LETREC-SIMPLE), $A \vdash M : \sigma$ *iff* $A \vdash \langle\!\langle M \rangle\!\rangle : \sigma$.

This is the intuition behind the rules (LETREC-VAR) and (LETREC) of Figure 2.3. (LETREC-VAR) is a straightforward generalization of the rule (REC) for terms of the form (**letrec** B **in** x), where x is a variable defined in B. Lemma 2.6.1 suggests that we type other **letrec** expressions by a rule of the form

$$
\frac{A \vdash \langle\!\langle \mathbf{letrec}\ B\ \mathbf{in}\ N \rangle\!\rangle : \sigma}{A \vdash (\mathbf{letrec}\ B\ \mathbf{in}\ N) : \sigma} \quad (N \text{ is not defined by } B)
$$

Our rule (LETREC) is obtained simply by desugaring the let-expression formed by $\langle\!\langle\cdot\rangle\!\rangle$ into abstractions and applications, and considering how the resulting term would be typed by (ABS) and (APP). We make the rule more compact by using the notation $A \vdash_\wedge M : (\bigwedge_{i\in I} \tau_i)$ to abbreviate $(\forall i \in I)\ A \vdash M : \tau_i$.

We write Λ_2^{R} for the system Λ_2 + (REC-SIMPLE) + (LETREC-SIMPLE), and $\mathbf{P}_2^{\mathrm{R}}$ for the system $\mathbf{P}_2$ + (REC) + (LETREC-VAR) + (LETREC).

Theorem 2.6.2 (Comparison of Λ_2^{R} and $\mathbf{P}_2^{\mathrm{R}}$) *If M is typable in Λ_2^{R}, then M is typable in $\mathbf{P}_2^{\mathrm{R}}$.*

Definition 2.6.3 ($\mathbf{P}_2^{\mathrm{R}}$ type inference) The type inference algorithm of Definition 2.4.1 can be extended to $\mathbf{P}_2^{\mathrm{R}}$ by adding the following cases.

- If $M = (\mu x N)$ and $\langle A, \sigma \rangle \in \mathrm{PP}(N)$, then:
  - If $x \notin \mathbf{dom}(A)$, and $U \in \mathbf{MGS}(\{\sigma \leq t\})$ where t is a fresh type variable, then $\langle UA, \mathrm{Gen}(UA, U\sigma) \rangle \in \mathrm{PP}(M)$.
  - If $x \in \mathbf{dom}(A)$ and $U \in \mathbf{MGS}(\{\sigma \leq A(x)\})$, then $\langle UA \backslash x, \mathrm{Gen}(UA \backslash x, U\sigma) \rangle \in \mathrm{PP}(M)$.
- If $M = (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ x_{i_0})$, where $i_0 \in I$, and $\langle A_i, \sigma_i \rangle \in \mathrm{PP}(M_i)$ for $i \in I$,
  $A' = \sum_{i \in I} A_i$,
  $A'' = A' \cup \{x_i : t_i \mid x_i \notin \mathbf{dom}(A'), t_i \text{ fresh}\}$,
  $U \in \mathbf{MGS}(\{\sigma_i \leq A''(x_i) \mid i \in I\})$,
  and $A = UA'' \backslash \{x_i \mid i \in I\}$,
  then $\langle A, \mathrm{Gen}(A, U\sigma_{i_0}) \rangle \in \mathrm{PP}(M)$.
- If $M = (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ N)$, where $N \notin \{x_i \mid i \in I\}$, and $\langle A, \sigma \rangle \in \mathrm{PP}(\langle\!\langle M \rangle\!\rangle)$,
  then $\langle A, \sigma \rangle \in \mathrm{PP}(M)$.

When it becomes necessary to distinguish this type inference algorithm from others, we will write it as $\mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}$.

Theorem 2.6.4 (Principal typings) *If M is typable in $\mathbf{P}_2^{\mathrm{R}}$, then there is a pair $\langle A, \sigma \rangle \in \mathrm{PP}(M)$ such that $A \vdash M : \sigma$ is a principal typing for M in $\mathbf{P}_2^{\mathrm{R}}$.*

An important limitation of our rules for mutual recursion is illustrated by the following well-known example of Mycroft [50]:

$$\begin{aligned} \mathit{map} &= \lambda f.\lambda l.\ \mathbf{if}\ \mathit{null}\ l\ \mathbf{then}\ \mathit{nil} \\ &\qquad\qquad \mathbf{else}\ f(\mathit{hd}\ l) :: \mathit{map}\ f\ (\mathit{tl}\ l) \\ \mathit{squarelist} &= \lambda l.\ \mathit{map}\ (\lambda x.\ x \times x)\ l \\ \mathit{complement} &= \lambda l.\ \mathit{map}\ (\lambda x.\ \mathit{not}\ x)\ l \end{aligned}$$

This program is not typable under our rules (or ML's rules) when presented as a single, mutually recursive definition. The function *map* is used polymorphically by the other functions, and our rules do not allow sufficient polymorphism for the program to type. Note that *map* does not depend on the other functions; if *map* is placed in a separate recursive definition, the program can be typed by our rules.

Thus to type an unordered set of definitions, it is necessary to examine the call graph of the program to determine an order in which to type the definitions. This complication must be addressed by the applications of the next section.[1]

[1] A generalization of our rules along the lines of Mycroft's (FIX') could handle the *map* example, but not all such examples.

2.7 Separate compilation

Any separate compilation system manages a collection of small program fragments that together make up a single large program. Two questions must be answered by such a system. First, does the program as a whole type check? And second, how do we generate code for each program fragment, and how can we combine these code fragments into an executable program?

We consider each of these questions in turn.

2.7.1 Incremental type inference

The problem of *incremental type inference* [1] can be described as follows. A user develops a program in an incremental fashion, by entering a sequence of definitions to a read-eval-print loop:

$$x_1 = M_1, x_2 = M_2, x_3 = M_3, \ldots$$

After each definition is entered, the compiler performs type inference to ensure the type-correctness of the partial program. Definitions may be re-defined as the programmer detects and corrects bugs, and they may be mutually recursive. Most relevant, a "bottom-up" style of program development is made possible by allowing definitions to refer to other definitions which have not yet been entered.

Incremental type inference is thus the type checking task of separate compilation on an extremely fine scale: not just every module, but every definition is typed and compiled separately.

Consider a partial program $x_1 = M_1, \ldots, x_n = M_n$, where duplicate definitions have been discarded. To check that the program is well-typed, it is sufficient to perform type inference on the expression

$$(\mathbf{letrec}\ B_1\ \mathbf{in}\ \cdots (\mathbf{letrec}\ B_m\ \mathbf{in}\ 0) \cdots)$$

derived from the call graph of the program: each B_i is a strongly connected component (SCC) of mutually recursive bindings, and the B_i are topologically sorted.

This can be accomplished by any type inference algorithm that works on terms with free variables. But this is not enough to solve the incremental problem efficiently: when the user enters the next definition, $x_{n+1} = M_{n+1}$, we must do better than just running the type inference algorithm on the new expression

$$(\mathbf{letrec}\ B'_1\ \mathbf{in}\ \cdots (\mathbf{letrec}\ B'_{m'}\ \mathbf{in}\ 0) \cdots).$$

A close inspection of the $\mathbf{P}_2^{\mathrm{R}}$ type inference algorithm will show that principal typings are the key to efficient incremental type inference.

If $P = (\textbf{letrec } B_1 \textbf{ in } \cdots (\textbf{letrec } B_m \textbf{ in } 0) \cdots)$ is our partial program, and the variables defined by each B_i are denoted $x_{i,1}, \ldots, x_{i,n_i}$, then by the $\mathbf{P}_2^{\mathrm{R}}$ equivalent of Lemma 2.6.1, type inference for P is equivalent to type inference for the expression

$$\begin{aligned} \langle\!\langle P \rangle\!\rangle = (\textbf{let } & x_{1,1} = (\textbf{letrec } B_1 \textbf{ in } x_{1,1}) \\ & \vdots \\ & x_{1,n_1} = (\textbf{letrec } B_1 \textbf{ in } x_{1,n_1}) \\ & \vdots \\ & x_{m,1} = (\textbf{letrec } B_1 \textbf{ in } x_{m,1}) \\ & \vdots \\ & x_{m,n_m} = (\textbf{letrec } B_m \textbf{ in } x_{m,n_m}) \\ \textbf{in } & 0), \end{aligned}$$

where **let**'s are desugared into applications of abstractions.

We now show that type inference for such an expression is equivalent to solving a subtype satisfaction problem constructed from the principal pair of each expression $(\textbf{letrec } B_i \textbf{ in } x_{i,j})$.

Definition 2.7.1 For any term M, we define the set $L^*(M)$ inductively as follows.

- If $M = (\lambda x M_1) M_2$,
  and $\langle A_1, \sigma, \pi \rangle \in L^*(M_1)$
  $\langle A_2, \sigma_2 \rangle \in \mathrm{PP}(M_2)$
  $(\bigwedge_{i \in I} \tau_i) = \begin{cases} t & \text{if } x \notin \mathbf{dom}(A_1) \text{ and } t \text{ is fresh,} \\ A_1(x) & \text{otherwise.} \end{cases}$
  $(\forall i \in I)$ S_i renames $\mathrm{FTV}(A_2, \sigma_2)$ to fresh type variables
  then $\langle A_1 + (\sum_{i \in I} S_i A_2), \sigma, \pi \cup \{ S_i \sigma_2 \le \tau_i \mid i \in I \} \rangle \in L^*(M)$.
- Otherwise, $\langle A, \sigma, \varnothing \rangle \in L^*(M)$ iff $\langle A, \sigma \rangle \in \mathrm{PP}(M)$.

Lemma 2.7.2 *$\langle A, \sigma \rangle \in \mathrm{PP}(M)$ iff for some A', σ', π and U, $\langle A', \sigma', \pi \rangle \in L^*(M)$, $U \in \mathbf{MGS}(\pi)$, and $\langle A, \sigma \rangle = \langle UA', \mathrm{Gen}(UA', U\sigma') \rangle$.*

Therefore, we can perform type inference for P by calculating $\langle A, \sigma, \pi \rangle \in L^*(\langle\!\langle P \rangle\!\rangle)$, and finding a solution to π. And $L^*(\langle\!\langle P \rangle\!\rangle)$ is calculated from the principal pair of each $(\textbf{letrec } B_i \textbf{ in } x_{i,j})$.

Now consider the incremental case. When the user enters the next definition, $x_{n+1} = M_{n+1}$, we must perform type inference on a new partial program, P'. Just as before, this means calculating a new π' from $L^*(\langle\!\langle P' \rangle\!\rangle)$. And again, this requires the principal pair of each $(\textbf{letrec } B'_{i'} \textbf{ in } x_{i',j'})$.

We now argue that $L^*(\langle\!\langle P' \rangle\!\rangle)$ can be constructed incrementally. First note that the new definition may not change the SCC's of the existing call graph: the SCCs

change only when the new definition is mutually recursive with a previous definition. So most often, (**letrec** $B'_{i'}$ **in** $x_{i',j'}$) will equal some previous (**letrec** B_i **in** $x_{i,j}$); and then, *by the principal typing property, the principal pair of* (**letrec** B_i **in** $x_{i,j}$) *is unchanged.*

We even benefit if the SCC's change. If $B = \{x_i = M_i \mid i \in I\}$ is a new SCC, we must calculate the principal pair of (**letrec** $\{x_i = M_i \mid i \in I\}$ **in** x_j) for all $j \in I$. This involves computing $\text{PP}(M_i)$ for each M_i; but if $i \neq n+1$, then by the principal typing property, $\text{PP}(M_i)$ is unchanged.

Thus the principal pair for each definition need only be computed once, as it is entered by the user; it does not need to be recomputed at each new definition or re-definition. This is not the case in the system of Aditya and Nikhil, where a new definition may cause the entire program to be reprocessed (see [19], p. 104).

We must also calculate a solution to the new satisfaction problem. However, the new problem may be almost identical to the previous problem. In particular, if the new definition does not change the SCCs of the call graph, the new satisfaction problem will be a superset of the old problem. We may be able to incorporate large parts of the old solution into the new solution. Our algorithm for subtype satisfaction, described in §2.3, solves problems by transforming them into equivalent, simpler problems until a solution is reached. Such an algorithm is ideally suited to incorporating parts of the old solution. The transformations that applied to the old problem will, for the most part, be identical to the transformations applicable to the new problem.

Finally, we remark that the SCCs and topological sort may be computed incrementally by off-the-shelf algorithms [23, 44].

2.7.2 Smartest recompilation

Once we have solved the type checking task of separate compilation, we face the task of code generation. Types determine data representations, calling conventions, and other implementation details. Thus we regard compilers as functions from typing judgments to machine code. For example, the compilation of a module M that imports a module x can be written

$$\text{Compile}(\{x : \sigma\} \vdash M : \tau) = \langle\text{machine code for } M\rangle.$$

There are two difficulties with this strategy. First, the compiler requires as input a typing judgment, or, at least, the types of external variables. The typical solution is to require the user to supply the types. A better solution is available in $\mathbf{P}_2$, where the compiler itself can infer a judgment $\{x : \sigma\} \vdash M : \tau$ for a term M with free variable x.

The second difficulty arises when we need to link all of the code fragments together into a single program. In particular, consider *re*compilation, in which a user changes a single module x and the system attempts to recompile as small a portion

of the entire program as possible. Certainly the definition of x must be recompiled. Moreover, an *unchanged* module M that imports x may have to be recompiled: if the type of x changes, then the *typing judgment* of M, and thus its compiled output, changes.

This is where principal typings help. Suppose that we have compiled a module M by compiling its principal typing, $A \vdash M : \tau$. At link time, we discover that in order to be consistent with the rest of the program, we should instead have compiled M by a different typing, $B \vdash M : \sigma$. The principal typing property tells us that the second judgment is an *instance* of the first: in $\mathbf{P}_2$, it can be obtained by substitution and subsumption from the principal typing. More formally,

$$\langle B, \sigma \rangle = \mathcal{C}\langle A, \tau \rangle,$$

where $\mathcal{C}$ is an operator that applies substitution and subsumption to the pair $\langle A, \tau \rangle$.

Stating the problem in this way lets us study the operator $\mathcal{C}$ in isolation. The operations of substitution and subsumption specified by $\mathcal{C}$ can be implemented via *coercions*. These coercions can be "wrapped" around the code generated for the typing $A \vdash M : \sigma$ at link time, making it behave like code generated for $B \vdash M : \tau$. That is,

$$\text{Compile}(B \vdash M : \sigma) \cong \text{Link}(\mathcal{C}, \text{Compile}(A \vdash M : \tau)),$$

where Link produces machine code that implements the coercions specified by $\mathcal{C}$.

Using this strategy, *a module need not be recompiled unless its definition changes.* This property was dubbed *smartest recompilation* by Shao and Appel [57]. They achieved smartest recompilation for ML by relating ML to a restriction of $\mathbf{P}_2$ with principal typings.

Shao and Appel identified the following problem with smartest recompilation. If a module references many free variables, e.g., functions from the standard library, then the type environment of the principal typing becomes large. This can be alleviated in the following way. Let B be a type environment specifying the $\mathbf{T}_{\forall 2}$ types of our library functions. We modify our type system to use two type environments, so that typings are of the form

$$A, B \vdash M : \sigma.$$

We modify our old rules to ignore this new type environment, and add a rule that allows us to use it:

$$(\text{VAR-NEW}) \quad A, B \cup \{x : \sigma\} \vdash x : \sigma$$

This system does not have principal typings, but it does have a useful "weak" form of principal typing property: given a term M typable in type environment B, there exists a typing $A, B \vdash M : \sigma$ representing all possible typings for M in B. We say that M has a principal typing *with respect to* the type environment B, and that we have smartest compilation *with respect to* B. Since B only specifies types for

identifiers that are relatively stable, we gain most of the benefits of full smartest recompilation.

As an aside, we remark that this immediately suggests an extension to the type system: restore let-expressions to the language and add the rule

$$(\text{LET})\quad \frac{A, B \vdash M : \sigma, \quad A, B\backslash x \cup \{x : \sigma\} \vdash N : \tau}{A, B \vdash (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau}$$

We call this a "rank 2.5" system, since it lies between ranks 2 and 3. For instance, it can type a term that is untypable in rank 2:

$$\mathbf{let}\ g = (\lambda x.xx)\ \mathbf{in}\ g(\lambda y.y) : \forall t.t \to t.$$

We will not pursue this further, because we already know how to extend $\mathbf{P}_2$ to a more general system, $\mathbf{P}$, that does not rely on let-polymorphism. We will describe $\mathbf{P}$ in a future paper.

We do not claim that we have solved the smartest recompilation problem for Standard ML. Standard ML has a rich module system, with type components in modules, and generative, user-definable, recursive datatypes. Our simple language does not support such features (nor does the work of Shao and Appel [57]). However, we have identified principal typings, or some equivalent, as the key ingredient of such a system.

2.8 Error messages

Up until now, we have concentrated on one benefit of principal typings: a term can be given a type without regard to the definitions of its free variables.

The flip side of this benefit is that a definition can be typed independently of its uses. We now show how this allows us to produce accurate error messages when our type inference algorithm is faced with a program containing type errors.

Consider a definition, $(\lambda xM)N$, in which some uses of the variable x cause type errors: they require types that N cannot satisfy. To perform type inference, we calculate the principal typings of both the operator and the operand, say

$$\begin{aligned} &A \vdash (\lambda xM) : (\textstyle\bigwedge_{i\in I} \sigma_i) \to \tau, \\ &A' \vdash N : \sigma'. \end{aligned}$$

By the principal typing property, we can calculate these principal typings in any order. To complete type inference, we simply check whether we can satisfy

$$\pi = \{S_i\sigma' \leq \sigma_i \mid i \in I\},$$

where each S_i renames $\mathrm{FTV}(A', \sigma')$ to fresh type variables. At this point we will discover all of the type errors related to x: for some i, the type $S_i\sigma'$ will not be able

to satisfy the constraints expressed by σ_i. If we take care to label each constraint with the use of x that produced it, we can output the offending uses, all in one batch.

Contrast this with the situation in ML. Assuming the definition is polymorphic, we must perform type inference on a let-expression (**let** $x = N$ **in** M). Without principal typings, we are forced to first calculate the principal type, σ, of N. We then process M, instantiating σ at each use of x. Errors are reported as they are encountered, at each use. But note, the errors of one definition can be interspersed with errors for other definitions, or with run-on errors. And the type σ may have been specialized for that particular (erroneous) use, leaving the programmer to understand a type only remotely related to the type σ of the definition.

2.9 Does ML have principal typings?

We have deliberately stated the principal typing property in a broad way, so that it can be applied to many different type systems.[2] In particular, we have not precisely defined what it means to *represent* all possible typings, because this will vary from one type system to another.

This imprecision makes it impossible for us to *prove* that a given type system lacks the principal typing property. Nevertheless, we do not know of a sensible formulation of principal typings for ML, and in particular, ML does not have principal typings in the sense of our Definition 2.2.3. For example, consider the following ML typings of the term xx.

$$\{x : \forall t.t\} \vdash xx : \forall t.t,$$
$$\{x : \forall t.t \to t\} \vdash xx : \forall t.t \to t.$$

Our intuition is that a principal typing EXPECTS LESS of its free variables and PROVIDES MORE than any other typing. We certainly cannot hope to derive a more general type for the term xx than $(\forall t.t)$, so the first judgment provides more than the second. However, the first judgment also makes a strong requirement on x: the type environment indicates that it too must have type $(\forall t.t)$. Thus the second judgment expects less than the first, and neither typing is more general than the other. Moreover, there is no typing more general than both the typings above. The obvious candidate,

$$\{x : \forall t.t \to t\} \vdash xx : \forall t.t,$$

is not derivable.

Why doesn't ML's principal type property imply the existence of principal typings? You might think that the principal typing of a term could be obtained from the principal type of the λ-closure of the term. But ML has only a restricted abstraction

[2] In fact, we could have stated it more broadly still: we assumed typing judgments were of the form $A \vdash M : \sigma$, but this is not always the case.

rule:

$$\frac{A \cup \{x : \tau_1\} \vdash M : \tau_2}{A \vdash (\lambda x M) : \tau_1 \to \tau_2} \quad \tau_1, \tau_2 \in \mathbf{T}_0$$

In ML, we cannot abstract over variables of polymorphic type; the only way of introducing polymorphic variables is through let-expressions.

Historically, there has been some confusion over the proper statement of properties like Properties A and B for ML. One reason is that Property A, which we have been calling the principal type property, is not strong enough to support type inference in ML. Milner's type inference algorithm W does require two inputs, a term M and type environment A, as suggested by Property A. But the output of the algorithm is not just a typing $A \vdash M : \sigma$. Instead, the output of the algorithm is a typing $A' \vdash M : \sigma$, where A' is a substitution instance of A. The reason is that in determining the type σ of M, the algorithm may discover that some of the types specified for external variables by A need to be further specialized.

Example 2.9.1 When Milner's algorithm is given inputs $A = \{x : \forall t.s \to t \to s\}$ and $M = x3$, its output is the typing

$$\{x : \forall t.s \to t \to s\} \vdash x3 : \forall t.\text{INT} \to t \to \text{INT}.$$

The property of ML that makes Milner's algorithm possible can be stated as follows.

Property C

Given: a term M typable in an instance of A.
There exists: a typing $A' \vdash M : \sigma$ representing all possible typings for M in instances of A.

Here we say that A' *is an instance of* A if $A' = SA$ for some $\mathbf{T}_0$ substitution S, and the principal typing "represents" the other typings in much the same way as Definition 2.2.3.

This property has some of the character of Property B, and may have led authors to claim, incorrectly, that ML has Property B. One widely cited paper that makes this claim is the study of Harper and Mitchell [20, p. 32]. This is corrected in the journal version of the paper [21], which instead claims that ML has the following property:

Property D

Given: an A-typable term M.
There exists: a typing $(A \cup A') \vdash M : \sigma$ representing all possible typings of M in A.

(A term M is said to be A-*typable* if there is a $\mathbf{T}_0$ type environment A' and type σ such that $(A \cup A') \vdash M : \sigma$.)

Property D tries to achieve as much of Property B as possible, while falling back on Property A when necessary. While ML does satisfy Property D, it should be pointed out that like Property A, Property D is insufficient for type inference. Instead we suggest a combination of Properties C and B:

Property E
Given: a term M typable in an extension of A.
There exists: a typing $A' \vdash M : \sigma$ representing all possible typings of M in extensions of A.

(An *extension* of a type environment A is a type environment $A' = SA \cup A''$, where S is a $\mathbf{T}_0$ substitution and A'' is a $\mathbf{T}_0$ type environment.)

This is exactly the sort of property we would need for type inference in the "rank 2.5" system described in §2.7.2.

2.10 Living without principal typings

If we want to work in a language lacking the principal typing property, we may still achieve some of its benefits by finding a "representation" for all possible typings. That is, we may relax the principal typing condition that the representatives themselves be typings.

Pushed to an extreme, this is nonsense—after all, M itself is a representation of all typings of M! But there is a middle ground. For example, the "representation" may be a typing *in another type system.*

This idea was the basis of the smartest recompilation system of Shao and Appel [57]. They defined a type system with the following property: for any ML typable term M, there is a judgement in the Shao-Appel system that encodes all of the ML typings for M, in an appropriate sense. They did not prove a principal typing property for their system, but it is essentially identical to a system of Damas [13]. Damas proved a principal typing theorem for his system, and showed that it types exactly the same terms as ML.

The systems $\mathbf{P}_2$ and Λ_2 are a second example of this phenomenon. We have already mentioned that $\mathbf{P}_2$ has principal typings and types exactly the same terms as Λ_2. However, Λ_2 does not have principal typings in the sense of Definition 2.2.3. The counterexample xx that we used for ML also works for Λ_2. Unlike ML, Λ_2 has a "true" abstraction rule; this is not a contradiction, because in addition to lacking principal typings, Λ_2 lacks principal types [35]. The equivalence of Λ_2 and $\mathbf{P}_2$ will be established in Chapter 3, and we will use this equivalence to define a type inference algorithm for Λ_2.

A third example of this phenomenon will be given in Chapter 4. Palsberg and Scott (personal communication, September 1995) have shown that the recursive type system of Amadio and Cardelli [4] types exactly the same terms as a type system

based on constraints [14]. In Chapter 4 we will define both of these systems, and show that the Palsberg-O'Keefe type inference algorithm for the Amadio-Cardelli system produces principal typings in the constraint-based system. An immediate corollary will be that the systems type precisely the same terms (exactly the result of Palsberg and Scott, but proved in a different way).

2.11 Related work

Principal typings are not a new concept. A number of existing type systems have principal typings, including the simply typed lambda calculus [67], the system of recursive types [8], the system of simple subtypes [49], and the system of intersection types [7]. Our contribution is to highlight the practical uses of the principal typing property, and to distinguish it from the principal type property. A number of authors have published offhand claims that ML possesses the principal typing property, despite the early remarks of Damas [13] to the contrary.

The system of rank 2 intersection types is also not new, but as with the principal typing property, it has attracted little attention. It was first suggested by Leivant in 1983 [41], but he did not give a formal definition of the type inference algorithm or proof of correctness. In an oft-referenced 1984 paper [45], McCracken gave a type inference algorithm for rank 2 of System F, inspired by Leivant's ideas. This algorithm is incorrect. A correct algorithm for rank 2 of System F was finally given by Kfoury and Wells [35] in 1993. Their algorithm is completely unrelated to Leivant's algorithm. The earliest formal definition and proof of Leivant's algorithm was published in 1993, by van Bakel [65]. All of the rank 2 systems will be discussed in more detail in the next chapter.

Our addition of top-level quantification is a useful technical improvement to the rank 2 intersection system. In particular, the simplicity of our rule for typing recursive definitions is due to the power of quantifiers and the subtyping relation $\leq_{\forall 2,1}$. It is possible to formulate an equivalent rule for typing recursive definitions without top-level quantification, but the machinery is cumbersome and simply duplicates the functionality of the quantifiers.

The constraint-based systems of Aiken and Wimmers [2], Jones [26], Kaes [28], and Smith [59] use ML's let-polymorphism, and, therefore, we believe they do not have principal typings. The restriction of these systems without let-polymorphism, though, is still of interest. In Chapter 4, we will show that one such restriction has principal typings.

Chapter 3

Rank 2 type systems

In this chapter we study the systems of rank 2 types originally introduced by Leivant [41]. One such system, $\mathbf{I}_2$, the system of rank 2 intersection types, is the basis of the type system $\mathbf{P}_2$ that we defined in last chapter. Here we will prove all of the results claimed there, including the existence of principal typings for $\mathbf{P}_2$, the correctness of the type inference algorithm, and the complexity of typability in the system.

Besides laying out the technical foundation of $\mathbf{P}_2$, we use this opportunity to demonstrate a way of designing a type inference algorithm for a type system without any notion of principal typing. Our example is rank 2 of System F, or Λ_2. We will show that Λ_2 is equivalent to $\mathbf{I}_2$ in that the systems type exactly the same terms. Thus type inference for $\mathbf{I}_2$ immediately solves the typability problem for Λ_2. Moreover, our proof of equivalence shows how we may obtain a Λ_2 typing from any $\mathbf{I}_2$ typing. Type inference for Λ_2 is achieved simply by constructing a Λ_2 typing from the principal $\mathbf{I}_2$ typing.

Historically, the rank 2 systems have attracted attention as an alternative to the let-polymorphism of ML. The limitations of let-polymorphism can be illustrated by the term $(\lambda x.xx)$. It is well known that this expression cannot be typed in ML: the only way for ML to type the self-application xx is by assigning a polymorphic type to x, and ML does not allow abstraction over variables with polymorphic type. In ML, the only mechanism for introducing variables of polymorphic type is the let-expression:

$$\begin{array}{l}\textbf{let } x = (\lambda y.y) \\ \textbf{in } xx.\end{array}$$

This let-expression binds x to the identity function $(\lambda y.y)$, which has the polymorphic type $\forall t.t \to t$ in ML. By ML's let-polymorphism, x is assigned the type $\forall t.t \to t$, which is sufficient to type xx.

The problem with this is that we cannot typecheck the uses of x (the application xx) separately from its definition (the function $(\lambda y.y)$). So ML must be extended

with a *module language* in order to support programming in the large, where it is impractical to require every polymorphic definition to appear in the same source file as every use.

In contrast, $(\lambda x.xx)$ is typable in all of the rank 2 systems we consider. Here are two rank 2 typings:

$$(\lambda x.xx) \;:\; (\forall t.t \to t) \to (\forall s.s \to s),$$

$$(\lambda x.xx) \;:\; (t \to t) \wedge ((t \to t) \to (t \to t)) \to (t \to t).$$

The first typing says that $(\lambda x.xx)$ is a function that, when given an argument with type $t \to t$ for *any* type t, produces a result with type $s \to s$, for any s. The identity function is an appropriate argument.

The second typing says that $(\lambda x.xx)$ is a function that, when given an argument having *both* the types $(t \to t)$ and $(t \to t) \to (t \to t)$, produces a result of type $(t \to t)$. Once again, the identity $(\lambda y.y)$ is an appropriate argument.

The rank 2 systems we consider are restrictions of two widely studied type systems, System F and the system of intersection types. System F, introduced independently by Girard [18] and by Reynolds [55], predates ML and can type many more terms. A recent result of Wells [69], however, shows that typability in the system is undecidable, putting type inference out of reach.

The system of intersection types, introduced independently by Coppo and Dezani [10] and by Sallé [56], can type even more terms than System F: it types all (and only) the strongly normalizing terms (without the type constant ω). The equivalence of typability and strong normalization implies that type inference, just as with System F, is unattainable.

With the goal of type inference in mind, we seek decidable restrictions of these type systems. Restrictions based on the *rank* of types were suggested by Leivant [41]. The rank of a type can be easily determined by examining it in tree form. A type is of rank k if no path from the root of the type to a type constructor of interest (either type intersection '$\wedge$' or type quantification '$\forall$') passes to the left of k arrows. The types shown in Figure 3.1 are rank 2 types, because no path from root to $\wedge$ or $\forall$ passes to the left of two arrows. But the types shown in Figure 3.2 go beyond rank 2 (they are rank 3 types). The types given above for $(\lambda x.xx)$ are rank 2 types.

Ranks 0 and 1 of Leivant's systems are equivalent to the simply typed lambda calculus, which can type fewer terms than ML. But starting with rank 2, the systems can type more terms than ML.

Rank 2 of System F, or Λ_2, has received the most study. McCracken [45] proposed a type inference algorithm for Λ_2 based on Leivant's ideas. This algorithm is incorrect. Kfoury and Tiuryn [30] show that the complexity of typability in Λ_2 is identical to that of ML. Kfoury and Wells [34, 35] give a correct type inference algorithm, and show that ranks 3 and higher in System F are undecidable.

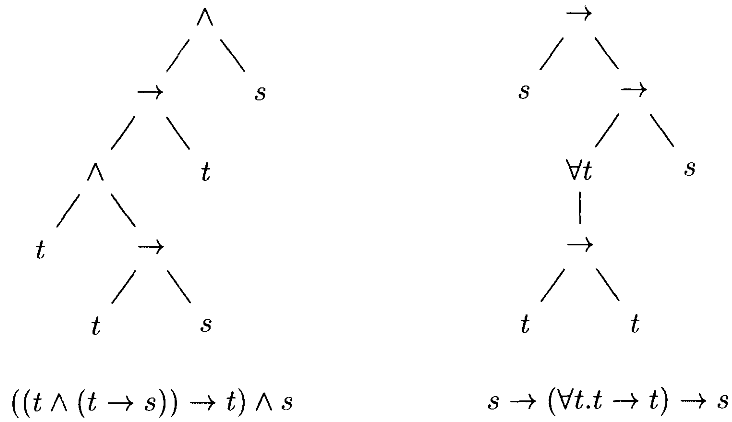


Figure 3.1: Examples of rank 2 types

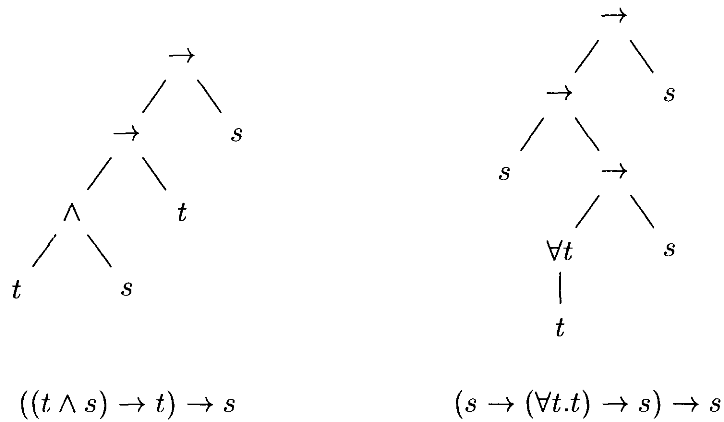
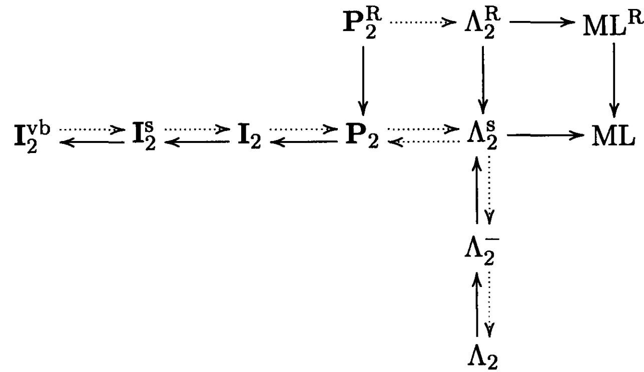


Figure 3.2: Types that go beyond rank 2

Leivant's original paper is almost the only work on $\mathbf{I}_2$, rank 2 of the intersection type discipline. Leivant sketched a type inference algorithm for $\mathbf{I}_2$, but the algorithm was not formalized and proved correct until recently [65]. Leivant also conjectured the undecidability of ranks 3 and higher in the intersection system; however, the details of his proof sketch have never been verified (personal communication, June 1995).

The following diagram summarizes the relationship of the type systems we will discuss in this chapter.



A solid arrow from a type system $\mathscr{T}$ to a type system $\mathscr{T}'$ indicates that every $\mathscr{T}'$ judgment is a $\mathscr{T}$ judgment. A dotted arrow from $\mathscr{T}$ to $\mathscr{T}'$ expresses a weaker implication, that every term typable in $\mathscr{T}'$ is typable in $\mathscr{T}$. We compare ML to the rank 2 systems by assuming that ML's let-expressions are considered as syntactic sugar in the rank 2 systems.

Organization of the chapter. In §3.1, we introduce $\mathbf{I}_2^{\mathrm{s}}$, a syntax-directed version of $\mathbf{I}_2$, and in §3.2 we introduce Λ_2^{s}, a syntax-directed version of Λ_2. We define ML in §3.3; its principal type property is used in §3.4 to prove the equivalence of Λ_2^{s} and $\mathbf{I}_2^{\mathrm{s}}$. The main result is that a term is typable in one system if and only if it is typable in the other. An immediate corollary is that typability in $\mathbf{I}_2^{\mathrm{s}}$ is DEXPTIME-complete, the same complexity as in ML and in Λ_2^{s}. In §3.5, we present the type inference algorithm for $\mathbf{I}_2^{\mathrm{s}}$, and in §3.6 we show how this gives a type inference algorithm for Λ_2^{s}. In §3.7, we discuss some other definitions of rank 2 intersection type systems, and establish their equivalence with $\mathbf{I}_2$. In §3.8, we discuss ways of typing recursive definitions in the rank 2 systems. Finally, in §3.9, we prove the correctness of our type inference algorithm for $\mathbf{P}_2$, and relate it to the other rank 2 systems.

(VAR) $A \cup \{x : (\bigwedge_{i \in I} \tau_i)\} \vdash x : \tau_{i_0}$ (where $i_0 \in I$)

(ABS) $$\frac{A \backslash x \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x M) : \sigma \to \tau}$$

(APP) $$\frac{A \vdash M : (\bigwedge_{i \in I} \tau_i) \to \sigma, \quad (\forall i \in I)\ A \vdash N : \tau_i}{A \vdash (MN) : \sigma}$$

Figure 3.3: Typing rules of $\mathbf{I}_2^{\mathrm{s}}$. Types in type environments are in $\mathbf{T}_1$, and derived types are in $\mathbf{T}_2$.

3.1 The rank 2 intersection type system

There are many different formulations of intersection type systems; see van Bakel [65] for a survey. We will present a very restricted intersection type system here, the system of rank 2 intersection types. Our system is a slight generalization of van Bakel's version (see §3.7.1).

The *terms of the intersection type system* are just the terms of the lambda calculus. The types of the system include the types $\mathbf{T}_0$, $\mathbf{T}_1$ and $\mathbf{T}_2$ defined in the last chapter (p. 18). We write $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \sigma$ if the judgment $A \vdash M : \sigma$ follows by the rules of Figure 3.3, with types appearing in type environments restricted to $\mathbf{T}_1$, and derived types restricted to $\mathbf{T}_2$. The superscript 's' in $\mathbf{I}_2^{\mathrm{s}}$ indicates that the system is syntax-directed, a useful technical property. A comparison with the system $\mathbf{P}_2$ of Chapter 2 shows that this has been accomplished by dropping the rule (ADD-HYP), and strengthening the rules (VAR) and (ABS).

The following lemma summarizes some basic facts about the various subtyping relations.

Lemma 3.1.1

i) *If* $\sigma \in \mathbf{T}_0$ *and* $\tau \in \mathbf{T}_1$, *then* $\sigma \leq_1 \tau$ *iff* $\sigma = \tau$.

ii) *If* $\sigma \in \mathbf{T}_2$ *and* $\tau \in \mathbf{T}_0$, *then* $\sigma \leq_2 \tau$ *iff* $\sigma = \tau$.

iii) *For* $i \in \{1,2\}$, *if* $\sigma \leq_i \tau$, *then* $S\sigma \leq_i S\tau$.

The next lemma states that derivable typings are closed under certain operations.

Lemma 3.1.2 (Sound operations on typings) *If* $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \sigma$, *then*

- $\mathbf{I}_2^{\mathrm{s}} \rhd A' \vdash M : \sigma$, *where* $A' \leq_1 A$; *and*
- $\mathbf{I}_2^{\mathrm{s}} \rhd SA \vdash M : S\sigma$ *for any* $\mathbf{T}_0$ *substitution* S.

Proof: An easy induction on typing derivations. □

The intuition behind the first operation is that whenever a set A of assumptions is sufficient to derive a type σ for a term M, then any set A' of *stronger* assumptions can also derive σ as a type for M. Because the typing $A' \vdash M : \sigma$ uses stronger assumptions to derive the same type, it is a weaker statement than the typing $A \vdash M : \sigma$. Therefore, we say that the typing $A' \vdash M : \sigma$ is obtained from the typing $A \vdash M : \sigma$ by *weakening*.

The second operation expresses the intuition that a type variable may be used in place of any simple type; we say that the typing $SA \vdash M : S\sigma$ is obtained from the typing $A \vdash M : \sigma$ by *substitutivity*.

3.2 Rank 2 of System F

The *terms of System F* are exactly the terms of the lambda calculus. The *types of System F* are defined by the following grammar:

$$\tau ::= t \mid (\tau_1 \to \tau_2) \mid (\forall t \tau)$$

We write $\mathbf{T}_\forall$ for the set of System F types, and we consider $\mathbf{T}_\forall$ types to be syntactically equal modulo renaming of bound type variables, reordering of adjacent quantifiers, and elimination of unnecessary quantifiers.

The types of System F can be organized into a hierarchy as follows. First, define $\mathbf{R}(0) = \mathbf{T}_0$. Then for $n \geq 0$, the set $\mathbf{R}(n+1)$ is defined to be the least set satisfying

$$\begin{aligned} \mathbf{R}(n+1) &= \mathbf{R}(n) \cup \{(\sigma \to \tau) \mid \sigma \in \mathbf{R}(n), \tau \in \mathbf{R}(n+1)\} \\ &\quad \cup \{(\forall t \sigma) \mid \sigma \in \mathbf{R}(n+1)\}. \end{aligned}$$

It will be useful to restrict types so that quantifiers do not appear to the immediate right of arrows. Therefore we define the sets

$$\begin{aligned} \mathbf{S} &= \mathbf{S}' \cup \{(\forall t \sigma) \mid \sigma \in \mathbf{S}\}, \\ \mathbf{S}' &= \mathbf{T}_0 \cup \{(\sigma \to \tau) \mid \sigma \in \mathbf{S}, \tau \in \mathbf{S}')\}. \end{aligned}$$

We write $\mathbf{S}(n)$ for $\mathbf{S} \cap \mathbf{R}(n)$ and $\mathbf{S}'(n)$ for $\mathbf{S}' \cap \mathbf{R}(n)$. Note that the $\mathbf{S}(1)$ types are exactly the ML type schemes.

Definition 3.2.1 Suppose $\sigma = \forall t_1 \cdots t_n.\tau \in \mathbf{S}(1)$, and $\tau, \tau' \in \mathbf{T}_0$. We say τ' *is an instance of* σ, written $\sigma \succ \tau'$, if and only if for some $\rho_1, \ldots, \rho_n \in \mathbf{T}_0$, we have $\tau' = \{t_1 := \rho_1, \ldots, t_n := \rho_n\}\tau$. We write $\sigma \succ (\forall s_1 \cdots s_m \tau')$ if and only if $s_1, \ldots, s_m$ are not free in σ and $\sigma \succ \tau'$.

$$(\text{VAR}) \quad A \cup \{x : \sigma\} \vdash x : \tau \quad (\text{where } \sigma \succ \tau)$$

$$(\text{ABS}) \quad \frac{A \backslash x \cup \{x : \tau_1\} \vdash M : \tau_2}{A \vdash (\lambda x M) : \tau_1 \to \tau_2}$$

$$(\text{APP}) \quad \frac{A \vdash M : (\forall \vec{t} \tau_1) \to \tau_2, \quad A \vdash N : \tau_1}{A \vdash (MN) : \tau_2} \quad (\text{each } t_i \notin \text{FTV}(A))$$

Figure 3.4: Typing rules of Λ_2^{s}. Types in type environments are in $\mathbf{S}(1)$, and derived types are in $\mathbf{S}'(2)$.

Note that the sense of '$\succ$' is opposite to that of our other subtyping relations; for example, both "$\sigma \leq_2 \tau$" and "$\sigma \succ \tau$" may be read, "σ is more general than τ." We make an exception in the case of '$\succ$' to be consistent with its use in ML [46].

We now define Λ_2^{s}, our version of the rank 2 fragment of System F. The superscript 's' in Λ_2^{s} indicates that the system is syntax-directed. See Kfoury and Tiuryn [30] for a definition of Λ_2, the non-syntax-directed version.

The judgments of the system are defined by the rules of Figure 3.4. We write $\Lambda_2^{\text{s}} \rhd A \vdash M : \tau$ if $A \vdash M : \tau$ is derivable from these rules, where types in type environments are restricted to $\mathbf{S}(1)$, and derived types are restricted to $\mathbf{S}'(2)$.

Λ_2^{s} is closely related to the system Λ_2^- studied by Kfoury et al. [30, 35]:

Theorem 3.2.2 (Comparison of Λ_2^{s} and Λ_2^-)

- *If* $\Lambda_2^{\text{s}} \rhd A \vdash M : \sigma$, *then* $\Lambda_2^- \rhd A \vdash M : \sigma$.
- *If* $\Lambda_2^- \rhd A \vdash M : \sigma$, *then* σ *is of the form* $\forall t_1 \cdots t_n \sigma'$, *where* $\sigma' \in \mathbf{S}'(2)$, *and* $\Lambda_2^{\text{s}} \rhd A \vdash M : \sigma'$.

This equivalence follows immediately from results of Kfoury and Wells [35]. It implies the following useful result:

Lemma 3.2.3 *If* $\Lambda_2^{\text{s}} \rhd A \vdash M : \sigma$ *and* $\text{Gen}(A, \sigma) \succ \sigma'$, *then* $\Lambda_2^{\text{s}} \rhd A \vdash M : \sigma'$.

The relationship of Λ_2^- and Λ_2 is established by Kfoury and Tiuryn [30].

3.3 ML

Many different formulations of the ML type system have been studied; we present a syntax-directed version here, cf. Clement et al. [9] or Tofte [64].

$$(\text{VAR}) \quad A \cup \{x : \sigma\} \vdash x : \tau \quad (\text{where } \sigma \succ \tau)$$

$$(\text{APP}) \quad \frac{A \vdash M : \tau_1 \to \tau_2, \quad A \vdash N : \tau_1}{A \vdash (MN) : \tau_2}$$

$$(\text{ABS}) \quad \frac{A \backslash x \cup \{x : \tau_1\} \vdash M : \tau_2}{A \vdash (\lambda x M) : \tau_1 \to \tau_2}$$

$$(\text{LET}) \quad \frac{A \vdash M_1 : \tau_1, \quad A \backslash x \cup \{x : \text{Gen}(A, \tau_1)\} \vdash M_2 : \tau_2}{A \vdash (\textbf{let } x = M_1 \textbf{ in } M_2) : \tau_2}$$

Figure 3.5: Typing rules of ML. Types in type environments are in $\mathbf{S}(1)$, and derived types are in $\mathbf{T}_0$.

The *types of ML* are the types $\mathbf{T}_0$, and the *ML type schemes* are the types $\mathbf{S}(1)$. The *terms of ML* are the terms of the lambda calculus extended with *let-expressions*:

$$M ::= x \mid (M_1 M_2) \mid (\lambda x M) \mid (\textbf{let } x = M_1 \textbf{ in } M_2)$$

The judgments of ML are defined inductively by the rules of Figure 3.5. We write ML $\triangleright A \vdash M : \tau$ if $A \vdash M : \tau$ is derivable by these rules, where types in type environments are restricted to $\mathbf{S}(1)$, and derived types are restricted to $\mathbf{T}_0$.

Definition 3.3.1 An ML type τ is a *principal type for M in A* if and only if ML $\triangleright$ $A \vdash M : \tau$, and for all ML types τ', if ML $\triangleright A \vdash M : \tau'$, then $\text{Gen}(A, \tau) \succ \tau'$.

Theorem 3.3.2 (Principal types for ML) *If M is typable by A, then there exists a principal type for M in A.*

Lemma 3.3.3 *If* ML $\triangleright A \vdash M : \tau$, *and* $\text{Gen}(A, \tau) \succ \tau'$, *then* ML $\triangleright A \vdash M : \tau'$.

3.4 Relationship of Λ_2^{s} and $\mathbf{I}_2^{\mathrm{s}}$

Here we show that a term is typable in Λ_2^{s} if and only if it is typable in $\mathbf{I}_2^{\mathrm{s}}$. Before stating the main theorem, we develop machinery that will allow us to relate $\mathbf{I}_2^{\mathrm{s}}$ and Λ_2^{s} typings in a precise way.

Definition 3.4.1

- We define a relation $\preceq_1$ between $\mathbf{S}(1)$ and $\mathbf{T}_1$ as follows. Suppose $\tau \in \mathbf{S}(1)$ and $\sigma_1, \ldots, \sigma_n \in \mathbf{T}_0$ $(n \geq 1)$. Then $\tau \preceq_1 (\bigwedge_{i \in I} \sigma_i)$ if and only if $\tau \succ \sigma_i$ for all $i \in I$.

- We define the relation $\preceq_2$ between $\mathbf{S}'(2)$ and $\mathbf{T}_2$ inductively:
  - For any type variable t, $t \preceq_2 t$.
  - If $\tau \preceq_1 \tau'$ and $\sigma \preceq_2 \sigma'$, then $(\tau \to \sigma) \preceq_2 (\tau' \to \sigma')$.

Note that the relation $\preceq_2$ is monotonic in the argument of function types, in contrast to the relation $\leq_2$. We extend the relation $\preceq_1$ to type environments as follows: $A \preceq_1 A'$ if and only if $x \in \mathbf{dom}(A)$ and $A(x) \preceq_1 A'(x)$ whenever $x \in \mathbf{dom}(A')$. Note that $A \preceq_1 (A' + A'')$ if $A \preceq_1 A'$ and $A \preceq_1 A''$, and $A \preceq_1 A'$ if $A \backslash x \preceq_1 A'$.

Theorem 3.4.2 (Comparison of $\mathbf{I}_2^{\mathrm{s}}$ and Λ_2^{s}) *$\Lambda_2^{\mathrm{s}} \rhd A \vdash M : \tau$ iff $\mathbf{I}_2^{\mathrm{s}} \rhd A' \vdash M : \tau'$ for some A' and τ' such that $A \preceq_1 A'$ and $\tau \preceq_2 \tau'$.*

Proof: By Lemmas 3.4.5 and 3.4.16 below. □

Corollary 3.4.3 *If M is a term of the pure lambda calculus, then M is typable in $\mathbf{I}_2^{\mathrm{s}}$ if and only if M is typable in Λ_2^{s}.*

This equivalence has been shown independently by Yokouchi [70].

Corollary 3.4.4 *Typability in $\mathbf{I}_2^{\mathrm{s}}$ is DEXPTIME-complete.*

Proof: Kfoury and Tiuryn [30] show that Λ_2^{s} typability is polynomial time equivalent to ML typability. ML typability was shown to be DEXPTIME-complete independently by Kfoury et al. [33] and by Mairson [43]. □

The left-to-right direction of Theorem 3.4.2 is proved by a straightforward induction on M.

Lemma 3.4.5 *If $\Lambda_2^{\mathrm{s}} \rhd A \vdash M : \tau$, then $\mathbf{I}_2^{\mathrm{s}} \rhd A' \vdash M : \tau'$ for some A' and τ' such that $A \preceq_1 A'$ and $\tau \preceq_2 \tau'$.*

Proof: By induction on M.

- If $M = x$, then $\Lambda_2^{\mathrm{s}} \rhd A \vdash x : \tau$ follows by the Λ_2^{s} rule (VAR), so $A(x) \succ \tau$.

  Let $A' = \{x : \tau\}$. Clearly $\mathbf{I}_2^{\mathrm{s}} \rhd A' \vdash M : \tau$, $A \preceq_1 A'$, and $\tau \preceq_2 \tau$.

- If $M = \lambda x N$, then $\tau = \sigma \to \tau_1$ and $\Lambda_2^{\mathrm{s}} \rhd A \vdash \lambda x N : \sigma \to \tau_1$ follows by the Λ_2^{s} rule (ABS).

  Then we must have
  $$\Lambda_2^{\mathrm{s}} \rhd A \backslash x \cup \{x : \sigma\} \vdash N : \tau_1.$$
  By induction, we have
  $$\mathbf{I}_2^{\mathrm{s}} \rhd A' \cup \{x : \sigma'\} \vdash N : \tau_1',$$

where $A\backslash x \preceq_1 A'$, $\sigma \preceq_1 \sigma'$, and $\tau_1 \preceq_2 \tau_1'$. So by the $\mathbf{I}_2^{\mathrm{s}}$ rule (ABS), we have

$$\mathbf{I}_2^{\mathrm{s}} \triangleright A' \vdash N : \sigma' \to \tau_1',$$

where $A \preceq_1 A'$, and $(\sigma \to \tau_1) \preceq_2 (\sigma' \to \tau_1')$, as desired.

- If $M = M_1 M_2$, then $\Lambda_2^{\mathrm{s}} \triangleright A \vdash M_1 M_2 : \tau$ follows by the Λ_2^{s} rule (APP). Then we must have, for some $\tau_0 \in \mathbf{T}_0$,

$$\begin{array}{l} \Lambda_2^{\mathrm{s}} \triangleright A \vdash M_1 : (\forall \vec{t}.\tau_0) \to \tau, \\ \Lambda_2^{\mathrm{s}} \triangleright A \vdash M_2 : \tau_0, \end{array}$$

where the type variables $\vec{t}$ do not appear in $\mathrm{FTV}(A)$. Then by induction we have

$$\mathbf{I}_2^{\mathrm{s}} \triangleright A_0' \vdash M_1 : (\bigwedge_{i \in I} \tau_i) \to \tau',$$

where $A \preceq_1 A_0'$, $\tau \preceq_2 \tau'$, and $(\forall \vec{t}.\tau_0) \preceq_1 (\bigwedge_{i \in I} \tau_i)$.

Then each τ_i is an instance of $(\forall \vec{t}.\tau_0)$, and therefore by Lemma 3.2.3, $\Lambda_2^{\mathrm{s}} \triangleright A \vdash M_2 : \tau_i$ for all $i \in I$.

By induction we have for all $i \in I$, $\mathbf{I}_2^{\mathrm{s}} \triangleright A_i' \vdash M_2 : \tau_i$, where $A \preceq_1 A_i'$. So if $A' = A_0' + \sum_{i \in I} A_i'$, then $A \preceq_1 A'$, and by weakening,

$$\begin{array}{ll} \mathbf{I}_2^{\mathrm{s}} \triangleright A' \vdash M_1 : (\bigwedge_{i \in I} \tau_i) \to \tau', & \\ \mathbf{I}_2^{\mathrm{s}} \triangleright A' \vdash M_2 : \tau_i & (\forall i \in I). \end{array}$$

Then by the $\mathbf{I}_2^{\mathrm{s}}$ rule (APP) we have

$$\mathbf{I}_2^{\mathrm{s}} \triangleright A' \vdash M_1 M_2 : \tau',$$

as desired.

□

We devote the rest of this section to proving the right-to-left direction of Theorem 3.4.2.

Convention 3.4.6 In the remainder of this section we do not consider terms to be identical modulo α-conversion, and we will assume the following convention regarding the names of bound and free variables:

- No variable is bound more than once.
- The bound and free variables are disjoint.

This convention is necessary to make the following function well-defined:

Definition 3.4.7 Let ϵ denote the empty sequence. The function, **act**, that maps terms to sequences of variables, is defined inductively by the following rules.[1]

- $\mathbf{act}(x) = \epsilon$.
- If $\mathbf{act}(M) = x_1, \ldots, x_n$ then $\mathbf{act}(\lambda y M) = y, x_1, \ldots, x_n$.
- If $\mathbf{act}(M) = y, x_1, \ldots, x_n$ $(n \geq 0)$ then $\mathbf{act}(MN) = x_1, \ldots, x_n$.
- If $\mathbf{act}(M) = \epsilon$ then $\mathbf{act}(MN) = \epsilon$.

Definition 3.4.8

- γ is the rule
$$(\lambda x(\lambda y M))N \rightarrow \lambda y((\lambda x M)N).$$
- $\rightarrow_\gamma$ is the compatible closure of γ.
- A γ-redex is any term matching the left-hand side of the rule γ. We say M is a γ-normal form, or γ-nf, if no subterm of M is a γ-redex.

Note that by our convention on the distinct naming of variables, there is no capture of variables in the γ rule. We use the name "γ" in accordance with Kfoury and Wells [36]. See Barendregt [6] for a definition of "compatible."

Lemma 3.4.9

i) $\rightarrow_\gamma$ *is strongly normalizing.*

ii) $\rightarrow_\gamma$ *satisfies the diamond property.*

iii) γ*-nf's are unique.*

Proof:

i) The proof is similar to the proof of Lemma 5.5 from Kfoury and Wells [35]:

Let $\mathbf{appl}(M)$ be the set of subterms of M that are applications, and let
$$\delta(M) = \sum_{(M_1 M_2) \in \mathbf{appl}(M)} \max(0, |\mathbf{act}(M_1)| - 1).$$

If $M \rightarrow_\gamma N$, then $\delta(M) = \delta(N) + 1$. Since for any M we have $\delta(M) \geq 0$, we can conclude that $\rightarrow_\gamma$ is strongly normalizing. In fact, $\delta(M) > 0$ iff M contains a γ-redex, and M normalizes in exactly $\delta(M)$ steps.

If $|M|$ is the size (number of subterms) of M, then clearly $|\mathbf{appl}(M)| \leq |M|$ and $|\mathbf{act}(M)| \leq |M|$. Thus $\delta(M) \leq |M|^2$. Therefore normalization of a term M takes $O(|M|^2)$ steps.

[1] Our definition is identical to the definition of [30], but differs from [29].

ii) This is a simple case analysis.

iii) This follows from (ii).

□

We write γ-nf(M) for the γ-nf of M.

Lemma 3.4.10 *For $\mathscr{T} \in \{\mathbf{I}_2^s, \Lambda_2^s\}$, the following hold:*

i) $\mathscr{T} \triangleright A \vdash (\lambda x(\lambda y M))N : \sigma$ *iff* $\mathscr{T} \triangleright A \vdash \lambda y((\lambda x M)N) : \sigma$.

ii) *If* $M \rightarrow_\gamma N$, *then* $\mathscr{T} \triangleright A \vdash M : \sigma$ *iff* $\mathscr{T} \triangleright A \vdash N : \sigma$.

iii) $\mathscr{T} \triangleright A \vdash M : \sigma$ *iff* $\mathscr{T} \triangleright A \vdash \gamma\text{-nf}(M) : \sigma$.

Proof:

i) Simple case analysis.

ii) Use (i) and induction on the definition of compatible.

iii) Use (ii) and induction on the length of rewriting.

□

Lemma 3.4.11 *If* $\mathbf{act}(M) = x_1, \ldots, x_n$ *and* $\mathbf{I}_2^s \triangleright A \vdash M : \sigma$, *then* σ *is of the form* $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$, *where* $\tau \in \mathbf{T}_0$.

Proof: By induction on the structure of M.

- If $M = x$, then $n = 0$ by the definition of **act**, and $\sigma \in \mathbf{T}_0$ by rule (VAR).
- If $M = \lambda x_1 N$, then $\mathbf{I}_2^s \triangleright A \vdash M : \sigma$ follows by rule (ABS), and therefore σ is of the form $\sigma_1 \rightarrow \sigma'$, where $\sigma_1 \in \mathbf{T}_1$.

  Also we must have $\mathbf{act}(N) = x_2, \ldots, x_n$ ($n \geq 1$) and $\mathbf{I}_2^s \triangleright A \cup \{x_1 : \sigma_1\} \vdash N : \sigma'$. By induction σ' must be of the form $\sigma_2 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$, where $\sigma_2, \ldots, \sigma_n \in \mathbf{T}_1$ and $\tau \in \mathbf{T}_0$.
- If $M = M_1 M_2$, then $\mathbf{I}_2^s \triangleright A \vdash M : \sigma$ follows by rule (APP), and therefore we have $\mathbf{I}_2^s \triangleright A \vdash M_1 : \sigma' \rightarrow \sigma$, where $\sigma' \in \mathbf{T}_1$.

  We consider two cases. If $\mathbf{act}(M_1) = y, x_1, \ldots, x_n$ for some variable y, then by induction, σ is of the form $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$, where $\sigma_1, \ldots, \sigma_n \in \mathbf{T}_1$ and $\tau \in \mathbf{T}_0$.

  Otherwise $\mathbf{act}(M_1) = \epsilon$, and therefore $\mathbf{act}(M) = \epsilon$, so we only need prove $\sigma \in \mathbf{T}_0$. And by induction, we have $(\sigma' \rightarrow \sigma) \in \mathbf{T}_0$, so $\sigma \in \mathbf{T}_0$.

□

A similar lemma holds for Λ_2^s, cf. Kfoury et al. [30], Lemma 15.

Lemma 3.4.12 *Suppose M is a γ-nf. Then*

$$\mathbf{act}(M) \neq \epsilon \quad \textit{iff} \quad M = \lambda y N \textit{ for some } y, N.$$

Proof: By induction on the structure of M. The cases $M = x$ and $M = \lambda y N$ are trivial, so assume $M = M_1 M_2$. We must show $\mathbf{act}(M) = \epsilon$.

By way of contradiction, assume that $\mathbf{act}(M) = x_1, \ldots, x_n$ $(n \geq 1)$. By the definition of **act**, we must have $\mathbf{act}(M_1) = y, x_1, \ldots, x_n$ for some y. Then $\mathbf{act}(M_1) \neq \epsilon$, so by induction we have $M_1 = \lambda y M_1'$, and $\mathbf{act}(M_1') = x_1, \ldots, x_n$. Since $n \geq 1$, $\mathbf{act}(M_1') \neq \epsilon$, and by induction $M_1' = \lambda x_1 M_1''$. But then M is a γ-redex, contradiction. □

Definition 3.4.13 We define a mapping, **ml**, from terms to ML terms:

- $\mathbf{ml}(x) = x$.
- $\mathbf{ml}(\lambda x M) = (\lambda x\ \mathbf{ml}(M))$.
- $\mathbf{ml}(M_1 M_2) = \begin{cases} (\mathbf{let}\ x = \mathbf{ml}(M_2)\ \mathbf{in}\ \mathbf{ml}(N)) & \text{if } M_1 = \lambda x N, \\ (\mathbf{ml}(M_1)\ \mathbf{ml}(M_2)) & \text{otherwise.} \end{cases}$

Definition 3.4.14

- A *generalization* of a set $\mathbf{T}$ of simple types is a type $\sigma \in \mathbf{S}(1)$ such that $\sigma \succ \tau$ for every $\tau \in \mathbf{T}$. A generalization σ of $\mathbf{T}$ is the *least common generalization* of $\mathbf{T}$ if $\sigma' \succ \sigma$ for any other generalization σ' of $\mathbf{T}$.
- If $(\bigwedge_{i \in I} \tau_i) \in \mathbf{T}_1$, we define $\mathbf{lcg}(\bigwedge_{i \in I} \tau_i)$ to be the least common generalization of $\{\tau_i \mid i \in I\}$. If $\sigma_1, \ldots, \sigma_n \in \mathbf{T}_1$ and $\tau \in \mathbf{T}_0$, then

$$\mathbf{lcg}(\sigma_1 \to \cdots \to \sigma_n \to \tau) = \mathbf{lcg}(\sigma_1) \to \cdots \to \mathbf{lcg}(\sigma_n) \to \tau.$$

The function **lcg** is extended to type environments in the usual way.

The use of "least" in the name "least common generalization" is consistent with the relation '$\succ$'. Recall that the sense of '$\succ$' is opposite to that of our other subtyping relations, so that "least" for '$\succ$' means "greatest" for the other relations.

The concept of least common generalizations was developed by Plotkin [52] and Reynolds [54]. They showed that any finite nonempty set of simple types has a least common generalization, and they gave an algorithm to compute it.

Lemma 3.4.15 *If M is a γ-nf and $\sigma \in \mathbf{T}_0$, then*

i) *if $A \preceq_1 A'$ and* $\mathbf{I}_2^{\rm s} \triangleright A' \vdash M : \sigma$, *then* $\mathrm{ML} \triangleright A \vdash \mathbf{ml}(M) : \sigma$; *and*

ii) $\Lambda_2^{\rm s} \triangleright A \vdash M : \sigma$ *if and only if* $\mathrm{ML} \triangleright A \vdash \mathbf{ml}(M) : \sigma$.

Proof:

i) By induction on the structure of M.

- The case $M = x$ is trivial.
- If $M = \lambda y N$, then $\mathbf{I}_2^{\mathrm{s}} \triangleright A' \vdash M : \sigma$ follows by the $\mathbf{I}_2^{\mathrm{s}}$ rule (ABS), so σ must be of the form $\tau \to \sigma'$ where $\tau, \sigma' \in \mathbf{T}_0$, and $\mathbf{I}_2^{\mathrm{s}} \triangleright A' \backslash y \cup \{y : \tau\} \vdash N : \sigma'$. Note that $A \backslash y \cup \{y : \tau\} \preceq_1 A' \backslash y \cup \{y : \tau\}$, and N is a γ-nf, so we can apply the induction hypothesis to get

$$\mathrm{ML} \triangleright A \backslash y \cup \{y : \tau\} \vdash \mathbf{ml}(N) : \sigma'.$$

  Finally since $\mathbf{ml}(\lambda y N) = \lambda y\ (\mathbf{ml}(N))$, by the ML rule (ABS),

$$\mathrm{ML} \triangleright A \vdash \mathbf{ml}(\lambda y N) : \tau \to \sigma'.$$

- If $M = (\lambda y M_1)M_2$, then our judgment must follow by the $\mathbf{I}_2^{\mathrm{s}}$ rules (ABS) and (APP). Thus we have

$$\begin{array}{rl} & \mathbf{I}_2^{\mathrm{s}} \triangleright A' \backslash y \cup \{y : (\bigwedge_{i \in I} \sigma_i)\} \vdash M_1 : \sigma, \\ (\forall i \in I) & \mathbf{I}_2^{\mathrm{s}} \triangleright A' \vdash M_2 : \sigma_i. \end{array}$$

  Let $\forall \vec{t} \tau = \mathbf{lcg}(\bigwedge_{i \in I} \sigma_i)$, where $\tau \in \mathbf{T}_0$, and no t_i appears in A. By induction, we have

$$\begin{array}{rl} & \mathrm{ML} \triangleright A \backslash y \cup \{y : \forall \vec{t} \tau\} \vdash \mathbf{ml}(M_1) : \sigma, \\ (\forall i \in I) & \mathrm{ML} \triangleright A \vdash \mathbf{ml}(M_2) : \sigma_i. \end{array}$$

  By the principal type property of ML, we have

$$\mathrm{ML} \triangleright A \vdash \mathbf{ml}(M_2) : \tau.$$

  Then since $\mathbf{ml}(M) = (\mathbf{let}\ y = \mathbf{ml}(M_2)\ \mathbf{in}\ \mathbf{ml}(M_1))$, we have

$$\mathrm{ML} \triangleright A \vdash \mathbf{ml}(M) : \sigma$$

  by the ML rule (LET).
- If $M = M_1 M_2$, where M_1 is not an abstraction, then by the $\mathbf{I}_2^{\mathrm{s}}$ rule (APP), we have for some $\sigma' \in \mathbf{T}_1$,

$$\mathbf{I}_2^{\mathrm{s}} \triangleright A' \vdash M_1 : \sigma' \to \sigma.$$

  Now M_1 is a γ-nf and is not an abstraction, so by Lemma 3.4.12, we have $\mathbf{act}(M_1) = \epsilon$. Then by Lemma 3.4.11, $\sigma' \to \sigma \in \mathbf{T}_0$, and therefore $\sigma' \in \mathbf{T}_0$. So by the $\mathbf{I}_2^{\mathrm{s}}$ rule (APP), we have

$$\mathbf{I}_2^{\mathrm{s}} \triangleright A' \vdash M_2 : \sigma'.$$

M_2 is also a γ-nf, so we may apply the induction hypothesis to both judgments above, to get

$$\mathrm{ML} \rhd A \vdash \mathbf{ml}(M_1) : \sigma' \to \sigma,$$
$$\mathrm{ML} \rhd A \vdash \mathbf{ml}(M_2) : \sigma'.$$

Then by the ML rule (APP), we have

$$\mathrm{ML} \rhd A \vdash \mathbf{ml}(M_1 M_2) : \sigma,$$

as desired.

ii) Similar, but easier.

□

The converse of Lemma 3.4.15(i) does not hold. For instance, if $\sigma = t_3$, $A' = \{x : t_1 \wedge t_2\}$, and $A = \{x : \forall t.t\}$, then $A \preceq_1 A'$, $\mathbf{ml}(xx) = xx$, and $\mathrm{ML} \rhd A \vdash xx : t_3$, but the judgment $A' \vdash xx : t_3$ cannot be derived in $\mathbf{I}_2^{\mathrm{s}}$.

Lemma 3.4.16 *If* $\mathbf{I}_2^{\mathrm{s}} \rhd A' \vdash M : \sigma'$, $A \preceq_1 A'$, *and* $\sigma \preceq_2 \sigma'$, *then* $\Lambda_2^{\mathrm{s}} \rhd A \vdash M : \sigma$.

Proof: Suppose $\mathbf{act}(M) = x_1, \ldots, x_n$. Then by Lemma 3.4.11, σ' is of the form $\sigma_1' \to \cdots \to \sigma_n' \to \tau$, where $\tau \in \mathbf{T}_0$, and by Lemma 3.4.12, the γ-nf of M is of the form $\lambda x_1 \cdots \lambda x_n N$, where N is a γ-nf. By Lemma 3.4.10(iii),

$$\mathbf{I}_2^{\mathrm{s}} \rhd A' \vdash \lambda x_1 \cdots \lambda x_n N : \sigma'.$$

This judgment must follow by n uses of the $\mathbf{I}_2^{\mathrm{s}}$ rule (ABS), so we have

$$\mathbf{I}_2^{\mathrm{s}} \rhd A' \backslash \{x_1, \ldots, x_n\} \cup \{x_1 : \sigma_1', \ldots, x_n : \sigma_n'\} \vdash N : \tau.$$

By the definition of $\preceq_2$, σ must be of the form $\sigma_1 \to \cdots \to \sigma_n \to \tau$, where $\sigma_i \preceq_1 \sigma_i'$ for $1 \le i \le n$. Then by Lemma 3.4.15, we have

$$\Lambda_2^{\mathrm{s}} \rhd A \backslash \{x_1, \ldots, x_n\} \cup \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \vdash N : \tau.$$

By n uses of the Λ_2^{s} rule (ABS), we have

$$\Lambda_2^{\mathrm{s}} \rhd A \vdash \lambda x_1 \cdots \lambda x_n N : \mathbf{lcg}(\sigma),$$

and by Lemma 3.4.10(iii), we have

$$\Lambda_2^{\mathrm{s}} \rhd A \vdash M : \sigma.$$

□

3.5 Type inference for $\mathbf{I}_2^{\mathrm{s}}$

We present the type inference algorithm for $\mathbf{I}_2^{\mathrm{s}}$ and a proof that it infers principal pairs. The algorithm is not new: it was described briefly in Leivant's original paper [41], and was defined rigorously by van Bakel in his dissertation [65]. We include it here because the algorithm provides a way to compare a variety of type systems based on rank 2 intersection types, and by the results of the last section, it gives a type inference algorithm for Λ_2^{s}. Our presentation differs from van Bakel's in one respect: he relies on unification while we use the more general machinery of subtype satisfaction.

Our algorithm takes as input a term M, and produces a pair $\langle A, \sigma \rangle$ such that $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \sigma$. Moreover, the pair $\langle A, \sigma \rangle$ is *principal* in the sense that any other acceptable pair of M can be obtained from $\langle A, \sigma \rangle$ by some well-understood operations.

Definition 3.5.1 (Principal typings) A typing $B \vdash M : \tau$ is an $\mathbf{I}_2^{\mathrm{s}}$ *instance* of a typing $A \vdash M : \sigma$ if there is a substitution S such that $B \leq_1 SA$ and $S\sigma \leq_2 \tau$, and τ is of the form $\tau_1 \to \cdots \to \tau_n \to \tau_0$ where $|\mathbf{act}(M)| = n$ and $\tau_0 \in \mathbf{T}_0$.

We say $A \vdash M : \sigma$ is an $\mathbf{I}_2^{\mathrm{s}}$ *principal typing* if $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \sigma$, and $\mathbf{I}_2^{\mathrm{s}} \rhd B \vdash M : \tau$ iff $B \vdash M : \tau$ is an $\mathbf{I}_2^{\mathrm{s}}$ instance of $A \vdash M : \sigma$.

If $A \vdash M : \sigma$ is an $\mathbf{I}_2^{\mathrm{s}}$ principal typing, we call $\langle A, \sigma \rangle$ an $\mathbf{I}_2^{\mathrm{s}}$ *principal pair for M*.

The definition of $\mathbf{I}_2^{\mathrm{s}}$ instance is complicated by the fact that $\mathbf{I}_2^{\mathrm{s}}$ typings are not closed under subsumption. For example,

$$\mathbf{I}_2^{\mathrm{s}} \rhd \{x : s \to t\} \vdash x : s \to t,$$

and $s \to t \leq_2 (s \wedge t) \to t$, but the judgment

$$\{x : s \to t\} \vdash x : (s \wedge t) \to t$$

is not derivable in $\mathbf{I}_2^{\mathrm{s}}$. For this reason, we have not formulated $\mathbf{I}_2^{\mathrm{s}}$ instances on pairs.

The next lemma shows that $\mathbf{I}_2^{\mathrm{s}}$ typings are closed under a limited form of subsumption, sufficient for the desired principal typing result.

Lemma 3.5.2 *If $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \sigma$, $\sigma \leq_2 \tau$, and τ is of the form $\tau_1 \to \cdots \to \tau_n \to \tau_0$ where $|\mathbf{act}(M)| = n$ and $\tau_0 \in \mathbf{T}_0$, then $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \tau$.*

Proof: By induction on M. Recall from Lemma 3.1.1(ii) that if $\sigma_1 \leq_2 \sigma_2 \in \mathbf{T}_0$, then $\sigma_1 = \sigma_2$.

- If $M = x$, then $|\mathbf{act}(M)| = 0$, so $\tau = \tau_0 \in \mathbf{T}_0$. Therefore $\sigma = \tau$ and $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \tau$.

- If $M = \lambda x M'$, then $|\mathbf{act}(M)| \geq 1$, $\mathbf{I}_2^{\mathrm{s}} \rhd A \backslash x \cup \{x : \sigma_1\} \vdash M' : \sigma_2$, and $\sigma = \sigma_1 \to \sigma_2$, where $\tau_1 \leq_1 \sigma_1$ and $\sigma_2 \leq_2 \tau_2 \to \cdots \to \tau_n \to \tau_0$.

  By induction, $\mathbf{I}_2^{\mathrm{s}} \rhd A \backslash x \cup \{x : \sigma_1\} \vdash M' : \tau_2 \to \cdots \to \tau_n \to \tau_0$.

  By weakening, $\mathbf{I}_2^{\mathrm{s}} \rhd A \backslash x \cup \{x : \tau_1\} \vdash M' : \tau_2 \to \cdots \to \tau_n \to \tau_0$.

  And by (ABS), $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \tau$.

- If $M = M_1 M_2$, then $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M_1 : \sigma' \to \sigma$ and $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M_2 : \sigma'$ for some $\sigma' \in \mathbf{T}_1$.

  - If $|\mathbf{act}(M_1)| = 0$ then $|\mathbf{act}(M)| = 0$, so $\tau = \tau_0 \in \mathbf{T}_0$. Therefore $\sigma = \tau$ and $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \tau$.
  - Otherwise $|\mathbf{act}(M_1)| = n + 1$, so $|\mathbf{act}(M)| = n$.

    Since $\sigma \leq_2 \tau$, we have $\sigma' \to \sigma \leq_2 \sigma' \to \tau$, and $\sigma' \to \tau$ is of the form $\sigma' \to \tau_1 \to \cdots \to \tau_n \to \tau_0$ where $\tau_0 \in \mathbf{T}_0$.

    Then by induction, $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M_1 : \sigma' \to \tau$, and by (APP), $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \tau$.

□

In order to define our type inference algorithm, we will need to solve subtype satisfaction problems of a particular kind. We define the relation $\leq_{2,1}$ between $\mathbf{T}_2$ and $\mathbf{T}_1$ to be the least relation closed under the rule:

- If $\sigma \leq_2 \tau_i$ for all $i \in I$, then $\sigma \leq_{2,1} (\bigwedge_{i \in I} \tau_i)$.

A $\leq_{2,1}$-*satisfaction problem* is a pair $\exists \vec{s}.P$, where P is a finite set whose every element is either: 1) an equality between simple types; or 2) an inequality between a $\mathbf{T}_2$ type and a $\mathbf{T}_1$ type. When $\vec{s}$ is empty $\exists \vec{s}$ may be omitted.

Note that any $\leq_{2,1}$-satisfaction problem is a $\leq_{\forall 2,1}$-satisfaction problem with the same set of solutions. Therefore we abuse notation and write $\mathbf{Solutions}(\pi)$, $\mathbf{MGS}(\pi)$, and $\mathbf{MGS}(\pi)[W]$ for the solutions, most general solutions, and most general solutions away from W of a $\leq_{2,1}$-satisfaction problem π. The algorithm of §2.3 can be used to solve $\leq_{2,1}$-satisfaction problems.

Definition 3.5.3 (Type inference) For any term M, we define the set $\mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M)$ of pairs by induction:

- If $M = x$, then for any type variable t, $\langle \{x : t\}, t \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(x)$.
- If $M = \lambda x N$, and $\langle A, \sigma \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(N)$, then:
  - If $x \notin \mathbf{dom}(A)$, and t is a type variable not appearing in $\langle A, \sigma \rangle$, then $\langle A, t \to \sigma \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(\lambda x N)$.
  - If $x \in \mathbf{dom}(A)$, then $\langle A \backslash x, A(x) \to \sigma \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(\lambda x N)$.

- If $M = M_1 M_2$, then:
  - If $\langle A_1, t\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M_1)$ and $\langle A_2, \sigma_2\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M_2)$ are disjoint, and $U \in \mathbf{MGS}(\{t = t_1 \to t_2, \sigma_2 \leq t_1\})$ where t_1, t_2 are fresh, then
  $$U\langle A_1 + A_2, t_2\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M_1 M_2).$$
  - If $\langle A_1, (\bigwedge_{i\in I} \sigma_i) \to \sigma_1\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M_1)$, and $\langle A_i, \tau_i\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M_2)$ for all $i \in I$, where all pairs are chosen disjoint, and $U \in \mathbf{MGS}(\{\tau_i \leq \sigma_i \mid i \in I\})$, then
  $$U\langle A_1 + \sum_{i\in I} A_i, \sigma_1\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M_1 M_2).$$

The following lemma establishes that the elements of $\mathrm{PP}_{\mathbf{I}_2^s}(M)$ are just trivial variants of each other. Therefore, the requirement of disjointness used in the definition of $\mathrm{PP}_{\mathbf{I}_2^s}$ is easily satisfied, and Definition 3.5.3 can be adapted to a type inference algorithm.

Lemma 3.5.4
i) *If $\langle A, \sigma\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M)$, then $x \in \mathbf{dom}(A)$ if and only if x is free in M.*

ii) *Suppose $\langle A_1, \sigma_1\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M)$. Then $\langle A_2, \sigma_2\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M)$ if and only if there is a bijection R of type variables such that $R\langle A_1, \sigma_1\rangle = \langle A_2, \sigma_2\rangle$.*

Proof: An easy induction on Definition 3.5.3. □

Theorem 3.5.5 *There is an algorithm that decides, for any M, whether the set $\mathrm{PP}_{\mathbf{I}_2^s}(M)$ is empty; and furthermore, if $\mathrm{PP}_{\mathbf{I}_2^s}(M)$ is not empty, it produces a member of $\mathrm{PP}_{\mathbf{I}_2^s}(M)$.*

Proof: Just follow the rules of Definition 3.5.3, generating "fresh" type variables as necessary, and use the algorithm of Corollary 2.3.5 to compute **MGS**. □

Example 3.5.6 We show how the algorithm finds the principal pair of $(\lambda x.xx)$.

- $\mathrm{PP}_{\mathbf{I}_2^s}(x)$ produces a pair $\langle\{x : t_1\}, t_1\rangle$.
- $\mathrm{PP}_{\mathbf{I}_2^s}(x)$ (again) produces a pair $\langle\{x : t_2\}, t_2\rangle$.
- To calculate $\mathrm{PP}_{\mathbf{I}_2^s}(xx)$, we find a most general solution to
$$\{t_2 \leq t_3, t_1 = t_3 \to t_4\},$$
such as $\{t_2 := t_3, t_1 := t_3 \to t_4\}$. Then $\langle\{x : t_3 \wedge (t_3 \to t_4)\}, t_4\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(xx)$.
- Finally, $\mathrm{PP}_{\mathbf{I}_2^s}(\lambda x.xx)$ produces $\langle\varnothing, (t_3 \wedge (t_3 \to t_4)) \to t_4\rangle$.

Theorem 3.5.7 (Principal typings) *If M is typable in $\mathbf{I}_2^{\mathrm{s}}$, then there is a principal pair $\langle A, \sigma \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M)$ for M.*

Proof: By Lemmas 3.4.11 and 3.5.2 above, and Theorems 3.5.8 and 3.5.9 below. □

Theorem 3.5.8 ($\mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}$ is sound) *If $\langle A, \sigma \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M)$, then $\langle A, \sigma \rangle \in \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(M)$.*

Proof: By induction on the definition of $\mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M)$.

- If $M = x$, then $\langle A, \sigma \rangle = \langle \{x : t\}, t \rangle$, and we have $\langle A, \sigma \rangle \in \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(x)$ by rule (VAR).
- If $M = \lambda x N$, then by Lemma 3.5.4(i) we have the following two cases:
  - x is not free in N, and $\sigma = t \to \sigma'$, where $\langle A, \sigma' \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(N)$.
    By induction and weakening, $\langle A \cup \{x : t\}, \sigma' \rangle \in \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(N)$ (note that $A \cup \{x : t\}$ is well-formed by Lemma 3.5.4(i)).
    So by rule (ABS), $\langle A, t \to \sigma' \rangle = \langle A, \sigma \rangle \in \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(\lambda x N)$.
  - x is free in N and $\langle A, \sigma \rangle = \langle A' \backslash x, A'(x) \to \sigma' \rangle$, where $\langle A', \sigma' \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(N)$.
    By induction $\langle A', \sigma' \rangle \in \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(N)$, so $\langle A, \sigma \rangle \in \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(\lambda x N)$ by rule (ABS).
- If $M = M_1 M_2$, then one of the following cases holds:
  - $\langle A, \sigma \rangle = U\langle A_1 + A_2, t_2 \rangle$, where $\langle A_1, t \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M_1)$, $\langle A_2, \sigma_2 \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M_2)$, and $U \in \mathbf{MGS}(\{t = t_1 \to t_2, \sigma_2 \leq t_1\})$.
    Then by induction, weakening, and substitutivity,
    $$\begin{array}{rcl} U\langle A_1 + A_2, t \rangle & \in & \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(M_1), \\ U\langle A_1 + A_2, \sigma_2 \rangle & \in & \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(M_2). \end{array}$$
    Since $U\sigma_2 \leq_2 Ut_1$, by Lemma 3.1.1(ii) we have $U\sigma_2 = Ut_1$. And $Ut = (Ut_1) \to (Ut_2)$, so by rule (APP) we have $U\langle A_1 + A_2, t_2 \rangle \in \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(M)$.
  - $\langle A, \sigma \rangle = U\langle A_1 + \sum_{i \in I} A_i, \sigma_1 \rangle$, where $\langle A_i, \tau_i \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M_2)$ for all $i \in I$, $\langle A_1, (\bigwedge_{i \in I} \sigma_i) \to \sigma_1 \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M_1)$, and $U \in \mathbf{MGS}(\{\tau_i \leq \sigma_i \mid i \in I\})$.
    Then by induction, weakening, and substitutivity,
    $$\begin{array}{rcl} U\langle A_1 + \sum_{i \in I} A_i, (\bigwedge_{i \in I} \sigma_i) \to \sigma_1 \rangle & \in & \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(M_1), \\ U\langle A_1 + \sum_{i \in I} A_i, \tau_i \rangle & \in & \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(M_2) \quad (\forall i \in I). \end{array}$$
    By Lemma 3.1.1(ii) and the fact that $U\tau_i \leq_2 U\sigma_i$, we have $U\tau_i = U\sigma_i$. Then by rule (APP) we have $U\langle A_1 + \sum_{i \in I} A_i, \sigma_1 \rangle \in \mathrm{AP}_{\mathbf{I}_2^{\mathrm{s}}}(M)$.

□

Theorem 3.5.9 ($\mathrm{PP}_{\mathbf{I}_2^s}$ **is complete**) *If* $\langle B,\tau\rangle \in \mathrm{AP}_{\mathbf{I}_2^s}(M)$*, then there is a pair* $\langle A,\sigma\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M)$ *and a substitution* S *such that* $B \leq_1 SA$ *and* $S\sigma \leq_2 \tau$.

Proof: By induction on the structure of M.

- If $M = x$, then $\langle B,\tau\rangle \in \mathrm{AP}_{\mathbf{I}_2^s}(M)$ by rule (VAR), and therefore, $B(x) = (\bigwedge_{i\in I}\tau_i)$ and $\tau = \tau_{i_0} \in \mathbf{T}_0$ for some $i_0 \in I$.

  Let $\sigma = t$ and $A = \{x : \sigma\}$ for any type variable t. Then $\langle A,\sigma\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M)$. Since $\tau \in \mathbf{T}_0$, $S = \{t := \tau\}$ is a well-formed substitution, and then $B \leq_1 \{x : \tau\} = SA$, and $S\sigma = \tau \leq_2 \tau$.

- If $M = \lambda x N$, then by the definition of $\mathbf{I}_2^s$, τ must be of the form $\tau_1 \to \tau_2$, and $\langle B\backslash x \cup \{x : \tau_1\}, \tau_2\rangle \in \mathrm{AP}_{\mathbf{I}_2^s}(N)$. By induction, there is a substitution S' and pair $\langle A', \sigma_2\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(N)$ such that $B\backslash x \cup \{x : \tau_1\} \leq_1 S'A'$ and $S'\sigma_2 \leq_2 \tau_2$.

  - If $x \notin \mathbf{dom}(A')$, let $\sigma = t \to \sigma_2$ where t is a fresh type variable, and let $A = A'$. Then $\langle A,\sigma\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M)$.
    Note that τ_1 is of the form $(\bigwedge_{i\in I}\tau_i)$, and therefore, we can pick $\sigma_1 \in \mathbf{T}_0$ such that $\tau_1 \leq_1 \sigma_1$ (choose any τ_i).
    Let $S = \{t := \sigma_1\} \cup S'$. Then $B \leq_1 B\backslash x \leq_1 S'A' = SA$, and $S(t \to \sigma_2) = \sigma_1 \to S'\sigma_2 \leq_2 \tau_1 \to \tau_2$, as desired.
  - If $x \in \mathbf{dom}(A')$, let $\sigma = A'(x) \to \sigma_2$ and $A = A'\backslash x$. Then $\langle A,\sigma\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M)$.
    Let $S = S'$. Then $B \leq_1 B\backslash x \leq_1 S'A'\backslash x = SA$, and $S\sigma = S'(A'(x) \to \sigma_2) \leq_2 \tau_1 \to \tau_2$, as desired.

- If $M = M_1M_2$, then by the definition of $\mathbf{I}_2^s$, $\langle B, (\bigwedge_{i\in I}\tau_i) \to \tau\rangle \in \mathrm{AP}_{\mathbf{I}_2^s}(M_1)$ and $\langle B,\tau_i\rangle \in \mathrm{AP}_{\mathbf{I}_2^s}(M_2)$ for all $i \in I$.

  By induction, $\mathrm{PP}_{\mathbf{I}_2^s}(M_1)$ is nonempty, and by Lemma 3.5.4(ii), it is sufficient to consider the following cases on the structure of pairs in $\mathrm{PP}_{\mathbf{I}_2^s}(M_1)$.

  - $\langle A_1, t\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M_1)$. By induction, there is a substitution S_1 such that $B \leq_1 S_1A_1$ and $S_1t \leq_2 (\bigwedge_{i\in I}\tau_i) \to \tau$.
    Since S_1 is a $\mathbf{T}_0$ substitution, $S_1t \in \mathbf{T}_0$, and by the definition of $\leq_2$, $S_1t = \tau_{i_0} \to \sigma_1$ for some $i_0 \in I$ and $\sigma_1 \in \mathbf{T}_0$, and $\sigma_1 \leq_2 \tau$.
    By induction and Lemma 3.5.4(ii), there is a disjoint pair $\langle A_2, \sigma_2\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M_2)$ and substitution S_2 such that $B \leq_1 S_2A_2$ and $S_2\sigma_2 \leq_2 \tau_{i_0}$.
    Let $\pi = \{t = t_1 \to t_2, \sigma_2 \leq t_1\}$, where t_1, t_2 are fresh. Then $R = S_1 \cup S_2 \cup \{t_1 := \tau_{i_0}, t_2 := \sigma_1\}$ is a solution to π.
    Therefore, $\mathbf{MGS}(\pi)$ is nonempty and we may pick $U \in \mathbf{MGS}(\pi)$. Let $A = U(A_1 + A_2)$ and $\sigma = Ut_2$. Then $\langle A,\sigma\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M)$.

By Convention 2.3.7, there exists an S such that $SA = R(A_1 + A_2)$ and $S\sigma = Rt_2$. Then $B \leq_1 S_1A_1 + S_2A_2 = R(A_1 + A_2) = SA$ and $S\sigma = Rt_2 = \sigma_1 \leq_2 \tau$, as desired.

- $\langle A_1, (\bigwedge_{j\in J} \sigma_j) \to \sigma' \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M_1)$.

  By induction there is a substitution S_1 such that $B \leq_1 S_1A_1$, and

$$S_1((\bigwedge_{j\in J} \sigma_j) \to \sigma') \leq_2 (\bigwedge_{i\in I} \tau_i) \to \tau.$$

  By the definition of $\leq_2$, $\{S_1\sigma_j \mid j \in J\} \subseteq \{\tau_i \mid i \in I\}$, so for all $j \in J$ there is an $i_j \in I$ such that $S_1\sigma_j = \tau_{i_j}$.

  By induction and Lemma 3.5.4(ii), for all $j \in J$ there are disjoint pairs $\langle A_j, \sigma'_j \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M_2)$ and substitutions S_j such that $B \leq_1 S_jA_j$ and $S_j\sigma'_j \leq_2 \tau_{i_j}$.

  Let $\pi = \{\sigma'_j \leq \sigma_j \mid j \in J\}$. Then $R = S_1 \cup (\bigcup_{j\in J} S_j)$ is a solution to π: $R\sigma'_j = S_j\sigma'_j \leq_2 \tau_{i_j} = S_1\sigma_j = R\sigma_j$.

  Therefore, $\mathbf{MGS}(\pi)$ is nonempty, and we may pick $U \in \mathbf{MGS}(\pi)$. Let $A = U(A_1 + \sum_{j\in J} A_j)$ and $\sigma = U\sigma'$. Then $\langle A, \sigma \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M)$.

  By Convention 2.3.7, there exists an S such that $SA = R(A_1 + \sum_{j\in J} A_j)$ and $S\sigma = R\sigma'$. Then

$$B \leq_1 S_1A_1 + \sum_{j\in J} S_jA_j = R(A_1 + \sum_{j\in J} A_j) = SA,$$

  and $S\sigma = R\sigma' = S_1\sigma' \leq_2 \tau$, as desired.

□

3.6 Type inference for Λ_2^{s}

In §3.4 we proved that every Λ_2^{s} type judgment closely corresponds to an $\mathbf{I}_2^{\mathrm{s}}$ judgment (Theorem 3.4.2), and in §3.5 we gave a type inference algorithm for $\mathbf{I}_2^{\mathrm{s}}$. By combining these results we can construct type inference algorithms for Λ_2^{s}.

Definition 3.6.1 (Λ_2^{s} type inference)

- For any lambda term M, $\mathrm{LCG}(M)$ is the least set of pairs satisfying

  - If $\langle A, \sigma \rangle \in \mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M)$, then $\langle \mathbf{lcg}(A), \mathbf{lcg}(\sigma) \rangle \in \mathrm{LCG}(M)$.

- For any lambda term M with $|\mathbf{act}(M)| = n$, $\mathrm{KW}(M)$ is the least set of pairs satisfying

Term	Algorithm	Inferred type
	ML	$t \to t$
	$\mathrm{PP}_{\mathbf{I}_2^s}$	$t \to t$
$\lambda x.x$	LCG	$t \to t$
	KW	$(\forall t.t) \to s$
	ML	$(s \to s) \to s \to s$
	$\mathrm{PP}_{\mathbf{I}_2^s}$	$(s \to t) \wedge (t \to u) \to s \to u$
$\lambda fx.f(fx)$	LCG	$(\forall tv.t \to v) \to s \to u$
	KW	$(\forall t.t) \to (\forall t.t) \to u$
	ML	$(s \to t \to u) \to (s \to t) \to s \to u$
	$\mathrm{PP}_{\mathbf{I}_2^s}$	$(s \to t \to u) \to (v \to t) \to (s \wedge v) \to u$
$\lambda xyz.xz(yz)$	LCG	$(s \to t \to u) \to (v \to t) \to (\forall t.t) \to u$
	KW	$(\forall t.t) \to (\forall t.t) \to (\forall t.t) \to u$
	ML	cannot be typed
	$\mathrm{PP}_{\mathbf{I}_2^s}$	$(s \wedge (s \to t)) \to t$
$\lambda x.xx$	LCG	$(\forall v.v) \to t$
	KW	$(\forall v.v) \to t$

Table 3.1: Comparison of type inference algorithms on some simple terms.

- If $\langle A, \sigma_1 \to \cdots \to \sigma_n \to \tau\rangle \in \mathrm{PP}_{\mathbf{I}_2^s}(M)$, then

$$\langle \{x : \forall t.t \mid x \in \mathbf{dom}(A)\}, (\forall t_1.t_1) \to \cdots \to (\forall t_n.t_n) \to \tau\rangle \in \mathrm{KW}(M).$$

Both algorithms are new. The algorithm KW is so named because its output is identical to that of the algorithm of Kfoury and Wells [35]. Table 3.1 compares the type inference algorithms on some simple terms.

3.7 Other systems of rank 2 intersection types

3.7.1 A restriction of $\mathbf{I}_2^s$

Van Bakel [65] defined a rank 2 intersection type system that is a slight restriction of our system $\mathbf{I}_2^s$. A version of his rules is presented below.

$$(\mathrm{VAR}) \quad \{x : \tau\} \vdash x : \tau \quad (\text{where } \tau \in \mathbf{T}_0)$$

$$(\mathrm{ABS}) \quad \frac{A \backslash x \cup \{x : \tau_1\} \vdash M : \tau_2}{A \backslash x \vdash (\lambda x M) : \tau_1 \to \tau_2}$$

$$(\mathrm{APP}) \quad \frac{A \vdash M : (\bigwedge_{i \in I} \tau_i) \to \tau, \quad (\forall i \in I)\ A_i \vdash N : \tau_i}{A + \sum_{i \in I} A_i \vdash (MN) : \tau}$$

We write $\mathbf{I}_2^{\mathrm{vb}} \rhd A \vdash M : \sigma$ if the judgment $A \vdash M : \sigma$ follows by these rules, under the following restrictions: environment types are in $\mathbf{T}_1$; derived types are in $\mathbf{T}_2$; and

in every judgment $A \vdash M : \tau$, the type environment A contains only assumptions actually used in the derivation of $A \vdash M : \tau$. For example, the rule (VAR) has been intentionally restricted to rule out a judgment such as

$$\{x : \sigma_1 \wedge \sigma_2\} \vdash x : \sigma_1,$$

in which the type σ_2 assumed for x is not used. Similarly, $\{x : \sigma_1, y : \sigma_2\} \vdash x : \sigma_1$ is not derivable because the assumption $y : \sigma_2$ is not used. Both of these examples show that $\mathbf{I}_2^{\mathrm{vb}}$ is not closed under weakening.

The exact relation between $\mathbf{I}_2^{\mathrm{vb}}$ and $\mathbf{I}_2^{\mathrm{s}}$ is summarized in the following lemma.

Theorem 3.7.1 (Comparison of $\mathbf{I}_2^{\mathrm{vb}}$ and $\mathbf{I}_2^{\mathrm{s}}$)

i) *If $\mathbf{I}_2^{\mathrm{vb}} \rhd A \vdash M : \sigma$, then $\mathbf{I}_2^{\mathrm{s}} \rhd A \vdash M : \sigma$. The converse does not hold.*

ii) *A term is typable in $\mathbf{I}_2^{\mathrm{vb}}$ if and only if it is typable in $\mathbf{I}_2^{\mathrm{s}}$.*

Proof:

i) Just note that the $\mathbf{I}_2^{\mathrm{vb}}$ rule (VAR) is a special case of the $\mathbf{I}_2^{\mathrm{s}}$ rule (VAR), that the $\mathbf{I}_2^{\mathrm{vb}}$ rule (ABS) is identical to the $\mathbf{I}_2^{\mathrm{s}}$ rule (ABS), and that the $\mathbf{I}_2^{\mathrm{vb}}$ rule (APP) follows from the $\mathbf{I}_2^{\mathrm{s}}$ rule (ABS) and weakening.

   The examples above show that the converse does not hold.

ii) This follows because the definition of principal pair in van Bakel's system is identical to our own.

□

3.7.2 An extension of $\mathbf{I}_2^{\mathrm{s}}$

A natural extension of $\mathbf{I}_2^{\mathrm{s}}$ is obtained by adding the rule of *subsumption* to the rules of $\mathbf{I}_2^{\mathrm{s}}$:

$$(\textsc{sub})\quad \frac{A \vdash M : \tau}{A \vdash M : \sigma} \quad (\text{where } \tau \leq_2 \sigma)$$

We write $\mathbf{I}_2 \rhd A \vdash M : \sigma$ if the judgment $A \vdash M : \sigma$ follows by the rules of $\mathbf{I}_2^{\mathrm{s}}$ plus (SUB), with types appearing in type environments restricted to $\mathbf{T}_1$, and derived types restricted to $\mathbf{T}_2$.

Clearly, every judgment of $\mathbf{I}_2^{\mathrm{s}}$ is a judgment of $\mathbf{I}_2$. The converse does not hold; for example, the judgment

$$\{x : \sigma \to \tau\} \vdash x : (\sigma \wedge \sigma') \to \tau$$

is derivable in $\mathbf{I}_2$ for any $\sigma \neq \sigma' \in \mathbf{T}_0$, but is not derivable in $\mathbf{I}_2^{\mathrm{s}}$.

Because $\mathbf{I}_2$ is closed under subsumption, there is a simple definition of $\mathbf{I}_2$ instances on pairs.

Definition 3.7.2 (Principal typings) A typing $B \vdash M : \tau$ is an $\mathbf{I}_2$ *instance* of a typing $A \vdash M : \sigma$ if there is a substitution S such that $B \leq_1 SA$ and $S\sigma \leq_2 \tau$.

A typing $A \vdash M : \sigma$ is an $\mathbf{I}_2$ *principal typing* if $\mathbf{I}_2 \rhd A \vdash M : \sigma$, and $\mathbf{I}_2 \rhd B \vdash M : \tau$ iff $B \vdash M : \tau$ is an $\mathbf{I}_2$ instance of $A \vdash M : \sigma$.

If $A \vdash M : \sigma$ is an $\mathbf{I}_2$ principal typing, we call $\langle A, \sigma \rangle$ an $\mathbf{I}_2$ *principal pair for* M.

It is easy to show that the $\mathbf{I}_2$ principal pairs are identical to the principal pairs of $\mathbf{I}_2^s$ (it requires only a simple extension of the proof of Theorem 3.5.9). An immediate consequence is that the terms typable in $\mathbf{I}_2$ are exactly the same as the terms typable in $\mathbf{I}_2^s$. In summary:

Theorem 3.7.3 (Comparison of $\mathbf{I}_2$ and $\mathbf{I}_2^s$)

- *If* $\mathbf{I}_2^s \rhd A \vdash M : \sigma$, *then* $\mathbf{I}_2 \rhd A \vdash M : \sigma$. *The converse does not hold.*
- *A term* M *is typable in* $\mathbf{I}_2$ *if and only if it is typable in* $\mathbf{I}_2^s$.

The simpler definition of instance makes $\mathbf{I}_2$ more attractive than either $\mathbf{I}_2^{yb}$ or $\mathbf{I}_2^s$, even though it does not type any more terms. However, it was still useful to develop $\mathbf{I}_2^s$. In particular, because $\mathbf{I}_2$ typings are closed under subsumption, Lemma 3.4.11 fails for $\mathbf{I}_2$; it was convenient to have Lemma 3.4.11 for the proof of the equivalence of typability with Λ_2^s.

3.8 Recursive definitions

We now consider ways of typing recursive definitions in the rank 2 systems. In Chapter 2, we introduced the rules (REC-SIMPLE) and (REC-POLY), either of which may be used with ML or with Λ_2.

$$(\text{REC-SIMPLE}) \quad \frac{A \backslash x \cup \{x : \tau\} \vdash M : \tau}{A \vdash (\mu x M) : \tau} \quad (\text{where } \tau \in \mathbf{T}_0)$$

$$(\text{REC-POLY}) \quad \frac{A \backslash x \cup \{x : \tau\} \vdash M : \tau}{A \vdash (\mu x M) : \tau} \quad (\text{where } \tau \in \mathbf{S}(1))$$

Example 3.8.1 When extended by (REC-POLY), both ML and Λ_2 can type the following terms:

$$\begin{aligned} (\mu w.(\lambda xy.y)(ww)) &: \forall t.t \to t, \\ (\mu w.(\lambda xyz.z)(w\ 3)(w\ \mathbf{true})) &: \forall t.t \to t, \\ (\mu x.xx) &: \forall t.t. \end{aligned}$$

Neither is typable with the rule (REC-SIMPLE). Other examples are given by Mycroft [50] and Kfoury et al. [31, 33], who introduced (REC-POLY) independently.

Unfortunately, type inference for Λ_2 or ML extended by (REC-POLY) is undecidable [32, 22], so (REC-SIMPLE) is used in practice.

The rule (REC-SIMPLE) is one way of typing recursive definitions in intersection type systems. However, as with ML and Λ_2, it seems overly restrictive. The rule (REC-POLY) involves $\mathbf{S}(1)$ types, so it is not appropriate for the intersection type systems. Instead, we might consider a rule like the following:

$$\frac{A\backslash x \cup \{x : \tau)\} \vdash M : \tau}{A \vdash (\mu x M) : \tau} \quad (\text{where } \tau \in \mathbf{T}_1)$$

Note that the full power of the rule is achieved only by allowing $\mathbf{T}_1$ derived types, so the rule is not compatible with the rank 2 intersection type systems that we have defined so far. However, the rule can be adapted to our systems as follows:

$$(\text{REC-INT}) \quad \frac{(\forall i \in I)\ A\backslash x \cup \{x : (\bigwedge_{j\in I} \tau_j)\} \vdash M : \tau_i}{A \vdash (\mu x M) : \tau_{i_0}} \quad (\text{where } i_0 \in I)$$

Example 3.8.2 The system $\mathbf{I}_2$ + (REC-INT) can type the following terms:

$$\begin{aligned} (\mu w.(\lambda xy.y)(ww)) &: \tau \to \tau, \\ (\mu w.(\lambda xyz.z)(w\ 3)(w\ \mathbf{true})) &: \tau \to \tau, \end{aligned}$$

where τ is any simple type. Neither term is typable in $\mathbf{I}_2$ + (REC-SIMPLE).

The close connection between $\mathbf{I}_2$ and Λ_2 casts some doubt on the decidability of the system $\mathbf{I}_2$ + (REC-INT). However, $\mathbf{I}_2$ + (REC-INT) cannot type all of the terms that can be typed by Λ_2 + (REC-POLY). For example, the term $(\mu x.xx)$ cannot be typed in $\mathbf{I}_2$ + (REC-INT). The decidability of $\mathbf{I}_2$ + (REC-INT) is an open question.

All of the rules discussed so far for typing recursive definitions exhibit the following anomaly in rank 2 systems: the term $(\lambda y.yy)$ is typable, but the term $(\mu x(\lambda y.yy))$ is not typable. This is because $(\lambda y.yy)$ is only typable at rank 2, and all of the rules we have discussed require a type of rank 1 or less for the type of x that will appear in the type environment.

This could be repaired by adding a special rule for the vacuous case:

$$(\text{REC-VAC}) \qquad \frac{A \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \qquad (\text{where } x \notin \mathbf{dom}(A))$$

3.9 The systems $\mathbf{P}_2$ and $\mathbf{P}_2^{\mathrm{R}}$

We defined the rank 2 system $\mathbf{P}_2$ in Chapter 2, but postponed the proof of correctness for its type inference algorithm. Here we give the proof in detail, and compare $\mathbf{P}_2$ to the other rank 2 systems defined in this chapter. We will actually prove correctness for the extension $\mathbf{P}_2^{\mathrm{R}}$ of $\mathbf{P}_2$; correctness for $\mathbf{P}_2$ follows immediately.

3.9.1 Preliminaries

Before proceeding with the proof of correctness we state some basic results.

Lemma 3.9.1

i) *If* $\sigma, \tau \in \mathbf{T}_0$, *then* $\sigma \leq_{\forall 2} \tau$ *iff* $\sigma \leq_{\forall 2,1} \tau$ *iff* $\sigma = \tau$.

ii) *If* $\sigma, \tau \in \mathbf{T}_2$, *then* $\sigma \leq_{\forall 2} \tau$ *iff* $\sigma \leq_2 \tau$.

iii) *If* $\sigma \leq_{\forall 2} \tau$, *then* $(\forall t \sigma) \leq_{\forall 2} (\forall t \tau)$.

iv) *If* $\sigma \in \mathbf{T}_2$ *and* $\tau \in \mathbf{T}_0$, *then* $\forall \vec{t} \sigma \leq_{\forall 2} \tau$ *iff for some substitution* S *with* $\mathbf{dom}(S) \subseteq \vec{t}$, *we have* $S\sigma = \tau$.

v) *For any substitution* S *and types* $\sigma, \tau \in \mathbf{T}_{\forall 2}$, *if* $S\sigma \leq_{\forall 2} \tau$, *then* $S(\forall t \sigma) \leq_{\forall 2} \tau$.

vi) *For any substitution* S, *types* $\sigma, \tau \in \mathbf{T}_{\forall 2}$, *and type environment* A, *if* $S\sigma \leq_{\forall 2} \tau$, *then* $S(\mathrm{Gen}(A, \sigma)) \leq_{\forall 2} \tau$.

vii) *If* $\sigma_1 \leq_{\forall 2} \sigma_2 \leq_{\forall 2,1} \sigma_3 \leq_1 \sigma_4$, *then* $\sigma_1 \leq_{\forall 2,1} \sigma_4$.

Lemma 3.9.2 *If* $M = (\mathbf{letrec}\ B\ \mathbf{in}\ N)$, *then* $\mathbf{P}_2^{\mathrm{R}} \triangleright A \vdash M : \sigma$ *iff* $\mathbf{P}_2^{\mathrm{R}} \triangleright A \vdash \langle\!\langle M \rangle\!\rangle : \sigma$.

Lemma 3.9.3

i) *If* $\langle A, \sigma \rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$, *then* $x \in \mathbf{dom}(A)$ *if and only if* x *is free in* M.

ii) *Suppose* $\langle A_1, \sigma_1 \rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$. *Then* $\langle A_2, \sigma_2 \rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ *if and only if there is a bijection* R *of type variables such that* $R\langle A_1, \sigma_1 \rangle = \langle A_2, \sigma_2 \rangle$.

Lemma 3.9.4 (Sound operations on typings) *If* $\mathbf{P}_2^{\mathrm{R}} \triangleright A \vdash M : \sigma$, *then*

- *if* $A' \leq_1 A$, *then* $\mathbf{P}_2^{\mathrm{R}} \triangleright A' \vdash M : \sigma$;
- *if* $\sigma \leq_{\forall 2} \sigma'$, *then* $\mathbf{P}_2^{\mathrm{R}} \triangleright A \vdash M : \sigma'$; *and*
- $\mathbf{P}_2^{\mathrm{R}} \triangleright SA \vdash M : S\sigma$ *for any* $\mathbf{T}_0$ *substitution* S.

Lemma 3.9.4 and Theorems 3.9.5 and 3.9.6 together imply the principal typing property for $\mathbf{P}_2$ and $\mathbf{P}_2^{\mathrm{R}}$ (Theorems 2.4.3 and 2.6.4). We prove Theorems 3.9.5 and 3.9.6 in the next two sections.

3.9.2 Soundness

Theorem 3.9.5 ($\mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}$ is sound) *If $\langle A, \sigma\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$, then $\langle A, \sigma\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.*

Proof: By induction on the definition of $\mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.

- If $M = x$, then $\langle A, \sigma\rangle = \langle\{x : t\}, t\rangle$ for some type variable t.
  Then we have $\langle A, \sigma\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(x)$ by rule (VAR).

- If $M = \lambda x N$, then by Lemma 3.9.3(i) we must consider two cases:

  - It x is not free in N, then $\langle A, \forall\vec{s}\sigma'\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$ for some σ', and $\sigma = \forall t\vec{s}(t \to \sigma')$ for some fresh type variable t.
    By induction, $\langle A, \forall\vec{s}\sigma'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$, and by weakening and subsumption,

    $$\langle A \cup \{x : t\}, \sigma'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(N).$$

    Note that $A \cup \{x : t\}$ is well-formed by Lemma 3.9.3(i).
    By rule (ABS), $\langle A, t \to \sigma'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(\lambda x N)$, and by (GEN),

    $$\langle A, \forall t\vec{s}(t \to \sigma')\rangle = \langle A, \sigma\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(\lambda x N).$$

  - If x is free in N, then $\langle A, \sigma\rangle = \langle A'\backslash x, \mathrm{Gen}(A'\backslash x, A'(x) \to \sigma')\rangle$, where $\langle A', \forall\vec{s}\sigma'\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$.
    By induction and rule (SUB), $\langle A', \sigma'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$, so by rule (ABS), $\langle A'\backslash x, A'(x) \to \sigma'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(\lambda x N)$. Then by (GEN),

    $$\langle A'\backslash x, \mathrm{Gen}(A'\backslash x, A'(x) \to \sigma')\rangle = \langle A, \sigma\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(\lambda x N).$$

- If $M = M_1 M_2$, then we have $\langle A_1, \forall\vec{s}\sigma_1\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M_1)$. By induction, we have $\langle A_1, \forall\vec{s}\sigma_1\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_1)$, and by (SUB), $\langle A_1, \sigma_1\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_1)$.

  - If σ_1 is a type variable t, then we must have a pair $\langle A_2, \sigma_2\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M_2)$ with fresh type variables, $A = U(A_1 + A_2)$ and $\sigma = \mathrm{Gen}(A, Ut_2)$, where $U \in \mathbf{MGS}(\{\sigma_2 \leq t_1, t = t_1 \to t_2\})$ for fresh type variables t_1 and t_2.
    By induction, $\langle A_2, \sigma_2\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_2)$. By substitutivity,

    $$U\langle A_1, \sigma_1\rangle = \langle UA_1, (Ut_1) \to (Ut_2)\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_1)$$

    and

    $$U\langle A_2, \sigma_2\rangle = \langle UA_2, U\sigma_2\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_2).$$

    By weakening and subsumption,

    $$\langle UA_1 + UA_2, (Ut_1) \to (Ut_2)\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_1)$$

and

$$\langle UA_1 + UA_2, Ut_1\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_2).$$

Then by rule (APP) we have

$$\langle UA_1 + UA_2, Ut_2\rangle = \langle A, Ut_2\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_1M_2),$$

and by rule (GEN),

$$\langle A, \mathrm{Gen}(A, Ut_2)\rangle = \langle A, \sigma\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_1M_2).$$

- If $\sigma_1 = (\bigwedge_{i\in I}\tau_i) \to \tau$, then we must have pairs $\langle A_i, \sigma_i\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M_2)$ with fresh type variables, $U \in \mathbf{MGS}\{\sigma_i \leq \tau_i \mid i \in I\}$, $A = U(A_1 + \sum_{i\in I} A_i)$, and $\sigma = \mathrm{Gen}(A, U\tau)$.
  By induction and substitutivity, $\langle UA_i, U\sigma_i\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_2)$ for all $i \in I$, and by substitutivity we have

$$\langle UA_1, U\sigma_1\rangle = \langle UA_1, (\bigwedge_{i\in I} U\tau_i) \to U\tau\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_1).$$

  By weakening and subsumption, $\langle A, (\bigwedge_{i\in I} U\tau_i) \to U\tau\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_1)$ and $\langle A, U\tau_i\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_2)$ for all $i \in I$.
  Then by rules (APP) and (GEN) we have

$$\langle A, \mathrm{Gen}(UA, U\tau)\rangle = \langle A, \sigma\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M).$$

- If $M = (\mu x N)$, we consider two cases.
  - If x is not free in N, then for some $\langle A', \sigma'\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$, fresh type variable t, and $U \in \mathbf{MGS}(\{\sigma' \leq t\})$,

$$\langle A, \sigma\rangle = \langle UA', \mathrm{Gen}(UA', U\sigma')\rangle.$$

    By induction and substitutivity, $\langle UA', U\sigma'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$. By weakening, $\langle UA' \cup \{x : Ut\}, U\sigma'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$. And by the rules (REC) and (GEN),

$$\langle UA', \mathrm{Gen}(UA', U\sigma')\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(\mu x N),$$

    as desired.
  - If x is free in N, then for some $\langle A', \sigma'\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$ and $U \in \mathbf{MGS}(\sigma' \leq A'(x))$, we have

$$\langle A, \sigma\rangle = \langle UA'\backslash x, \mathrm{Gen}(UA'\backslash x, U\sigma')\rangle.$$

    By induction, $\langle A', \sigma'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$. Then $\langle UA', U\sigma'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$ by substitutivity. Since $U\sigma' \leq_{\forall 2,1} UA'(x)$, by rule (REC) we have $\langle UA'\backslash x, U\sigma'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(\mu x N)$. Finally by rule (GEN),

$$\langle UA'\backslash x, \mathrm{Gen}(UA'\backslash x, U\sigma')\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(\mu x N).$$

- If $M = (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ x_{i_0})$ where $i_0 \in I$,

  then $\langle A, \sigma\rangle = \langle A''', \mathrm{Gen}(A''', U\sigma_{i_0})\rangle$, where

  $\langle A_i, \sigma_i\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M_i)$ for $i \in I$,
  $A' = \sum_{i\in I} A_i$,
  $A'' = A' \cup \{x_i : t_i \mid i \in I,\ x_i \notin \mathbf{dom}(A'),\ t_i \text{ fresh}\}$
  $U \in \mathbf{MGS}(\{\sigma_i \leq A''(x_i) \mid i \in I,\})$,
  and $A''' = UA''\backslash\{x_i \mid i \in I\}$,

  By induction, $\langle A_i, \sigma_i\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_i)$ for $i \in I$.

  By weakening and substitutivity, $\langle UA'', U\sigma_i\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_i)$ for $i \in I$.

  Then by rule (LETREC-VAR), $\langle A''', U\sigma_{i_0}\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$, and by (GEN), $\langle A, \sigma\rangle = \langle A''', \mathrm{Gen}(A''', U\sigma_{i_0})\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.

- If $M = (\mathbf{letrec}\ x_i = M_i \mid i \in I\ \mathbf{in}\ N)$ where $N \notin \{x_i \mid i \in I\}$, the result follows by Lemma 3.9.2 and induction.

□

3.9.3 Completeness

Theorem 3.9.6 **($\mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}$ is complete)** *If $\langle B, \tau\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$, then there is a pair $\langle A, \sigma\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ and a substitution S such that $B \leq_1 A$ and $S\sigma \leq_{\forall 2} \tau$.*

Proof: By induction on the definition of $\mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.

- If $\langle B, \tau\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ by rule (VAR), then $M = x$ for some variable x, $B(x) = (\bigwedge_{i\in I} \tau_i)$, and $\tau = \tau_{i_0} \in \mathbf{T}_0$ for some $i_0 \in I$.

  Let $\sigma = t$ and $A = \{x : \sigma\}$, where t is a fresh type variable. Then $\langle A, \sigma\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.

  Since $\tau \in \mathbf{T}_0$, $S = \{t := \tau\}$ is a well-formed substitution, $B \leq_1 \{x : \tau\} = SA$, and $S\sigma = \tau \leq_{\forall 2} \tau$.

- If $\langle B, \tau\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ by rule (ABS), then $M = \lambda x N$, τ is of the form $\tau_1 \to \tau_2$, and $\langle B \cup \{x : \tau_1\}, \tau_2\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$.

  By induction, there is a substitution S' and pair $\langle A', \forall\vec{s}\sigma'\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$ such that $B \cup \{x : \tau_1\} \leq_1 S'A'$, and $S'(\forall\vec{s}\sigma') \leq_{\forall 2} \tau_2$.

  - If $x \notin \mathbf{dom}(A')$, let $A = A'$ and $\sigma = \forall t\vec{s}(t \to \sigma')$ for any fresh type variable t. Then $\langle A, \sigma\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.

    Let $S = S'$. We already have $B \leq_1 S'A' = SA$, so we only need show $S\sigma = S'(\forall t\vec{s}(t \to \sigma')) \leq_{\forall 2} \tau$.

We can assume $t, \vec{s}$ are fresh, so that

$$S\sigma = S'(\forall t\vec{s}(t \to \sigma')) = \forall t\vec{s}(t \to S'\sigma').$$

And

$$\{t := \tau_1\}(\forall \vec{s}(t \to S'\sigma')) = \forall \vec{s}(\tau_1 \to S'\sigma') \leq_{\forall 2} \tau_1 \to \tau_2 = \tau,$$

so by the definition of $\leq_{\forall 2}$, $S\sigma \leq_{\forall 2} \tau$, as desired.

- If $x \in \mathbf{dom}(A')$, let $A = A' \backslash x$ and $\sigma = \text{Gen}(A, A'(x) \to \sigma')$. Then $\langle A, \sigma\rangle \in \text{PP}_{\mathbf{P}_2^{\text{R}}}(M)$.
  Let $S = S'$. Then $B \leq_1 S'A' \backslash x = SA$. And we may assume that $\vec{s}$ are fresh, so that $S\sigma = S'(\text{Gen}(A, A'(x) \to \sigma')) \leq_{\forall 2} \tau_1 \to \tau_2 = \tau$, as desired.

- If $\langle B, \tau\rangle \in \text{AP}_{\mathbf{P}_2^{\text{R}}}(M)$ by rule (APP), then $M = M_1 M_2$, and $\langle B, (\bigwedge_{i\in I} \tau_i) \to \tau\rangle \in \text{AP}_{\mathbf{P}_2^{\text{R}}}(M_1)$ and $\langle B, \tau_i\rangle \in \text{AP}_{\mathbf{P}_2^{\text{R}}}(M_2)$ for all $i \in I$.

  By induction, $\text{PP}_{\mathbf{P}_2^{\text{R}}}(M_1)$ is nonempty, and by Lemma 3.9.3(ii), it is sufficient to consider the following cases on the structure of pairs in $\text{PP}_{\mathbf{P}_2^{\text{R}}}(M_1)$.

  - $\langle A_1, \forall \vec{s}t\rangle \in \text{PP}_{\mathbf{P}_2^{\text{R}}}(M_1)$.
    We may assume that the type variables $\vec{s}$ are fresh, so by induction and the definition of $\leq_{\forall 2}$, there is a substitution S_1 such that $B \leq_1 S_1 A_1$ and $S_1 t \leq_{\forall 2} (\bigwedge_{i\in I} \tau_i) \to \tau$.
    Since S_1 is a $\mathbf{T}_0$ substitution, by the definition of $\leq_{\forall 2}$, $S_1 t = (\tau_{i_0} \to \sigma')$ for some $i_0 \in I$ and $\sigma' \in \mathbf{T}_0$ with $\sigma' \leq_{\forall 2} \tau$.
    By induction and Lemma 3.9.3(ii), there is a disjoint pair $\langle A_2, \sigma_2\rangle \in \text{PP}_{\mathbf{P}_2^{\text{R}}}(M_2)$ and substitution S_2 such that $B \leq_1 S_2 A_2$, and $S_2\sigma_2 \leq_{\forall 2} \tau_{i_0}$.
    Let $\pi = \{\ t = t_1 \to t_2, \sigma_2 \leq t_1\ \}$, where t_1, t_2 are fresh. Note that π has a solution, $R = S_1 \cup S_2 \cup \{t_1 := \tau_{i_0}, t_2 := \sigma'\}$.
    Therefore $\mathbf{MGS}(\pi)$ is nonempty, and we may pick $U \in \mathbf{MGS}(\pi)$. Let $A = U(A_1 + A_2)$ and $\sigma = \text{Gen}(A, Ut_2)$. Then $\langle A, \sigma\rangle \in \text{PP}_{\mathbf{P}_2^{\text{R}}}(M)$.
    By Convention 2.3.7, there exists an S such that $SA = R(A_1 + A_2)$ and $SUt_2 = Rt_2$. Then

    $$B \leq_1 S_1 A_1 + S_2 A_2 = R(A_1 + A_2) = SA,$$

    and

    $$SUt_2 = Rt_2 = \sigma' \leq_{\forall 2} \tau.$$

    Finally by Lemma 3.9.1(vi), $S\sigma = S(\text{Gen}(A, Ut_2)) \leq_{\forall 2} \tau$, as desired.

- $\langle A_1, \forall \vec{s}(\bigwedge_{j\in J}\sigma_j) \to \sigma'\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M_1)$.
  We may assume that the type variables $\vec{s}$ are fresh, so by induction and the definition of $\leq_{\forall 2}$, there is a substitution S_1 such that $B \leq_1 S_1A_1$, and $S_1((\bigwedge_{j\in J}\sigma_j) \to \sigma' \leq_{\forall 2} (\bigwedge_{i\in I}\tau_i) \to \tau$.
  Then $\{S_1\sigma_j \mid j \in J\} \subseteq \{\tau_i \mid i \in I\}$, so for all $j \in J$ there is an $i_j \in I$ such that $S_1\sigma_j = \tau_{i_j}$.
  By induction and Lemma 3.9.3(ii), for all $j \in J$ there are disjoint pairs $\langle A_j, \sigma'_j\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M_2)$ and substitutions S_j such that $B \leq_1 S_jA_j$ and $S_j\sigma'_j \leq_{\forall 2} \tau_{i_j}$.
  Let $\pi = \{\sigma'_j \leq \sigma_j \mid j \in J\}$. Then $R = S_1 \cup (\bigcup_{j\in J} S_j)$ is a solution to π: $R\sigma'_j = S_j\sigma'_j \leq_{\forall 2} \tau_{i_j} = S_1\sigma_j = R\sigma_j$.
  Therefore $\mathbf{MGS}(\pi)$ is nonempty, and we may pick $U \in \mathbf{MGS}(\pi)$. Let $A = U(A_1 + \sum_{j\in J} A_j)$ and $\sigma = \mathrm{Gen}(A, U\sigma')$. Then $\langle A, \sigma\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.
  By Convention 2.3.7, there exists an S such that $SU\sigma' = R\sigma'$ and $SA = R(A_1 + \sum_{j\in J} A_j)$. Then

$$B \leq_1 S_1A_1 + \sum_{j\in J} S_jA_j = R(A_1 + \sum_{j\in J} A_j) = SA,$$

  and $SU\sigma' = R\sigma' = S_1\sigma' \leq_{\forall 2} \tau$.
  Finally by Lemma 3.9.1(vi), $S\sigma = S(\mathrm{Gen}(A, U\sigma')) \leq_{\forall 2} \tau$, as desired.

- If $\langle B, \tau\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ by rule (GEN), then $\tau = \forall t\tau'$ where $t \notin \mathrm{FTV}(B)$, and we have a shorter derivation of $\langle B, \tau'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.

  By induction there is a pair $\langle A', \sigma'\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ and a substitution S' such that $B \leq_1 S'A'$, and $S'\sigma' \leq_{\forall 2} \tau'$.

  Simply let $A = A'$ and $\sigma = \sigma'$ to find $\langle A, \sigma\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.

  We now show that $t \notin \mathrm{FTV}(S'\sigma')$. Then since $S'\sigma = S'\sigma' \leq_{\forall 2} \tau'$, we have $S'\sigma \leq_{\forall 2} \forall t\tau' = \tau$, and we simply let $S = S'$.

  Assume by way of contradiction that $t \in \mathrm{FTV}(S'\sigma')$. Since $B \leq_1 S'A'$, $\mathrm{FTV}(S'A') \subseteq \mathrm{FTV}(B)$. Therefore, $t \notin \mathrm{FTV}(B) \Rightarrow t \notin \mathrm{FTV}(S'A')$.

  Since $t \notin \mathrm{FTV}(S'A')$ and $t \in \mathrm{FTV}(S'\sigma')$, there must be some $u \in \mathrm{FTV}(\sigma') - \mathrm{FTV}(A')$ such that $t \in \mathrm{FTV}(S'u)$. However, it is easily checked that $\langle A', \sigma'\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M) \Rightarrow \mathrm{FTV}(\sigma') - \mathrm{FTV}(A') = \varnothing$, so we have reached a contradiction.

- If $\langle B, \tau\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ by rule (SUB), then for some $\tau' \leq_{\forall 2} \tau$, we have a shorter derivation of $\langle B, \tau'\rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.

  By induction there is a pair $\langle A', \sigma'\rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ and a substitution S' such that $B \leq_1 S'A'$, and $S'\sigma' \leq_{\forall 2} \tau'$.

Let $A = A'$, $\sigma = \sigma'$, and $S = S'$. Then $\langle A, \sigma \rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$, $B \leq_1 S'A' = SA$, and $S\sigma = S'\sigma' \leq_{\forall 2} \tau$ by transitivity.

- If $\langle B, \tau \rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ by rule (ADD-HYP), then $B = B \backslash x \cup \{x : \tau'\}$ and $\langle B \backslash x, \tau \rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.

  By induction there is a pair $\langle A', \sigma' \rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ and a substitution S' such that $B \backslash x \leq_1 S'A'$, and $S'\sigma' \leq_{\forall 2} \tau$.

  Let $A = A'$, $\sigma = \sigma'$, and $S = S'$. Then $\langle A, \sigma \rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$, $B \leq_1 B \backslash x \leq_1 S'A' = SA$, and $S\sigma = S'\sigma' \leq_{\forall 2} \tau$ as desired.

- If $\langle B, \tau \rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ by rule (REC), then $M = (\mu x N)$, and for some $\tau' \in \mathbf{T}_1$, we have $\langle B \cup \{x : \tau'\}, \tau \rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$ and $\tau \leq_{\forall 2,1} \tau'$.

  By induction, we have a pair $\langle A', \sigma' \rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(N)$ and a substitution S' such that $B \cup \{x : \tau'\} \leq_1 S'A'$, and $S'\sigma' \leq_{\forall 2} \tau$. We consider two cases.

  - If $x \notin \mathbf{dom}(A')$, let t be fresh and $\pi = \{\sigma' \leq t\}$. Now $S'\sigma' \leq_{\forall 2} \tau \leq_{\forall 2,1} \tau'$, and since $\tau' \in \mathbf{T}_1$, there must be some $\tau'' \in \mathbf{T}_0$ such that $\tau' \leq_1 \tau''$. Then $S'\sigma' \leq_{\forall 2,1} \tau''$, so $R = S' \cup \{t := \tau''\}$ is a solution to π.

    Therefore $\mathbf{MGS}(\pi)$ is nonempty and we may pick $U \in \mathbf{MGS}(\pi)$. Let $A = UA'$ and $\sigma = \mathrm{Gen}(A, U\sigma')$. Then $\langle A, \sigma \rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(\mu x N)$.

    By Convention 2.3.7, there exists an S such that $SA = RA'$ and $SU\sigma' = R\sigma'$.

    Then $B \leq_1 S'A' = RA' = SA$, and $SU\sigma' = R\sigma' = S'\sigma' \leq_{\forall 2} \tau$. Finally, by Lemma 3.9.1(vi) we have $S\sigma = S(\mathrm{Gen}(A, U\sigma')) \leq_{\forall 2} \tau$.

  - If $x \in \mathbf{dom}(A')$, then $S'\sigma' \leq_{\forall 2} \tau \leq_{\forall 2,1} \tau' \leq_1 S'A'(x)$, so S' is a solution to $\pi = \{\sigma' \leq A'(x)\}$.

    Therefore, $\mathbf{MGS}(\pi)$ is nonempty, and we may pick $U \in \mathbf{MGS}(\pi)$. Let $A = UA' \backslash x$ and $\sigma = \mathrm{Gen}(A, U\sigma')$. Then $\langle A, \sigma \rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(\mu x N)$.

    By Convention 2.3.7, there exists an S such that $SA = S'A' \backslash x$ and $SU\sigma' = S'\sigma'$.

    Then $B \leq_1 S'A' \backslash x = SA$, and $SU\sigma' = S'\sigma' \leq_{\forall 2} \tau$. Finally, $S\sigma = S(\mathrm{Gen}(A, U\sigma')) \leq_{\forall 2} \tau$ by Lemma 3.9.1(vi).

- If $\langle B, \tau \rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ by rule (LETREC-VAR),

  then $M = (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ x_{i_0})$ for some $i_0 \in I$, and for some $B_0 = \{x_i : \tau_i' \mid i \in I\}$, we have $\langle B \cup B_0, \tau_i \rangle \in \mathrm{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M_i)$ and $\tau_i \leq_{\forall 2,1} \tau_i'$ for all $i \in I$, and $\tau = \tau_{i_0}$.

  By induction, for all $i \in I$ we have disjoint pairs $\langle A_i, \sigma_i' \rangle \in \mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M_i)$ and substitutions S_i such that $B \cup B_0 \leq_1 S_i A_i$ and $S_i \sigma_i' \leq_{\forall 2} \tau_i$.

Then $S_i\sigma_i' \leq_{\forall 2,1} \tau_i'$, and since each $\tau_i' \in \mathbf{T}_1$, there must be a $\tau_i'' \in \mathbf{T}_0$ such that $\tau_i' \leq_1 \tau_i''$, and thus $S_i\sigma_i' \leq_{\forall 2,1} \tau_i''$.

Let $A' = (\sum_{i \in I} A_i)$, $A'' = A' \cup \{x_i : t_i \mid i \in I,\ x_i \notin \mathbf{dom}(A'),\ t_i \text{ fresh}\}$, $\pi = \{\sigma_i' \leq A''(x_i) \mid i \in I\}$, and $R = \{t_i := \tau_i'' \mid i \in I,\ x_i \notin \mathbf{dom}(A')\} \cup (\bigcup_{i \in I} S_i)$.

If $x_i \in \mathbf{dom}(A')$, then

$$R\sigma_i' = S_i\sigma_i' \leq_{\forall 2,1} \tau_i' \leq_1 RA'(x_i) = RA''(x_i).$$

Otherwise $x_i \notin \mathbf{dom}(A')$ and

$$R\sigma_i' = S_i\sigma_i' \leq_{\forall 2,1} \tau_i'' = Rt_i = RA''(x_i).$$

Thus R is a solution to π.

Therefore $\mathbf{MGS}(\pi)$ is nonempty, and we may pick $U \in \mathbf{MGS}(\pi)$. Let $A = UA' \backslash \{x_i \mid i \in I\}$ and $\sigma = \text{Gen}(A, U\sigma_{i_0}')$. Then $\langle A, \sigma \rangle \in \text{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$.

By Convention 2.3.7, there exists an S such that $SA = RA' \backslash \{x_i \mid i \in I\}$ and $SU\sigma_{i_0}' = R\sigma_{i_0}'$. Then

$$A \leq_1 RA' \backslash \{x_i \mid i \in I\} = SA$$

and

$$SU\sigma_{i_0}' = R\sigma_{i_0}' \leq_{\forall 2} \tau_{i_0} = \tau.$$

And by Lemma 3.9.1(vi), $S\sigma = S(\text{Gen}(A, U\sigma_{i_0}')) \leq_{\forall 2} \tau$, as desired.

- If $\langle B, \tau \rangle \in \text{AP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$ by rule (LETREC), the result follows by Lemma 3.9.2 and induction.

□

3.9.4 Comparison with the other rank 2 systems

Of the rank 2 systems we have defined in this chapter, $\mathbf{P}_2$ is most closely related to $\mathbf{I}_2$. It adds rules (GEN) and (ADD-HYP), and strengthens the rule (SUB) by using $\leq_{\forall 2}$ in place of $\leq_2$. The $\mathbf{P}_2$ rules (VAR) and (ABS) are weaker than their $\mathbf{I}_2$ counterparts, but the $\mathbf{I}_2$ versions are derivable using (ADD-HYP). In summary, $\mathbf{P}_2$ extends $\mathbf{I}_2$:

Lemma 3.9.7 *If* $\mathbf{I}_2 \triangleright A \vdash M : \sigma$, *then* $\mathbf{P}_2 \triangleright A \vdash M : \sigma$.

In fact, a stronger connection can be shown: a term is typable in one system if and only if it is typable in the other.

Theorem 3.9.8 (Comparison of $\mathbf{P}_2$ and $\mathbf{I}_2$) $\mathbf{P}_2 \triangleright A \vdash M : \forall \vec{t} \sigma$ *for some* $\vec{t}$ *if and only if* $\mathbf{I}_2 \triangleright A \vdash M : \sigma$.

Proof: The right-to-left direction follows by Lemma 3.9.7. The left-to-right direction is proved by induction on derivations; the only non-trivial case is (SUB), which can be shown as follows.

If $\mathbf{P}_2 \triangleright A \vdash M : \forall\vec{t}\sigma$ follows by rule (SUB), then we must have a shorter derivation of

$$\mathbf{P}_2 \triangleright A \vdash M : \forall\vec{s}\tau,$$

and $(\forall\vec{s}\tau) \leq_{\forall 2} (\forall\vec{t}\sigma)$. We must show $\mathbf{I}_2 \triangleright A \vdash M : \sigma$.

By induction, $\mathbf{I}_2 \triangleright A \vdash M : \tau$. Furthermore, by the definition of $\leq_{\forall 2}$, for some sequence $\vec{\rho}$ of simple types, we have $\{\vec{s} := \vec{\rho}\}\tau \leq_2 \sigma$. We may assume that the type variables $\vec{s}$ do not appear in A. Then by substitutivity for $\mathbf{I}_2$,

$$\mathbf{I}_2 \triangleright A \vdash M : \{\vec{s} := \vec{\rho}\}\tau,$$

and by the $\mathbf{I}_2$ rule (SUB), we have $\mathbf{I}_2 \triangleright A \vdash M : \sigma$, as desired. □

Now we show that our rules for typing recursive definitions are at least as powerful as the usual ones for ML.

We write $\mathrm{ML}^{\mathrm{R}} \triangleright A \vdash M : \tau$ if $A \vdash M : \tau$ follows by the rules of ML and the rules (REC-SIMPLE) and (LETREC-SIMPLE), and $\Lambda_2^{\mathrm{R}} \triangleright A \vdash M : \tau$ if $A \vdash M : \tau$ follows by the rules of Λ_2^{s} and (REC-SIMPLE) and (LETREC-SIMPLE).

It is well known (cf. [32]) that Λ_2^{R} types strictly more terms than ML^{R}.

Theorem 3.9.9 (Comparison of ML^{R} and Λ_2^{R}) *If $\mathrm{ML}^{\mathrm{R}} \triangleright A \vdash M : \tau$, then $\Lambda_2^{\mathrm{R}} \triangleright A \vdash M : \tau$. The converse does not hold.*

To show the relationship between Λ_2^{R} and $\mathbf{P}_2^{\mathrm{R}}$, we first state the following result, without proof.

Lemma 3.9.10 *If $M = (\mathbf{letrec}\ B\ \mathbf{in}\ N)$, then $\Lambda_2^{\mathrm{R}} \triangleright A \vdash M : \sigma$ iff $\Lambda_2^{\mathrm{R}} \triangleright A \vdash \langle\langle M \rangle\rangle : \sigma$.*

Theorem 3.9.11 (Comparison of Λ_2^{R} and $\mathbf{P}_2^{\mathrm{R}}$) *If $\Lambda_2^{\mathrm{R}} \triangleright A \vdash M : \tau$, then $\mathbf{P}_2^{\mathrm{R}} \triangleright A' \vdash M : \tau'$, where $A \preceq_1 A'$ and $\tau \preceq_2 \tau'$.*

Proof: We will use the following facts, which we state without proof:

- If $\tau \preceq_1 \tau'$ and $\tau \in \mathbf{T}_0$, then $\tau = \tau'$.
- If $\tau \preceq_2 \tau'$ and $\tau \in \mathbf{T}_0$, then $\tau = \tau'$.
- If $A \preceq_1 A'$, then $SA \preceq_1 SA'$.
- If $\Lambda_2^{\mathrm{R}} \triangleright A \vdash M : \sigma$ and $\mathrm{Gen}(A, \sigma) \succ \sigma'$, then $\Lambda_2^{\mathrm{R}} \triangleright A \vdash M : \sigma'$.

We prove the theorem by induction on M. By Lemmas 3.9.2 and 3.9.10, we need not consider the case $M = (\textbf{letrec } B \textbf{ in } N)$ where N is not a variable defined in B. The cases $M = x$, $M = (\lambda x N)$, and $M = (M_1 M_2)$ can be proved just as in Lemma 3.4.5, and the case $M = (\mu x N)$ is trivial. That leaves only the following case.

- $M = (\textbf{letrec } \{x_i = M_i \mid i \in I\} \textbf{ in } x_{i_0})$, where $i_0 \in I$.

  Then for some $\mathbf{T}_0$ type environment $A_0 = \{x_j : \tau_j \mid j \in I\}$, we have

$$\Lambda_2^{\mathrm{R}} \triangleright A \cup A_0 \vdash M_i : \tau_i$$

  for all $i \in I$, and $\mathrm{Gen}(A, \tau_{i_0}) \succ \tau$.

  By induction, for all $i \in I$ we have

$$\mathbf{P}_2^{\mathrm{R}} \triangleright A_i' \vdash M_i : \tau_i',$$

  where $(A \cup A_0) \preceq_1 A_i'$ and $\tau_i \preceq_2 \tau_i'$. Since $\tau_i \in \mathbf{T}_0$ we have $\tau_i = \tau_i'$.
  Let $A' = (\sum_{i \in I} A_i') + A_0$; then $(A \cup A_0) \preceq_1 A'$, and $A'(x_j) = \tau_j$ for any $j \in I$. By weakening, $\mathbf{P}_2^{\mathrm{R}} \triangleright A' \vdash M_i : \tau_i$ for all $i \in I$. By rule (LETREC-VAR),

$$\mathbf{P}_2^{\mathrm{R}} \triangleright A'' \vdash (\textbf{letrec } \{x_i = M_i \mid i \in I\} \textbf{ in } x_{i_0}) : \tau_{i_0},$$

  where $A'' = A' \backslash \{x_j \mid j \in I\}$. Since $\mathrm{Gen}(A, \tau_{i_0}) \succ \tau$, for some substitution S we have $S\tau_{i_0} = \tau$ and $\mathbf{dom}(S) = \mathrm{FTV}(\tau_{i_0}) - \mathrm{FTV}(A)$. By substitutivity,

$$\mathbf{P}_2^{\mathrm{R}} \triangleright SA'' \vdash (\textbf{letrec } \{x_i = M_i \mid i \in I\} \textbf{ in } x_{i_0}) : \tau,$$

  and $A \preceq_1 A'' \Rightarrow A = SA \preceq_1 SA''$.

□

Chapter 4

Type inference with subtyping

In this chapter we study type inference for a broad class of type systems that extend the rules of the simply typed lambda calculus with subtyping and the rule of subsumption:

$$(\text{SUB}) \quad \frac{A \vdash M : \tau_1}{A \vdash M : \tau_2} \quad \tau_1 \leq \tau_2$$

Type systems without subtyping, such as the simply typed lambda calculus [12] and the system of recursive types [8], are included in this class, since the equality of types is a trivial subtyping relation.

Some systems in this class have a notion of principal typing. Others have no such notion, and these are of particular interest to us. A case in point is the system of recursive types and subtyping introduced by Amadio and Cardelli [3, 4]. The system has no known notion of principal typing; nevertheless, Palsberg and O'Keefe [51] succeeded in designing a type inference algorithm for the system.

To put this in context with the results of the previous chapters, we must address the question: does the Palsberg-O'Keefe algorithm have anything to do with principal typings, or is an entirely different methodology at work? The answer, it turns out, is that their algorithm relies on a weak form of principal typings. The algorithm takes a term M and produces a triple $\langle C_M, A_M, t_M\rangle$ where C_M is a set of type inequalities (a subtype satisfaction problem), A_M is a type environment, and t_M is a type variable. They prove the following theorem:

> **Theorem.** $A \vdash M : \sigma$ if and only if there is a solution S of C_M such that $SA_M \subseteq A$ and $S(t_M) = \sigma$.

This is almost a principal typing theorem: it says that $\langle C_M, A_M, t_M\rangle$ represents all typings of M, and lacks only that $\langle C_M, A_M, t_M\rangle$ is not itself a typing. A method for finding solutions to C_M completes the type inference algorithm.

This algorithm is not the only example of type inference by reduction to a system of type inequalities. Indeed, there is a large body of work on just this subject; besides

the Palsberg-O'Keefe paper, we mention [67, 49, 38, 61, 62, 16] as just a few examples. Our interest in the Amadio-Cardelli system in particular stems from a recent result of Palsberg and Smith, who showed that the system types exactly the same terms as a type system based on constraints (personal communication, September 1995).

We go further, and show that the Palsberg-O'Keefe algorithm not only identifies the typable terms of the two systems, it also infers *principal* typings in the constraint-based system (the equivalence of the two type systems is an immediate corollary of this result). The relationship of the systems is much like the relationship between the rank 2 type systems we studied in the last chapter: the terms typable in the two systems are identical, one system has principal typings, and the other system does not.

Moreover, we prove our result in a general setting. The reduction of type inference in lambda calculi with or without subtyping to solving systems of type equalities or inequalities is an idea so fundamental that it has been called "folkloric" [67, 24]. However, in practice this idea has been implemented differently in different type systems. Not only does the reduction itself vary from system to system, but the notions of instance and of principal typings or representatives vary as well. Consequently, the correctness of the reduction and the property of principal typings or representatives must be proved anew for each system.

Our contribution is to show that for a wide class of type systems, a single reduction suffices. We prove an equivalence between a class of type systems with subtyping (including the Amadio-Cardelli system) and a class of constraint-based type systems. We give an algorithm for reducing type inference for any language in the first class into subtype satisfaction, and show that the same algorithm produces principal typings in the class of constraint-based systems. And finally, we show how type systems with and without constraints are related.

Organization of the chapter. We describe the system of recursive types and subtyping in §4.1. In §4.2 we introduce its generalization, and show how to reduce the type inference problem to subtype satisfaction. In §4.3 we define the constraint-based systems, and we show how they satisfy principal typings. Finally in §4.4 we show how the systems with and without constraints are related.

4.1 Recursive types with subtyping

We begin by describing a particular member of our class of type systems, namely, the Amadio-Cardelli system of recursive types with subtyping. The original Amadio-Cardelli system is a "Church style," explicitly typed language in which every variable is annotated with its type, and the typing of a term is determined entirely by its syntax. Since we are interested in type inference, we will work with its "Curry style," implicitly typed variant, which we call Λ_μ. We give a minimal presentation based on that of Palsberg and O'Keefe [51].

The terms of Λ_μ are defined by the following grammar.

$$M ::= x \mid (\lambda x M) \mid (M_1 M_2) \mid 0 \mid \mathbf{succ}$$

That is, we extend the terms of the lambda calculus with two constants, 0 and the successor function **succ**. These constants are not added to the language in order to make it more realistic; rather, they are added to make the typing problem non-trivial—without constants, every term would be typable in Λ_μ. With 0 and **succ**, we can write untypable programs such as $(0x)$ and $(\mathbf{succ}\ (\lambda x.x))$.

The recursive types $\mathbf{T}_\mu$ are defined by the following grammar.

$$\tau ::= t \mid (\tau_1 \to \tau_2) \mid (\mu t \tau) \mid \bot \mid \top \mid \text{INT}$$

We may think of a recursive type as its infinite unfolding under the rule

$$(\mu t \tau) \to \begin{cases} \bot & \text{if } \tau = (\mu t_1 \cdots \mu t_n t) \quad \text{for } n \geq 0, \\ \{t := (\mu t \tau)\}\tau & \text{otherwise.} \end{cases}$$

This unfolding expands away all uses of μ, so that types are a class of regular trees [11] over the alphabet $\Sigma_\mu = \{\to, \text{INT}, \bot, \top\} \cup \mathbf{Tv}$.[1]

We may also think of a recursive type as a partial function from *paths* to Σ_μ, where a path is a string over $\{0, 1\}$, with 0 indicating "left subtree," and 1 indicating "right subtree." We write $\mathbf{dom}(\tau)$ for the domain of τ when it is thought of as such a partial function.

The *parity* of a path is the number mod 2 of 0's in the path. We write $\mathbf{parity}(\alpha)$ for the parity of a path α. Let $\leq_\mu^0$ be the least partial order on Σ_μ satisfying

$$\begin{array}{lcll} \bot \leq_\mu^0 \to & \text{and} & \to \leq_\mu^0 \top & \text{and} \\ \bot \leq_\mu^0 \text{INT} & \text{and} & \text{INT} \leq_\mu^0 \top, & \end{array}$$

and let $\leq_\mu^1$ be its reverse, the least partial order satisfying

$$\begin{array}{lcll} \top \leq_\mu^1 \to & \text{and} & \to \leq_\mu^1 \bot & \text{and} \\ \top \leq_\mu^1 \text{INT} & \text{and} & \text{INT} \leq_\mu^1 \bot. & \end{array}$$

Then we define the subtyping relation $\leq_\mu$ by

$$\sigma \leq_\mu \tau \text{ iff } \sigma(\alpha) \leq_\mu^{\mathbf{parity}(\alpha)} \tau(\alpha) \text{ for all } \alpha \in \mathbf{dom}(\sigma) \cap \mathbf{dom}(\tau).$$

Note in particular that $\leq_\mu$ satisfies the property

$$\sigma_1 \to \sigma_2 \leq_\mu \tau_1 \to \tau_2 \quad \text{if and only if} \quad \tau_1 \leq_\mu \sigma_1 \text{ and } \sigma_2 \leq_\mu \tau_2.$$

Kozen et al. [39] have shown that the relation $\leq_\mu$ is decidable in $O(n^2)$ time.

(ZERO) $A \vdash 0 : \text{INT}$

(SUCC) $A \vdash \mathbf{succ} : \text{INT} \to \text{INT}$

(VAR) $A \vdash x : A(x)$

(ABS) $$\frac{A \backslash x \cup \{x : \tau_1\} \vdash M : \tau_2}{A \vdash \lambda x M : \tau_1 \to \tau_2}$$

(APP) $$\frac{A \vdash M_1 : \tau_1 \to \tau_2, \quad A \vdash M_2 : \tau_1}{A \vdash M_1 M_2 : \tau_2}$$

(SUB) $$\frac{A \vdash M : \tau_1}{A \vdash M : \tau_2} \quad \tau_1 \leq_\mu \tau_2$$

Figure 4.1: Typing rules of Λ_μ.

The typing rules of Λ_μ are given in Figure 4.1. In every Λ_μ judgment $A \vdash M : \sigma$, the type environment A holds $\mathbf{T}_\mu$ types, and the derived type σ is a $\mathbf{T}_\mu$ type.

We give some example Λ_μ typings below.

- $(\lambda x.xx)$ has both the types $(\mu s.s \to t)$ and $(\mu s.s \to t) \to t$, so the divergent term $(\lambda x.xx)(\lambda x.xx)$ has type t in Λ_μ.
- For any pure term M, if $\tau_0 = (\mu t.t \to t)$ and A maps every variable in $\mathbf{FV}(M)$ to τ_0, then $\Lambda_\mu \triangleright A \vdash M : \tau_0$. Thus every pure term is typable in the system.
- The fixed point operator $Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ has type $(\sigma \to \sigma) \to \sigma$ in Λ_μ, where σ is any $\mathbf{T}_\mu$ type.

4.2 Lambda calculi with subtyping

We now generalize the definitions of the last section to a wide class of type systems based on subtyping. Many of these systems have no known notion of principal typing; for example, our definitions apply to type systems without type variables, and any such system will not have principal typings in the sense of Chapter 2.

[1]Palsberg and O'Keefe do not include type variables in their alphabet. This restriction in effect requires all types to be closed, which has no effect on the set of terms typable in the system. Free type variables can be added to their system without affecting any of the constructions or results of their work (personal communication, Palsberg, November 1995).

(CONST) $A \vdash c^\sigma : \sigma$

(VAR) $A \vdash x : A(x)$

(ABS) $$\frac{A \backslash x \cup \{x : \tau_1\} \vdash M : \tau_2}{A \vdash \lambda x M : \tau_1 \rightarrow \tau_2}$$

(APP) $$\frac{A \vdash M_1 : \tau_1 \rightarrow \tau_2, \quad A \vdash M_2 : \tau_1}{A \vdash M_1 M_2 : \tau_2}$$

(SUB) $$\frac{A \vdash M : \tau_1}{A \vdash M : \tau_2} \quad \tau_1 \leq \tau_2$$

Figure 4.2: Typing rules of $\Lambda_\leq$.

Nevertheless, we will show that *every* type system in this class satisfies a "principal representatives" theorem, stating that for any typable M, there is a triple $\langle C_M, A_M, t_M \rangle$ encoding all of the possible typings of M. Throughout we are careful to minimize our assumptions, so that our results will apply to as wide a class of type systems as possible.

We define type systems $\Lambda(\mathbf{T}, \leq, \mathbf{C})$, where $\mathbf{T}$ is a set of types, $\leq$ is a preorder on $\mathbf{T}$, and $\mathbf{C}$ is a set of explicitly typed constants ranged over by c^σ. We make the following minimal requirements on $\mathbf{T}$ and $\mathbf{C}$.

- If $\sigma \in \mathbf{T}$ and $\tau \in \mathbf{T}$, then $(\sigma \rightarrow \tau) \in \mathbf{T}$.
- If $(\sigma \rightarrow \tau) \in \mathbf{T}$ and $\sigma \in \mathbf{T}$, then $\tau \in \mathbf{T}$.
- If $c^\sigma \in \mathbf{C}$, then σ is closed and $\sigma \in \mathbf{T}$.

These requirements are necessary to insure that the typing rules cannot derive types outside of $\mathbf{T}$.

We will abbreviate $\Lambda(\mathbf{T}, \leq, \mathbf{C})$ by $\Lambda_\leq$, assuming that $\mathbf{T}$ and $\mathbf{C}$ can be recovered from context. The terms of $\Lambda_\leq$ are defined by the grammar

$$M ::= x \mid (\lambda x M) \mid (M_1 M_2) \mid c^\sigma$$

and the typing rules of $\Lambda_\leq$ are given in Figure 4.2. Type environments are required to map variables to types in $\mathbf{T}$ only. Then it is guaranteed that $\sigma \in \mathbf{T}$ in any derivable judgment $A \vdash M : \sigma$.

The rules (CONST), (VAR), (ABS), and (APP) are the usual typing rules of the lambda calculus with constants, and rule (SUB) is the rule of subsumption. Subtyping can be removed from the system by using equality of types as the subtyping relation.

For example:

- The pure simply typed lambda calculus is the type system $\Lambda(\mathbf{T}_0, =, \varnothing)$.
- The system Λ_μ of recursive types from the last section is

$$\Lambda(\mathbf{T}_\mu, \leq_\mu, \{0^{\text{INT}}, \mathbf{succ}^{\text{INT}\to\text{INT}}\}).$$

At this point it is worth pointing out how few assumptions we are making about the parameters of the system. We do not require that $\leq$ be a partial order (it need not be anti-symmetric); we do not require that $\mathbf{T}$ be closed under $\mathbf{T}$ substitutions; we do not require that $\mathbf{T}$ contain type variables; we do not require $\leq$ to obey contravariant function subtyping, or make any assumptions about function subtyping whatsoever.

Nevertheless, the assumptions we have made are enough for us to prove all of the results of this section.

Lemma 4.2.1 (Characterization of typings) $\Lambda_{\leq} \triangleright A \vdash M : \tau$ *iff*

- $M = c^\sigma$ *and* $\sigma \leq \tau$*;*
- $M = x$ *and* $A(x) \leq \tau$*;*
- $M = \lambda x M'$*, and for some* τ_1 *and* τ_2*,* $\Lambda_{\leq} \triangleright A\backslash x \cup \{x : \tau_1\} \vdash M' : \tau_2$ *and* $\tau_1 \to \tau_2 \leq \tau$*; or*
- $M = M_1 M_2$*, and for some* τ_1 *and* τ_2*,* $\Lambda_{\leq} \triangleright A \vdash M_1 : \tau_1 \to \tau_2$*,* $\Lambda_{\leq} \triangleright A \vdash M_2 : \tau_1$*, and* $\tau_2 \leq \tau$*.*

Proof: ($\Leftarrow$) Trivial.

($\Rightarrow$) An easy induction on the derivation of $\Lambda_{\leq} \triangleright A \vdash M : \tau$, using the reflexivity and transitivity of $\leq$. □

Lemma 4.2.2 (Sound operations on typings) *If* $\Lambda_{\leq} \triangleright A \vdash M : \tau$*, then*

- $\Lambda_{\leq} \triangleright A' \vdash M : \tau$*, where* $A' \leq A$*; and*
- $\Lambda_{\Vdash} \triangleright A \vdash M : \tau'$*, where* $\tau \leq \tau'$*.*

Proof: The first case follows by a simple induction on typing derivations, and the second follows immediately by the rule (SUB). □

We now define the reduction from type inference to sets of type inequalities. For convenience, in the rest of this chapter we assume that all free and bound variables in terms are distinct.

Definition 4.2.3 (Type inference) For any term M, we define X_M, Y_M, A_M, and C_M as follows.

- X_M is a set of fresh type variables. It contains a type variable s_x for every term variable x appearing in M.
- Y_M is a set of fresh type variables. It contains a type variable $t_{M'}$ for each occurrence of a subterm M' of M, and a type variable $s_{M'}$ for each occurrence of an applicative subterm $M' = (M_1 M_2)$ of M. (If M' occurs more than once in M, then $t_{M'}$ and $s_{M'}$ are ambiguous. However, it will always be clear from context which occurrence is meant.)
- A_M is a type environment, defined by

$$A_M = \{x : s_x \mid x \text{ is free in } M\}.$$

- C_M is the set of the following constraints over X_M and Y_M:
  - For each occurrence in M of a subterm c^σ, the constraint

$$\sigma \leq t_{c^\sigma}.$$

  - For each occurrence in M of a subterm x, the constraint

$$s_x \leq t_x.$$

  - For each occurrence in M of a subterm $(\lambda x M')$, the constraint

$$s_x \to t_{M'} \leq t_{\lambda x M'}.$$

  - For each occurrence in M of a subterm $(M_1 M_2)$, the constraints

$$t_{M_1} \leq t_{M_2} \to s_{M_1 M_2},$$
$$s_{M_1 M_2} \leq t_{M_1 M_2}.$$

Except for minor notional differences, this definition is almost identical to that of Palsberg and O'Keefe [51]. The only significant differences are that we handle general constants c^σ instead of just 0 and **succ**, and our constraints for applications $(M_1 M_2)$ are different because we make fewer assumptions about the subtyping relation $\leq$. In particular, Palsberg and O'Keefe use the following property of $\leq_\mu$:

$$\text{If } \tau \leq_\mu \tau' \text{ then } \sigma \to \tau \leq_\mu \sigma \to \tau'.$$

This is a fairly common property; for example, it is satisfied by any subtyping relation with contravariant function subtyping. If we were to assume that $\leq$ satisfied this property, the application case could be simplified to the single constraint

$$t_{M_1} \leq t_{M_2} \to t_{M_1 M_2}.$$

We have said that C_M is a set of constraints, without giving a precise definition of what a constraint is.

Definition 4.2.4 (Constraints and solutions) If $\mathbf{T}'$ is a set of types, then a $\mathbf{T}'$ *constraint* is a pair (σ, τ) where $\sigma, \tau \in \mathbf{T}'$. We use C to range over sets of constraints. If $\mathbf{T}$ is a set of types and $\leq$ is a binary relation on $\mathbf{T}$, then a $(\mathbf{T}, \leq)$ *solution* to a set C of $\mathbf{T}'$ constraints is a substitution S such that for every $(\sigma, \tau) \in C$, both $S\sigma \in \mathbf{T}$ and $S\tau \in \mathbf{T}$, and $S\sigma \leq S\tau$.

Notice that we do not require the types $\mathbf{T}'$ of constraints and the types $\mathbf{T}$ of the subtyping preorder $\leq$ to be the same. Our type inference algorithm above requires constraints whose types are one of: a type σ of a constant c^σ; a type variable t; or a type $t_1 \to t_2$. However, the types t or $t_1 \to t_2$ need not be members of $\mathbf{T}$ by our definition. Thus our main theorem below will apply to type systems without type variables.

By convention, a constraint (σ, τ) is written $(\sigma \leq \tau)$; in this notation, the symbol $\leq$ is simply syntactic, and does not indicate that (σ, τ) is a member of the relation $\leq$. Again by convention, we call $\mathbf{T}'$ constraints just *constraints*, and $(\mathbf{T}, \leq)$ solutions just *solutions*, when the parameters $\mathbf{T}'$, $\mathbf{T}$, and $\leq$ can be recovered from context.

The main theorem is stated and proved as follows.

Theorem 4.2.5 (Principal representatives) $\Lambda_{\leq} \triangleright A \vdash M : \tau$ *iff there is a solution S of C_M such that $S(t_M) = \tau$ and $SA_M \subseteq A$.*

The proof is unremarkable. We include it only to convince the reader that the assumptions we have made so far are sufficient for the proof.

Proof: ($\Rightarrow$) An easy induction on M using Lemma 4.2.1.

- If $M = c^\sigma$, then $\sigma \leq \tau$, $A_M = \varnothing$, and $C_M = \{\sigma \leq t_M\}$.

  Then $S = \{t_M := \tau\}$ is a solution of C_M, $S(t_M) = \tau$, and $SA_M = \varnothing \subseteq A$.

- If $M = x$, then $A(x) \leq \tau$, $A_M = \{x : s_x\}$, and $C_M = \{s_x \leq t_x\}$.

  Then $S = \{s_x := A(x), t_x := \tau\}$ is a solution to C_M, $S(t_M) = S(t_x) = \tau$, and $SA_M = \{x : A(x)\} \subseteq A$.

- If $M = \lambda x M'$, then for some τ_1 and τ_2, $\Lambda_{\leq} \triangleright A \backslash x \cup \{x : \tau_1\} \vdash M' : \tau_2$, $\tau_1 \to \tau_2 \leq \tau$, $A_M = (A_{M'}) \backslash x$, and $C_M = C_{M'} \cup \{s_x \to t_{M'} \leq t_M\}$.

  By induction, there is a solution S' of $C_{M'}$ such that such that $S'(t_{M'}) = \tau_2$ and $S'A_{M'} \subseteq A \backslash x \cup \{x : \tau_1\}$.

  We may assume that $\mathbf{dom}(S') = X_{M'} \cup Y_{M'}$, so that $S = S' \cup \{s_x := \tau_1, t_M := \tau\}$ is well-defined.

  Then S solves C_M, $SA_M = S'A_{M'} \backslash x \subseteq A \backslash x \subseteq A$, and $S(t_M) = \tau$.

- If $M = M_1 M_2$, then for some τ_1 and τ_2, $\Lambda_{\leq} \triangleright A \vdash M_1 : \tau_1 \to \tau_2$, $\Lambda_{\leq} \triangleright A \vdash M_2 : \tau_1$, $\tau_2 \leq \tau$, A_M is the union of A_{M_1} and A_{M_2}, and $C_M = C_{M_1} \cup C_{M_2} \cup \{t_{M_1} \leq t_{M_2} \to s_M, s_M \leq t_M\}$.

By induction, there are solutions S_1 of C_{M_1} and S_2 of C_{M_2} such that $S_1(t_{M_1}) = \tau_1 \to \tau_2$, $S_2(t_{M_2}) = \tau_1$, $S_1 A_{M_1} \subseteq A$ and $S_2 A_{M_2} \subseteq A$.

We may assume that $\mathbf{dom}(S_i) = X_{M_i} \cup Y_{M_i}$ for $i \in \{1,2\}$. Note that Y_{M_1}, Y_{M_2}, and $\{t_M\}$ are disjoint, and that if $s_x \in X_{M_1} \cap X_{M_2}$ then $S_1(s_x) = S_2(s_x) = A(x)$. Therefore $S = S_1 \cup S_2 \cup \{s_M := \tau_2, t_M := \tau\}$ is well-defined.

Then S solves C_M, $SA_M \subseteq A$, and $S(t_M) = \tau$.

(This is where we would need stronger conditions on function subtyping if we used the constraints of Palsberg-O'Keefe: to solve the constraint $t_{M_1} \leq t_{M_2} \to t_M$ we need $\tau_1 \to \tau_2 \leq \tau_1 \to \tau$, but we only know $\tau_2 \leq \tau$.)

($\Leftarrow$) Prove the following statement by induction on M_0:

If S is a solution of C_M, then $\Lambda_{\leq} \triangleright SA_{M_0} \vdash M_0 : S(t_{M_0})$ for every subterm M_0 of M.

- If $M_0 = c^\sigma$, then $A_{M_0} = \varnothing$, and $C_M \supseteq \{\sigma \leq t_{M_0}\}$.

  Then $\Lambda_{\leq} \triangleright SA_{M_0} \vdash c^\sigma : \sigma$ by (CONST), and $\Lambda_{\leq} \triangleright SA_{M_0} \vdash c^\sigma : S(t_{M_0})$ by (SUB).

- If $M_0 = x$, then $A_{M_0} = \{x : s_x\}$, and $C_M \supseteq \{s_x \leq t_{M_0}\}$.

  Then $\Lambda_{\leq} \triangleright SA_{M_0} \vdash x : S(s_x)$ by (VAR), and $\Lambda_{\leq} \triangleright SA_{M_0} \vdash x : S(t_{M_0})$ by (SUB).

- If $M_0 = \lambda x M'$, then $A_{M_0} = (A_{M'})\backslash x$, and $C_M \supseteq \{s_x \to t_{M'} \leq t_{M_0}\}$.

  By induction, $\Lambda_{\leq} \triangleright SA_{M'} \vdash M' : S(t_{M'})$. Note that $A_{M'} \subseteq (A_{M'})\backslash x \cup \{x : s_x\} = A_{M_0} \cup \{x : s_x\}$. So by weakening, $\Lambda_{\leq} \triangleright SA_{M_0} \cup \{x : S(s_x)\} \vdash M' : S(t_{M'})$, and by (ABS), $\Lambda_{\leq} \triangleright SA_{M_0} \vdash \lambda x M' : S(s_x) \to S(t_{M'})$. Since S solves C_M, $S(s_x) \to S(t_{M'}) \leq S(t_{M_0})$. So by (SUB), $\Lambda_{\leq} \triangleright SA_{M_0} \vdash M_0 : S(t_{M_0})$.

- If $M_0 = M_1 M_2$, then A_M is the union of A_{M_1} and A_{M_2}, and $C_M \supseteq \{t_{M_1} \leq t_{M_2} \to s_{M_0}, s_{M_0} \leq t_{M_0}\}$.

  By induction $\Lambda_{\leq} \triangleright SA_{M_1} \vdash M_1 : S(t_{M_1})$ and $\Lambda_{\leq} \triangleright SA_{M_2} \vdash M_2 : S(t_{M_2})$. By weakening, $\Lambda_{\leq} \triangleright SA_{M_0} \vdash M_1 : S(t_{M_1})$ and $\Lambda_{\leq} \triangleright SA_{M_0} \vdash M_2 : S(t_{M_2})$. By (SUB), $\Lambda_{\leq} \triangleright SA_{M_0} \vdash M_1 : S(t_{M_2}) \to S(s_{M_0})$. By (APP), $\Lambda_{\leq} \triangleright SA_{M_0} \vdash M_0 : S(s_{M_0})$. Finally by (SUB), $\Lambda_{\leq} \triangleright SA_{M_0} \vdash M_0 : S(t_{M_0})$.

□

4.3 Constraint-based type systems

We now introduce a class of type systems based on constraints, generalizing the systems introduced by Mitchell [47, 49]. Our main result will be that the algorithm that produced principal *representatives* for the type systems of the last section produces principal *typings* for the constraint-based systems. Once again, we are careful to minimize the assumptions we make, so that our results apply to as many type systems as possible.

Definition 4.3.1 (Pre-entailment) A *pre-entailment relation on* $\mathbf{T}$ is a relation, $\Vdash$, between $\mathbf{T}$ constraint sets and $\mathbf{T}$ constraints satisfying the axioms and rules below.

$$\text{(HYP)} \qquad C \cup \{\tau_1 \leq \tau_2\} \Vdash \tau_1 \leq \tau_2$$

$$\text{(REFL)} \qquad C \Vdash \tau \leq \tau \qquad \text{for any } \tau \in \mathbf{T}$$

$$\text{(TRANS)} \qquad \frac{C \Vdash \tau_1 \leq \tau_2, \quad C \Vdash \tau_2 \leq \tau_3}{C \Vdash \tau_1 \leq \tau_3}$$

We write $C \Vdash C'$ if $C \Vdash \tau_1 \leq \tau_2$ for every $(\tau_1 \leq \tau_2) \in C'$, and $C \Vdash A \leq A'$ if $\mathbf{dom}(A') \subseteq \mathbf{dom}(A)$, and $C \Vdash A(x) \leq A'(x)$ for all $x \in \mathbf{dom}(A')$.

We define type systems $\Lambda(\mathbf{T}, \Vdash, \mathbf{C})$, where $\mathbf{T} \supseteq \mathbf{T}_0$ is a set of types, $\Vdash$ is a pre-entailment relation on $\mathbf{T}$, and $\mathbf{C}$ is a set of explicitly typed constants ranged over by c^σ. Just as with $\Lambda_\leq$, we make the following minimal requirements on $\mathbf{T}$ and $\mathbf{C}$ to insure that the typing rules cannot derive types outside of $\mathbf{T}$.

- If $\sigma \in \mathbf{T}$ and $\tau \in \mathbf{T}$, then $(\sigma \to \tau) \in \mathbf{T}$.
- If $(\sigma \to \tau) \in \mathbf{T}$ and $\sigma \in \mathbf{T}$, then $\tau \in \mathbf{T}$.
- If $c^\sigma \in \mathbf{C}$, then σ is closed and $\sigma \in \mathbf{T}$.

We will abbreviate $\Lambda(\mathbf{T}, \Vdash, \mathbf{C})$ by $\Lambda_\Vdash$, assuming that $\mathbf{T}$ and $\mathbf{C}$ can be recovered from context. The terms of $\Lambda_\Vdash$ are defined by the grammar

$$M ::= x \mid (\lambda x M) \mid (M_1 M_2) \mid c^\sigma$$

Because we want every system $\Lambda_\Vdash$ to have principal typings, we have made two requirements that were not made for $\Lambda_\leq$. First, we require $\mathbf{T}_0 \subseteq \mathbf{T}$; type variables, and therefore all of $\mathbf{T}_0$, are needed for our algorithm to deduce sound typings (Theorem 4.3.5). Second, in $\Lambda_\Vdash$ the types in constraints, the types in type environments, and the derived types all come from the same set. It would be possible to relax this requirement, but there does not seem much point to it. The reason for allowing the constraint types to be different in $\Lambda_\leq$ was that it let us talk about type systems without type variables.

(CONST) $C, A \vdash c^\sigma : \sigma$

(VAR) $C, A \vdash x : A(x)$

(ABS) $$\frac{C, A\backslash x \cup \{x : \tau_1\} \vdash M : \tau_2}{C, A \vdash \lambda x M : \tau_1 \to \tau_2}$$

(APP) $$\frac{C, A \vdash M_1 : \tau_1 \to \tau_2, \quad C, A \vdash M_2 : \tau_1}{C, A \vdash M_1 M_2 : \tau_2}$$

(SUB) $$\frac{C, A \vdash M : \tau_1}{C, A \vdash M : \tau_2} \quad C \Vdash \tau_1 \leq \tau_2$$

Figure 4.3: Typing rules of $\Lambda_{\Vdash}$.

We now extend our notion of typing judgment to include quadruples of the form $C, A \vdash M : \tau$. For any pre-entailment relation $\Vdash$, we write $\Lambda_{\Vdash} \triangleright C, A \vdash M : \sigma$ if $C, A \vdash M : \sigma$ follows by the rules of Figure 4.3.

Lemma 4.3.2 (Characterization of typings) *$\Lambda_{\Vdash} \triangleright C, A \vdash M : \tau$ iff*

- *$M = c^\sigma$ and $C \Vdash \sigma \leq \tau$;*
- *$M = x$ and $C \Vdash A(x) \leq \tau$;*
- *$M = \lambda x M'$, and for some τ_1, τ_2, we have $\Lambda_{\Vdash} \triangleright C, A\backslash x \cup \{x : \tau_1\} \vdash M' : \tau_2$, and $C \Vdash \tau_1 \to \tau_2 \leq \tau$; or*
- *$M = M_1 M_2$, and for some τ_1, τ_2, we have $\Lambda_{\Vdash} \triangleright C, A \vdash M_1 : \tau_1 \to \tau_2$, $\Lambda_{\Vdash} \triangleright C, A \vdash M_2 : \tau_1$, and $C \Vdash \tau_2 \leq \tau$.*

Proof: ($\Leftarrow$) Trivial.
($\Rightarrow$) An easy induction on the derivation of $\Lambda_{\Vdash} \triangleright C, A \vdash M : \tau$, using (REFL) and (TRANS). The rule (HYP) is not needed. □

Lemma 4.3.3 (Sound operations on typings) *If $\Lambda_{\Vdash} \triangleright C, A \vdash M : \tau$, then*

- *$\Lambda_{\Vdash} \triangleright C, A' \vdash M : \tau$, where $C \Vdash A' \leq A$; and*
- *$\Lambda_{\Vdash} \triangleright C, A \vdash M : \tau'$, where $C \Vdash \tau \leq \tau'$.*

Proof: The first case follows by induction on the derivation of $\Lambda_{\Vdash} \triangleright C, A \vdash M : \tau$ using (TRANS), and the second follows immediately by (SUB) and (TRANS). □

Definition 4.3.4 (Instance) We write $\langle C, A, \tau\rangle \leq_{\Vdash} \langle C', A', \tau'\rangle$, and say $\langle C', A', \tau'\rangle$ is an *instance* of $\langle C, A, \tau\rangle$, iff there exists a substitution S such that

- $C' \Vdash SC$;
- $C' \Vdash A' \leq SA$; and
- $C' \Vdash S\tau \leq \tau'$.

Note that by (REFL) and (HYP), instantiation is reflexive. However, we cannot show that instantiation is transitive or sound in general; additional assumptions on the pre-entailment relation are required, as we describe below.

We now demonstrate that the constraints generated by the algorithm of the last section (Definition 4.2.3) give principal typings in this system.

Theorem 4.3.5 (Soundness) *$\Lambda_{\Vdash} \rhd C_M, A_M \vdash M : t_M$ for any term M and pre-entailment relation $\Vdash$.*

Proof: An easy induction on M using (HYP). □

Theorem 4.3.6 (Completeness) *If $\Lambda_{\Vdash} \rhd C, A \vdash M : \tau$, then $\langle C_M, A_M, t_M\rangle \leq_{\Vdash} \langle C, A, \tau\rangle$, for any pre-entailment relation $\Vdash$.*

Proof: We assume that the type variables X_M and Y_M are disjoint from all other type variables. By induction on M we prove the following statement:

> If $\Lambda_{\Vdash} \rhd C, A \vdash M : \tau$, then there is a substitution S such that $C \Vdash SC_M$, $S(t_M) = \tau$, $\mathbf{dom}(S) = X_M \cup Y_M$, and $s_x \in X_M \Rightarrow S(s_x) = A(x)$.

Recall that $x \in \mathbf{dom}(A_M)$ iff $s_x \in X_M$ iff $A_M(x) = s_x$. Therefore the last condition implies $x \in \mathbf{dom}(A_M) \Rightarrow C \Vdash A(x) \leq S(A_M(x))$. Thus our induction hypothesis is enough to prove the theorem.

- If $M = c^\sigma$, then $C \Vdash \sigma \leq \tau$, $A_M = \varnothing$, and $C_M = \{\sigma \leq t_M\}$.

  Let S be the substitution mapping t_M to τ; then $C \Vdash SC_M$, and $C \Vdash S(t_M) \leq \tau$ by (REFL). It is easy to see that the last two conditions of the induction hypothesis hold.

- If $M = x$, then $C \Vdash A(x) \leq \tau$, $A_M = \{x : s_x\}$, and $C_M = \{s_x \leq t_x\}$.

  Let $S = \{s_x := A(x), t_x := \tau\}$. Then $C \Vdash SC_M$, and $S(t_M) = \tau$. It is easy to see that the last two conditions of the induction hypothesis hold.

- If $M = \lambda y M'$, then for some τ_1, τ_2, we have $\Lambda_{\Vdash} \triangleright C, A\backslash y \cup \{y : \tau_1\} \vdash M' : \tau_2$ and $C \Vdash \tau_1 \to \tau_2 \leq \tau$.

  By induction there is a substitution S' such that $C \Vdash S'C_{M'}$, $S'(t_{M'}) = \tau_2$, $\mathbf{dom}(S') = X_{M'} \cup Y_{M'}$, and $s_x \in X_{M'} \Rightarrow S'(s_x) = (A\backslash y \cup \{y : \tau_1\})(x)$.

  Let $S = S' \cup \{t_M := \tau\} \cup \{s_y := \tau_1\}$. This is well defined, because $t_M \notin \mathbf{dom}(S')$, and $s_y \in \mathbf{dom}(S') \Rightarrow S'(s_y) = \tau_1$ by induction. Note that the last two conditions of the induction hypothesis are satisfied by this S.

  Now $C_M = C_{M'} \cup \{s_y \to t_{M'} \leq t_M\}$. Since $SC_{M'} = S'C_{M'}$, we already have $C \Vdash SC_{M'}$. And $S\{s_y \to t_{M'} \leq t_M\} = \{\tau_1 \to \tau_2 \leq \tau\}$, and therefore $C \Vdash SC_M$.

- If $M = M_1 M_2$, then for some τ_1 and τ_2 we have $\Lambda_{\Vdash} \triangleright C, A \vdash M_1 : \tau_1 \to \tau_2$, $\Lambda_{\Vdash} \triangleright C, A \vdash M_2 : \tau_1$, and $C \Vdash \tau_2 \leq \tau$.

  Let $\sigma_1 = \tau_1 \to \tau_2$, and $\sigma_2 = \tau_1$. By induction, for $i \in \{1,2\}$ there are substitutions S_i such that $C \Vdash S_i C_{M_i}$, $S_i(t_{M_i}) = \sigma_i$, $\mathbf{dom}(S_i) = X_{M_i} \cup Y_{M_i}$, and $s_x \in X_{M_i} \Rightarrow S_i(s_x) = A(x)$.

  Let $S = S_1 \cup S_2 \cup \{s_M := \tau_2, t_M := \tau\}$. This is well defined, because $t \in \mathbf{dom}(S_1) \cap \mathbf{dom}(S_2)$ iff $t \in X_{M_1} \cap X_{M_2}$, and the last condition of our induction hypothesis guarantees that $S_1 t = S_2 t$.

  Note that $C_M = C_{M_1} \cup C_{M_2} \cup \{t_{M_1} \leq t_{M_2} \to s_M, s_M \leq t_M\}$. Then for $i \in \{1,2\}$, $SC_{M_i} = S_i C_{M_i}$, so $C \Vdash SC_{M_i}$. And $S\{s_M \leq t_M\} = \{\tau_2 \to \tau\}$, so $C \Vdash S\{s_M \leq t_M\}$. And $S\{t_{M_1} \leq t_{M_2} \to s_M\} = \{\tau_1 \to \tau_2 \leq \tau_1 \to \tau_2\}$, so $C \Vdash S\{t_{M_1} \leq t_{M_2} \to s_M\}$ by (REFL).

  Finally, the last two conditions of the induction hypothesis are easily checked.

□

Notice that (REFL) and (TRANS) were the only properties of the pre-entailment relation needed in the proof; (HYP) was not used.

Typically, constraint-based systems divide constraint sets into those that are *consistent* in some sense, and those that are *inconsistent*.

Definition 4.3.7 (Consistency) For a given pre-entailment relation $\Vdash$, a *notion of consistency* is a predicate on constraint sets such that

- if C is consistent and $C \Vdash C'$, then C' is consistent; and
- if SC is consistent, then C is consistent, for any $\mathbf{T}$ substitution S.

Typings in a constraint-based system with a notion of consistency are restricted to contain consistent constraint sets only. Then a term M is only typable if there is a typing $\Lambda_{\Vdash} \triangleright C, A \vdash M : \tau$ for M where C is consistent. In this setting, we desire a stronger soundness theorem.

Theorem 4.3.8 (Soundness II) *For any pre-entailment relation $\Vdash$ and notion of consistency, if M is typable in $\Lambda_{\Vdash}$, then $\Lambda_{\Vdash} \rhd C_M, A_M \vdash M : t_M$, and C_M is consistent.*

Proof: By our previous soundness theorem, we only need show C_M consistent. Since M is typable, there is a typing $\Lambda_{\Vdash} \rhd C, A \vdash M : \tau$ such that C is consistent. By our completeness theorem, there is a substitution S such that $C \Vdash SC_M$. Then the two conditions on notions of consistency immediately imply that C_M is consistent. □

Notice that the predicate that is true of *all* constraint sets is a notion of consistency. Therefore our second soundness theorem implies the first. We proved the theorems in this order to emphasize that soundness has little to do with consistency.

Henceforth, we will assume that every system $\Lambda_{\Vdash}$ is implicitly associated with a notion of consistency, just as a notion of solvability is implicitly associated with every $\Lambda_{\leq}$.

4.4 Relationship of $\Lambda_{\leq}$ and $\Lambda_{\Vdash}$

When the notion of *solvability* in a type system $\Lambda(\mathbf{T}, \leq, \mathbf{C})$ and the notion of *consistency* in a type system $\Lambda(\mathbf{T}', \Vdash, \mathbf{C})$ coincide, it immediate follows that the systems type exactly the same terms. If, moreover, the pre-entailment relation corresponds with the subtyping relation, then every typing in the constraint-based system corresponds with typings in the subtyping based system.

Formally, we say that $\Lambda_{\leq}$ and $\Lambda_{\Vdash}$ *agree on constraint set C* if C is solvable in $\Lambda_{\leq}$ iff C is consistent in $\Lambda_{\Vdash}$, and that $\Vdash$ *is sound with respect to* $\leq$ if a $\mathbf{T}$ substitution S is a solution to C' whenever S is a solution to C and $C \Vdash C'$.

Theorem 4.4.1 (Equivalence of $\Lambda_{\leq}$ and $\Lambda_{\Vdash}$) *Suppose that $\Lambda_{\leq} = \Lambda(\mathbf{T}, \leq, \mathbf{C})$, and $\Lambda_{\Vdash} = \Lambda(\mathbf{T}', \Vdash, \mathbf{C})$.*

i) *If $\Lambda_{\leq}$ and $\Lambda_{\Vdash}$ agree on every C_M, then $\Lambda_{\leq}$ and $\Lambda_{\Vdash}$ type the same terms.*

ii) *Suppose $\Lambda_{\leq}$ and $\Lambda_{\Vdash}$ agree on every $\mathbf{T}'$ constraint set, and $\Vdash$ is sound with respect to $\leq$. If $\Lambda_{\Vdash} \rhd C, A \vdash M : \tau$ and C is consistent, then $\Lambda_{\leq} \rhd SA \vdash M : S\tau$ for any $\mathbf{T}$ solution S of C.*

Proof:

i) By Theorems 4.2.5, 4.3.5, and 4.3.8.

ii) By completeness, $\langle C_M, A_M, t_M \rangle \leq_{\Vdash} \langle C, A, \tau \rangle$, so there is a substitution R such that $C \Vdash RC_M$, $C \Vdash A \leq RA_M$, and $C \Vdash R(t_M) \leq \tau$.

Let S be a solution to C. Since $\Vdash$ is sound, S is a solution to RC_M. Then (SR) is a solution to C_M, and by the principal representatives theorem for $\Lambda_{\leq}$, $\Lambda_{\leq} \triangleright SRA_M \vdash M : SR(t_M)$.

Finally, since $\Vdash$ is sound and S solves C, we have $SA \leq SRA_M$ and $SR(t_M) \leq S\tau$, so by Lemma 4.2.2, $\Lambda_{\leq} \triangleright SA \vdash M : S\tau$.

□

4.5 Entailment and Principal Typings

Together, the Soundness and Completeness Theorems above do not quite give us a Principal Typings Theorem. The reason is that instantiation is not sound in general. If we assume that the pre-entailment relation satisfies the following additional properties, then we can show that instantiation is transitive and sound.

Definition 4.5.1 (Entailment) A pre-entailment relation $\Vdash$ on $\mathbf{T}$ is called an *entailment relation on* $\mathbf{T}$ if it satisfies the following rules.

$$(\textsc{subst}) \quad \frac{C \Vdash C'}{SC \Vdash SC'} \quad \text{for any } \mathbf{T} \text{ substitution } S$$

$$(\textsc{impl}) \quad \frac{C_1 \Vdash C_2, \quad C_2 \Vdash C_3}{C_1 \Vdash C_3}$$

Notice that the rule (SUBST) implies that the types of $\mathbf{T}$ are closed under $\mathbf{T}$ substitutions; previously we have not made this requirement.

Lemma 4.5.2 (Sound operations on typings) *If* $\Vdash$ *is an entailment relation, and* $\Lambda_{\Vdash} \triangleright C, A \vdash M : \tau$, *then*

- $\Lambda_{\Vdash} \triangleright SC, SA \vdash M : S\tau$ *for any* $\mathbf{T}$ *substitution* S*; and*
- $\Lambda_{\Vdash} \triangleright C', A \vdash M : \tau$ *where* $C' \Vdash C$.

Theorem 4.5.3 (Principal Typings) *If* $\Vdash$ *is an entailment relation and* M *is typable in* $\Lambda_{\Vdash}$, *then* $\Lambda_{\Vdash} \triangleright C_M, A_M \vdash M : t_M$ *and* C_M *is consistent. Furthermore,* $\Lambda_{\Vdash} \triangleright C, A \vdash M : \tau$ *and* C *is consistent iff* $\langle C_M, A_M, t_M \rangle \leq_{\Vdash} \langle C, A, \tau \rangle$.

As with soundness, an important special case of this theorem is when all constraint sets are considered consistent.

Principal typings of the form $\varnothing, A \vdash M : \sigma$ seem particularly simple. Recently, Hoang and Mitchell [24] have shown that principal typings with empty constraint sets do not always exist. Here we show that if such a principal typing exists, it corresponds to a principal typing in the system without constraint sets (assuming that typings are closed under substitution).

$$\text{(ARROW)} \quad \frac{C \Vdash \tau_1 \leq \sigma_1, \quad C \Vdash \sigma_2 \leq \tau_2}{C \Vdash \sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2}$$

$$\text{(ARROW-INVERSE-1)} \quad \frac{C \Vdash \sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2}{C \Vdash \tau_1 \leq \sigma_1}$$

$$\text{(ARROW-INVERSE-2)} \quad \frac{C \Vdash \sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2}{C \Vdash \sigma_2 \leq \tau_2}$$

Figure 4.4: Rules of $\Vdash_0$.

Theorem 4.5.4 *Suppose $\Lambda_{\leq}$ and $\Lambda_{\Vdash}$ agree on every $\mathbf{T}'$ constraint set, and $\Vdash$ is sound with respect to $\leq$. If $\varnothing, A \vdash M : \sigma$ is principal in $\Lambda_{\Vdash}$, then $\Lambda_{\leq} \triangleright A \vdash M : \sigma$, and if $\Lambda_{\leq} \triangleright B \vdash M : \tau$, then there is a substitution S such that $B \leq SA$ and $S\sigma \leq \tau$.*

Proof: Note that $\Lambda_{\leq} \triangleright A \vdash M : \sigma$ follows by Theorem 4.4.1, since the identity substitution (or indeed, *any* $\mathbf{T}$ substitution) solves $\varnothing$.

Now suppose $\Lambda_{\leq} \triangleright B \vdash M : \tau$. We must find a substitution S such that $B \leq SA$ and $S\sigma \leq \tau$.

By the principal representatives theorem, there is a solution R to C_M such that $RA_M \subseteq B$ and $R(t_M) = \tau$.

Since $\varnothing, A \vdash M : \sigma$ is principal, and $\Lambda_{\Vdash} \triangleright C_M, A_M \vdash M : t_M$, there is a substitution R' such that $C_M \Vdash R'\varnothing$, $C_M \Vdash A_M \leq R'A$, and $C_M \Vdash R'\sigma \leq t_M$.

By substitutivity, $\Lambda_{\Vdash} \triangleright \varnothing, R'A \vdash M : R'\sigma$.

Since R solves $\varnothing$, $\Lambda_{\leq} \triangleright RR'A \vdash M : RR'\sigma$.

Since $\Vdash$ is sound, $RA_M \leq RR'A$ and $RR'\sigma \leq R(t_M)$.

Therefore, let $S = RR'$; we have $B \leq RA_M \leq RR'A = SA$, and $S\sigma = RR'\sigma \leq R(t_M) = \tau$. □

4.6 An example

We now define a particular entailment relation and notion of consistency that result in a constraint-based system equivalent to the system Λ_μ of recursive types.

Let $\mathbf{T}_{\text{INT}}$ be the set of types generated by the grammar

$$\tau ::= \text{INT} \mid t \mid (\tau_1 \to \tau_2),$$

and let $\Vdash_0$ be the smallest pre-entailment relation on $\mathbf{T}_{\text{INT}}$ closed under the rules of Figure 4.4. It is easy to show that $\Vdash_0$ is an entailment relation.

Now say that a constraint set C is $\Vdash_0$-*inconsistent* if for some τ, τ' we have $C \Vdash_0 \text{INT} \leq \tau \to \tau'$ or $C \Vdash_0 \tau \to \tau' \leq \text{INT}$. We say C is $\Vdash_0$-*consistent* if it is not $\Vdash_0$-inconsistent.

The properties (SUBST) and (IMPL) of $\Vdash_0$ immediately imply that this is a notion of consistency.

Define the set $\mathbf{C}_{\text{INT}}$ by

$$\mathbf{C}_{\text{INT}} = \{0^{\text{INT}}, \mathbf{succ}^{\text{INT}\to\text{INT}}\}.$$

We write $\Lambda_{\Vdash_0}$ for the system $\Lambda(\mathbf{T}_{\text{INT}}, \Vdash_0, \mathbf{C}_{\text{INT}})$.

Theorem 4.6.1 *M is typable in Λ_μ iff M is typable in $\Lambda_{\Vdash_0}$.*

Proof: Palsberg and O'Keefe [51, Theorem 4.2.3] prove that a $\mathbf{T}_{\text{INT}}$ constraint set is $\mathbf{T}_\mu$ solvable if and only if it is $\Vdash_0$-consistent. Then the result follows by Theorem 4.4.1. □

Example 4.6.2 Let $M = \lambda x.xx$. We will denote the two occurrences of x as x_1 and x_2. The algorithm produces

- $X_M = \{s_x\}$,
- $Y_M = \{t_{x_1}, t_{x_2}, t_{x_1x_2}, s_{x_1x_2}, t_{\lambda x.x_1x_2}\}$,
- $C_M = \{s_x \leq t_{x_1},$
  $s_x \leq t_{x_2},$
  $t_{x_1} \leq t_{x_2} \to s_{x_1x_2},$
  $s_{x_1x_2} \leq t_{x_1x_2},$
  $s_x \to t_{x_1x_2} \leq t_{\lambda x.xx}\}$,
- $A_M = \varnothing$.

Note that if S is a $(\mathbf{T}_\mu, \leq_\mu)$ solution to C_M, then S is a $(\mathbf{T}_\mu, \leq_\mu)$ solution to

$$C = \{s_x \leq s_x \to t_{\lambda x.xx}\},$$

and if S is a $(\mathbf{T}_\mu, \leq_\mu)$ solution to C, then there exists a $(\mathbf{T}_\mu, \leq_\mu)$ solution S' to C_M such that $S(t_{\lambda x.xx}) = S'(t_{\lambda x.xx})$. Therefore $\Lambda_\mu \rhd A \vdash \lambda x.xx : \tau$ iff there is a $(\mathbf{T}_\mu, \leq_\mu)$ solution S to C such that $S(t_{\lambda x.xx}) = \tau$.

Chapter 5

Type inference without principal typings

In this chapter we study the type system F_η, obtained by adding to System F a rule that asserts that extensionally equivalent terms have the same typings:

$$\frac{A \vdash (\lambda x.Mx) : \sigma}{A \vdash M : \sigma} \quad x \notin \mathbf{FV}(M)$$

More terms are typable in F_η than in System F, and System F typable terms have more typings in F_η, so F_η is a proper extension of System F. Mitchell [48] showed that F_η is equivalent to (gives terms the same typings as) the extension of System F by a subtyping relation, $\leq_\forall$, based on the idea that a quantified type is a subtype of all of its instances. Until recently, even the decidability of this subtyping relation was an open question.

Here we consider the following six problems for the system F_η.

1. **Typability:** Given M, is $A \vdash M : \sigma$ derivable for some A and σ?

2. **Type inference:** Given M, produce A and σ such that $A \vdash M : \sigma$, or halt and fail if no such A and σ exist.

3. **Subtype satisfaction:** Given pairs (σ_1, τ_1), ..., (σ_n, τ_n), is there a substitution S such that $S\sigma_1 \leq_\forall S\tau_1$, ..., $S\sigma_n \leq_\forall S\tau_n$?

4. **Subtyping:** Given σ and τ, is $\sigma \leq_\forall \tau$?

5. **Type checking closed terms:** Given σ and a closed term M, is $\varnothing \vdash M : \sigma$ derivable?

6. **Type checking:** Given A, σ, and M, is $A \vdash M : \sigma$ derivable?

There are trivial reductions (1) $\Rightarrow$ (2), and (4) $\Rightarrow$ (5) $\Rightarrow$ (6). We complete the reduction (1) $\Rightarrow$ (6) by giving reductions (2) $\Rightarrow$ (3) and (3) $\Rightarrow$ (4). We also give a reduction (5) $\Rightarrow$ (4), thus (4) and (5) are equivalent.

Recently, Tiuryn and Urzyczyn [63] have shown that subtyping is undecidable, and Wells [68] has shown that subtyping and typability are undecidable. Wells's result implies that all of the problems (1–6) are undecidable.

Our reduction from type inference to subtype satisfaction is of independent interest. It shows that F_η satisfies a "principal representatives" theorem, stating that the typings of every term can be represented by a subtype satisfaction problem. The same kind of theorem was proved for the systems of Chapter 4. However, the reduction of Chapter 4 was particularly simple in that the set of type inequalities was constructed compositionally (the inequalities of the program as a whole were constructed from the inequalities of its parts).

In contrast, our reduction for F_η is not compositional, and no compositional reduction is known. Compositionality in F_η seems destroyed by the presence of type quantifiers. All of the quantifier-free systems that we have studied in previous chapters satisfy the following property (cf. Lemma 4.2.1):

- $A \vdash \lambda x M : \sigma$ iff for some τ_1, τ_2 we have $A \cup \{x : \tau_1\} \vdash M : \tau_2$ and $\tau_1 \rightarrow \tau_2 \leq \sigma$.

The type inequalities of $\lambda x M$ are then the type inequalities of M plus an inequality expressing $\tau_1 \rightarrow \tau_2 \leq \sigma$.

However, in F_η the above property does not hold. Instead, we have

- $A \vdash \lambda x M : \sigma$ iff for some τ_1, τ_2 we have $A \cup \{x : \tau_1\} \vdash M : \tau_2$ and $\mathrm{Gen}(A, \tau_1 \rightarrow \tau_2) \leq \sigma$.

That is, once a type is derived for $\lambda x M$, it can be further generalized, and this generalization must somehow be accounted for in the set of type inequalities.

We therefore abandon the compositional approach and base our reduction on the following observation: although F_η seems to lack a notion of principal typings in general, some terms do have principal typings. We show that an arbitrary term M is β-equivalent to a term of the form $(M_0 M_1 \cdots M_n)$, where each M_i has a principal typing, and moreover, that $A \vdash M : \sigma$ if and only if $A \vdash (M_0 M_1 \cdots M_n) : \sigma$. The typings of M can then be expressed in terms of a set of type inequalities (a subtype satisfaction problem) derived from the principal typings of the M_i.

Our contribution is thus to show that typings in F_η have principal "representatives," in spite of the lack of compositionality.

Of course, our reduction does not lead to a type inference algorithm for F_η, because subtype satisfaction is undecidable for F_η. To show that this technique can lead to type inference algorithms for decidable type systems, we show that it gives a reduction from the type inference problem for Λ_μ, the system of recursive types and subtyping, to a subtype satisfaction problem. Since subtype satisfaction in Λ_μ is decidable, this gives a second type inference algorithm for the system.

Organization of the chapter. The definition of F_η is given in §5.1. We show how to find principal typings for two classes of terms in §5.2 and §5.3. We give the reduction from F_η type inference to subtype satisfaction in §5.4, and in §5.5 we show that subtype satisfaction reduces to subtyping. In §5.6 we show that type checking closed terms reduces to subtyping. Finally in §5.7 we show how our technique applies to Λ_μ.

5.1 System F_η

We now define the system F_η. Our definition is based on Mitchell's subtyping relation rather than the η rule; for a proof of equivalence of the two systems, see Mitchell [48].

The terms of F_η are the terms of the lambda calculus, and the types of F_η are just the types $\mathbf{T}_\forall$ of System F, which were defined in §3.2 by the grammar:

$$\tau ::= t \mid (\tau_1 \to \tau_2) \mid (\forall t \tau)$$

Mitchell's subtyping relation, $\leq_\forall$, is based on the notion that a quantified type is a subtype of all of its instances. For example,

$$(\forall t.t) \leq_\forall \tau \quad \text{for any } \tau,$$

and

$$(\forall t.t \to t) \leq_\forall (\forall t.t \to t) \to (\forall t.t \to t).$$

The full definition of Mitchell's relation is given in Figure 5.1. An alternate presentation, by Longo et al. [42], is given in Figure 5.2. This presentation has some nice technical properties; in particular, it avoids the rule (TRANS) and is therefore more syntax-directed.

The typing rules of F_η are given in Figure 5.3. The rules (VAR), (ABS), and (APP) are exactly as in System F, while the rule (GENSUB) combines the System F rule (GEN) with the rule of subsumption:

$$(\text{GEN}) \quad \frac{A \vdash M : \sigma}{A \vdash M : (\forall t \sigma)} \quad t \notin \text{FTV}(A)$$

$$(\text{SUB}) \quad \frac{A \vdash M : \sigma}{A \vdash M : \tau} \quad \sigma \leq_\forall \tau$$

The reason we use (GENSUB) instead of (GEN) and (SUB) is that the system is more syntax-directed. By the following lemma, we may assume that in any derivation, the rule (GENSUB) is never used twice in succession.

Lemma 5.1.1 *If* $\text{Gen}(A, \sigma) \leq_\forall \tau$ *and* $\text{Gen}(A, \tau) \leq_\forall \rho$, *then* $\text{Gen}(A, \sigma) \leq_\forall \rho$.

(REFL) $\sigma \leq_\forall \sigma$

(ARROW) $$\frac{\tau_1 \leq_\forall \sigma_1, \quad \sigma_2 \leq_\forall \tau_2}{(\sigma_1 \to \sigma_2) \leq_\forall (\tau_1 \to \tau_2)}$$

($\forall$-INTRO) $\sigma \leq_\forall (\forall t \sigma) \quad t \notin \mathrm{FTV}(\sigma)$

($\forall$-ELIM) $(\forall t \sigma) \leq_\forall \{t := \tau\}\sigma$

(DIST) $(\forall t(\sigma \to \tau)) \leq_\forall (\forall t \sigma) \to (\forall t \tau)$

(CONG) $$\frac{\sigma \leq_\forall \tau}{(\forall t \sigma) \leq_\forall (\forall t \tau)}$$

(TRANS) $$\frac{\tau_1 \leq_\forall \tau_2, \quad \tau_2 \leq_\forall \tau_3}{\tau_1 \leq_\forall \tau_3}$$

Figure 5.1: Mitchell's subtyping relation.

(REFL) $\sigma \leq_\forall \sigma$

(ARROW) $$\frac{\tau_1 \leq_\forall \sigma_1, \quad \sigma_2 \leq_\forall \tau_2}{(\sigma_1 \to \sigma_2) \leq_\forall (\tau_1 \to \tau_2)}$$

($\forall$-LEFT) $$\frac{\{t := \sigma'\}\sigma \leq_\forall \tau}{(\forall t \sigma) \leq_\forall \tau}$$

($\forall$-RIGHT) $$\frac{\sigma \leq_\forall (\tau_1 \to \cdots \to \tau_n \to \tau)}{\sigma \leq_\forall (\tau_1 \to \cdots \to \tau_n \to (\forall t \tau))}$$
(where $n \geq 0$ and t is not free in $\sigma, \tau_1, \ldots, \tau_n$)

Figure 5.2: Alternate definition of Mitchell's subtyping.

$$(\text{VAR}) \quad A \vdash x : \sigma \quad \text{if } A(x) = \sigma$$

$$(\text{APP}) \quad \frac{A \vdash M_1 : \sigma \to \tau, \quad A \vdash M_2 : \sigma}{A \vdash (M_1 M_2) : \tau}$$

$$(\text{ABS}) \quad \frac{A \backslash x \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x M) : \sigma \to \tau}$$

$$(\text{GENSUB}) \quad \frac{A \vdash M : \sigma}{A \vdash M : \tau} \quad \text{if } \mathrm{Gen}(A, \sigma) \leq_\forall \tau$$

Figure 5.3: Typing rules of F_η.

Then the following standard result is easily proved by induction on typing derivations.

Lemma 5.1.2 (Characterization of typings) $\mathrm{F}_\eta \triangleright A \vdash M : \sigma$ *iff*

- $M = x$ *and* $A(x) \leq_\forall \sigma$;
- $M = \lambda x M'$, *and for some* τ_1 *and* τ_2 *we have* $\mathrm{F}_\eta \triangleright A \backslash x \cup \{x : \tau_1\} \vdash M' : \tau_2$ *and* $\mathrm{Gen}(A, \tau_1 \to \tau_2) \leq_\forall \sigma$; *or*
- $M = M_1 M_2$, *and for some* τ_1 *and* τ_2 *we have* $\mathrm{F}_\eta \triangleright A \vdash M_1 : \tau_1 \to \tau_2$, $\mathrm{F}_\eta \triangleright A \vdash M_2 : \tau_1$, *and* $\mathrm{Gen}(A, \tau_2) \leq_\forall \sigma$.

F_η satisfies the usual properties of substitutivity, weakening, and subsumption:

Lemma 5.1.3 (Sound operations on typings) *If* $\mathrm{F}_\eta \triangleright A \vdash M : \sigma$, *then*

- $\mathrm{F}_\eta \triangleright SA \vdash M : S\sigma$ *for any* $\mathbf{T}_\forall$ *substitution* S;
- $\mathrm{F}_\eta \triangleright B \vdash M : \sigma$, *where* $B \leq_\forall A$; *and*
- $\mathrm{F}_\eta \triangleright A \vdash M : \tau$, *where* $\sigma \leq_\forall \tau$.

5.2 Principal typings for a restricted set of terms

We now give an ordering on pairs that will allow us to define a notion of principal typings for a class of terms.

Definition 5.2.1 (Instance) The ordering $\leq_{\forall 0}$ on pairs is the least relation satisfying the following rules.

$$(\text{ENV}) \quad \frac{B \leq_{\forall} A}{\langle A, \sigma \rangle \leq_{\forall 0} \langle B, \sigma \rangle}$$

$$(\text{SUB}) \quad \frac{\sigma \leq_{\forall} \tau}{\langle A, \sigma \rangle \leq_{\forall 0} \langle A, \tau \rangle}$$

$$(\text{SUBST}) \quad \langle A, \sigma \rangle \leq_{\forall 0} S\langle A, \sigma \rangle \quad \text{for any substitution } S$$

$$(\text{TRANS}) \quad \frac{\langle A_1, \sigma_1 \rangle \leq_{\forall 0} \langle A_2, \sigma_2 \rangle, \quad \langle A_2, \sigma_2 \rangle \leq_{\forall 0} \langle A_3, \sigma_3 \rangle}{\langle A_1, \sigma_1 \rangle \leq_{\forall 0} \langle A_3, \sigma_3 \rangle}$$

We say that $\langle B, \tau \rangle$ is a $\leq_{\forall 0}$-*instance* of $\langle A, \sigma \rangle$ if $\langle A, \sigma \rangle \leq_{\forall 0} \langle B, \tau \rangle$.

The ordering $\leq_{\forall 0}$ can be expressed in a more familiar way.

Lemma 5.2.2 $\langle A, \sigma \rangle \leq_{\forall 0} \langle B, \tau \rangle$ *iff there exists a substitution S such that $B \leq_{\forall} SA$ and $S\sigma \leq_{\forall} \tau$.*

Proof: ($\Leftarrow$) Trivial.

($\Rightarrow$) By case analysis show that $\langle A, \sigma \rangle \leq_{\forall 0} \langle B, \tau \rangle$ must follow by some applications of rule (SUBST), followed by some applications of (ENV), followed by some applications of (SUB). Then any sequence of (SUBST) applications can be combined into one, any sequence of (ENV) applications can be combined into one, and any sequence of (SUB) applications can be combined into one. □

Note that $\leq_{\forall 0}$ is reflexive, by (ENV) and (SUB) and the fact that $\leq_{\forall}$ is reflexive on types and type environments. By Lemma 5.1.3, $\leq_{\forall 0}$-instantiation is a sound operation on typings.

Lemma 5.2.3 (Instantiation is sound) *If* $\mathrm{F}_\eta \rhd A \vdash M : \sigma$ *and* $\langle A, \sigma \rangle \leq_{\forall 0} \langle B, \tau \rangle$, *then* $\mathrm{F}_\eta \rhd B \vdash M : \tau$.

Any ordering on pairs induces a notion of principal typing as follows.

Definition 5.2.4 (Principal typings) For any term M and relation R on pairs, an *R-principal pair for M* is a pair $\langle A, \sigma \rangle$ such that $\mathrm{F}_\eta \rhd A \vdash M : \sigma$, and $\mathrm{F}_\eta \rhd B \vdash M : \tau$ if and only if $\langle A, \sigma \rangle \; R \; \langle B, \tau \rangle$. An *$R$-principal typing for M* is a typing $A \vdash M : \sigma$ such that $\langle A, \sigma \rangle$ is an R-principal pair for M.

We now show that we may define, for a particular class of terms,[1] pairs that are principal under the relation $\leq_{\forall 0}$.

[1] Our DOF terms are a subset of a class of terms identified by Mitchell [48] as having principal typings. However, we have not been able to verify the details of his proof.

Definition 5.2.5 (DOF type inference) We say x *appears as an operator in* M if $M = \cdots(xN)\cdots$ for some N. We say M is in *distinct operator form (DOF)* if it is λ-free, and all variables that appear as operators in M have exactly one occurrence in M. The function $\mathrm{PP}_{\mathrm{DOF}}$ from terms in DOF to pairs is defined as follows. Associate every term variable x with a distinct type variable t_x. Then for any term $M = (xM_1 \cdots M_n)$ in DOF, if $\mathrm{PP}_{\mathrm{DOF}}(M_i) = \langle A_i, \sigma_i \rangle$ for $i \leq n$, then

$$\mathrm{PP}_{\mathrm{DOF}}(M) = \langle A_1 \cup \cdots \cup A_n \cup \{x : \sigma_1 \to \cdots \to \sigma_n \to t_x\}, t_x \rangle.$$

For example, x is in DOF, wxx is in DOF, and $x(yz)(wz)$ is in DOF, but xx is not in DOF. And

$$\begin{aligned}
\mathrm{PP}_{\mathrm{DOF}}(x) &= \langle \{x : t_x\}, t_x \rangle, \\
\mathrm{PP}_{\mathrm{DOF}}(wxx) &= \langle \{w : t_x \to t_x \to t_w, x : t_x\}, t_w \rangle, \\
\mathrm{PP}_{\mathrm{DOF}}(x(yz)(wz)) &= \langle \{w : t_z \to t_w, x : t_y \to t_w \to t_x, y : t_z \to t_y, z : t_z\}, t_x \rangle.
\end{aligned}$$

Note that if $\mathrm{PP}_{\mathrm{DOF}}(M) = \langle A, \sigma \rangle$, then $\mathbf{dom}(A) = \mathbf{FV}(M)$.

Lemma 5.2.6 (Soundness) *If M is in DOF, then $\mathrm{PP}_{\mathrm{DOF}}(M)$ is well-defined, and if $\mathrm{PP}_{\mathrm{DOF}}(M) = \langle A, \sigma \rangle$, then $\mathrm{F}_\eta \rhd A \vdash M : \sigma$.*

Proof: To show that $\mathrm{PP}_{\mathrm{DOF}}(M)$ is well-defined it is sufficient to show that

$$A_1 \cup \cdots \cup A_n \cup \{x : \sigma_1 \to \cdots \to \sigma_n \to t_x\}$$

is well-defined. Note that only non-operators appear in more than one A_i, and a non-operator y is always assigned the type t_y. Therefore, if $y \in (\mathbf{dom}(A_i) \cap \mathbf{dom}(A_j))$, then $A_i(y) = A_j(y)$. And since x is an operator, it cannot appear in any M_i, and therefore it appears in no A_i.

To see that the judgment $A \vdash M : \sigma$ is derivable in F_η, use induction and weakening. □

We will now show that $\mathrm{PP}_{\mathrm{DOF}}$ gives a $\leq_{\forall 0}$-principal typing for any term M in DOF. The next two results develop some additional properties of typings of terms in DOF.

Lemma 5.2.7 $\mathrm{F}_\eta \rhd A \vdash M_1 M_2 : \sigma$ *iff for some τ we have* $\mathrm{F}_\eta \rhd A \vdash M_1 : \tau \to \sigma$ *and* $\mathrm{F}_\eta \rhd A \vdash M_2 : \tau$.

Proof: ($\Leftarrow$) By rule (APP).

($\Rightarrow$) By Lemma 5.1.2, for some τ, τ' we have $\mathrm{F}_\eta \rhd A \vdash M_1 : \tau \to \tau'$, $\mathrm{F}_\eta \rhd A \vdash M_2 : \tau$, and $\mathrm{Gen}(A, \tau') \leq_\forall \sigma$.

If $\mathrm{Gen}(A, \tau') \leq_\forall \sigma$, then $\mathrm{Gen}(A, \tau \to \tau') \leq_\forall \tau \to \sigma$. Then by (GENSUB), $\mathrm{F}_\eta \rhd A \vdash M_1 : \tau \to \sigma$. □

Corollary 5.2.8 *$\mathrm{F}_\eta \rhd A \vdash M_0 M_1 \cdots M_n : \sigma$ iff for some $\tau_0, \tau_1, \ldots, \tau_n$ we have $\tau_0 \leq_\forall \tau_1 \to \cdots \to \tau_n \to \sigma$ and $\mathrm{F}_\eta \rhd A \vdash M_i : \tau_i$ for $0 \leq i \leq n$.*

Corollary 5.2.9 *$\mathrm{F}_\eta \rhd A \vdash x M_1 \cdots M_n : \sigma$ iff for some $\tau_1, \ldots, \tau_n$ we have $A(x) \leq_\forall \tau_1 \to \cdots \to \tau_n \to \sigma$ and $\mathrm{F}_\eta \rhd A \vdash M_i : \tau_i$ for $1 \leq i \leq n$.*

Theorem 5.2.10 (Principal typings) *If M is in DOF, then $\mathrm{PP_{DOF}}(M)$ is a $\leq_{\forall 0}$-principal pair for M.*

Proof: Lemma 5.2.6 shows that $\mathrm{PP_{DOF}}(M)$ is an acceptable pair of M, and Lemma 5.1.3 shows that any $\leq_{\forall 0}$-instance of $\mathrm{PP_{DOF}}(M)$ is an acceptable pair of M. Therefore we only need show that if $\mathrm{F}_\eta \rhd B \vdash M : \tau$, then $\mathrm{PP_{DOF}}(M) \leq_{\forall 0} \langle B, \tau \rangle$. To do this we prove the following stronger statement:

If M is in DOF, $\mathrm{PP_{DOF}}(M) = \langle A, \sigma \rangle$, and $\mathrm{F}_\eta \rhd B \vdash M : \tau$, then there exists a substitution S such that

- $\mathbf{dom}(S) = \{t_y \mid y \in \mathbf{FV}(M)\}$;
- $S\sigma \leq_\forall \tau$; and
- for all $y \in \mathbf{FV}(M)$,

$$\begin{array}{lcll} B(y) & \leq_\forall & S(A(y)) & \text{if } y \text{ appears as an operator in } M \text{, and} \\ B(y) & = & S(A(y)) & \text{otherwise.} \end{array}$$

The proof is by induction on the structure of M. Note that M must be of the form $x M_1 \cdots M_n$, where $n \geq 0$, $M_1, \ldots, M_n$ are in DOF, and no operator occurs twice in M. Thus we have

$$\mathrm{F}_\eta \rhd B \vdash x M_1 \cdots M_n : \tau.$$

By Corollary 5.2.9, for some $\tau_1, \ldots, \tau_n$ we have

$$\begin{array}{c} \mathrm{F}_\eta \rhd B \vdash M_1 : \tau_1 \\ \vdots \\ \mathrm{F}_\eta \rhd B \vdash M_n : \tau_n \end{array}$$

and $B(x) \leq_\forall \tau_1 \to \cdots \to \tau_n \to \tau$.

Suppose $\mathrm{PP_{DOF}}(M_i) = \langle A_i, \sigma_i \rangle$ for $1 \leq i \leq n$, so that

$$\mathrm{PP_{DOF}}(M) = \langle A_1 \cup \cdots \cup A_n \cup \{x : \sigma_1 \to \cdots \to \sigma_n \to t_x\}, t_x \rangle.$$

By induction, for $1 \leq i \leq n$ we have a substitution S_i such that $\mathbf{dom}(S_i) = \{t_y \mid y \in \mathbf{FV}(M_i)\}$, $S_i\sigma_i \leq_\forall \tau_i$, and for all $y \in \mathbf{FV}(M_i)$,

$$\begin{array}{lcll} B(y) & \leq_\forall & S_i(A_i(y)) & \text{if } y \text{ appears as an operator in } M_i, \text{ and} \\ B(y) & = & S_i(A_i(y)) & \text{otherwise.} \end{array}$$

Let $\tau' = B(x)$ if $n = 0$, and $\tau' = \tau$ otherwise. Note that if y appears in more than one M_i, it is not an operator. So if $t_y \in (\mathbf{dom}(S_i) \cap \mathbf{dom}(S_j))$, we have $S_i(t_y) = B(y) = S_j(t_y)$. Therefore, the substitution

$$S = S_1 \cup \cdots \cup S_n \cup \{t_x := \tau'\}$$

is well-defined, $\mathbf{dom}(S) = \{t_y \mid y \in \mathbf{FV}(M)\}$, and $S(\sigma) = S(t_x) = \tau' \leq_\forall \tau$.

If $y \in \mathbf{FV}(M)$ and $y \neq x$, then y must appear in some M_i, so $B(y) \leq_\forall S(A(y))$ if y appears as an operator in M, and $B(y) = S(A(y))$ otherwise.

Now consider x. There are two cases. If $n = 0$, then x is not an operator and $B(x) = S(A(x))$.

If $n > 0$, then x is an operator. We have

$$\begin{array}{rcll} B(x) & \leq_\forall & \tau_1 \to \cdots \to \tau_n \to \tau & \\ & \leq_\forall & S_1\sigma_1 \to \cdots \to S_n\sigma_n \to \tau & (\text{since } S_i\sigma_i \leq_\forall \tau_i) \\ & = & S(\sigma_1 \to \cdots \to \sigma_n \to t_x) & \\ & = & S(A(x)) & \end{array}$$

as desired. □

5.3 Principal typings for abstractions

We would like to extend $\mathrm{PP_{DOF}}$ to obtain principal typings for more terms than just those in DOF. In particular, we would like to have principal typings for abstractions (λxM). Unfortunately, in this case the relation $\leq_{\forall 0}$ does not seem to be the appropriate notion of principal typing, as the following example shows.

Example 5.3.1 Consider the term $(\lambda y.xy)$. Since $\mathrm{PP_{DOF}}(xy) = \langle\{x : t_y \to t_x, y : t_y\}, t_x\rangle$, it seems natural to consider the pair $\langle\{x : t_y \to t_x\}, t_y \to t_x\rangle$ as the principal pair for $(\lambda y.xy)$. However, this pair is not principal under $\leq_{\forall 0}$, as shown by the following derivable judgment:

$$\mathrm{F}_\eta \rhd \{x : \forall u.u\} \vdash (\lambda y.xy) : \forall st.s \to t.$$

Suppose $\langle\{x : t_y \to t_x, y : t_y\}, t_x\rangle \leq_{\forall 0} \langle\{x : \forall u.u\}, \forall st.s \to t\rangle$. Then there is a substitution S such that

$$S(t_y \to t_x) \leq_\forall \forall st.s \to t.$$

However, $\forall st.s \to t \leq_\forall t_1 \to t_2$ and $\forall st.s \to t \leq_\forall t_3 \to t_2$, so we must have $t_1 \leq S(t_y)$ and $t_3 \leq S(t_y)$. This is a contradiction, since t_1 and t_3 do not have an upper bound.

Thus the pair $\langle\{x : t_y \to t_x\}, t_y \to t_x\rangle$ is not $\leq_{\forall 0}$-principal for $(\lambda y.xy)$.

In order to obtain a notion of principal typings for terms including abstractions, we extend $\leq_{\forall 0}$ as follows.

Definition 5.3.2 (Instance) The ordering $\leq_{\forall 1}$ on pairs is the least relation satisfying the rules of $\leq_{\forall 0}$ and the rule (GEN):

$$\text{(GEN)} \qquad \langle A, \sigma \rangle \leq_{\forall 1} \langle A, \forall t \sigma \rangle \quad t \notin \mathrm{FTV}(A)$$

$$\text{(ENV)} \qquad \frac{B \leq_{\forall} A}{\langle A, \sigma \rangle \leq_{\forall 1} \langle B, \sigma \rangle}$$

$$\text{(SUB)} \qquad \frac{\sigma \leq_{\forall} \tau}{\langle A, \sigma \rangle \leq_{\forall 1} \langle A, \tau \rangle}$$

$$\text{(SUBST)} \quad \langle A, \sigma \rangle \leq_{\forall 1} S\langle A, \sigma \rangle \quad \text{for any substitution } S$$

$$\text{(TRANS)} \quad \frac{\langle A_1, \sigma_1 \rangle \leq_{\forall 1} \langle A_2, \sigma_2 \rangle, \quad \langle A_2, \sigma_2 \rangle \leq_{\forall 1} \langle A_3, \sigma_3 \rangle}{\langle A_1, \sigma_1 \rangle \leq_{\forall 1} \langle A_3, \sigma_3 \rangle}$$

Note that $\leq_{\forall 1}$ is reflexive, by rules (ENV) and (SUB) and the fact that $\leq_{\forall}$ is reflexive on types and type environments. By Lemma 5.1.3 and the rule (GENSUB), $\leq_{\forall 1}$ is a sound operation on typings.

Lemma 5.3.3 (Instantiation is sound) *If* $\mathrm{F}_\eta \triangleright A \vdash M : \sigma$ *and* $\langle A, \sigma \rangle \leq_{\forall 1} \langle B, \tau \rangle$, *then* $\mathrm{F}_\eta \triangleright B \vdash M : \tau$.

The key result that lets us construct principal typings for abstractions is

Theorem 5.3.4 *Suppose* $A \vdash M : \sigma$ *is a* $\leq_{\forall 1}$*-principal typing for* M.

- *If* $x \in \mathbf{dom}(A)$, *then* $A \backslash x \vdash (\lambda x M) : \mathrm{Gen}(A \backslash x, A(x) \to \sigma)$ *is a* $\leq_{\forall 1}$*-principal typing for* $(\lambda x M)$.
- *If* $x \notin \mathbf{dom}(A)$ *and* t *is a fresh type variable, then* $A \vdash \lambda x M : \forall t.t \to \sigma$ *is a* $\leq_{\forall 1}$*-principal typing for* $(\lambda x M)$.

Example 5.3.5 We will be able to show that the pair

$$\langle \{x : t_y \to t_x\}, t_y \to t_x \rangle$$

of Example 5.3.1 is $\leq_{\forall 1}$-principal for $(\lambda y.xy)$. For example, we have

$$\begin{aligned} \langle \{x : t_y \to t_x\}, t_y \to t_x \rangle \;&\leq_{\forall 1}\; \langle \{x : \forall u.u\}, t_y \to t_x \rangle \\ &\leq_{\forall 1}\; \langle \{x : \forall u.u\}, \forall t_x t_y . t_y \to t_x \rangle \\ &=\; \langle \{x : \forall u.u\}, \forall st.s \to t \rangle. \end{aligned}$$

Recall from Example 5.3.1 that this inequality does not hold for $\leq_{\forall 0}$; so Theorem 5.3.4 does not hold when $\leq_{\forall 1}$ is replaced by $\leq_{\forall 0}$.

After first developing some technical properties of $\leq_{\forall 1}$, we prove the theorem.

Lemma 5.3.6 $\langle A, \sigma\rangle \leq_{\forall 1} \langle B, \tau\rangle$ *iff there exists a substitution* S *such that* $B \leq_{\forall} SA$ *and* $\mathrm{Gen}(B, S\sigma) \leq_{\forall} \tau$.

Proof: ($\Leftarrow$) Trivial.

($\Rightarrow$) By case analysis show that $\langle A, \sigma\rangle \leq_{\forall 1} \langle B, \tau\rangle$ must follow by some applications of rule (SUBST), followed by some applications of (ENV), followed by some applications of (GEN) and (SUB). Then any sequence of (SUBST) applications can be combined into one, any sequence of (ENV) applications can be combined into one, and by Lemma 5.1.1, any sequence of (GEN) and (SUB) applications can be combined into one. □

Once this result has been established, it is easy to prove the following two lemmas.

Lemma 5.3.7 *If* $\langle A \cup \{x : \sigma_1\}, \sigma_2\rangle \leq_{\forall 1} \langle B \cup \{x : \tau_1\}, \tau_2\rangle$, *then* $\langle A, \sigma_1 \rightarrow \sigma_2\rangle \leq_{\forall 1} \langle B, \tau_1 \rightarrow \tau_2\rangle$.

Lemma 5.3.8 *If* $x \notin \mathbf{dom}(A)$, $x \in \mathbf{dom}(B)$, *and* $\langle A, \sigma\rangle \leq_{\forall 1} \langle B, \tau\rangle$, *then* $\langle A \cup \{x : t\}, \sigma\rangle \leq_{\forall 1} \langle B, \tau\rangle$ *for any fresh type variable* t.

Proof of Theorem 5.3.4: Clearly the typings stated in Theorem 5.3.4 are derivable typings for $(\lambda x M)$ if $\mathrm{F}_\eta \rhd A \vdash M : \sigma$. We only need show that the typings are $\leq_{\forall 1}$-principal.

Suppose $\mathrm{F}_\eta \rhd B \vdash (\lambda x M) : \tau$. Then by Lemma 5.1.2, for some τ_1, τ_2 we have $\mathrm{F}_\eta \rhd B \backslash x \cup \{x : \tau_1\} \vdash M : \tau_2$ and $\mathrm{Gen}(B, \tau_1 \rightarrow \tau_2) \leq_{\forall} \tau$.

Since $A \vdash M : \sigma$ is $\leq_{\forall 1}$-principal, $\langle A, \sigma\rangle \leq_{\forall 1} \langle B \backslash x \cup \{x : \tau_1\}, \tau_2\rangle$. We consider two cases.

- If $x \in \mathbf{dom}(A)$, then

$$\begin{array}{llll}
\langle A\backslash x, \mathrm{Gen}(A\backslash x, A(x) \rightarrow \sigma)\rangle & \leq_{\forall 1} & \langle A\backslash x, A(x) \rightarrow \sigma\rangle & \text{by (SUB)} \\
& \leq_{\forall 1} & \langle B\backslash x, \tau_1 \rightarrow \tau_2\rangle & \text{by Lemma 5.3.7} \\
& \leq_{\forall 1} & \langle B, \tau_1 \rightarrow \tau_2\rangle & \text{by (ENV)} \\
& \leq_{\forall 1} & \langle B, \mathrm{Gen}(B, \tau_1 \rightarrow \tau_2)\rangle & \text{by (GEN)} \\
& \leq_{\forall 1} & \langle B, \tau\rangle. &
\end{array}$$

- If $x \notin \mathbf{dom}(A)$, then

$$\begin{array}{llll}
\langle A, \forall t.t \rightarrow \sigma\rangle & \leq_{\forall 1} & \langle A, t \rightarrow \sigma\rangle & \text{by (SUB)} \\
& \leq_{\forall 1} & \langle B\backslash x, \tau_1 \rightarrow \tau_2\rangle & \text{by Lemmas 5.3.8 and 5.3.7} \\
& \leq_{\forall 1} & \langle B, \tau\rangle & \text{as above.}
\end{array}$$

□

Definition 5.3.9 (λDOF type inference) We say M is in λ *distinct operator form (λDOF)* if M is in DOF, or $M = (\lambda x M')$ where M' is in λDOF. The type inference algorithm $\mathrm{PP}_{\lambda\mathrm{DOF}}$ is defined by the following cases.

- If M is in DOF then $\mathrm{PP}_{\lambda\mathrm{DOF}}(M) = \mathrm{PP}_{\mathrm{DOF}}(M)$.
- If $M = \lambda x M'$ where M' is in λDOF, and $\mathrm{PP}_{\lambda\mathrm{DOF}}(M') = \langle A, \sigma \rangle$, then
  - if $x \in \mathbf{dom}(A)$ then $\mathrm{PP}_{\lambda\mathrm{DOF}}(M) = \langle A \backslash x, \mathrm{Gen}(A \backslash x, A(x) \to \sigma) \rangle$; and
  - if $x \notin \mathbf{dom}(A)$ then $\mathrm{PP}_{\lambda\mathrm{DOF}}(M) = \langle A, \forall t_x . t_x \to \sigma \rangle$.

Lemma 5.3.3 and Theorems 5.2.10 and 5.3.4 immediately imply the following principal typing theorem.

Theorem 5.3.10 (Principal typings) *If M is in λDOF, then* $\mathrm{PP}_{\lambda\mathrm{DOF}}(M)$ *is a $\leq_{\forall 1}$-principal pair for M.*

Note that for any closed λDOF term M, $\mathrm{PP}_{\lambda\mathrm{DOF}}(M)$ will be of the form $\langle \varnothing, \sigma \rangle$ where σ is closed. In this case, σ is a principal *type* for M.

Theorem 5.3.11 (Principal types) *If σ has no free type variables, then $\langle \varnothing, \sigma \rangle \leq_{\forall 1} \langle B, \tau \rangle$ iff $\sigma \leq_{\forall} \tau$. Therefore, if M is a closed λDOF, then* $\mathrm{PP}_{\lambda\mathrm{DOF}}(M) = \langle \varnothing, \sigma \rangle$, *and $\mathrm{F}_\eta \rhd A \vdash M : \tau$ iff $\sigma \leq_{\forall} \tau$.*

5.4 From type inference to subtype satisfaction

The results of the last two sections show that we can find principal typings for any term in λDOF, and that the principal typings of DOF terms and closed λDOF terms are principal in a particularly nice way. In this section, we show how this can be used to reduce type inference for an arbitrary term to a subtype satisfaction problem.

Our idea is to give a transformation that starts with an arbitrary term and produces a term with exactly the same typings, but which is made up from DOF and closed λDOF pieces. Each piece will have a principal typing, and the principal typings of the pieces will be combined into a subtype satisfaction problem whose solutions give all of the possible typings for the term.

Our transformation is based on two term equivalences, β_{var} and β_{one}:

$$\begin{aligned} \beta_{\mathrm{var}} &: \quad (\lambda x M) y = M[x := y], \\ \beta_{\mathrm{one}} &: \quad (\lambda x M) N = M[x := N] \quad \text{if } x \text{ appears exactly once in } M. \end{aligned}$$

We say that M and N are $\beta_{\mathrm{var,one}}$*-equivalent*, and write $M =_{\beta_{\mathrm{var,one}}} N$, if M can be converted to N by the rules β_{var} and β_{one}.

The notion of $\beta_{\mathrm{var,one}}$-equivalence is useful because it preserves F_η typings (unlike the unrestricted β rule).

Lemma 5.4.1 ($\beta_{\text{var,one}}$ is sound) *If M and N are $\beta_{\text{var,one}}$-equivalent, then $F_\eta \triangleright A \vdash M : \sigma$ iff $F_\eta \triangleright A \vdash N : \sigma$.*

The following instance of $\beta_{\text{var,one}}$-equivalence will be particularly useful in our transformation.

Lemma 5.4.2 *If $C[\cdot]$ is a context with one hole, no variable in $\mathbf{FV}(\vec{N})$ or $\vec{w}$ is bound in $C[\cdot]$, and each w_i appears exactly once in M, then*

$$C[(\lambda\vec{w}.M)\vec{N}] =_{\beta_{\text{var,one}}} (\lambda\vec{w}.C[M])\vec{N}.$$

Two problems must be addressed in transforming terms into terms made up of DOF's and closed λDOF's: operators that appear twice, and open abstractions. Intuitively, our transformation addresses these problems as follows.

- Operators that appear twice can be eliminated by the following idea: for each operator that appears in our term,

  $$(\ldots(xN)\ldots),$$

  introduce a fresh term variable w and use instead the β_{one}-equivalent term

  $$(\lambda w(\ldots(wxN)\ldots))I,$$

  where I is the identity function $(\lambda y.y)$.

  Then the only variable operators will be w's, and since we introduce a fresh w for each operator occurrence, each w appears exactly once. The result will be an application of λDOF's.

  For example,

  - $(\lambda x.xx)$ becomes $(\lambda w\lambda x.wxx)I$, and
  - $(\lambda x.x(xx))$ becomes $(\lambda w_1\lambda w_2\lambda x.w_1x(w_2xx))II$.

- An open abstraction such as

  $$(\ldots(\lambda y.x)\ldots)$$

  can be closed by β_{var}:

  $$(\ldots((\lambda x\lambda y.x)x)\ldots).$$

  The resulting term contains an application of an abstraction. As in the last case, we seek to produce an application of λDOF's, so we use β_{one} to lift out the closed abstractions: introduce a new term variable w and use the term

  $$(\lambda w(\ldots(wx)\ldots))(\lambda x\lambda y.x).$$

  Again we have introduced a new variable operator, but it appears only once.

We now formalize this intuition.

Definition 5.4.3 (The transformation $|\cdot|$) For any term M we define a term $|M|$ by induction on the structure of M. In this definition, when we write $(\lambda\vec{w}M')\vec{N}$ we assume that $\vec{w}$ and $\vec{N}$ have the same length. Therefore when we write $M = (\lambda\vec{w}M')\vec{N}$, the $\vec{w}$, M', and $\vec{N}$ are completely determined.

- If $M = xM_1\cdots M_n$, $(\forall i \leq n)\ (\lambda\vec{w}_iM_i')\vec{N}_i = |M_i|$, and w is a fresh variable, then
$$|M| = (\lambda w\vec{w}_1\cdots\vec{w}_n.wxM_1'\cdots M_n')I\vec{N}_1\cdots\vec{N}_n.$$
- If $M = (\lambda\vec{x}.M_0)$ where $\vec{x}$ is nonempty and M_0 is not an abstraction, and $(\lambda\vec{w}_0M_0')\vec{N}_0 = |M_0|$, $\vec{y} = \mathbf{FV}(\lambda\vec{x}.M_0')$, and w is a fresh variable, then
$$|M| = (\lambda w\vec{w}_0.w\vec{y})(\lambda\vec{y}\vec{x}M_0')\vec{N}_0.$$
- If $M = M_0M_1\cdots M_n$ where M_0 is an abstraction and $n \geq 1$, w is a fresh type variable, and $(\lambda\vec{w}'.M')\vec{N}' = |(wM_0M_1\cdots M_n)|$, then
$$|M| = (\lambda w\vec{w}'.M')I\vec{N}'.$$

Lemma 5.4.4 *For any M, $|M| = (\lambda\vec{w}.M')\vec{N}$, where*

- *M' is in DOF;*
- *$\mathrm{FTV}(\lambda\vec{w}.M') = \mathrm{FTV}(M)$;*
- *$\vec{w}$ and $\vec{N}$ are of equal length;*
- *every variable of $\vec{w}$ appears exactly once in M';*
- *the variables $\vec{w}$ are the only variables that appear as operators in M';*
- *every N_i is a closed λDOF; and*
- *$M =_{\beta_{\mathrm{var,one}}} |M|$.*

Proof: A straightforward induction on the definition of $|\cdot|$, using Lemma 5.4.2. □

The transformation $|\cdot|$ lets us transform any term into an application $M_0M_1\cdots M_n$ where M_0 is a λDOF and $M_1, \ldots, M_n$ are closed λDOF's. In order to handle free variables that might appear in M_0, we introduce a second transformation, $\|\cdot\|$.

Definition 5.4.5 (The transformation $\|\cdot\|$) For any term M we define a term $\|M\|$ as follows. If $M_0M_1\cdots M_n = |M|$ where M_0 is a λDOF, and $\vec{y} = \mathbf{FV}(M_0)$, then $\|M\| = (\lambda\vec{y}.M_0)\vec{y}M_1\cdots M_n$.

An immediate consequence of this definition is

Lemma 5.4.6 *For any term M, $\|M\|$ is of the form $M_0 M_1 \cdots M_n$, where M_0 is a closed λDOF, and each M_i is a closed λDOF or distinct variable; and $M =_{\beta_{\text{var,one}}} \|M\|$.*

The last step of our reduction requires that we define subtype satisfaction. In anticipation of the result of the next section, we will define a more general class of subtype satisfaction problems than we need here.

Definition 5.4.7 (Subtype satisfaction)

- We define $\leq_\forall$-*satisfaction problems* π by the following grammar:

$$\pi ::= (\sigma \leq \tau) \mid (\exists t \pi) \mid (\pi_1 \wedge \pi_2)$$

  We write $\exists \vec{t} \pi$ for the problem $(\exists t_1 (\cdots (\exists t_n \pi) \cdots))$, where $n \geq 0$ and $\vec{t} = t_1, \ldots, t_n$.

- The notion of $\leq_\forall$-*satisfaction* for problems is defined by the following rules:

  - $S \models (\sigma \leq \tau)$ iff $S(\sigma) \leq_\forall S(\tau)$.
  - $S \models (\exists t \pi)$ iff $S_t \cup \{t := \tau\} \models \pi$ for some type τ.
  - $S \models (\pi_1 \wedge \pi_2)$ iff $S \models \pi_1$ and $S \models \pi_2$.

  We say a problem π is $\leq_\forall$-*satisfiable* if there is a $\mathbf{T}_\forall$ substitution S such that $S \models \pi$.

Finally, we show that F_η type inference reduces to subtype satisfaction.

Theorem 5.4.8 *For any term M, if $M_0 M_1 \cdots M_n = \|M\|$, $\langle A_i, \sigma_i \rangle = \mathrm{PP}_{\lambda\mathrm{DOF}}(M_i)$ for $i \leq n$, t is a fresh type variable, and $\pi = (\sigma_0 \leq \sigma_1 \to \cdots \to \sigma_n \to t)$, then $\mathrm{F}_\eta \rhd A \vdash M : \tau$ iff $St = \tau$ and $S(\bigcup_{i=0}^n A_i) \subseteq A$ for some $S \models \pi$.*

Proof: By Lemmas 5.4.6 and 5.4.1, Corollary 5.2.8, and Theorems 5.2.10 and 5.3.11. □

5.5 From subtype satisfaction to subtyping

The last section showed that the type inference problem for F_η can be reduced to solving subtype satisfaction. We now show that subtype satisfaction itself can be reduced to subtyping alone. We first develop some technical properties of Mitchell's subtyping relation.

Lemma 5.5.1 *$(\sigma_1 \to \sigma_2) \leq_\forall (\tau_1 \to \tau_2)$ iff $\tau_1 \leq_\forall \sigma_1$ and $\sigma_2 \leq_\forall \tau_2$.*

Proof: ($\Rightarrow$) Immediate by rule (ARROW).

($\Leftarrow$) By induction on derivations in the presentation of Longo et al. [42] (see Figure 5.2).

- If $(\sigma_1 \to \sigma_2) \leq_\forall (\tau_1 \to \tau_2)$ follows by (REFL), then $\tau_1 = \sigma_1$ and $\sigma_2 = \tau_2$, and the result follows by (REFL).
- If $(\sigma_1 \to \sigma_2) \leq_\forall (\tau_1 \to \tau_2)$ follows by (ARROW), the result is immediate.
- Otherwise $(\sigma_1 \to \sigma_2) \leq_\forall (\tau_1 \to \tau_2)$ by ($\forall$-RIGHT), and

$$\begin{aligned} \tau_1 \to \tau_2 &= \tau_1 \to \tau_{2,1} \to \cdots \to \tau_{2,n} \to (\forall t\tau), \\ \sigma_1 \to \sigma_2 &\leq_\forall \tau_1 \to \tau_{2,1} \to \cdots \to \tau_{2,n} \to \tau, \end{aligned}$$

  and t is not free in $\sigma_1 \to \sigma_2$, τ_1, $\tau_{2,1}, \ldots, \tau_{2,n}$.

  By induction, $\tau_1 \leq_\forall \sigma_1$ and $\sigma_2 \leq_\forall (\tau_{2,1} \to \cdots \to \tau_{2,n} \to \tau)$. Then by ($\forall$-RIGHT), $\sigma_2 \leq_\forall (\tau_{2,1} \to \cdots \to \tau_{2,n} \to (\forall t\tau)) = \tau_2$, as desired.

Note that rule ($\forall$-LEFT) cannot apply. □

Lemma 5.5.2 *If $t \notin \mathrm{FTV}(\tau)$ and τ is of the form $\tau_1 \to \cdots \to \tau_n \to s$, then $(\forall t\sigma) \leq_\forall \tau$ iff $\{t := \sigma'\}\sigma \leq_\forall \tau$ for some σ'.*

Proof: ($\Rightarrow$) Immediate by rule ($\forall$-LEFT).

($\Leftarrow$) Just note that only rule ($\forall$-LEFT) applies to $(\forall t\sigma)$ and τ of such form. □

Lemma 5.5.3

i) *π is $\leq_\forall$-satisfiable if and only if $(\exists t\pi)$ is $\leq_\forall$-satisfiable.*

ii) *If $S(\tau) = \tau_1 \to \cdots \to \tau_n \to s$, where $n \geq 0$, and t does not appear free in τ or S, then $S \models \exists t(\sigma \leq \tau)$ iff $S \models (\forall t\sigma) \leq \tau$.*

Proof: (i) is trivial, and (ii) follows immediately by Lemma 5.5.2. □

We say problems π_1 and π_2 are *equivalent* if for all substitutions S, $S \models \pi_1$ iff $S \models \pi_2$. The following lemma develops some basic equivalences.

Lemma 5.5.4

i) *$((\pi_1 \wedge \pi_2) \wedge \pi_3)$ is equivalent to $\pi_1 \wedge (\pi_2 \wedge \pi_3)$.*

ii) *$(\pi_1 \wedge \pi_2)$ is equivalent to $(\pi_2 \wedge \pi_1)$.*

iii) *If t does not appear in π, then π is equivalent to $\exists t\pi$.*

iv) *If t does not appear in $(\exists s\pi)$, then $(\exists t(\{s := t\}\pi))$ is equivalent to $(\exists s\pi)$.*

v) *If t is not free in π_2, then $(\exists t \pi_1) \wedge \pi_2$ is equivalent to $(\exists t(\pi_1 \wedge \pi_2))$.*

vi) *$(\sigma_1 \to \sigma_2) \leq (\tau_1 \to \tau_2)$ is equivalent to $(\tau_1 \leq \sigma_1) \wedge (\sigma_2 \leq \tau_2)$.*

vii) *π is equivalent to $\pi \wedge (\sigma \leq \sigma)$ for any σ.*

viii) *If t does not appear in σ or τ, then $\sigma \leq \tau$ is equivalent to $\exists t((\sigma \leq t) \wedge (t \leq \tau))$.*

Proof: Case (vi) follows by Lemma 5.5.1, and all of the other cases are trivial. □

Corollary 5.5.5 *Every problem is equivalent to a problem of the form $\exists \vec{t}(\sigma \leq \tau)$.*

The main part of our reduction is given by the following lemma.

Lemma 5.5.6 *Suppose σ and τ are types with free type variables $\vec{t}$, and π is the problem $\exists \vec{t}(\sigma \leq \tau)$. Define*

$$\begin{aligned} \sigma_\pi &= \forall \vec{t}((\sigma \to \tau) \to u) \to u, \\ \tau_\pi &= (\forall s(s \to s) \to u) \to u, \end{aligned}$$

where s and u are fresh type variables. Then

$$\pi \text{ is } \leq_\forall\text{-satisfiable iff } \sigma_\pi \leq_\forall \tau_\pi.$$

Proof: Since π is closed, it is either $\leq_\forall$-satisfiable by all substitutions, or none. Therefore, π is $\leq_\forall$-satisfiable iff $S_{\text{id}} \models \exists \vec{t}(\sigma \leq \tau)$. Then

$$\begin{array}{lll}
S_{\text{id}} \models \exists \vec{t}(\sigma \leq \tau) & & \\
\quad \text{iff} & S_{\text{id}} \models \exists \vec{t} s(\sigma \leq s \wedge s \leq \tau) & \text{by Lemma 5.5.4(viii)} \\
\quad \text{iff} & S_{\text{id}} \models \exists \vec{t} s(\sigma \to \tau \leq s \to s) & \text{by Lemma 5.5.4(vi)} \\
\quad \text{iff} & S_{\text{id}} \models \exists \vec{t} s((\sigma \to \tau \leq s \to s) \wedge (u \leq u)) & \text{by Lemma 5.5.4(vii)} \\
\quad \text{iff} & S_{\text{id}} \models \exists \vec{t} s((s \to s) \to u \leq (\sigma \to \tau) \to u) & \text{by Lemma 5.5.4(vi)} \\
\quad \text{iff} & S_{\text{id}} \models \exists \vec{t}(\forall s(s \to s) \to u \leq (\sigma \to \tau) \to u) & \text{by Lemma 5.5.3(ii)} \\
\quad \text{iff} & S_{\text{id}} \models \exists \vec{t}(((\sigma \to \tau) \to u) \to u & \text{by Lemma 5.5.4(vi,vii)} \\
& \qquad \leq (\forall s(s \to s) \to u) \to u) & \\
\quad \text{iff} & S_{\text{id}} \models \forall \vec{t}((\sigma \to \tau) \to u) \to u & \text{by Lemma 5.5.3(ii)} \\
& \qquad \leq (\forall s(s \to s) \to u) \to u & \\
\quad \text{iff} & \forall \vec{t}((\sigma \to \tau) \to u) \to u & \text{by } \leq_\forall\text{-satisfaction.} \\
& \qquad \leq_\forall (\forall s(s \to s) \to u) \to u &
\end{array}$$

□

The main result of this section is an immediate corollary.

Theorem 5.5.7 *$\leq_\forall$-satisfaction reduces to Mitchell's subtyping relation.*

Proof: By Lemma 5.5.3(i), we may assume that π has no free type variables, and by Corollary 5.5.5, we may assume π is of the form $\exists \vec{t}(\sigma \leq \tau)$. The result then follows by the previous lemma. □

5.6 From type checking closed terms to subtyping

We have already proven the results we need for this final reduction.

Theorem 5.6.1 *Type checking closed terms reduces to Mitchell's subtyping relation.*

Proof: Suppose we have a type σ and a closed term M. Let $(M_0 M_1 \cdots M_n) = \|M\|$. Since M is closed, each M_i is a closed λDOF and has a principal type τ_i. Then by Lemmas 5.1.2, 5.4.1, and 5.4.6,

$$\varnothing \vdash M : \sigma \quad \text{iff} \quad \varnothing \vdash M_0 M_1 \cdots M_n : \sigma \quad \text{iff} \quad \tau_0 \leq_\forall \tau_1 \to \cdots \tau_n \to \sigma.$$

□

5.7 Type inference for Λ_μ, revisited

We briefly sketch how the method of this chapter results in a type inference algorithm for the system Λ_μ, the Amadio-Cardelli system of recursive types and subtyping discussed in the last chapter. We will not prove the result for the general class $\Lambda_\leq$, because we will need to use properties of Λ_μ that do not hold for systems in this class in general. For example, we will need that $\leq_\mu$ satisfies the contravariant function subtyping rule

$$(\text{ARROW}) \quad \frac{\tau_1 \leq_\mu \sigma_1, \quad \sigma_2 \leq_\mu \tau_2}{(\sigma_1 \to \sigma_2) \leq_\mu (\tau_1 \to \tau_2)}$$

As we saw in the last chapter, Palsberg and O'Keefe [51] have already given a type inference algorithm for Λ_μ, and indeed, their algorithm is simpler than the one we will describe here. However, we present the algorithm as a "proof of concept," demonstrating that our method applies to type systems other than F_η.

We first adapt the notion of principal typing introduced in Chapter 2 to Λ_μ. We do not know whether this notion of principal typings holds for all terms typable in Λ_μ. However, we will be able to show that it holds for terms in λDOF.

Definition 5.7.1 (Instance) We write $\langle A, \sigma \rangle \leq_\mu \langle B, \tau \rangle$ iff for some $\mathbf{T}_\mu$ substitution S, $B \leq_\mu SA$ and $S\sigma \leq_\mu \tau$. We say that $\langle B, \tau \rangle$ is a $\leq_\mu$-*instance* of $\langle A, \sigma \rangle$ if $\langle A, \sigma \rangle \leq_\mu \langle B, \tau \rangle$.

Recall that the terms of Λ_μ include constants c^σ, where σ is a closed type (in particular, 0^{INT} and $\mathbf{succ}^{\text{INT} \to \text{INT}}$). Note that any c^σ has the principal typing $\langle \varnothing, \sigma \rangle$. Informally, we will extend our transformation $|\cdot|$ as follows: the term

$$(\ldots c^\sigma \ldots)$$

will be replaced by

$$(\lambda w (\ldots w \ldots)) c^\sigma,$$

choosing w fresh each time.

The next two lemmas follow immediately from the characterization of typings (Lemma 4.2.1).

Lemma 5.7.2 *$\Lambda_\mu \rhd A \vdash M_0 M_1 \cdots M_n : \sigma$ iff for some $\tau_0, \ldots, \tau_n$ we have $\tau_0 \leq_\mu \tau_1 \to \cdots \to \tau_n \to \sigma$ and $\Lambda_\mu \rhd A \vdash M_i : \tau_i$ for $i \leq n$.*

Lemma 5.7.3 *$\Lambda_\mu \rhd A \vdash x M_1 \cdots M_n : \sigma$ iff for some $\tau_1, \ldots, \tau_n$ we have $A(x) \leq_\mu \tau_1 \to \cdots \to \tau_n \to \sigma$ and $\Lambda_\mu \rhd A \vdash M_i : \tau_i$ for $1 \leq i \leq n$.*

Lemma 5.7.4 *If M is in DOF and $\mathrm{PP}_{\mathrm{DOF}}(M) = \langle A, \sigma \rangle$, then $\Lambda_\mu \rhd B \vdash M : \tau$ iff there is a substitution S such that $B \leq_\mu SA$ and $S\sigma \leq_\mu \tau$.*

Proof: Just as in the proof of Theorem 5.2.10, using Lemmas 4.2.1 and 5.7.3. □

Lemma 5.7.5 *If $\langle A \cup \{x : \sigma_1\}, \sigma_2 \rangle \leq_\mu \langle B \cup \{x : \tau_1\}, \tau_2 \rangle$, then $\langle A, \sigma_1 \to \sigma_2 \rangle \leq_\mu \langle B, \tau_1 \to \tau_2 \rangle$.*

Proof: There exists a substitution S such that $B \cup \{x : \tau_1\} \leq_\mu S(A \cup \{x : \sigma_1\})$ and $S\sigma_2 \leq_\mu \tau_2$.

Then $B \leq_\mu SA$ and $\tau_1 \leq_\mu S\sigma_1$, so by (ARROW), $S\sigma_1 \to S\sigma_2 \leq_\mu \tau_1 \to \tau_2$. Therefore $\langle A, \sigma_1 \to \sigma_2 \rangle \leq_\mu \langle B, \tau_1 \to \tau_2 \rangle$, as desired. □

Lemma 5.7.6 *If $x \notin \mathbf{dom}(A)$, $x \in \mathbf{dom}(B)$, and $\langle A, \sigma \rangle \leq_\mu \langle B, \tau \rangle$, then $\langle A \cup \{x : t\}, \sigma \rangle \leq_\mu \langle B, \tau \rangle$ for any fresh type variable t.*

Proof: There exists a substitution S such that $B \leq_\mu SA$ and $S\sigma \leq_\mu \tau$. We may assume that $\mathbf{dom}(S) \subseteq \mathrm{FTV}(A) \cup \mathrm{FTV}(\sigma)$; else use its restriction. Then $S' = S \cup \{t := B(x)\}$ is well-defined, $B \leq_\mu S'(A \cup \{x : t\})$, and $S'\sigma = S\sigma \leq_\mu \tau$. Therefore $\langle A \cup \{x : t\}, \sigma \rangle \leq_\mu \langle B, \tau \rangle$, as desired. □

In contrast to F_η, we will not need a separate notion of principal typing for abstractions.

Lemma 5.7.7 *Suppose $\Lambda_\mu \rhd A \vdash M : \sigma$ is a $\leq_\mu$-principal typing for M.*

- *If $x \in \mathbf{dom}(A)$, then $\Lambda_\mu \rhd A \backslash x \vdash (\lambda x M) : A(x) \to \sigma$ is a $\leq_\mu$-principal typing for $(\lambda x M)$.*
- *If $x \notin \mathbf{dom}(A)$ and t is a fresh type variable, then $\Lambda_\mu \rhd A \vdash \lambda x M : t \to \sigma$ is a $\leq_\mu$-principal typing for $(\lambda x M)$.*

Proof: Just as for Theorem 5.3.4, using Lemmas 5.7.5 and 5.7.6. □

Therefore, it would be possible to use a simpler transformation for Λ_μ than the one we used for F_η. However, the F_η transformation suffices, so for simplicity we will not introduce another transformation.

Lemma 5.7.8 *If M and N are $\beta_{\mathrm{var,one}}$-equivalent, then $\Lambda_\mu \rhd A \vdash M : \sigma$ iff $\Lambda_\mu \rhd A \vdash N : \sigma$.*

We extend the definition of $|\cdot|$, and therefore $\|\cdot\|$, to handle terms with constants as follows:

- If $M = c^\sigma$ and w is a fresh variable, then $|M| = (\lambda w.w)c^\sigma$.

Lemma 5.7.9 *For any term M, $\|M\|$ is of the form $M_0 M_1 \cdots M_n$, where M_0 is a closed λDOF and $M_1, \ldots, M_n$ are either closed λDOF's, distinct variables, or constants; and $M =_{\beta_{\mathrm{var,one}}} \|M\|$.*

Definition 5.7.10 For M a λDOF, DOF, or constant, we define $\mathrm{PP}_{\Lambda_\mu}(M)$ as follows.

- If $M = c^\sigma$ then $\mathrm{PP}_{\Lambda_\mu}(M) = \langle \varnothing, \sigma \rangle$.
- If M is in DOF, then $\mathrm{PP}_{\Lambda_\mu}(M) = \mathit{PPdof}(M)$.
- If $M = \lambda x M'$ and $\langle A, \sigma \rangle = \mathrm{PP}_{\Lambda_\mu}(M')$, then
  - if $x \in \mathbf{dom}(A)$ then $\langle A \backslash x, A(x) \to \sigma \rangle \in \mathrm{PP}_{\Lambda_\mu}(M)$; and
  - if $x \notin \mathbf{dom}(A)$ then $\langle A, t \to \sigma \rangle \in \mathrm{PP}_{\Lambda_\mu}(M)$ where t is a fresh type variable.

By Lemmas 5.7.4 and 5.7.7, $\mathrm{PP}_{\Lambda_\mu}$ computes $\leq_\mu$-principal typings.

Theorem 5.7.11 *For any term M, if $M_0 M_1 \cdots M_n = \|M\|$, $\langle A_i, \sigma_i \rangle = \mathrm{PP}_{\Lambda_\mu}(M_i)$ for $i \leq n$, t is a fresh type variable, then $\mathrm{F}_\eta \rhd A \vdash M : \tau$ iff $St = \tau$ and $S(\bigcup_{i=0}^{n} A_i) \subseteq A$ for some S such that $S\sigma_0 \leq S(\sigma_1 \to \cdots \to \sigma_n \to t)$.*

Bibliography

[1] Shail Aditya and Rishiyur Nikhil. Incremental polymorphism. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 379–405. Springer-Verlag, 1991.

[2] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41. ACM Press, June 1993.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 104–118, 1991.

[4] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[5] Franz Baader and Jörg Siekmann. Unification theory. In Dov M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 41–125. Claredon Press, 1994.

[6] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics (Revised Edition)*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, 1984.

[7] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4):931–940, December 1983.

[8] Felice Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, May 1991.

[9] Dominique Clement, Joelle 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, 1986.

[10] M. Coppo and M. Dezani. A new type-assignment for lambda terms. *Archiv für Math. Logik*, 19:139–156, 1978.

[11] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[12] H.B. Curry and R. Feys. *Combinatory Logic I.* North-Holland, 1958.

[13] Luis Manuel Martins Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1984.

[14] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and it application to OOP. In *Proc. Mathematical Foundations of Programming Semantics*, 1995. To appear.

[15] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT '89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, Volume 2*, volume 352 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, March 1989.

[16] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 1990.

[17] J.H. Gallier and W. Snyder. Designing unification procedures using transformations: A survey. In Y.N. Moschovakis, editor, *Logic from Computer Science*, volume 21 of *Mathematical Sciences Research Institute Publications*, pages 153–215. Springer-Verlag, 1992.

[18] Jean-Yves Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, 2nd *Scandinavian Logic Symp.*, pages 63–92. North-Holland Publishing Co., 1971.

[19] Shail Aditya Gupta. An incremental type inference system for the programming language Id. Master's thesis, Massachusetts Institute of Technology, November 1990. Available as MIT/LCS Technical Report TR–488.

[20] Robert Harper and John C. Mitchell. The essence of ML. In POPL 15 [53], pages 28–46.

[21] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[22] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[23] Monika Rauch Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. To appear in Proc. 36th Annual Symp. on Foundations of Computer Science, 1995.

[24] My Hoang and John Mitchell. Lower bounds on type inference with subtypes. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 176–185, 1995.

[25] IEEE Computer Society Press. *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, 1995.

[26] Mark P. Jones. *Qualified Types: Theory and Practice.* Cambridge University Press, November 1994.

[27] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, chapter 8, pages 257–321. MIT Press, 1991.

[28] Stefan Kaes. Typing in the presence of overloading, subtyping, and recursive types. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 193–204, 1992.

[29] A.J. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the polymorphic λ-calculus (extended summary). In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 2–11. IEEE Computer Society Press, 1990.

[30] A.J. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order λ-calculus. *Information and Computation*, 98(2):228–257, June 1992.

[31] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. A proper extension of ML with an effective type-assignment. In POPL 15 [53], pages 58–69.

[32] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[33] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2), March 1994.

[34] A.J. Kfoury and J.B. Wells. A direct algorithm for type inference in the rank 2 fragment of the second-order lambda-calculus. Technical Report 93–017, Boston University, November 1993.

[35] A.J. Kfoury and J.B. Wells. A direct algorithm for type inference in the rank 2 fragment of the second-order lambda-calculus. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 196–207, 1994.

[36] A.J. Kfoury and J.B. Wells. New notions of reduction and non-semantic proofs of strong β-normalization in typed λ-calculi. In LICS 10 [25], pages 311–321.

[37] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Reviews*, 21(1):93–124, March 1989.

[38] Dexter Kozen, Jens Palsberg, and Michael I. Schartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.

[39] Dexter Kozen, Jens Palsberg, and Michael I. Schartzbach. Efficient recursive subtyping. *Math. Struct. in Comp. Science*, 1995. To appear.

[40] J.-L. Lassez, M.J. Maher, and K. Marriot. Unification revisited. In Jack Minker, editor, *Deductive Databases and Logic Programming*, chapter 15, pages 587–625. Morgan Kaufman, 1988.

[41] Daniel Leivant. Polymorphic type inference. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 88–98, 1983.

[42] Giuseppe Longo, Kathleen Milsted, and Sergei Soloviev. A logic of subtyping. In LICS 10 [25], pages 300–310.

[43] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 382–401, 1990.

[44] Marchetti-Spaccamela, Nanni, and Rohnert. On-line graph algorithms for incremental compilation. In *Graph-Theoretic Concepts in Computer Science, International Workshop WG*, 1993.

[45] Nancy McCracken. The typechecking of programs with implicit type structure. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 301–315, June 1984.

[46] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[47] John Mitchell. Coercion and type inference. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, 1984.

[48] John Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, February/March 1988. Special Issue on Semantics of Data Types.

[49] John Mitchell. Type inference with simple subtypes. *J. Functional Programming*, 1(3):245–285, July 1991.

[50] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming, Toulouse*, volume 167 of *Lecture Notes in Computer Science*, pages 217–239. Springer-Verlag, 1984.

[51] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, 1995.

[52] Gordon D. Plotkin. A note on inductive generalization. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, 1970.

[53] *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988.

[54] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, pages 135–151, 1970.

[55] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium: Proceedings, Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[56] Patrick Sallé. Une extension de la theorie des types en λ-calcul. In G. Ausiello and C. Böhm, editors, *Automata, Languages and Programming: Fifth Colloquium*, volume 62 of *Lecture Notes in Computer Science*, pages 398–410. Springer-Verlag, July 1978.

[57] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 439–450, 1993.

[58] Jörg H. Siekmann. Unification theory. *J. Symbolic Computation*, 7(3&4):207–274, March/April 1989.

[59] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[60] Wayne Snyder. *A Proof Theory for General Unification*, volume 11 of *Progress in Computer Science and Applied Logic*. Birkhäuser, 1991.

[61] Satish R. Thatte. Type inference with partial types. *Theoretical Computer Science*, 124:127–148, 1994.

[62] Jerzy Tiuryn. Subtype inequalities. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 308–315. IEEE Computer Society Press, 1992.

[63] Jerzy Tiuryn and Paweł Urzyczyn. The subtyping problem for second-order types is undecidable. Draft manuscript, 19 November 1995.

[64] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, May 1988.

[65] Steffen van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems.* PhD thesis, Mathematisch Centrum, Amsterdam, February 1993.

[66] Mitchell Wand. Finding the source of type errors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–43, 1986.

[67] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987.

[68] J.B. Wells. Typability is undecidable for F+eta. Unpublished draft, December 1995.

[69] J.B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. Technical Report 93–011, Boston University, 1993.

[70] Hirofumi Yokouchi. Embedding a second-order type system into an intersection type system. *Information and Computation*, 117(2):206–220, March 1995.

Index

Types

- $\mathbf{R}(n)$, 46
- $\mathbf{S}(n)$, 46
- $\mathbf{S}'(n)$, 46
- $\mathbf{T}_\forall$, 46, 97
- $\mathbf{T}_{\forall 2}$, 18
- $\mathbf{T}_\mu$, 79
- $\mathbf{T}_0$, 18
- $\mathbf{T}_1$, 18
- $\mathbf{T}_2$, 18
- $\mathbf{Tv}$, 16
- types of ML, 48
- type schemes of ML, 48
- types of System F, 46, 97

Type systems

- $\Lambda(\mathbf{T}, \leq, \mathbf{C})$, 81
- $\Lambda(\mathbf{T}, \Vdash, \mathbf{C})$, 86
- $\Lambda_\leq$, 81
- $\Lambda_\Vdash$, 86
- Λ_μ, 80
- Λ_2, 47
- Λ_2^{s}, 47
- Λ_2^{R}, 74
- F_η, 97
- $\mathbf{I}_2$, 63
- $\mathbf{I}_2^{\mathrm{s}}$, 45
- $\mathbf{I}_2^{\mathrm{vb}}$, 62
- ML, 48
- ML^{R}, 74
- $\mathbf{P}_2$, 20
- $\mathbf{P}_2^{\mathrm{R}}$, 29

Relations and operators

- $\sigma \leq_\mu \tau$, 79
- $\sigma \leq_1 \tau$, 19
- $\sigma \leq_2 \tau$, 19
- $\sigma \leq_{2,1} \tau$, 57
- $\sigma \leq_{\forall 2} \tau$, 19
- $\sigma \leq_{\forall 2,1} \tau$, 19
- $\sigma \leq_\forall \tau$, 97
- $\sigma \succ \tau$, 46
- $\sigma \preceq_1 \tau$, 48
- $\sigma \preceq_2 \tau$, 49
- $\langle A, \sigma \rangle \leq_{\forall 0} \langle A', \sigma' \rangle$, 100
- $\langle A, \sigma \rangle \leq_{\forall 1} \langle A', \sigma' \rangle$, 104
- $\langle A, \sigma \rangle \leq_\mu \langle A', \sigma' \rangle$, 112
- $\langle C, A, \tau \rangle \leq_\Vdash \langle C', A', \tau' \rangle$, 88
- $A \leq A'$, 16
- $S \leq S'$, 22
- $S \leq_V S'$, 22
- $S = S'$, 22
- $S =_V S'$, 22
- $A + A'$, 25
- $A \cup A'$, 16
- $S \cup S'$, 17
- $M \to_\gamma M'$, 51
- $\langle\!\langle M \rangle\!\rangle$, 29
- $|M|$, 108, 114
- $\|M\|$, 108, 114
- β_{one}, 106
- β_{var}, 106
- $\beta_{\mathrm{var,one}}$, 106
- γ-nf(M), 52
- ϵ, 51
- Σ_μ, 79
- $A \backslash x$, 16
- $A \backslash X$, 16
- A_M, 83

act(M), 51
AP(M), 17
appl(M), 51
C_M, 83
dom(A), 16
dom(S), 17
dom(τ), 79
FTV, 16
Gen(A, σ), 16
KW(M), 61
LCG(M), 61
lcg(σ), 53
MGS(π), 22, 57
MGS$(\pi)[W]$, 22, 57
ml(M), 53
$\mathrm{PP}_{\Lambda_\mu}(M)$, 114
$\mathrm{PP}_{\mathbf{I}_2^{\mathrm{s}}}(M)$, 57
$\mathrm{PP}_{\mathbf{P}_2}(M)$, 25
$\mathrm{PP}_{\mathbf{P}_2^{\mathrm{R}}}(M)$, 29
$\mathrm{PP}_{\mathrm{DOF}}(M)$, 101
$\mathrm{PP}_{\lambda\mathrm{DOF}}(M)$, 106
rng(S), 17
Solutions(π), 22, 57
X_M, 83
Y_M, 83

General concepts

pair, 17
  acceptable, 17
  disjoint, 17
  principal, 56, 64
substitution, 17
  disjoint, 17
  idempotent, 22
substitutivity, 46
subsumption, 20, 63
term, 16
  of Λ_μ, 79
  of $\Lambda_\leq$, 81
  of $\Lambda_\Vdash$, 86
  of ML, 48
typable, 17
type environments, 16
weakening, 46