Programming with Intersection Types and Bounded Polymorphism

Benjamin C. Pierce

December 20, 1991
CMU-CS-91-205

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This draft is identical to the printed thesis, except that it has been typeset using TEX's Computer Modern fonts for greater portability, whereas the printed version uses Palatino. Citations to this draft should be by section number, not page number.

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597; in part by the Office of Naval Research under Contract N00013-84-K-0415; in part by the National Science Foundation under Contract CCR-8922109; and in part by Siemens.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: Lambda calculus and related systems, language theory, programming, type structure, data types and structures, polymorphism, subtyping, bounded quantification, intersection types.

Abstract

Intersection types and bounded quantification are complementary mechanisms for extending the expressive power of statically typed programming languages. They begin with a common framework: a simple, typed language with higher-order functions and a notion of subtyping. Intersection types extend this framework by giving every pair of types σ and τ a greatest lower bound, $\sigma \wedge \tau$, corresponding intuitively to the intersection of the sets of values described by σ and τ. Bounded quantification extends the basic framework along a different axis by adding polymorphic functions that operate uniformly on all the subtypes of a given type. This thesis unifies and extends prior work on intersection types and bounded quantification, previously studied only in isolation, by investigating theoretical and practical aspects of a typed λ-calculus incorporating both.

The practical utility of this calculus, called $F_\wedge$, is established by examples showing, for instance, that it allows a rich form of "coherent overloading" and supports an analog of abstract interpretation during typechecking; for example, the addition function is given a type showing that it maps pairs of positive inputs to a positive result, pairs of zero inputs to a zero result, etc. More familiar programming examples are presented in terms of an extension of Forsythe (an Algol-like language with intersection types), demonstrating how parametric polymorphism can be used to simplify and generalize Forsythe's design. We discuss the novel programming and debugging styles that arise in $F_\wedge$.

We prove the correctness of a simple semi-decision procedure for the subtype relation and the partial correctness of an algorithm for synthesizing minimal types of $F_\wedge$ terms. Our main tool in this analysis is a notion of "canonical types," which allow proofs to be factored so that intersections are handled separately from the other type constructors.

A pair of negative results illustrates some subtle complexities of $F_\wedge$. First, the subtype relation of $F_\wedge$ is shown to be undecidable; in fact, even the subtype relation of pure second-order bounded quantification is undecidable, a surprising result in its own right. Second, the failure of an important technical property of the subtype relation — the existence of least upper bounds — indicates that typed semantic models of $F_\wedge$ will be more difficult to construct and analyze than the known typed models of intersection types. We propose, for future study, some simpler fragments of $F_\wedge$ that share most of its essential features, while recovering decidability and least upper bounds.

We study the semantics of $F_\wedge$ from several points of view. An untyped model based on partial equivalence relations demonstrates the consistency of the typing rules and provides a simple interpretation for programs, where "σ is a subtype of τ" is read as "σ is a subset of τ." More refined models can be obtained using a translation from $F_\wedge$ into the pure polymorphic λ-calculus; in these models, "σ is a subtype of τ" is interpreted by an explicit coercion function from σ to τ. The nonexistence of least upper bounds shows up here in the failure of known techniques for proving the coherence of the translation semantics. Finally, an equational theory of equivalences between $F_\wedge$ terms is presented and its soundness for both styles of model is verified.

To Angela, Bern, Dave, and Susan

Contents

1 Introduction **7**
- 1.1 Motivation . . . 7
  - 1.1.1 Programming languages . . . 7
  - 1.1.2 Types . . . 8
  - 1.1.3 Typed λ-calculi . . . 9
  - 1.1.4 Subtyping . . . 9
- 1.2 Claims . . . 11
- 1.3 Outline of Results . . . 12

2 Background **14**
- 2.1 Notational Preliminaries . . . 14
- 2.2 Simply Typed λ-Calculus with Subtyping . . . 16
- 2.3 Intersection Types . . . 19
- 2.4 Semantic Frameworks for Intersection Types . . . 22
  - 2.4.1 Untyped Semantics . . . 22
  - 2.4.2 Typed Semantics . . . 23
  - 2.4.3 Operational semantics . . . 27
  - 2.4.4 Discussion . . . 28
- 2.5 Expressiveness of the Intersection Type Discipline . . . 29
- 2.6 Bounded Polymorphism . . . 30

3 The $F_\wedge$ Calculus **35**
- 3.1 Explicit Alternation: The *for* Construct . . . 36
- 3.2 Syntax, Subtyping, and Typing . . . 37
- 3.3 Linear Notation for Derivations . . . 38
- 3.4 Discussion . . . 40
  - 3.4.1 *Top* vs. $\top$ . . . 40
  - 3.4.2 Encoding Primitive Subtyping . . . 41
- 3.5 Alternative Formulations . . . 42
  - 3.5.1 Unbounded Quantifiers . . . 42
  - 3.5.2 Additional Subtyping Rules . . . 42
  - 3.5.3 Bounded Existential Types . . . 43

4 Typechecking **44**
- 4.1 Basic Properties . . . 44
- 4.2 Subtyping . . . 48
  - 4.2.1 Canonical Types . . . 48

4.2.2 Canonical Subtyping . . . 51
4.2.3 Weakening and Narrowing . . . 52
4.2.4 Subtyping Derivation Normalization Rules . . . 55
4.2.5 Termination of the Normalization Rules . . . 58
4.2.6 Shapes of Normal-Form Subtyping Derivations . . . 60
4.2.7 Equivalence of Ordinary and Canonical Subtyping . . . 61
4.2.8 Subtyping Algorithm . . . 64
4.3 Typechecking . . . 67
4.3.1 Finite Bases for Applications . . . 68
4.3.2 Type Synthesis . . . 72
4.3.3 Conservativity . . . 75

5 Semantics **77**
5.1 Untyped Semantics . . . 77
5.1.1 Total Combinatory Algebras . . . 78
5.1.2 Partial Equivalence Relations . . . 79
5.1.3 PER Interpretation of $F_\wedge$ . . . 80
5.2 Nonexistence of Least Upper Bounds . . . 84
5.3 Translation Semantics . . . 89
5.3.1 Target Calculus . . . 89
5.3.2 Ordinary Derivations . . . 91
5.3.3 Algorithmic Derivations . . . 93
5.4 Coherence (Preliminary Results) . . . 94
5.5 Equational Theory . . . 94
5.5.1 Definitions . . . 95
5.5.2 Basic Properties . . . 96
5.5.3 Soundness for the Untyped Semantics . . . 100
5.5.4 Soundness for the Translation Semantics . . . 103

6 Undecidability of Subtyping **109**
6.1 A Flawed Decidability Argument for $F_\leq$ . . . 110
6.2 Nontermination of the $F_\leq$ Subtyping Algorithm . . . 111
6.3 A Deterministic Fragment of $F_\leq$ . . . 112
6.4 Eager Substitution . . . 114
6.5 Rowing Machines . . . 117
6.6 Encoding Rowing Machines as Subtyping Problems . . . 119
6.7 Two-counter Machines . . . 122
6.8 Encoding Two-counter Machines as Rowing Machines . . . 123
6.9 Undecidability of $F_\leq$ Typechecking . . . 126
6.10 Undecidability of $F_\wedge$ . . . 127
6.11 Related Systems . . . 128
6.12 Discussion . . . 128

7 Examples **129**
7.1 Conventions . . . 129
7.2 Examples from the Forsythe Report . . . 131
7.3 Procedures With Optional Arguments . . . 135
7.4 User-defined Coherent Overloading . . . 136

7.5 Modeling Abstract Interpretation . . . 137
7.5.1 Booleans . . . 137
7.5.2 Lists . . . 139
7.5.3 Natural Numbers . . . 141
7.6 Modelling Strictness Analysis . . . 142
7.7 Refining Pure Encodings of Inductive Types . . . 144
7.7.1 Church Arithmetic . . . 144
7.7.2 Booleans . . . 148
7.8 Observations on Programming with $F_\wedge$ . . . 149
7.9 An Experiment with a Simpler Formulation of $F_\wedge$ . . . 151

8 Evaluation and Future Work **154**
8.1 Alternative Formulations . . . 155
8.2 Foundations . . . 156
8.2.1 Semantics . . . 157
8.2.2 Coherence . . . 157
8.3 Extensions . . . 159
8.3.1 Records . . . 159
8.3.2 Recursive Types . . . 160
8.3.3 Union Types . . . 160
8.3.4 Type Reconstruction . . . 160
8.4 Implementation . . . 160

A Summary of Major Definitions **162**
A.1 $F_\wedge$ . . . 162
A.1.1 Subtyping . . . 162
A.1.2 Typing . . . 162
A.1.3 Syntax-Directed Subtyping . . . 163
A.1.4 Type Synthesis . . . 163
A.2 $F_\leq$ . . . 164
A.2.1 Subtyping . . . 164
A.2.2 Typing . . . 164
A.3 $\lambda_\wedge$ . . . 164
A.3.1 Subtyping . . . 164
A.3.2 Typing . . . 165

B Glossary of Notation **166**

Bibliography **167**

Acknowledgements

A dissertation is supposed to be the ultimate individual project. But if it were written in a vacuum, the result would not be remotely worth the trouble. If this document is the product of my efforts these two years, then it is just the harvest of the kindness and generosity of the countless people who have helped, taught, supported, and loved me for these past two years and the previous twenty six.

I'm especially grateful to my parents, Alexandra and Roger Pierce, and my grandparents, Louise and Ralph Young, for introducing me to the life of the mind; to my early mentors at Stanford, Brian Reid, Terry Winograd, and Forest Baskett, for opening the doors to computer science; and to my colleagues and teachers at CMU for creating the friendly and stimulating environment I've enjoyed here.

While writing my thesis, I have had the good fortune to work with four of the finest computer scientists I know. The vision, creativity, and energy of Luca Cardelli, Nico Habermann, Bob Harper, and John Reynolds will be an example to me for the rest of my life.

I've also learned a great deal from Peter Lee, Frank Pfenning, David Garlan, Nevin Heintze, Spiro Michaylov, QingMing Ma, Tim Freeman, and the rest of the Gandalf and Ergo groups.

Sharon Burks helped chart a safe course through more perils than I care to think about.

During the final months of research and writing, Juliet Langman kept me more-or-less sane and made this an unexpectedly happy time. She and all my other dear friends, especially Penny Anderson, Violetta Cavalli-Sforza, Susan Finger, Nevin Heintze, Angela Hickman, Bernadette Kowalski, Kim McCall, Spiro Michaylov, Jessica Pierce, and Dave Plaut, have meant more to me than I could begin to say.

Pittsburgh, PA
December 20, 1991

Chapter 1

Introduction

This thesis describes an experiment in the foundations of programming languages. Our aim is to study the interaction of two powerful linguistic primitives, bounded quantification and intersection types, both of which have attracted significant attention recently in the research community. The result of the study is a new typed λ-calculus, an elegant "core programming language" combining the power of second-order polymorphism with the fine-grained expressiveness of intersection types.

1.1 Motivation

The activity of programming — the formalization of ideas and their expression in forms suitable for interpretation by computers — is characterized by a certain cognitive mode, a state of mind simultaneously creative and analytical. Like writing, it begins with a rough idea, an intuition, picture, sketch, or analogy, which is gradually refined and clarified, details added, and internal consistency established. In both writing and programming, formalization is by far the most difficult part, since it often involves redefining the original idea over and over as its ramifications are better understood. Then (or concurrently) the detailed, formalized mental picture is written out in some concrete, external form.

1.1.1 Programming languages

In programming, the original mental picture is of some computational behavior: a task to be performed or a value to be calculated by some machine, real or imaginary, operating according to a well-understood set of formal rules. A *programming language* is a concrete, formal notation for effectively describing such computational behaviors.

Natural languages like English are clearly not programming languages according to this definition: they are neither formal, since a great deal of "speaking English" involves the shared social and physical context of speakers and listeners, nor are they, in our sense, effective. Specification languages like Larch and Z, though they are perfectly formal, are also not programming languages because the connection between descriptions and behaviors is not effective: like the rules of classical harmony in music composition, they constrain a class of behaviors, but, in themselves, do not provide any method for constructing an element of this class.

On the other hand, our definition does admit all the forms of conventional programming languages: machine and assembly codes; imperative languages (Fortran, Algol, Ada); functional languages (Haskell, Miranda); mixed functional and imperative languages (Lisp, Scheme, ML);

object-oriented languages (Simula, Smalltalk); and macro languages (TeX, Lotus 123). Logic programming languages (Prolog) are also admitted, since their declarative style of presentation is completely formal and the connection between description and behavior is effective; similar arguments can be made for process-control languages; simulation languages; languages for statistical calculations; and the many declarative languages used in the AI community. Also included are languages whose notion of behavior is more abstract (Turing machines, λ-calculi, recursive function theory, term rewriting systems, cellular automata) and languages that include behavioral primitives like nondeterminacy, randomness, concurrency, and oracles for undecidable problems.

Depending on the discussion at hand, it may be useful to loosen the requirements of effectiveness or precision so that notations like pseudo-code or the fragments of English used in recipes and tax forms are also admitted. Likewise, the very simple languages used to describe regular expressions, typesetting styles in display-oriented editors, and the behavior of microwave ovens can either be taken as programming languages or relegated to some broader category.

1.1.2 Types

The appropriateness of a programming language for a given task can significantly affect both aspects — formalization and expression — of programming. The process of translation from an internal, mental description of a desired behavior into a concrete program implementing this behavior is much smoother if the conceptual primitives of the mental picture are reflected by analogous constructs in the programming language. Ideally, the concrete language provides such a clear and useful set of conceptual structures that this translation step is almost trivial. Conversely, and more importantly, the formal concepts embodied in a programming language tend to become a part of the internal conceptual language that programmers use to develop and refine their ideas.

The notion of *type* plays a crucial role in facilitating this translation of ideas from abstract conceptual structures to concrete realizations and vice versa.

Essentially all programming languages have some notion — at least informally — of a collection of conceptual categories, or *types*, appropriate to the intended domain of discourse. Machine languages deal with registers, words, memory pages, and device interrupts. Typesetting languages manipulate characters, words, boxes and glue, paragraphs, and pages. Functional languages use numbers, records, lists, and higher-order functions. The more coherent, clean, and simple the system of types, the better. Ideally, the type system becomes an organizing principle for the entire language, guiding its design, application, and even implementation.

Statically typed programming languages take the point of view that the type system should be made simple enough and given a sufficiently rigorous foundation that it becomes possible to detect certain kinds of category errors in programs automatically. Though this requirement often restricts the expressiveness of the type system, it has a number of practical benefits. The most obvious is that a compiler for a statically typed language can, in principle, guarantee the absence of these type errors. Since many of the errors made by programmers are of exactly this sort, compilers with static typecheckers are valuable tools for pinpointing mistakes early in the process of developing a program. More subtly, it has often been observed that for certain languages and programming tasks there is such a close correspondence between the type structures provided (and checked) by the language and the appropriate conceptual structures for imagining the behavior of the program that once all type errors have been removed, the program is usually completely correct — not only with regard to simple category errors, but even with regard to properties of its behavior that lie completely outside the apparent purview of the type system. In a sense, each type contains very few programs, one of which is the intended one. Other well-known benefits of static type

systems include the fact that they can support the generation of smaller and faster object code by giving compilers better information about possible optimizations; that they sometimes suggest an appropriate architecture for the compiler itself; and that they can form part of the "documentation" of a program, allowing it to be understood more easily by other programmers.

1.1.3 Typed λ-calculi

One valuable tool in the study of statically typed programming languages is a class of formal systems known as *typed λ-calculi*. These calculi are programming languages in their own right, since they can be described and reasoned about mathematically, they incorporate notions of behavior, and they admit an effective translation from programs to the behaviors they describe. But they are languages of a much simpler order than those used in the day-to-day work of most programmers. They omit all niceties of punctuation and syntax, sacrifice readability for compactness, provide only the most impoverished collections of built-in types and operations (or sometimes none at all), and usually include only a tiny collection of basic conceptual structures instead of the rich and varied facilities offered by most full-fledged languages. In short, they are intended as objects of study rather than vehicles for expression of complex ideas. Nevertheless, experience has shown that new ideas in language design — in particular, the behavior and interactions of various kinds of type structures — can be studied very productively in these isolated settings and the results transferred to larger languages constructed on the basis of the same ideas. Of course, significant attention must still be paid to the numerous engineering issues involved in full-scale language design; but the core type system will retain its essential properties.

Of course, some care must be used in generalizing from properties of λ-calculi to larger programming languages with "similar" type systems, since this process is only sound when the core conceptual structures of the small and large languages are truly analogous. One important situation where the correspondence is sometimes misstated is the case where the λ-calculus and the larger programming language have different notions of evaluation. For example, the core type system of the Standard ML language, which uses a call-by-value evaluation regime, where arguments are fully evaluated before being passed to functions, is often analyzed in terms of a λ-calculus with normal-order evaluation, where arguments are passed to functions unevaluated. For simple fragments of the full SML language, this mismatch turns out to be harmless; but the addition of computational effects such as updateable cells, exceptions, or unbounded recursion leads to unsoundness of the naive type system for the full language [68] (this point is also discussed in recent unpublished manuscripts of Robert Harper).

1.1.4 Subtyping

During the past decade, researchers in static type systems have been particularly successful in developing formal accounts of the notion of *subtyping*. In 1984, Cardelli [23] suggested that the basic concepts of object-oriented programming [8, 54, 67] could be understood type-theoretically using the following rough correspondence:

Object-oriented languages	**Typed lambda-calculi**
Classes	Record types
Objects	Records
Subclass	Subtype
Methods	Functions
Message passing	Function call

Some parts of this picture have been filled in during the intervening years [33, 133, 134, 113, 29, 101, 32, 71, 11, 130, 82, 26, 112, 18, 124, 39, 63, 83, 24, 135, 115, 133, 21, 22, 114, 25, 96, 111, 64, etc.]:

Object-oriented languages	**Typed lambda-calculi**
Classes	Record types
Objects	Records
Subclass	Subtype
Methods	Functions
Message passing	Function call
Object modification	Functional record update
Method inheritance	Cascaded record construction
self	Recursive records
SelfType	Recursive record types

Accounting for other aspects of object-oriented programming in this framework — method inheritance, in particular — remains the subject of active research. But whatever the ultimate success of this research program, the basic framework has proven to be a fruitful source of innovations in language design.

Informally, a type σ is a *subtype* of a type τ if any element of σ may sensibly be considered as an element of τ. In the simplest case, this just means that every element of σ *is* an element of τ, as when σ is "monkey" and τ is "mammal." In general, though, this need not be the case. On many computers, integers are represented in a completely different format from floating-point numbers, so, although we may abstractly think of the integers as being a subset of the floating-point numbers, the truth is that every integer may be *coerced* to an equivalent floating-point number. A more accurate formulation of the notion of subtype, then, is that σ is a subtype of τ if every element of σ contains sufficient information that it can be coerced to an appropriate element of τ.

Hiding in the word "appropriate" is another important observation: when we speak of a coercion from one type to another, we do not intend that this be any mapping whatsoever; it must preserve the identity of the original values, as much as possible, in their new forms. The coercion from integers to floating point numbers must take each integer to "the same number" in the floating-point representation.

This discussion can be formalized as a general architecture for constructing typed λ-calculi with subtyping. We begin with a collection of *types* (σ, τ, ...), a collection of *expressions* or *terms* (e, f, ...), and some formal rules describing the circumstances under which we may validly assert that a term e has a type τ. To handle subtyping, we introduce an order structure on the collection of types — a relation $\sigma \leq \tau$. Again, this relation is presented as a collection of rules describing the circumstances under which we may validly assert that a type σ is a subtype of τ. To make a connection between the two systems of rules, we add to the typing rules a rule of *subsumption* formalizing the intuitive notion that when $\sigma \leq \tau$ every term that may validly be considered an element of σ may also be considered an element of τ.

This skeletal framework can be extended in many ways, depending on the definitions of the sets of types and terms and the typing and subtyping relations. Two instances that are of special importance here are the *first-order (simply typed) λ-calculus with intersection types* — called $\lambda_\wedge$ ("lambda-meet") here — and the *second-order (polymorphic) λ-calculus with bounded quantification* — usually called $F_\leq$ ("F-sub").

The idea of *intersection types* is extremely simple and natural, though its ramifications in programming are only beginning to be understood. Essentially, it consists of enriching the collection

of types with a new type $\sigma \wedge \tau$ for every pair of types σ and τ (including the case where σ and τ themselves contain intersections). This new type is thought of as containing all the elements of σ that are also elements of τ; using our more general notion of subtyping, every element of $\sigma \wedge \tau$ contains enough information to coerce it either to an element of σ or to an element of τ. Furthermore, $\sigma \wedge \tau$ should be the "best" such type, in the sense that it is a supertype of every other type whose elements contain sufficient information to coerce them to either σ or τ. In terms of the order structure, this is precisely the greatest lower bound of σ and τ.

The most intriguing and potentially useful property of intersection types is their ability to express an essentially unbounded (though of course finite) amount of information about the components of a program. For example, the addition function + can be given the type $\mathit{Int} \to \mathit{Int} \to \mathit{Int} \wedge \mathit{Real} \to \mathit{Real} \to \mathit{Real}$, capturing both the general fact that the sum of two real numbers is always a real and the more specialized fact that the sum of two integers is always an integer. A compiler for a language with intersection types might even provide two different object-code sequences for the two versions of +, one using a floating point addition instruction and one using integer addition. For each instance of + in a program, the compiler can decide whether both arguments are integers and generate the more efficient object code sequence in this case. This kind of *finitary polymorphism* or *coherent overloading* is so expressive, that (in a sense that can be made theoretically precise; c.f. Section 2.5) the set of all valid typings for a program amounts to a complete characterization of the program's behavior.

Intersection types can also be viewed as a natural type-theoretic analog of *multiple inheritance*. If $\sigma \leq \theta$ is read as "σ is a subclass of θ," then $\sigma \wedge \tau$ is a name for a class with all the common properties of σ and τ. Of course, this analogy, like the subtype$\leftrightarrow$subclass analogy, is not exact. In particular, it says nothing about the complex mechanisms supporting code reuse in object-oriented programming languages with multiple inheritance. But it is intuitively appealing and, like the rest of Cardelli's analogy, can perhaps be made more precise in a sufficiently enriched calculus based on intersection types. We shall return to this point in Chapter 8.

Bounded quantification is an extension of the simpler notion of ordinary second-order quantification, or polymorphism. This was introduced in the early 1970's by Girard and Reynolds to capture the intuitive concept of a function that takes a type as a parameter. For instance, the "polymorphic reverse" function, which accepts a type τ and returns the *monomorphic* reverse function that reverses lists whose elements are all of type τ, has the quantified type $\forall\alpha.\ \mathit{List}(\alpha) \to \mathit{List}(\alpha)$. Cardelli and Wegner integrated this mechanism with the notion of subtyping by allowing a quantified type to give a *bound* for its parameter; for example, $\forall\alpha \leq \mathit{Student}.\ \mathit{List}(\alpha) \to \mathit{List}(\alpha)$ takes, as its first parameter, an arbitrary subtype of the type *Student* and returns a function on lists of this type. This form of *universal* or *parametric polymorphism* is both broader (since the number of possible instantiations of a polymorphic type is infinite) and more rigid (since all instances must have the same basic shape) than the finitary polymorphism provided by intersection types. Its main practical advantage is compile-time efficiency: it allows polymorphic expressions to be written, typechecked, and compiled just once.

1.2 Claims

This thesis is a detailed investigation of a typed λ-calculus combining intersection types and bounded quantification.

Since intersection types and polymorphism can each be formulated in different ways with varying degrees of expressiveness and technical difficulty, there is actually not just a single calculus combining the two notions but a whole space of such calculi; our first task is selecting one or more

of these as our object of study. We chose to focus on just one in order to study it in the greatest possible depth. This calculus, called $F_\wedge$, was formed by combining the most expressive formulations of intersection types and bounded quantification, so that any positive results obtained for it would apply to as many as possible of the other calculi combining intersections and polymorphism. (This choice is discussed in greater depth in Section 3.5.)

Our major claims are as follows:

- Bounded quantification and intersection types fit together very naturally. The syntax of our calculus combining them is elegant and relatively simple. The two different kinds of polymorphism — finitary and parametric — complement each other, leading to a variety of novel and useful programming idioms.
- Natural algorithms exist for subtyping and typechecking; these can (with some work) be proved partially correct. However, the subtype relation of $F_\wedge$ turns out to be undecidable. This comes as a surprise, since the undecidability result also applies to the pure calculus of bounded quantification, which was generally thought to be decidable.
- Untyped semantic models of $F_\wedge$, where subtyping is interpreted as simple inclusion, are unproblematic. Appealing typed models, where subtyping is interpreted by actual coercion functions, can also be sketched, but we encounter difficulties with the details.
- Our negative results — the undecidability of subtyping and the difficulty of constructing typed models — indicate that in some ways $F_\wedge$ is too powerful. Future investigations in this area might profitably concentrate on weaker fragments for which the same positive results can be proven more easily and for which comparable negative results do not hold, provided that such fragments retain most of $F_\wedge$'s expressive power. We propose some likely candidates in Chapter 8.

1.3 Outline of Results

The development of the technical chapters may be easier to follow for readers with some background in λ-calculus [5, 77], type systems [33, 120], and, for Section 2.4, basic category theory [3, 7, 90, 106].

Chapter 2 is a self-contained introduction to the major precursors of the $F_\wedge$ calculus: the simply typed λ-calculus with subtyping, the first-order calculus of intersection types, and the second-order calculus of bounded quantification. Besides introducing the notation and conceptual background needed for later development, this chapter gives a conceptual and terminological framework in which the semantics of these languages can be understood. Although it presents no novel results, this framework may contribute to the organization and clarification of terminology for these systems in the literature.

The $F_\wedge$ calculus itself is introduced in Chapter 3 and some basic design issues arising in its formulation are discussed. In particular, we introduce a new language construct, the *for* expression, which controls the search behavior of the typechecker and the introduction of intersection types. This construct generalizes similar ideas from Reynolds' Forsythe language and provides a cleaner separation of mechanism than earlier formulations.

Chapter 4 undertakes a thorough proof-theoretic investigation of the properties of $F_\wedge$. We begin by analyzing the subtype relation using "canonical types," a well-behaved fragment with sufficient expressive power to capture the essential aspects of the whole language. Using an extension of Curien and Ghelli's method of proof normalization by rewriting [50, 63], we show that every canonical derivation can be transformed into a derivation in a restricted normal form. This fact

is used to prove the soundness and semi-completeness of a straightforward algorithm for checking the subtype relation for arbitrary types. This algorithm forms a major component of a type synthesis procedure for $F_\wedge$ expressions, which is shown to be sound and semi-complete, in the sense that whenever it terminates it computes a minimal type. The definition of this algorithm can be thought of as defining a class of normal-form typing derivations similar to the normal-form subtyping derivations that arise in the subtyping algorithm; the existence of these normal forms yields a simple proof of the conservativity of $F_\wedge$ over first-order intersection types.

Chapter 5 offers a preliminary semantic investigation of $F_\wedge$. First, a simple untyped model is defined using partial equivalence relations and the soundness of the typing rules is proved. We next present the major obstacle to a full account of the typed semantics of $F_\wedge$: the fact that there are pairs of $F_\wedge$ types with no least upper bound. This result implies that standard methods for constructing models of intersection types and proving them coherent cannot be extended straightforwardly to $F_\wedge$. Nevertheless, we can give a partial account of the typed interpretation of $F_\wedge$ as a programming language by exhibiting a translation from $F_\wedge$ typing derivations into a calculus with a number well-studied typed models: the ordinary polymorphic λ-calculus extended with surjective tuples. This translation extends work on the semantics of $F_\leq$ by Breazu-Tannen, Coquand, Gunter, and Scedrov [10] by interpreting intersection types as "coherent tuples" in the target language. An alternative explanation of the semantics of $F_\wedge$ is given by an equational theory of equivalences between $F_\wedge$ terms; this theory is shown to be a sound description of the earlier untyped and translation semantics (assuming in the latter case that the translation is coherent).

Next (Chapter 6), we consider the completeness of the typechecking algorithm and discover, unfortunately, that the typechecking problem for $F_\wedge$ turns is undecidable. Indeed, we prove a stronger result: the subtyping problem ("given a set of assumptions Γ and two types σ and τ, is it the case that σ is a subtype of τ under Γ?") is already undecidable in $F_\leq$, the pure calculus of second-order bounded quantification. This result is of significant independent interest, since $F_\leq$ subtyping has long been thought to be decidable and numerous theoretical studies and language designs have been based on it. However, we argue that both $F_\leq$ and $F_\wedge$ are "decidable in practice," in the sense that the natural type synthesis algorithms terminate for every case of conceivable practical importance. Moreover, it is easy to show that some large and useful fragments of these calculi are decidable.

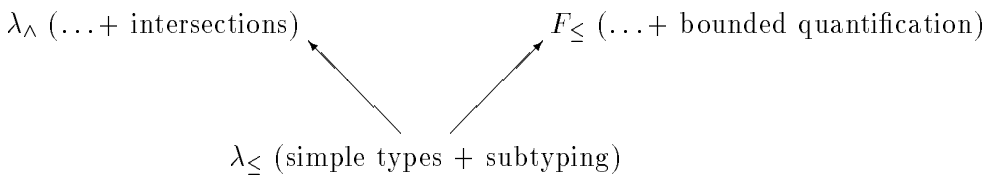
The last technical chapter, Chapter 7, presents a collection of programming examples in $F_\wedge$ including many new examples of programming with intersection types and type-theoretic formulations of simple abstract interpretation and strictness analysis. We close with some observations on programming with the *for* construct and techniques for debugging programs written in this style.

Chapter 8 presents a critical evaluation of our results and outlines a program for further study.

Chapter 2

Background

This chapter sets the stage for the $F_\wedge$ calculus by establishing notational conventions and reviewing its two immediate ancestors. We first define a common core calculus, a simply typed λ-calculus with subtyping, and then discuss two extensions of this core, a first-order calculus with intersection types and a second-order calculus with bounded quantification.



Each section includes additional notation and terminology for the mechanisms it introduces; these are carried over or trivially extended in later sections.

2.1 Notational Preliminaries

2.1.1. Definition: A finite sequence with elements x_1 through x_n is written $[x_1..x_n]$. Concatenation of finite sequences is written $X_1 * X_2$ or X_1, X_2. Single elements are adjoined to the right or left of sequences with a comma: $[x_a, X_b]$ or $[X_a, x_b]$. The length of a finite sequence X is written $len(X)$.

Sequences are sometimes written in a "comprehension" notation: $[x \mid \ldots]$. For example, if $T \equiv [\sigma{\to}\tau,\ \psi{\to}\psi,\ \sigma{\to}\theta]$, then the comprehension $[\zeta_2 \mid \zeta_1{\to}\zeta_2 \in T \text{ and } \zeta_1{\equiv}\sigma]$ stands for the finite sequence $[\tau, \theta]$.

2.1.2. Definition: It is sometimes convenient to ignore the ordering of a finite sequence and treat it as a finite set. Conversely, we say that a finite sequence X *enumerates* a finite set Y if $Y = \{x \mid x \in X\}$.

2.1.3. Notation: Throughout the thesis, the metavariables α and β range over type variables; σ, τ, θ, ϕ, ψ, and ζ range over types; S, T, U, V, M, and P range over finite sequences of types; e and f range over terms; and x and y range over term variables. (See Appendix B for a complete glossary of metavariables.)

2.1.4. Definition: A *context* Γ is a finite sequence of typing and/or subtyping assumptions for a set of variables and/or type variables, with no variable listed twice. The empty context is written $\{\}$. More explicit definitions of the contexts of particular calculi are given below.

2.1.5. Definition: A *subtyping statement* is a phrase of the form $\Gamma \vdash \sigma \leq \tau$, where σ and τ are types. A *typing statement* is a phrase of the form $\Gamma \vdash e \in \tau$, where e is a term and τ is a type. The *body* of a statement is the portion to the right of the turnstile.

2.1.6. Definition: A *derivation* of a subtyping or typing statement J is a proof tree, valid according to some collection of inference rules, whose root is J. We write $d :: J$ to indicate that d is a derivation of J.

2.1.7. Notation: The metavariable J ranges over both subtyping and typing statements; c and d range over derivations of subtyping statements; s and t range over derivations of typing statements.

2.1.8. Convention: When two or more systems of inference rules are being considered simultaneously, the turnstile symbol will often be annotated with a superscript indicating which calculus a derivation belongs to; for example, derivations in the pure $F_\wedge$ calculus are marked $\vdash^\wedge$.

2.1.9. Convention: Types, terms, contexts, statements, and derivations that differ only in the names of bound variables are considered identical.

It is formally clearer to think of variables not as names but, as suggested by deBruijn [56], as pointers into the surrounding context. This point of view is notationally too inconvenient to adopt explicitly in what follows, but will be a significant aid in understanding the behavior of the rules that manipulate variables.

2.1.10. Definition: When X and X' are identical phrases (types, terms, contexts, finite sequences of types, statements, derivations, etc.) up to renaming of bound variables, we write $X \equiv X'$. If X' contains free metavariables, then $X \equiv X'$ denotes pattern matching; for example

"if $\tau \equiv \tau_1 \rightarrow \tau_2$, then..."

means

"if τ has the form $\tau_1 \rightarrow \tau_2$ for some τ_1 and τ_2, then..."

2.1.11. Definition: The number of nodes in a derivation d is written $size(d)$.

2.1.12. Definition: The capture-avoiding substitution of e for x in f is written $\{e/x\}f$. The capture-avoiding substitution of σ for α in τ or e is written $\{\sigma/\alpha\}\tau$ or $\{\sigma/\alpha\}e$. The capture-avoiding substitution of σ for α in the range of Γ is written $\{\sigma/\alpha\}\Gamma$.

2.1.13. Notation: The functional application of one phrase to another is normally denoted by juxtaposition: $e_1\ e_2$. When e_1 or e_2 is a long or complex expression, the application is sometimes emphasized with an explicit marker: $e_1 \cdot e_2$. Similarly, type applications, usually written $e\ [\tau]$, sometimes appear as $e \cdot [\tau]$ to improve readability.

2.1.14. Notation: Sessions with the prototype typechecker for $F_\wedge$ are set in a typewriter font using only ascii symbols. The mathematical symbols used in the λ-calculus notation are transliterated as follows:

Ascii	TEX
`s, t`	$\sigma,\ \tau$
`A, B`	$\alpha,\ \beta$
`->`	$\rightarrow$
`/\`	$\bigwedge,\ \wedge$
`T`	$\top$
`\x:s. e`	$\lambda x{:}\sigma.\ \ e$
`\\a<s. e`	$\Lambda\alpha{\leq}\sigma.\ \ e$
`All a<s. t`	$\forall\alpha{\leq}\sigma.\ \ \tau$
`<`	$\leq$
`plus, times`	$+,\ \times$

Lines of input to the running typechecker are prefixed with a `>` character and followed by the system's response:

```
> f = \x:Int. x;
f : Int -> Int
```

2.1.15. Convention: The type constructors $\rightarrow$ and $\forall$ are assumed to bind more tightly than $\wedge$, allowing most parentheses to be dropped. Also, $\rightarrow$ associates to the right and $\forall$ obeys the usual "dot rule" where the body τ of a quantified type $\forall\alpha{\leq}\sigma.\ \tau$ is taken to extend to the right as far as possible.

For example, the type expression

$$(\sigma{\rightarrow}(\tau{\rightarrow}(\theta_1{\wedge}\theta_2))) \wedge (\forall\alpha{\leq}\sigma.\ (\forall\beta{\leq}\tau.\ (\phi{\wedge}\psi)))$$

is written

$$\sigma{\rightarrow}\tau{\rightarrow}(\theta_1{\wedge}\theta_2) \wedge \forall\alpha{\leq}\sigma.\ \forall\beta{\leq}\tau.\ \phi{\wedge}\psi.$$

2.1.16. Remark: The word *algorithm* is used throughout the thesis in the sense of "recursively defined procedure," with no intended connotation of totality. When a given algorithm is known to terminate for all inputs, we call it a *decision procedure*.

2.2 Simply Typed λ-Calculus with Subtyping

The $F_\wedge$ calculus may be viewed as a "least upper bound" of two calculi: a first-order λ-calculus with intersection types ($\lambda_\wedge$) and a second-order λ-calculus with bounded quantification ($F_\leq$). These, in turn, are both extensions of the simply typed λ-calculus enriched with a subtyping relation ($\lambda_\leq$). The latter system was proposed by Cardelli [20, 23] as a "core calculus of subtyping" in a foundational framework for object-oriented programming languages.

2.2.1. Definition: The types of $\lambda_\leq$ consist of a set of *primitive types* (ranged over by the metavariable ρ) closed under the *function space* type constructor $\rightarrow$:

$$\tau \quad ::= \quad \rho \quad | \quad \tau_1{\rightarrow}\tau_2$$

2.2.2. Definition: The terms of $\lambda_\leq$ consist of a countable set of variables (ranged over by x), together with all the phrases that can be built from these by functional abstraction and application:

$$e \quad ::= \quad x \quad | \quad \lambda x{:}\tau.\ e \quad | \quad e_1\ e_2$$

2.2.3. Remark: The presence of the *domain-type annotation* τ in the syntax of λ-abstractions marks a fundamental design choice, which we shall maintain throughout the thesis: all of the calculi

we consider are *explicitly typed* systems (as opposed to *type assignment* systems). This requires that a programmer exert firm control over the typechecker's behavior, making programs more verbose but rendering typechecking decidable in many cases where type inference would be undecidable. Section 2.4 discusses these issues in more detail.

2.2.4. Definition: A $\lambda_{\leq}$ context is a sequence of typing assumptions

$$\Gamma \quad ::= \quad \{\} \quad | \quad \Gamma, x{:}\tau$$

with no variable mentioned twice. The function $dom(\Gamma)$ denotes the set of variables defined by Γ; the *range* of Γ is the collection of right-hand sides of bindings in Γ. $\Gamma(x)$ denotes the type of x in Γ, if it has one.

2.2.5. Definition: The set of free variables of a term e is written $FV(e)$.

2.2.6. Definition: A term e is *closed* with respect to a context Γ if $FV(e) \subseteq dom(\Gamma)$. A typing statement $\Gamma \vdash e \in \tau$ is closed if e is closed with respect to Γ.

2.2.7. Convention: In the following, we assume that all statements under discussion are closed. In particular, we allow only closed statements in instances of inference rules.

The typing relation of $\lambda_{\leq}$ is formalized as a collection of inference rules for deriving typing statements of the form $\Gamma \vdash e \in \tau$ ("under assumptions Γ, expression e has type τ"), where Γ contains a typing assumption for each of the free variables of e. The rules for variables, abstractions, and applications are exactly the same as in the ordinary simply typed λ-calculus [37]. In addition, we introduce a rule of *subsumption* stating that whenever a term e has a type σ and σ is a subtype of another type τ, the type of e may be promoted to τ.

2.2.8. Definition: The $\lambda_{\leq}$ typing relation $\Gamma \vdash e \in \tau$ is the least three-place relation closed under the following rules:

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(VAR)}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e \in \tau_1 \to \tau_2} \qquad \text{(ARROW-I)}$$

$$\frac{\Gamma \vdash e_1 \in \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash e_1\, e_2 \in \tau_2} \qquad \text{(ARROW-E)}$$

$$\frac{\Gamma \vdash e \in \tau_1 \qquad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e \in \tau_2} \qquad \text{(SUB)}$$

2.2.9. Remark: This definition may be viewed in two different ways:

1. as a three-place relation constructed as the limit of a sequence beginning with the empty relation and successively enriching it according to the rules VAR, ARROW-I, ARROW-E, and SUB, or
2. as a simple logic whose derivable judgements are those appearing as conclusions of valid derivation trees built from these rules.

We adopt both views, interchangeably, in what follows. For example, the consequent in the sentence

If $\Gamma \vdash e \in \tau$, then $\Gamma \vdash f \in \theta$

may be read as asserting the existence of a valid derivation with conclusion $\Gamma \vdash f \in \theta$, as well as the presence of the tuple $\langle \Gamma, f, \theta \rangle$ in the graph of the typing relation.

2.2.10. Remark: By analogy with types, we might expect the definition of $\lambda_{\leq}$ terms to include a collection of constants. These can be added to the calculus, but they are not strictly necessary:

we can write programs involving "built-in values" like numbers and arithmetic operators simply by considering these as variables and providing a *pervasive context* — call it Γ_P — assigning them appropriate types. When contexts are extended in Section 2.6 to allow assumptions for type variables, primitive types can also be dropped, although not every conceivable ordering on the primitive types can be encoded as a pervasive context (c.f. 2.6.6 and 3.4.2.5).

It remains to define the subtype relation. Intuitively, a subtyping statement $\Gamma \vdash \sigma \leq \tau$ corresponds to the assertion that σ is a *refinement* of τ, in the sense that every element of σ contains enough information to meaningfully be regarded as an element of τ. In some models this means simply that σ is a subset of τ; more generally, it implies the existence of a distinguished *coercion function* from σ to τ.

These considerations immediately entail that the subtype relation should be both reflexive and transitive — i.e., that it should be a preorder. We assume that the subtype relation on primitive types is given in advance by some preorder $\leq_P$. This relation is extended to the smallest preorder closed under the following subtyping rule for function types: $\sigma_1 \rightarrow \sigma_2$ is a subtype of $\tau_1 \rightarrow \tau_2$ iff τ_1 is a subtype of σ_1 and σ_2 is a subtype of τ_2. Notice that, as usual, this relation is *covariant* in the right-hand side and *contravariant* in the left-hand side of the $\rightarrow$ constructor: a collection of functions can be refined either by narrowing the range into which their results must fall or by enlarging the domain over which they must behave properly.

2.2.11. Remark: A key feature of this notion of subtyping is that it is *structural*: the ordering of two arrow types is completely determined by their left- and right-hand sides. In logical terms, the only extended theories we consider are those whose non-logical rules are restricted to statements about primitive types. This feature is retained in the other calculi we consider in the thesis.

In fact, the subtype relation of $\lambda_\leq$ is not only structural, but *compositional*: the ordering on arrow types may be computed as a function of the ordering of their left- and right-hand sides. The introduction of intersection types in the next section will invalidate this stronger property.

2.2.12. Definition: The $\lambda_\leq$ subtyping relation $\Gamma \vdash \sigma \leq \tau$ is the least three-place relation closed under the following rules:

$$\Gamma \vdash \tau \leq \tau \qquad \text{(Sub-Refl)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \tau_2 \qquad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3} \qquad \text{(Sub-Trans)}$$

$$\frac{\Gamma \vdash \rho_1 \leq_P \rho_2}{\Gamma \vdash \rho_1 \leq \rho_2} \qquad \text{(Sub-Prim)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2 \leq \tau_1 \rightarrow \tau_2} \qquad \text{(Sub-Arrow)}$$

2.2.13. Remark: The context Γ plays no part in these rules and could be dropped without changing the system. We include it here for notational compatibility with later systems, in which contexts will also contain subtyping assumptions about type variables.

2.2.14. Notation: When $\Gamma \vdash \sigma \leq \tau$, we call τ a *supertype* of σ.

2.2.15. Notation: When $\Gamma \vdash \sigma \leq \tau$ and $\Gamma \vdash \tau \leq \sigma$, we say that σ and τ are *equivalent* under Γ, written $\Gamma \vdash \sigma \sim \tau$.

2.2.16. Notation: We write $\Gamma \not\vdash \sigma \leq \tau$ to *deny* the derivability of the statement $\Gamma \vdash \sigma \leq \tau$.

2.2.17. Convention: When necessary to prevent confusion with other calculi, turnstiles in $\lambda_\leq$ derivations are written $\vdash^{\lambda\leq}$.

2.3 Intersection Types

Intersection types in the pure λ-calculus were developed in the late 1970s by Coppo and Dezani-Ciancaglini [40], and independently by Sallé [46, 127] and Pottinger [110]. Since then, they have been studied extensively by members of the group at the university of Turin and many others [6, 35, 41, 42, 43, 44, 45, 57, 58, 75, 76, 121, 123, 125, 126, 131, 132]. The original motivation for their introduction was the desire for a type-assignment system in the spirit of Curry [52], but with two additional properties:

1. The typing of a term should be invariant under β-conversion. (Under Curry's system, β-reduction preserves types but β-expansion, in general, does not.)
2. Every term possessing a normal form should be given a meaningful typing.

Various extensions of the original intersection type discipline have also been explored. These include the notion of infinite intersections [88], the dual notion of union types [4, 73, 105, 107], and the relationship between intersection types and models of polymorphism [81, 104, 136]. Some related extensions to ML-style type inference systems are represented by the notions of refinement types [60, 72, 107] and soft typing [36, 59].

Reynolds provided the first demonstration that intersection types can be used as the basis for practical programming languages [118, 121]. A primary goal of this thesis is to extend Reynolds' work by studying the interaction of intersection types with other important type-theoretic principles, primarily parametric polymorphism, and to develop a larger suite of interesting examples illustrating their utility in programming.

2.3.1. Definition: The first-order calculus of intersection types, $\lambda_\wedge$, is formed from $\lambda_\leq$ by adding intersections to the language of types:

$$\tau \quad ::= \quad \rho \quad | \quad \tau_1 \rightarrow \tau_2 \quad | \quad \bigwedge[\tau_1..\tau_n]$$

2.3.2. Definition: For presenting examples, our formulation of the $\lambda_\wedge$ type system in terms of n-ary intersections is somewhat cumbersome. We therefore introduce the following abbreviations:

$$\begin{array}{lcl} \sigma \wedge \tau & \stackrel{\text{def}}{=} & \bigwedge[\sigma, \tau] \\ \top & \stackrel{\text{def}}{=} & \bigwedge[\,]. \end{array}$$

Of course, the whole system could equally well be formulated in terms of a binary constructor $\wedge$ and a nullary constructor $\top$. However, for the theoretical analysis of the system (and its extension, $F_\wedge$), the n-ary formulation leads to shorter and clearer proofs of its properties.

2.3.3. Remark: The notations $\sigma \cap \tau$, $\sigma \& \tau$, and $\sigma \wedge \tau$ have all been used to denote the intersection of σ and τ; the universal type (usually corresponding to a nullary intersection) has been written as both ω and *ns* ("nonsense"). To emphasize the order-theoretic intuition that the intersection of σ and τ is their greatest lower bound (or "formal meet") in the subtype preorder — and to *de*-emphasize the common intuition that $\sigma \wedge \tau$ denotes the set-theoretic intersection of the denotations of σ and τ — we use the $\wedge$ symbol for binary and n-ary intersections. The phrase $\sigma \wedge \tau$ (or $\bigwedge[\sigma, \tau]$) is pronounced "σ meet τ," "σ intersect τ," or "σ and τ." For the nullary intersection, we use the symbol $\top$ ("top") by analogy with the binary case.

Two new subtyping rules capture the order-theoretic properties of the $\wedge$ operator:

$$\frac{\text{for all } i,\ \Gamma \vdash \sigma \leq \tau_i}{\Gamma \vdash \sigma \leq \bigwedge[\tau_1..\tau_n]} \qquad \text{(Sub-Inter-G)}$$

$$\Gamma \vdash \bigwedge[\tau_1..\tau_n] \leq \tau_i \qquad \text{(Sub-Inter-LB)}$$

(Complete sets of inference rules for $\lambda_\wedge$ and the other major systems introduced in the thesis appear in Appendix A. Here we discuss just the extensions to $\lambda_\leq$ that are required to form $\lambda_\wedge$.) Note that the premise of the first rule actually stands for n different premises, one for each $i \in \{1..n\}$; similarly, the second rule stands for n different rules, one for each value of i.

One additional subtyping rule captures the relation between intersections and function spaces, allowing the two constructors to "distribute" when an intersection appears on the right-hand side of an arrow:

$$\Gamma \vdash \bigwedge[\sigma{\to}\tau_1 \,..\, \sigma{\to}\tau_n] \leq \sigma \to \bigwedge[\tau_1..\tau_n] \qquad \text{(Sub-Dist-IA)}$$

(This inclusion is actually an equivalence, since the the other direction may be proved from the rules for meets and arrows.)

This rule, though intuitively reasonable, will have a strong effect on both syntactic and semantic properties of the language. For example, it implies that $\top \leq \sigma{\to}\top$ for any σ.

The typing rules must also be extended slightly. As for any type constructor, we expect to find a pair of an introduction rule, by which terms can be shown to possess intersection types, and an elimination rule, by which this fact may later be exploited. The introduction rule allows an intersection type to be derived for a term whenever each of the elements of the intersection can be derived for it separately:

$$\frac{\text{for all } i,\ \Gamma \vdash e \in \tau_i}{\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]} \qquad \text{(Inter-I)}$$

The corresponding elimination rule would allow us to infer, on the basis of a derivation of a statement like $\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]$, that e possesses every τ_i individually. But this follows already from the rule Sub-Inter-G and the rule of subsumption; we need not add the elimination rule explicitly to the calculus.

The nullary case of this rule is worth particular notice, since it allows the type $\top$ to be derived for *every* term of the calculus, including terms whose evaluation intuitively encounters a run time error

```
> 5 true;
it : T
```

or fails to terminate:

```
> (\x:T. x x) (\x:T. x x);
it : T
```

The system as we have described it so far supports the use of intersection types in programming only to a limited degree. Suppose, for example, that the primitive subtype relation has $Int \leq Real$ and the addition function in the pervasive context is overloaded to operate on both integers and reals:

$$\Gamma_P(+) = Int{\to}Int{\to}Int \wedge Real{\to}Real{\to}Real.$$

Expressions involving addition of integers and reals will be given type *Int* if possible, otherwise type *Real*:

```
> plus 0 0;
it : Int

> plus pi pi;
it : Real

> plus 0 pi;
it : Real
```

(Note that the typechecker attempts to simplify the type it derives for each term, so that, for example, the type of `plus 0 0` is printed as `Int` instead of as `Int/\Real`.)

But using just the constructs introduced so far, there is no way of writing our own functions that behave in this way. For example, the doubling function $\lambda x{:}?.\ x + x$ cannot be given the type $\mathit{Int}{\to}\mathit{Int} \wedge \mathit{Real}{\to}\mathit{Real}$, since replacing the ? with either *Int* or *Real* (or even $\mathit{Int}{\wedge}\mathit{Real}$) gives a typing that is too restrictive:

```
> double1 = \x:Int. plus x x;
double1 : Int -> Int

> double2 = \x:Real. plus x x;
double2 : Real -> Real

> double3 = \x:Int/\Real. plus x x;
double3 : Int -> Int
```

This led Reynolds [121] to introduce a generalized form of λ-abstraction allowing explicit programmer-controlled generation of alternative typings for terms:

$$e \quad ::= \quad \ldots \quad | \quad \lambda x{:}\tau_1..\tau_n.\ e$$

The typing rule for this form allows the typechecker to make a choice of any of the σ's as the type of x in the body:

$$\frac{\Gamma, x{:}\sigma_i \vdash e \in \tau_i}{\Gamma \vdash \lambda x{:}\sigma_1..\sigma_n.\ e \in \sigma_i{\to}\tau_i} \qquad \text{(Arrow-I')}$$

This rule can be used together with Inter-I to generate a set of up to n alternative typings for the body and then form their intersection as the type of the whole λ-abstraction:

```
> double = \x:Int,Real. plus x x;
double : Int->Int /\ Real->Real
```

One peculiar property of the generalized λ is that adding extra alternatives to the set of possible domain types for x can only improve the typing of the whole expression. If some alternative results in a "typechecking failure," the best type for the body under this assumption will be equivalent to $\top$ (typically via the Sub-Dist-IA rule), and may therefore be dropped from the final type of the expression without changing its equivalence class in the subtype ordering:

```
> double = \x:Int,Real,Char. plus x x;
double : Int->(Int/\Real) /\ Real->/\[Real] /\ Char->T
    i.e. Int->Int /\ Real->Real
```

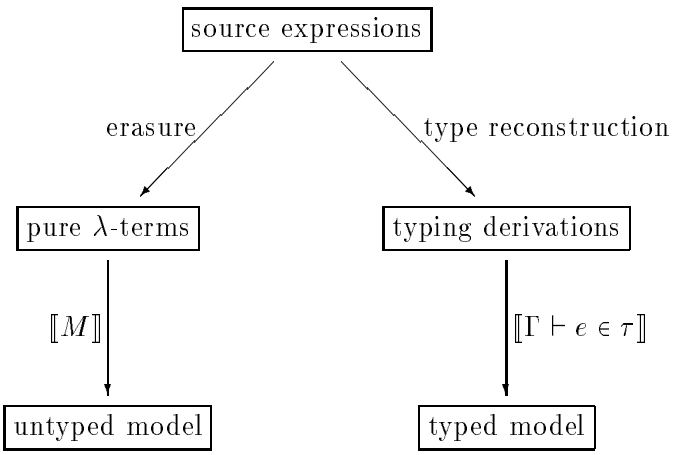
Another point worth noting is that, in $\lambda_{\leq}$, the embedding of the primitive subtype relation $\leq_P$ into the full relation $\leq$ preserves both greatest lower bounds and least upper bounds. In $\lambda_{\wedge}$, this is true only for least upper bounds; that is:

- In both $\lambda_{\leq}$ and $\lambda_{\wedge}$, if ρ' is a least upper bound of ρ_1 and ρ_2 in the $\leq_P$ relation, then it is also a least upper bound in $\leq$.
- In $\lambda_{\leq}$, if ρ' is a greatest lower bound of ρ_1 and ρ_2 in the $\leq_P$ relation, then it is also a greatest lower bound in $\leq$.
- In $\lambda_{\wedge}$, if ρ' is a greatest lower bound of ρ_1 and ρ_2 in the $\leq_P$ relation, then it is *not* a greatest lower bound in $\leq$; in particular, it is strictly greater than $\rho_1{\wedge}\rho_2$.

2.3.4. Convention: When necessary to prevent confusion with other calculi, turnstiles in $\lambda_{\wedge}$ derivations are written $\vdash^{\lambda\wedge}$.

2.4 Semantic Frameworks for Intersection Types

Work in the semantics of typed programming languages and λ-calculi may roughly be divided into two philosophical camps. One, sometimes called *Curry-style* semantics, takes the semantics of an expression to be the semantics of the pure λ-term found by erasing any type annotations it may contain. The other, sometimes called *Church-style* semantics, views the expressions of a typed calculus as a linear shorthand for fully typed forms in which every phrase and subphrase is annotated with its typing; it is these fully explicit forms, i.e., the *typing derivations* of the calculus, to which a semantic interpretation is given. In general, Curry-style systems correspond to the left-hand side of the following diagram, while Church-style presentations correspond to the right-hand side:



These two perspectives have also been called the *epistemological* and the *ontological* views of types [87], since one is primarily concerned with *knowledge*, the other with *being*; *extrinsic* and *intrinsic* have also been suggested. Both views yield sensible and useful interpretations of systems with intersection types, and of $F_\wedge$ in particular.

2.4.1. Remark: The distinction between typed and untyped models is not a clear as it might appear from this sketch. For example, one of the mysteries of the polymorphic λ-calculus is that every known typed model is based on an underlying untyped model [Reynolds, personal communication, 1991]. Even so, the difference between typed and untyped semantic interpretations of terms is quite distinct.

2.4.1 Untyped Semantics

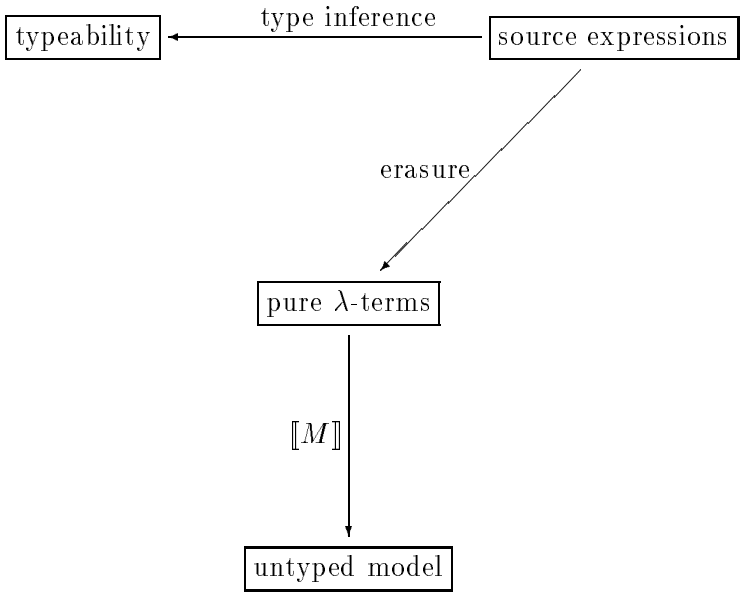
Curry-style type systems are often called *type assignment systems*. Terms in these systems typically contain no type annotations at all, in which case the erasure step is trivial. The *interpretation* of a term M is some element m of an untyped model D, given by a semantic function $[\![—]\!]$, which is defined by induction on the structure of terms. Typing is a matter of *predication*: a typing statement involving a term M is an assertion about $[\![M]\!]$.

According to this point of view, the interpretation of a type θ is a predicate, a set of elements of D for which the assertion expressed by θ is true. For example, the interpretation of $\sigma{\to}\tau$ is

$$[\![\sigma{\to}\tau]\!] = \{m \in D \mid \text{for all } n \in [\![\sigma]\!],\ m \cdot n \in [\![\tau]\!]\}.$$

A typechecker, in this context, can be thought of as proving theorems about programs — theorems that show, on the basis of a set of typing rules that are known to be sound descriptions of the semantics of terms, that the interpretations of terms behave in certain ways. A *type inference*

procedure is a deterministic procedure for discovering a *principal* theorem — a theorem of which all other theorems about the behavior of the program are corollaries. (The term "type inference" is occasionally used in an even more general sense, to describe algorithms that determine whether a term is *typeable*, without actually synthesizing any particular type or set of types for it ([85, for example]).



When σ and τ are regarded as predicates, the assertion that $\sigma \leq \tau$ simply means $[\![\sigma]\!] \subseteq [\![\tau]\!]$; the syntactic term *subtype* coincides with the semantic term *subset*. Similarly, the natural interpretation of $\sigma \wedge \tau$ is the logical conjunction of the predicates σ and τ, i.e., the intersection of the subsets of D denoted by σ and τ,

$$[\![\sigma \wedge \tau]\!] = [\![\sigma]\!] \cap [\![\tau]\!],$$

and $[\![\top]\!] = D$.

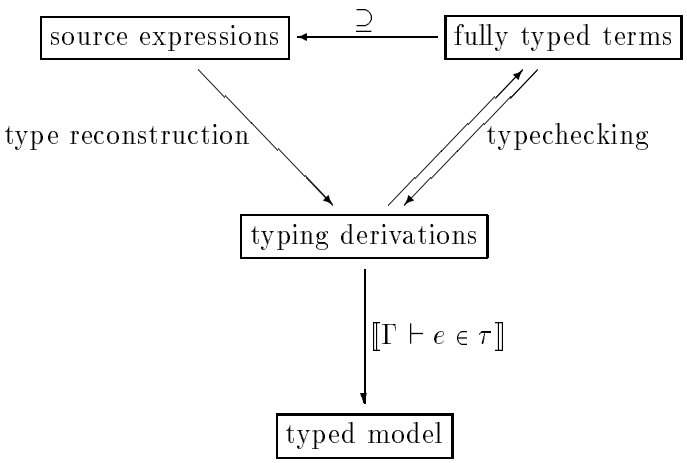
2.4.2 Typed Semantics

In Church-style type systems, commonly referred to as *typed λ-calculi*, the picture is somewhat more complicated. Typing, here, has behavioral force: it is not a *description* of semantics, but an integral *part* of semantics. The interpretation function $[\![—]\!]$ is defined by induction on typing derivations, not on the underlying terms. In cases where a typing derivation contains a subsidiary subtyping derivation, the latter is mapped into a function between semantic domains — a derivation whose conclusion is $\sigma \leq \tau$ is mapped into a *coercion function* from $[\![\sigma]\!]$ to $[\![\tau]\!]$. This function will sometimes be just an identity injection, as in the untyped case, but in general it may transform its inputs in some substantial way. For example, the coercion from $[\![Int]\!]$ to $[\![Real]\!]$ may involve a change of representation, and the coercion from a record with many fields to one with fewer fields may involve actually dropping the extra fields.

The typed interpretation of an expression e of the source calculus involves the intermediate step of constructing a typing derivation of $\Gamma \vdash e \in \tau$ for some Γ and τ. In other words, a bare source expression *underdetermines* its semantic interpretation: its image in the semantic domain depends on "more than meets the eye." Indeed, even when Γ and τ are fixed, there may be a number of different derivations of the statement $\Gamma \vdash e \in \tau$, and these might, in general, lead to different interpretations.

For practical programming languages, there must be some effective means of calculating a type τ and a valid derivation $s :: \Gamma \vdash e \in \tau$, whenever possible, for any given source expression e and context Γ. An algorithm that performs this task is a *type reconstruction procedure*.

For certain calculi (e.g., languages based on system F with *partial type reconstruction* [9, 102, 109]), it is possible to pick out a subset of the source expressions, the *fully typed terms*, with the special property that each fully typed term has at most one typing derivation for a particular choice of Γ, and for which the type reconstruction problem is therefore particularly simple. Type reconstruction for these terms might better be called *typechecking*, since this connotes a simpler sort of algorithm than "reconstruction." For such a calculus, our general picture of typed semantics may be refined as follows:



None of the languages considered in this thesis have a notion of fully typed terms, however, so the left-hand side of this diagram is the one that is important here.

Higher-order polymorphic λ-calculi sometimes blur the syntactic distinction between ordinary applications and type applications [47, 48]. In such situations, type reconstruction can be generalized to a notion of *argument synthesis* [109]. On the other hand, some second-order calculi with subtyping, such as $F_{\leq}$ and $F_{\wedge}$, require fully explicit type abstractions and applications, but allow the types of terms to be implicitly promoted to supertypes at any point. In these cases, *coercion reconstruction* may be a more appropriate term.

Just as some typings are more informative than others (for example, $Int \to Int \to Int \wedge Real \to Real \to Real$ is a better type than $Int \to Int \to Int$ for the + operator, and this is better, in turn, than $\top$), there may be some interpretations of (possible typings of) a source expression that are more useful than others. In general, more informative typings yield more useful interpretations, so it is desirable that the type reconstruction procedure calculate as good a typing as possible for each source expression. In the best case, a calculus may have the property that there is a "least" or *principal* typing for every typable source expression. Best of all is the case where an effective procedure exists for computing this typing.

All of the calculi studied in this thesis have the principal typing property. Both $\lambda_{\wedge}$ and $\lambda_{\leq}$ have decidable principal typings. (Indeed, the type assignment system corresponding to our explicitly typed $\lambda_{\leq}$has decidable principal typings.) Principal typings for $F_{\leq}$, unfortunately, fail to be decidable (c.f. Chapter 6), but a slightly weaker property does hold: whenever the type synthesis procedure described in Section 2.6 terminates, it yields a principal typing; in fact, for every source expression with *any* $F_{\leq}$ typing, the procedure is guaranteed to terminate and yield a principal typing. This property is not meaningful in $F_{\wedge}$, since there, as in $\lambda_{\wedge}$, every source expression has

some typing. But the cases where the natural type synthesis procedure diverges seem extremely unlikely to arise in practice.

For the sake of precision in what follows, we pause to sketch a framework for the category-theoretic semantics of a first-order typed λ-calculus with subtyping, as suggested by Reynolds [121, 123]. (For the present discussion, we drop the context Γ from subtyping derivations to emphasize that we are working with a first-order calculus. To deal rigorously with the more general case, where τ might depend on Γ, we would need a category-theoretic model for second-order bounded quantification. The structure of such models is not well understood.)

- The semantics of derivations is based on a category $\mathbf{K}$ and a subcategory $\mathbf{S}$ sharing all of $\mathbf{K}$'s objects. The objects of $\mathbf{K}$ and $\mathbf{S}$ correspond to the types of the source calculus. We require that $\mathbf{K}$ be Cartesian closed and that there be a functor $\underset{S}{\Longrightarrow} \in \mathbf{S}^{\mathrm{op}} \times \mathbf{S} \rightarrow \mathbf{S}$ that is a restriction of $\underset{K}{\Longrightarrow} \in \mathbf{K}^{\mathrm{op}} \times \mathbf{K} \rightarrow \mathbf{K}$, the exponentiation functor. $\mathbf{S}$, the subcategory of coercions, must include morphisms corresponding to the primitive coercions, must be closed under all the coercion constructors needed to interpret compound subtyping derivations, and must possess certain limits (discussed below), including at least products. Moreover, a limit in $\mathbf{S}$ should also be a limit of the same diagram in $\mathbf{K}$. In other words, we require that every $\mathbf{S}$-diagram of the appropriate form have a $\mathbf{K}$-limit.
- Each type τ of the source calculus is interpreted as (*denotes*) an object $[\![\tau]\!]$ of $\mathbf{K}$.
- A subtyping derivation $c :: \sigma \leq \tau$ is interpreted as a $\mathbf{S}$-morphism $[\![c]\!] \in [\![\sigma]\!] \rightarrow [\![\tau]\!]$.
- A context $\Gamma \equiv x_1{:}\tau_1 \,..\, x_n{:}\tau_n$ is interpreted as the $\mathbf{S}$-product $[\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$.
- A typing derivation $s :: \Gamma \vdash e \in \tau$ is interpreted as a $\mathbf{K}$-morphism $[\![s]\!] \in [\![\Gamma]\!] \rightarrow [\![\tau]\!]$.

The fact that information must be added to the source text of a program to find its behavior should be invisible to the programmer, in the sense that, whenever the same source expression may lead to different proofs of the same typing statement and thus to different interpretations, it is critical that these interpretations should behave identically. That is, the source expression should completely determine the observable behavior of all its possible interpretations: "What you see is what you get." A language that violates this requirement may still be perfectly well defined, but it will be impossible for programmers to predict the behavior of their programs, except, perhaps, by according to the details of some particular algorithm for computing principal typings. (Algol-68 and PL/I have both been criticized on this score.)

This idea of invariant behavior under alternative interpretations was introduced by Reynolds [117] for the specific case of the coercions associated with primitive operators like +. Breazu-Tannen, Coquand, Gunter, and Scedrov later considered the problem in a more general setting [10] and coined the term *coherence* to describe it.

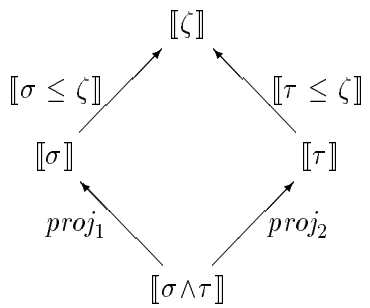
An important special case of the coherence of the interpretation of typing is the coherence of the interpretation of subtyping: for every pair of subtyping derivations c and d with the same conclusion, it must be the case that $[\![c]\!] = [\![d]\!]$. (The appropriate notion of "=" depends on the model.) If this requirement fails, we can easily use rule SUB to construct an incoherent pair of typing derivations.

If we consider the subtype preorder as a category, then the coherence condition for subtyping says precisely that the interpretation function is a functor.

The desire for a coherent semantics leaves very little choice in the interpretation of intersection types. For the sake of simplicity, consider just the binary intersection of two types σ and τ. Then the following observations follow directly:

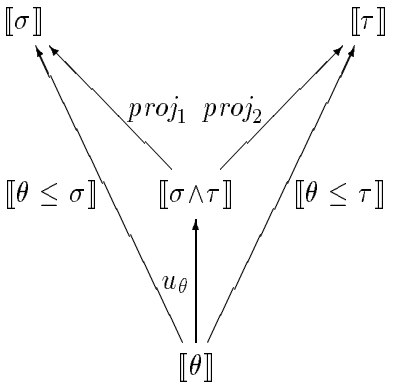
1. Since rule SUB-INTER-LB stipulates that $\sigma \wedge \tau \leq \sigma$ and $\sigma \wedge \tau \leq \tau$, there must be coercion functions $proj_1 \in [\![\sigma \wedge \tau]\!] \rightarrow [\![\sigma]\!]$ and $proj_2 \in [\![\sigma \wedge \tau]\!] \rightarrow [\![\tau]\!]$ in $\mathbf{S}$.

2. For every type ζ such that $\sigma \leq \zeta$ and $\tau \leq \zeta$, the composite coercions $proj_1$; $[\![\sigma \leq \zeta]\!]$ and $proj_2$; $[\![\tau \leq \zeta]\!]$ must be equal in order to achieve coherence — i.e., the **S**-diagram



must commute. (Here ";" denotes composition in diagrammatic order.)

3. For every type θ such that $\theta \leq \sigma$ and $\theta \leq \tau$, rule Sub-Inter-G implies that there must be a coercion from θ to $\sigma \wedge \tau$. Call this coercion u_θ. To achieve coherence, the coercions $[\![\theta \leq \sigma]\!]$ and $[\![\theta \leq \tau]\!]$ that map directly from θ to σ and τ must "factor through" u_θ — that is, the **S**-diagram

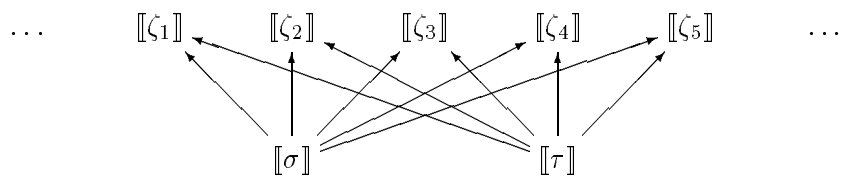


must commute.

In order that the interpretation of subtyping derivations be well-defined, this coercion u_θ should be determined by the coercions $[\![\theta \leq \sigma]\!]$ and $[\![\theta \leq \tau]\!]$.

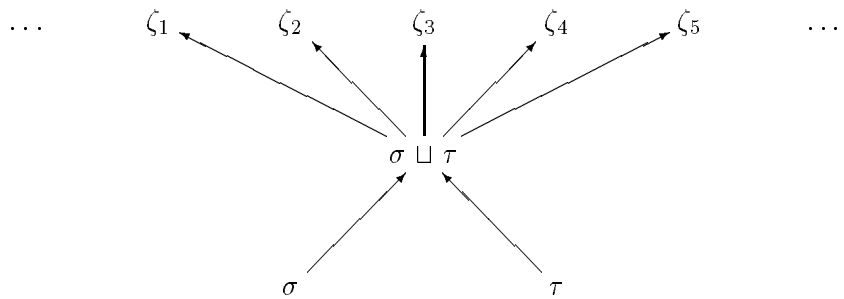
Taken together, these considerations lead us to a straightforward definition of the interpretation of $\sigma \wedge \tau$ as a categorical *limit*. (Note that we are not *forced* by the above considerations to interpret $\sigma \wedge \tau$ as a limit: any **S**-object that makes all the appropriate diagrams commute would do. But the limit is a particularly natural choice.)

2.4.2.1. Definition: The interpretation $[\![\sigma \wedge \tau]\!]$ and the associated coercions $proj_1$ and $proj_2$ are given as the limit, in **S**, of the diagram

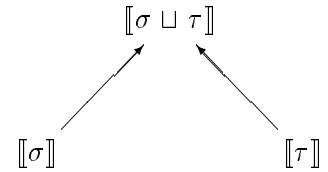


where ζ_1, ζ_2, etc. are all the common supertypes of σ and τ. (N.b. To avoid cluttering the picture, the ζs have been drawn as though they were mutually unrelated. Of course, the coercions between them must also be included in the limit diagram.)

If σ and τ happen to have a *least* upper bound $\sigma \sqcup \tau$ in the subtype ordering (again omitting subtype relations between the ζs),



then — because, to achieve coherence, all the composite arrows must commute — the diagram defining $[\![\sigma \wedge \tau]\!]$ may be summarized by a much simpler diagram with the same limit:

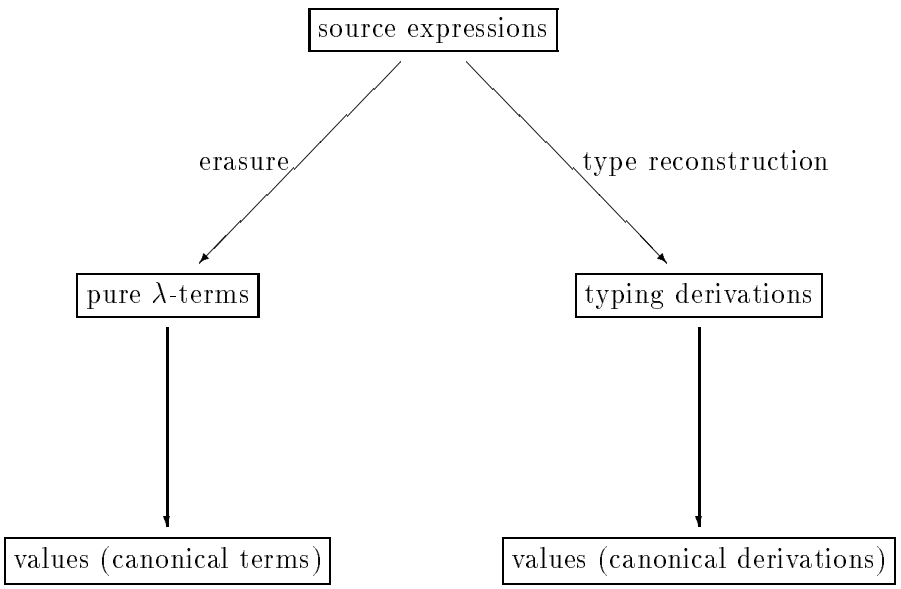


The limit of this diagram is precisely the categorical *pullback* of $[\![\sigma]\!]$ and $[\![\tau]\!]$ with respect to the coercions $[\![\sigma \leq \sigma \sqcup \tau]\!]$ and $[\![\tau \leq \sigma \sqcup \tau]\!]$.

The first-order calculus of intersection types, $\lambda_\wedge$, can indeed be shown to possess a least upper bound for every finite collection of types (under certain assumptions about the primitive subtype relation), so we can use the latter interpretation of intersections, appropriately generalized to n elements. The **S**-limits needed to make this construction well defined are guaranteed to exist if we require that **S** have limits of all finite diagrams.

2.4.3 Operational semantics

A similar sort of distinction may be drawn between typed and untyped styles of operational semantics:



An untyped operational semantics first strips a source expression e of any type annotations, yielding a pure λ-term M, which is then evaluated according to some collection of reduction rules to produce a *value* v (typically also a λ-term, though some formulations make use of auxiliary notions like closures). A typed operational semantics, on the other hand, begins from a typing derivation $\Gamma \vdash e \in \tau$. This derivation itself is then transformed according to some collection of evaluation rules to produce a derivation in a restricted *canonical form*. (Again, auxiliary notions like the closure of a derivation may also be needed.)

The notion of an operational semantics of typing derivations is somewhat unfamiliar, even though typed operational semantics have often been given for typed languages. The reason for this is that these semantics have usually been presented as operating on typed terms, not derivations. (Curien and Ghelli's rewriting systems on $F_{\leq}$ typing derivations [50, 51, 63] might be viewed as an exception.) This works because the calculi involved (explicitly typed presentations of system F for example) are degenerate in a certain sense: there is a one-to-one correspondence between valid typing statements and their typing derivations. We might say, in such cases, that the type systems involved are *unitary*.

2.4.4 Discussion

The key distinctions between the two views of typing can be summarized as follows:

Type Assignment Systems	Typed Calculi
Curry-style extrinsic epistemological	Church-style intrinsic ontological
"type annotations are directives to the typechecker"	"type annotations are part of the program"
untyped models	typed models
type inference	typechecking type reconstruction argument synthesis coercion reconstruction
subset interpretation of subtyping	coercion interpretation of subtyping
intersection = intersection	intersection = coherent overloading

2.4.4.1. Remark: In comparing Church-style and Curry-style semantics, we have been cautious in our usage of a number of terms that are sometimes employed to distinguish the two. In some neighborhoods of the programming language community where clear and rigorous meanings have been agreed on for these terms, they function as useful shorthands. But the field as a whole has seen them put to so many, often subtly different, uses that they can cause confusion among broader audiences.

The phrases *implicitly typed* and *explicitly typed calculus* — referring to the presence or absence of concrete-syntax annotations such as type abstractions, type applications, and our *for* construct — are particularly confusing, since Church-style systems tend to include such annotations and Curry-style systems tend to omit them. In the context of the above discussion, it can be seen that the other pairings also make perfect sense: a type assignment system may be formulated so that type annotations in source expressions limit the possible typing derivations involving them; conversely, a typed semantics may be given to a language where the entire burden of discovering typing derivations is placed on the typechecker. Indeed, Harper and Mitchell [69] have promoted

a view of ML, the quintessential implicitly typed polymorphic language, as a typed λ-calculus in the style of Church. Thus, the distinction between explicit and implicit typing should be viewed as a syntactic question ("what are the subjects of the type inference rules") and an algorithmic question ("how much and what kind of help must the programmer give to the typechecker"), but not as a semantic question.

We have also been careful to use the phrase "type inference" only in the context of systems with untyped semantic models. When the same term is used for typed systems, its historical association with Curry-style presentations tends to lead to confusion. We prefer "type reconstruction" for the typed case.

2.4.4.2. Remark: Intersection types are also sometimes called *conjunctive types*, but this terminology has fallen into disfavor because it suggests a false analogy with the "and" connective of intuitionistic logic. The well-known Curry-Howard isomorphism establishes a fruitful intuitive correspondence between the type constructors of λ-calculi and the connectives of intuitionistic logic: functional types correspond to logical implication, polymorphic types to polymorphic quantifiers, product types to logical conjunction, disjoint sum types to disjunction, and so on. Roughly speaking, a term of a given type can be viewed as *evidence* (in the sense of a constructive proof) of the truth of the corresponding logical proposition. The difference between the product type $\sigma \times \tau$ and the intersection $\sigma \wedge \tau$ is that an element of $\sigma \times \tau$ consists of *two* pieces of evidence, one establishing σ and the other establishing τ, whereas an element of $\sigma \wedge \tau$ consists of a *single* piece of evidence that establishes *both* σ and τ. (To be completely faithful to the explicitly typed perspective, we should go a step further and say that an element of the intersection type contains a piece of evidence that can be *coerced* to evidence for σ and, perhaps via a different coercion, to evidence for τ.)

2.5 Expressiveness of the Intersection Type Discipline

The expressive power of intersection types is clearly illustrated by the fact that they can be used to type many pure λ-terms that have no typed counterparts in $\lambda_{\leq}$. For example, it is easy to show that there the term $\lambda x.\ xx$ has no simple typing (i.e., there are no σ and τ such that $\vdash^{\lambda\leq} \lambda x{:}\sigma.\ xx \in \tau$). But in $\lambda_{\wedge}$ it has many types — for example,

```
> \x:A/\A->B. x x;
it : (A/\A->B) -> B
```

for any `A` and `B`.

This particular λ-term can also be typed in the pure polymorphic λ-calculus (for example, by the typed term $\lambda x{:}(\forall\alpha.\,\alpha{\to}\alpha).\,x\ [\forall\alpha.\,\alpha{\to}\alpha]\ x$), but it is known that there are strongly normalizing terms that cannot be given polymorphic typings [65]. By contrast, various formulations of intersection types can be used to exactly characterize the set of strongly normalizing terms and two similar important sets of pure terms (see [35] for a more complete discussion, sketches of proofs, and pointers to full proofs).

2.5.1. Remark: The classical results about the intersection type discipline have been proved for a type assignment system, not an explicitly typed calculus. Strictly speaking, they cannot be taken as having any immediate bearing on the explicitly typed calculi we consider here. For example, it does not make sense to talk about "typing terms of the pure λ-calculus" in an explicitly typed calculus whose semantic models are also typed, since there is no straightforward way to relate the semantics of a typed term to the semantics of its erasure. Nevertheless, these results do provide a good intuitive gauge of the expressive power we may expect from intersection types in typed calculi.

2.5.2. Definition:

1. The type assignment system $\vdash_{\wedge,\top,\leq}$ is the implicitly typed analog of $\lambda_\wedge$.
2. The type assignment system $\vdash_\wedge$ is the implicitly typed analog of a version of $\lambda_\wedge$ where the $\leq$ relation and the rule SUB are dropped in favor of an explicit INTER-E rule and the $\wedge$ constructor is restricted to the binary case.

2.5.3. Definition:

1. A type τ is *tail-proper* if $\tau \equiv \sigma_1 \to \sigma_2 \to \cdots \to \sigma_n \to \rho$ for some $n \geq 0$.
2. The *positive* and *negative occurrences* of a subphrase σ in a type τ are defined inductively as follows:
   - if $\tau \equiv \sigma$, then σ occurs positively in τ;
   - if σ occurs positively (resp. negatively) in τ_1, then it occurs negatively (positively) in $\tau_1 \to \tau_2$; if it occurs positively (negatively) in τ_2, then it occurs positively (negatively) in $\tau_1 \to \tau_2$;
   - if σ occurs positively (negatively) in τ_i, then it occurs positively (negatively) in $\bigwedge[\tau_1..\tau_n]$.
3. A type is *proper* if it contains only negative occurrences of $\top$, *anti-proper* if it contains only positive occurrences of $\top$, and *strictly proper* if it contains no occurrences at all of $\top$.
4. A context is proper (anti-proper, etc.) if its range contains only proper types.

2.5.4. Theorem: Let M be a pure λ-term. Then M is normalizable iff there is an anti-proper $\vdash_{\wedge,\top,\leq}$ context Γ and a proper type σ such that $\Gamma \vdash_{\wedge,\top,\leq} M \in \sigma$.

2.5.5. Theorem: Let M be a pure λ-term. Then M is head-normalizable iff there is some $\vdash_{\wedge,\top,\leq}$ context Γ and a tail-proper type σ such that $\Gamma \vdash_{\wedge,\top,\leq} M \in \sigma$.

2.5.6. Theorem: Let M be a pure λ-term. Then M is strongly normalizing iff there is a context Γ in $\vdash_\wedge$ and a strictly proper type σ such that $\Gamma \vdash_\wedge M \in \sigma$.

2.5.7. Remark: An easy corollary of these theorems is that the full type inference problem for the type assignment presentation of intersection types is undecidable.

2.5.8. Remark: The classical intersection type discipline can be formulated without the subtype relation, distributivity law, or nullary intersections, while retaining much of the expressive power of the full-blown systems with these features. This leads one to wonder whether a simpler formulation would also suffice for the uses of intersection types in a programming language setting, although, for the sake of generality in this thesis we have chosen to deal with the richer, more complex formulation. This point is discussed in more detail in Section 3.4.1.

2.6 Bounded Polymorphism

The notion of *bounded quantification* was introduced by Cardelli and Wegner [33] in the language Fun. Based on informal ideas by Cardelli and formalized using techniques developed by Mitchell [94], Fun integrated Girard-Reynolds polymorphism [66, 116] with Cardelli's first-order calculus of subtyping [20, 23].

Fun and its relatives have been studied extensively by programming language theorists and designers. Cardelli and Wegner's survey paper gives the first programming examples using bounded quantification; more are developed in Cardelli's study of power kinds [24]. Curien and Ghelli [50,

63] address a number of syntactic properties of $F_{\leq}$. Semantic aspects of closely related systems have been studied by Bruce and Longo [12], Martini [91], Breazu-Tannen, Coquand, Gunter, and Scedrov [10], Cardone [34], Cardelli and Longo [29], Cardelli, Martini, Mitchell, and Scedrov [30], Curien and Ghelli [50, 51], and Bruce and Mitchell [14]. $F_{\leq}$ has been extended to include record types and richer notions of inheritance by Cardelli and Mitchell [32], Bruce [11], Cardelli [26], and Canning, Cook, Hill, Olthoff, and Mitchell [18]. Bounded quantification also plays a key role in Cardelli's programming language Quest [25, 29] and in the Abel language developed at HP Labs [17, 18, 19, 39].

The original Fun was simplified and slightly generalized by Bruce and Longo [12], and again by Curien and Ghelli [50]. Curien and Ghelli's formulation, called *minimal Bounded Fun* or $F_{\leq}$ ("F-sub"), is the one considered here.

Like other second-order λ-calculi, the terms of $F_{\leq}$ include the variables, abstractions, and applications of $\lambda_{\leq}$, plus the type abstractions and type applications of the second-order λ-calculus — slightly refined to take account of the subtype relation: each type abstraction gives a *bound* for the type variable it introduces and each type application must satisfy the constraint that the argument type is a subtype of the bound of the polymorphic function being applied. Also, like that of $\lambda_{\wedge}$, the $F_{\leq}$ subtype ordering includes a maximal element. (Since the two are not exactly the same (c.f. 3.4.1), the maximal $F_{\leq}$ type is called here by its conventional name, *Top*, instead of $\top$.)

$$\begin{array}{lcl} \tau & ::= & \mathit{Top} \mid \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha{\leq}\tau_1.\ \tau_2 \\ e & ::= & x \mid \lambda x{:}\tau.\ e \mid e_1\ e_2 \mid \Lambda \alpha{\leq}\tau.\ e \mid e[\tau] \end{array}$$

To accomodate the subtyping assumptions introduced by type abstractions, we enrich the notion of "context" to include bindings for both term variables (as before) and type variables:

2.6.1. Definition: A *context* is a finite sequence of distinct type variables with associated bounds and term variables with associated types:

$$\Gamma \quad ::= \quad \{\} \mid \Gamma, \alpha{\leq}\tau \mid \Gamma, x{:}\tau$$

The function $dom(\Gamma)$ gives the set of type and term variables defined by Γ. $\Gamma(\alpha)$ denotes the bound of α in Γ if it has one; $\Gamma(x)$ denotes the type of x in Γ if it has one.

2.6.2. Definition: The set of *free type variables* of a type τ or a term e is written $FTV(\tau)$ or $FTV(e)$. The set of free type variables of a context Γ is the union of the sets of free type variables of the elements of the range of Γ. The set of *all* type variables in a type τ is written $TV(\tau)$.

2.6.3. Definition: A type τ is *closed* with respect to a context Γ if $FTV(\tau) \subseteq dom(\Gamma)$. A term e is closed with respect to Γ if $FTV(e) \cup FV(e) \subseteq dom(\Gamma)$. A context Γ is closed if

1. $\Gamma \equiv \{\}$, or
2. $\Gamma \equiv \Gamma_1, \alpha{\leq}\tau$, with Γ_1 closed and τ closed with respect to Γ_1, or
3. $\Gamma \equiv \Gamma_1, x{:}\tau$, with Γ_1 closed and τ closed with respect to Γ_1.

A subtyping statement $\Gamma \vdash \sigma \leq \tau$ is closed if Γ is closed and σ and τ are closed with respect to Γ; a typing statement $\Gamma \vdash e \in \tau$ is closed if Γ is closed and e and τ are closed with respect to Γ

2.6.4. Convention: (c.f. 2.2.7) In the following, we assume that all statements under discussion are closed. In particular, we allow only closed statements in instances of inference rules.

Type abstractions have almost the same typing rule as in other second-order λ-calculi; they are checked by moving their stated bound for the type variable they introduce into the context and checking the body of the abstraction under the enriched set of assumptions:

$$\frac{\Gamma, \alpha{\leq}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \Lambda\alpha{\leq}\tau_1.\ e \in \forall\alpha{\leq}\tau_1.\ \tau_2} \qquad \text{(All-I)}$$

Type applications check that the type being passed as a parameter is indeed a subtype of the bound of the corresponding quantifier:

$$\frac{\Gamma \vdash e \in \forall\alpha{\leq}\tau_1.\ \tau_2 \qquad \Gamma \vdash \tau \leq \tau_1}{\Gamma \vdash e[\tau] \in \{\tau/\alpha\}\tau_2} \qquad \text{(All-E)}$$

The subtype relation of $\lambda_{\leq}$ is also extended with several rules. First, we stipulate that *Top* is a maximal element of the subtype order:

$$\Gamma \vdash \sigma \leq \mathit{Top} \qquad \text{(Sub-Top)}$$

(One of the main uses of *Top* — in fact, the original reason it was introduced by Cardelli and Wegner — is to recover ordinary unbounded quantification as a special case of bounded quantification: $\forall\alpha.\,\tau$ becomes $\forall\alpha{\leq}\mathit{Top}.\ \tau$.)

Type variables are subtypes of the bounds given for them in the prevailing context:

$$\Gamma \vdash \alpha \leq \Gamma(\alpha) \qquad \text{(Sub-TVar)}$$

Like arrow types, subtyping of quantified types is contravariant in their bounds and covariant in their bodies:

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma, \alpha{\leq}\tau_1 \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \forall\alpha{\leq}\sigma_1.\ \sigma_2 \leq \forall\alpha{\leq}\tau_1.\ \tau_2} \qquad \text{(Sub-All)}$$

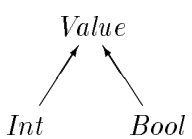
This rule deserves a closer look, since it causes considerable difficulties (c.f. Chapter 6 in particular). Intuitively, it reads as follows:

> A type $\tau \equiv \forall\alpha{\leq}\tau_1.\ \tau_2$ describes a collection of polymorphic values (functions from types to values), each mapping subtypes of τ_1 to instances of τ_2. If τ_1 is a subtype of σ_1, then the domain of τ is smaller than that of $\sigma \equiv \forall\alpha{\leq}\sigma_1.\ \sigma_2$, so σ is a stronger constraint and describes a smaller collection of polymorphic values. Moreover, if, for each type θ that is an acceptable argument to the functions in both collections (i.e., one that satisfies the more stringent requirement $\theta \leq \tau_1$), the θ-instance of σ_2 is a subtype of the θ-instance of τ_2, then σ is a "pointwise stronger" constraint and again describes a smaller collection of polymorphic values.

Thus, quantified types may be thought of as a kind of function spaces. We sometimes abuse this analogy and speak of the body and bound of a quantified type as its "left-hand" and "right-hand" sides.

2.6.5. Convention: When necessary to prevent confusion, turnstiles in $F_{\leq}$ derivations are written $\vdash^{\leq}$.

2.6.6. Remark: (c.f. 2.2.10) Since contexts now include assumptions about free type variables, we can drop the separate notion of "primitive types." For each primitive type of $\lambda_{\leq}$, we add a variable with the same name to the pervasive context Γ_P. The bounds given to these variables encode the subtype relation $\leq_P$. For example, the primitive subtype relation



is represented by the pervasive context

$$\Gamma_P \equiv \mathit{Value} \leq \mathit{Top},\ \mathit{Int} \leq \mathit{Value},\ \mathit{Bool} \leq \mathit{Value}.$$

Note, however, that this encoding sacrifices a degree of flexibility: it will only work for "tree-shaped" order structures where each primitive type has at most one immediate parent; for example, the preorders

cannot be encoded in this way. Luckily, some of this lost flexibility is regained in $F_\wedge$, which allows cycle-free directed graphs like the one on the left to be encoded as well.

2.6.7. Remark: The encoding of primitive types as type variables also results in a modest *gain* in expressiveness over their formulation in $\lambda_\leq$ and $\lambda_\wedge$: in both of these calculi, the only things that can be supertypes of primitive types are other primitive types (and, in $\lambda_\wedge$, intersections involving only primitive types). Here, the upper bound given in Γ_P for a variable may be any $F_\leq$ type whose free variables are all defined to the left of the variable being introduced.

This extra freedom might be used, for example, to express the fact that natural numbers can be viewed as iterators (c.f. Section 7.7.1), in terms of an implicit coercion from a primitive type *Nat* to the appropriate polymorphic type:

$$\Gamma_P(\mathit{Nat}) = \forall\alpha.\ (\alpha\to\alpha)\to\alpha\to\alpha$$

2.6.8. Definition: We use the following abbreviations in examples:

$$\begin{array}{lcl} \forall\alpha.\ \tau & \stackrel{\text{def}}{=} & \forall\alpha\leq\mathit{Top}.\ \tau \\ \forall\alpha_1\leq\phi_1..\alpha_n\leq\phi_n.\ \tau & \stackrel{\text{def}}{=} & \forall\alpha_1\leq\phi_1.\ ..\forall\alpha_n\leq\phi_n.\ \tau. \end{array}$$

The rules defining $F_\leq$ do not directly constitute an algorithm for checking the subtype relation, since they are not syntax-directed. In particular, the rule TRANS cannot effectively be applied backwards, since this would involve "guessing" an appropriate value for the intermediate type τ_2. Curien and Ghelli (as well as Cardelli and others) use the following reformulation:

2.6.9. Definition: $F^N_\leq$ ("N" for "normal form") is the least relation closed under the following rules:

$$\Gamma \vdash \sigma \leq \mathit{Top} \qquad \text{(NTOP)}$$

$$\Gamma \vdash \alpha \leq \alpha \qquad \text{(NREFL)}$$

$$\frac{\Gamma \vdash \Gamma(\alpha) \leq \tau}{\Gamma \vdash \alpha \leq \tau} \qquad \text{(NVAR)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \sigma_1\to\sigma_2 \leq \tau_1\to\tau_2} \qquad \text{(NARROW)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma, \alpha{\leq}\tau_1 \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \forall\alpha{\leq}\sigma_1.\ \sigma_2 \leq \forall\alpha{\leq}\tau_1.\ \tau_2} \quad (\textsc{NAll})$$

The reflexivity rule here is restricted to type variables. Transitivity is eliminated, except for instances of the form

$$\frac{\Gamma \vdash \alpha \leq \Gamma(\alpha) \qquad \Gamma \vdash \Gamma(\alpha) \leq \tau}{\Gamma \vdash \alpha \leq \tau,}$$

which are packaged together as instances of the new rule NVar.

2.6.10. Definition: The rules defining $F^N_{\leq}$ may be read as an algorithm (i.e., a recursively defined procedure) for checking the subtype relation:

$check(\Gamma \vdash \sigma \leq \tau) =$

1. *if* $\tau \equiv Top$
   then true
2. *else if* $\sigma \equiv \sigma_1{\to}\sigma_2$ *and* $\tau \equiv \tau_1{\to}\tau_2$
   then $check(\Gamma \vdash \tau_1 \leq \sigma_1)$
   and $check(\Gamma \vdash \sigma_2 \leq \tau_2)$
3. *else if* $\sigma \equiv \forall\alpha{\leq}\sigma_1.\ \sigma_2$ *and* $\tau \equiv \forall\alpha{\leq}\tau_1.\ \tau_2$
   then $check(\Gamma \vdash \tau_1 \leq \sigma_1)$
   and $check(\Gamma, \alpha{\leq}\tau_1 \vdash \sigma_2 \leq \tau_2)$
4. *else if* $\sigma \equiv \alpha$ *and* $\tau \equiv \alpha$
   then true
5. *else if* $\sigma \equiv \alpha$
   then $check(\Gamma \vdash \Gamma(\alpha) \leq \tau)$

n. *else*
   false.

We write $F^N_{\leq}$ to refer either to the algorithm or to the inference system, depending on context.

2.6.11. Lemma: [Curien and Ghelli] The relations $F_{\leq}$ and $F^N_{\leq}$ coincide: $\Gamma \vdash \sigma \leq \tau$ is derivable in $F_{\leq}$ iff it is derivable in $F^N_{\leq}$.

The algorithm $F^N_{\leq}$ may be thought of as incrementally attempting to build a normal form derivation of a statement J, starting from the root and recursively building subderivations for the premises. By Lemma 2.6.11, if there is any derivation whatsoever of a statement J, there is one in normal form; the algorithm is guaranteed to recapitulate this derivation and halt in finite time.

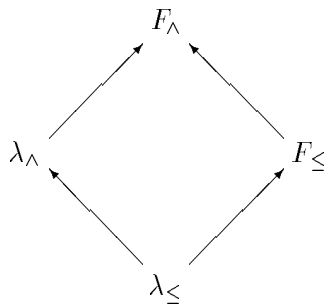
2.6.12. Fact: [Curien and Ghelli] $\Gamma \vdash \sigma \leq \tau$ is derivable in $F^N_{\leq}$ iff the algorithm $F^N_{\leq}$ halts and returns *true* when given this statement as input.

Unfortunately, the algorithm is *not* a decision procedure for the subtype relation. Indeed, the main result of Chapter 6 is that this relation is undecidable. We shall see, however, that the convergence of the algorithm in practice is perfectly acceptable (c.f. 6.12).

Chapter 3

The $F_\wedge$ Calculus

This chapter introduces $F_\wedge$, an explicitly typed second-order lambda calculus with bounded quantification and intersection types. $F_\wedge$ can roughly be characterized as the union of the concrete syntax and typing rules for the systems $\lambda_\wedge$ and $F_\leq$:



To achieve a compact and symmetric calculus, however, a few small modifications and extensions are needed:

- Since $F_\leq$ allows primitive types to be encoded as elements of the pervasive context, we drop the primitive types of $\lambda_\leq$ and $\lambda_\wedge$ and the rule Sub-Prim.
- Since $\top$ and *Top* both function as maximal elements of their respective subtype orderings, we drop *Top* and let $\top$ take over its job. Section 3.4 discusses this design decision in more detail.
- Since $\forall$ behaves like a kind of function space constructor, we add a new law Sub-Dist-IQ, analogous to Sub-Dist-IA, which allows intersections to be distributed over quantifiers on the right-hand side.
- We use the ordinary form of λ-abstraction from $F_\leq$ rather than the generalized one introduced by Reynolds for $\lambda_\wedge$. The notion of deriving alternative typings for subphrases under different sets of assumptions is captured by a new syntactic form, *for*.

The *for* construct is described in detail in Section 3.1. Section 3.2 then summarizes the concrete syntax, subtyping rules, and typing rules of $F_\wedge$. Section 3.3 introduces a linear shorthand for $F_\wedge$ derivations. Section 3.4 discusses the major design choices in the formulation of $F_\wedge$.

3.1 Explicit Alternation: The *for* Construct

The notions of type variables and type substitution inherited from $F_{\leq}$ can be used to define an elegant generalization of the alternation inherent in $\lambda_{\wedge}$'s generalized λ-abstraction construct. We extend the concrete syntax of terms with a *for* form

$$e \quad ::= \quad \ldots \quad | \quad for\,\alpha\ in\,\sigma_1..\sigma_n.\ e$$

whose typing rule allows a choice of any of the σ's as a replacement for α in the body:

$$\frac{\Gamma \vdash \{\sigma_i/\alpha\}e \in \tau_i}{\Gamma \vdash for\,\alpha\ in\,\sigma_1..\sigma_n.\ e \in \tau_i} \qquad \text{(For)}$$

This rule, like the generalized arrow introduction rule Arrow-I' of $\lambda_{\wedge}$, can be used together with Inter-I to generate a set of up to n alternative typings for the body and then form their intersection as the type of the whole *for* expression:

```
> double = for A in Int,Real. \x:A. plus x x;
double : Int->Int /\ Real->Real
```

Indeed, $\lambda_{\wedge}$'s generalized λ-abstraction may be reintroduced as a simple syntactic abbreviation:

3.1.1. Notation: $\lambda x{:}\sigma_1..\sigma_n.\ e \stackrel{\text{def}}{=} for\,\alpha\ in\,\sigma_1..\sigma_n.\ \lambda x{:}\alpha.\ e,$ where α is fresh.

Besides separating the mechanisms of functional abstraction and alternation, the introduction of the *for* construct extends the expressive power of the language by providing a *name* for the "current choice" being made by the type checker:

```
> for A in Int,Real.
>   \\B<A. \f:A->B. \x:A.
>     f (double x);
it : All B<Int. (Int->B)->Int->B /\ All B<Real. (Real->B)->Real->B
```

Indeed, the finer control over alternation allowed by the explicit *for* construct may be used to improve the efficiency of typechecking even for first-order languages with intersections, like Forsythe. For example, Forsythe's generalized λ-abstraction allows the definition of polynomial functions like the following:

```
> poly =
>   \w:Int,Real. \x:Int,Real. \y:Int,Real. \z:Int,Real.
>       plus (double x) (plus (plus w y) z);
poly : Int->Int->Int->Int->Int /\ Real->Real->Real->Real->Real
```

But the behavior of the typechecker on this program is unnecessarily inefficient. It is easy to see, from the types of `plus` and `double`, that when all the arguments to `poly` have type `Int`, the result type will be `Int`, and that otherwise the result type will be `Real`. So if we choose `Real` for the type of any of the four parameters, we might as well choose `Real` for the others too. We can realize a substantial gain in typechecking efficiency by making this observation explicit with a *for* expression:

```
> poly =
>   for A in Int,Real.
>     \w:A. \x:A. \y:A. \z:A.
>         plus (double x) (plus (plus w y) z);
poly : Int->Int->Int->Int->Int /\ Real->Real->Real->Real->Real
```

The second version of `poly` requires that the body be checked only twice, as compared to sixteen times for the first version.

3.2 Syntax, Subtyping, and Typing

We now give a precise definition of the $F_\wedge$ calculus, which forms the main object of study for the remainder of the thesis. Since all of its components have already been discussed in detail, we present just the bare facts here. (These definitions are also summarized in Appendix A for easy reference.)

3.2.1. Definition: The set of $F_\wedge$ types is defined by the following abstract grammar:

$$\begin{array}{lcl} \tau & ::= & \alpha \\ & | & \tau_1 \to \tau_2 \\ & | & \forall \alpha{\leq}\tau_1.\ \tau_2 \\ & | & \bigwedge[\tau_1..\tau_n] \end{array}$$

3.2.2. Definition: The set of $F_\wedge$ terms is defined by the following abstract grammar:

$$\begin{array}{lcl} e & ::= & x \\ & | & \lambda x{:}\tau.\ e \\ & | & e_1\ e_2 \\ & | & \Lambda \alpha{\leq}\tau.\ e \\ & | & e[\tau] \\ & | & \mathit{for}\ \alpha\ \mathit{in}\ \tau_1..\tau_n.\ e \end{array}$$

3.2.3. Definition: The three-place $F_\wedge$ *subtype relation* $\Gamma \vdash \sigma \leq \tau$ is the least relation closed under the following rules:

$$\Gamma \vdash \tau \leq \tau \qquad \text{(Sub-Refl)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \tau_2 \qquad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3} \qquad \text{(Sub-Trans)}$$

$$\Gamma \vdash \alpha \leq \Gamma(\alpha) \qquad \text{(Sub-TVar)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2} \qquad \text{(Sub-Arrow)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma, \alpha{\leq}\tau_1 \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \forall \alpha{\leq}\sigma_1.\ \sigma_2 \leq \forall \alpha{\leq}\tau_1.\ \tau_2} \qquad \text{(Sub-All)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash \sigma \leq \tau_i}{\Gamma \vdash \sigma \leq \bigwedge[\tau_1..\tau_n]} \qquad \text{(Sub-Inter-G)}$$

$$\Gamma \vdash \bigwedge[\tau_1..\tau_n] \leq \tau_i \qquad \text{(Sub-Inter-LB)}$$

$$\Gamma \vdash \bigwedge[\sigma \to \tau_1\ ..\ \sigma \to \tau_n] \leq \sigma \to \bigwedge[\tau_1..\tau_n] \qquad \text{(Sub-Dist-IA)}$$

$$\Gamma \vdash \bigwedge[\forall \alpha{\leq}\sigma.\ \tau_1\ ..\ \forall \alpha{\leq}\sigma.\ \tau_n] \leq \forall \alpha{\leq}\sigma.\ \bigwedge[\tau_1..\tau_n] \qquad \text{(Sub-Dist-IQ)}$$

3.2.4. Definition: The three-place $F_\wedge$ *typing relation* $\Gamma \vdash e \in \tau$ is the least relation closed under the following rules:

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(Var)}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ e \in \tau_1 \to \tau_2} \qquad \text{(Arrow-I)}$$

$$\frac{\Gamma \vdash e_1 \in \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash e_1\ e_2 \in \tau_2} \qquad \text{(Arrow-E)}$$

$$\frac{\Gamma, \alpha{\leq}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \Lambda\alpha{\leq}\tau_1.\ e \in \forall\alpha{\leq}\tau_1.\ \tau_2} \quad \text{(All-I)}$$

$$\frac{\Gamma \vdash e \in \forall\alpha{\leq}\tau_1.\ \tau_2 \qquad \Gamma \vdash \tau \leq \tau_1}{\Gamma \vdash e[\tau] \in \{\tau/\alpha\}\tau_2} \quad \text{(All-E)}$$

$$\frac{\Gamma \vdash \{\sigma_i/\alpha\}e \in \tau_i}{\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e \in \tau_i} \quad \text{(For)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash e \in \tau_i}{\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]} \quad \text{(Inter-I)}$$

$$\frac{\Gamma \vdash e \in \tau_1 \qquad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e \in \tau_2} \quad \text{(Sub)}$$

3.2.5. Convention: When necessary to prevent confusion, turnstiles in $F_\wedge$ derivations are written $\vdash^\wedge$.

3.3 Linear Notation for Derivations

It is convenient to have a linear notation for typing and subtyping derivations, so that operations on proofs such as cut-elimination transformations can be expressed as textual rules rather than as pictures. Our notation is a modified version of Curien and Ghelli's [50].

3.3.1. Definition: The sets of *subtyping derivation abbreviations* c and *typing derivation abbreviations* s are defined by the following abstract grammar:

$$\begin{array}{lcl}
c & ::= & \mathit{id} \\
& | & c_1\ ;\ c_2 \\
& | & V_\alpha \\
& | & c_1{\rightarrow}c_2 \\
& | & \forall\alpha{\leq}c_1.\ c_2 \\
& | & \langle c_1\ ..\ c_n\rangle \\
& | & \mathit{proj}_i \\
& | & \mathit{dist\text{-}ia} \\
& | & \mathit{dist\text{-}iq}
\end{array}$$

$$\begin{array}{lcl}
s & ::= & V_x \\
& | & \lambda x{:}\sigma.\ s \\
& | & s_1\ s_2 \\
& | & \Lambda\alpha{\leq}\sigma.\ s \\
& | & s\ [c] \\
& | & \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ s_i \\
& | & \langle s_1\ ..\ s_n\rangle \\
& | & c \uparrow s
\end{array}$$

3.3.2. Definition: The translation function $(\text{—})^\dagger$, which maps derivation trees to their abbreviated forms, is defined as follows. (Recall that $c :: J$ is read as "c is a derivation whose conclusion is the judgement J.")

$$\left(\frac{\text{(SUB-REFL)}}{\Gamma \vdash \tau \leq \tau}\right)^\dagger \quad = \quad \mathit{id}$$

$$\left(\frac{c_1 :: \Gamma \vdash \tau_1 \le \tau_2 \qquad c_2 :: \Gamma \vdash \tau_2 \le \tau_3}{\Gamma \vdash \tau_1 \le \tau_3}\,\text{(SUB-TRANS)}\right)^\dagger \quad = \quad c_1^\dagger \,;\, c_2^\dagger$$

$$\left(\frac{}{\Gamma \vdash \alpha \le \Gamma(\alpha)}\,\text{(SUB-TVAR)}\right)^\dagger \quad = \quad V_\alpha$$

$$\left(\frac{c_1 :: \Gamma \vdash \tau_1 \le \sigma_1 \qquad c_2 :: \Gamma \vdash \sigma_2 \le \tau_2}{\Gamma \vdash \sigma_1 \to \sigma_2 \le \tau_1 \to \tau_2}\,\text{(SUB-ARROW)}\right)^\dagger \quad = \quad c_1^\dagger \to c_2^\dagger$$

$$\left(\frac{c_1 :: \Gamma \vdash \tau_1 \le \sigma_1 \qquad c_2 :: \Gamma, \alpha \le \tau_1 \vdash \sigma_2 \le \tau_2}{\Gamma \vdash \forall\alpha \le \sigma_1.\ \sigma_2 \le \forall\alpha \le \tau_1.\ \tau_2}\,\text{(SUB-ALL)}\right)^\dagger \quad = \quad \forall\alpha \le c_1^\dagger.\ c_2^\dagger$$

$$\left(\frac{\text{for all } i,\ c_i :: \Gamma \vdash \sigma \le \tau_i}{\Gamma \vdash \sigma \le \bigwedge[\tau_1..\tau_n]}\,\text{(SUB-INTER-G)}\right)^\dagger \quad = \quad \langle c_1^\dagger..c_n^\dagger \rangle$$

$$\left(\frac{}{\Gamma \vdash \bigwedge[\tau_1..\tau_n] \le \tau_i}\,\text{(SUB-INTER-LB)}\right)^\dagger \quad = \quad proj_i$$

$$\left(\frac{}{\Gamma \vdash \bigwedge[\sigma \to \tau_1\,..\sigma \to \tau_n] \le \sigma \to \bigwedge[\tau_1..\tau_n]}\,\text{(SUB-DIST-IA)}\right)^\dagger \quad = \quad dist\text{-}ia$$

$$\left(\frac{}{\Gamma \vdash \bigwedge[\forall\alpha \le \sigma.\ \tau_1\,..\forall\alpha \le \sigma.\ \tau_n] \le \forall\alpha \le \sigma.\ \bigwedge[\tau_1..\tau_n]}\,\text{(SUB-DIST-IQ)}\right)^\dagger \quad = \quad dist\text{-}iq$$

$$\left(\frac{}{\Gamma \vdash x \in \Gamma(x)}\,\text{(VAR)}\right)^\dagger \quad = \quad V_x$$

$$\left(\frac{s :: \Gamma, x{:}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ e \in \tau_1 \to \tau_2}\,\text{(ARROW-I)}\right)^\dagger \quad = \quad \lambda x{:}\tau_1.\ s^\dagger$$

$$\left(\frac{s_1 :: \Gamma \vdash e_1 \in \tau_1 \to \tau_2 \qquad s_2 :: \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash e_1\ e_2 \in \tau_2}\,\text{(ARROW-E)}\right)^\dagger \quad = \quad s_1^\dagger\ s_2^\dagger$$

$$\left(\frac{s :: \Gamma, \alpha \le \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \Lambda\alpha \le \tau_1.\ e \in \forall\tau_1.\ \tau_2}\,\text{(ALL-I)}\right)^\dagger \quad = \quad \Lambda\alpha \le \tau_1.\ s^\dagger$$

$$\left(\frac{s :: \Gamma \vdash e \in \forall\alpha \le \tau_1.\ \tau_2 \qquad c :: \Gamma \vdash \tau \le \tau_1}{\Gamma \vdash e\ [\tau] \in \{\tau/\alpha\}\tau_2}\,\text{(ALL-E)}\right)^\dagger \quad = \quad s^\dagger\ [c^\dagger]$$

$$\left(\frac{s_i :: \Gamma \vdash \{\sigma_i/\alpha\}e \in \tau_i}{\Gamma \vdash for\,\alpha\ in\,\sigma_1..\sigma_n.\ e \in \tau_i}\,\text{(FOR)}\right)^\dagger \quad = \quad for\,\alpha\ in\,\sigma_1..\sigma_n.\ s_i^\dagger$$

$$\left(\frac{\text{for all } i,\ s_i :: \Gamma \vdash e \in \tau_i}{\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]}\,\text{(INTER-I)}\right)^\dagger \quad = \quad \langle s_i^\dagger..s_n^\dagger \rangle$$

$$\left(\frac{s :: \Gamma \vdash e \in \sigma \qquad c :: \Gamma \vdash \sigma \le \tau}{\Gamma \vdash e \in \tau}\,\text{(SUB)}\right)^\dagger \quad = \quad s^\dagger \uparrow c^\dagger$$

For example, the subtyping derivation

$$\dfrac{\dfrac{}{\Gamma_P \vdash Int \leq Real}\,\text{(SUB-TVAR)} \qquad \dfrac{\dfrac{}{\Gamma_P, \alpha{\leq}Int \vdash \alpha \leq \alpha}\,\text{(SUB-REFL)} \qquad \dfrac{}{\Gamma_P, \alpha{\leq}Int \vdash Int \leq \top}\,\text{(SUB-INTER-G)}}{\Gamma_P, \alpha{\leq}Int \vdash \alpha{\to}Int \leq \alpha{\to}\top}\,\text{(SUB-ARROW)}}{\Gamma_P \vdash \forall\alpha{\leq}Real.\ \alpha{\to}Int \leq \forall\alpha{\leq}Int.\ \alpha{\to}\top}\,\text{(SUB-ALL)}$$

is abbreviated by the linear shorthand

$$\forall\alpha{\leq}V_{Int}.\ id{\to}\langle\rangle.$$

3.3.3. Remark: Strictly speaking, our linear abbreviations should contain sufficient information that they uniquely determine proof trees; in other words, $(\text{—})^\dagger$ should be injective. Clearly, we could decorate our linear abbreviations with additional information and extend $(\text{—})^\dagger$ to a bijection. For example, each abbreviation could include an explicit indication of the judgement it derives. However, this would make the abbreviations much larger and less readable, eliminating most of the benefit of introducing them in the first place. We therefore use the present abbreviatory forms *as if* they contained sufficient information to unique determine derivation trees, relying on the reader to imagine the necessary annotations.

Again, when we need to be explicit about the conclusion of a derivation written in linear form, we use the notation $c :: J$.

3.4 Discussion

We pause now to discuss the design choices that arise in the formulation of $F_\wedge$ and explore some of its properties.

3.4.1 *Top* vs. $\top$

In forming $F_\wedge$ from $\lambda_\wedge$ and $F_\leq$, we find, pleasantly, that most of their features are quite orthogonal: for a given feature, either it is found already in $\lambda_\leq$ or else it exists in either $\lambda_\wedge$ or $F_\leq$ in a form that interacts smoothly with all the features of the other. The one exception is the maximal types $\top$ and *Top*. In the best case, we might hope that these would coincide in $F_\wedge$, but this, unfortunately, is not the case.

The difference arises from the INTER-I rule of $\lambda_\wedge$, which, in its nullary form, states that any term whatsoever has type $\top$. $F_\leq$ has no such rule: the only way a term e can be assigned type *Top* is by the rules SUB and SUB-TOP, which require that the term already have some type σ with $\sigma \leq \tau$. In other words,

- *Top* is the type of all *well-typed* terms;
- $\top$ is the type of *all* terms.

Order-theoretically, the two types are equivalent (each is a subtype of the other), since each is explicitly axiomatized as a maximal type.

For the sake of conceptual parsimony, we drop *Top* here and retain $\top$, since $\top$ can perform the same jobs as *Top* (in particular, it allows unbounded quantification to be recovered from bounded quantification), while requiring no extra typing or subtyping rules beyond those defining the behavior of general n-ary intersections.

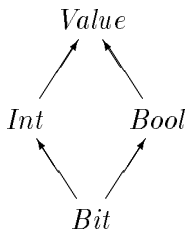
The alternative system with both $\top$ and *Top*, though messier and probably not much more expressive than $F_\wedge$, makes reasonable syntactic sense and does not seem much harder to typecheck.

More interesting, though, would be the system with *Top* instead of $\top$, where intersections are restricted to two or more elements: this language would have much of the practical expressiveness of $F_\wedge$ while avoiding the unfamiliar notion of a "type of all terms, even ill-behaved ones." This system supports the notion of *typechecking failure*, which in $F_\wedge$ is simply equated with $\top$.

3.4.2 Encoding Primitive Subtyping

As in $F_\leq$, both term and type constants are absent from $F_\wedge$, and programs involving them are expressed as terms with free variables whose typing and subtyping behavior are declared in a pervasive context Γ_P.

The presence of intersection types allows a greater variety of primitive subtype relations to be encoded in this way than was possible in $F_\leq$. For example, the relation



is represented by the pervasive context

$$\Gamma_P \equiv \mathit{Value} \leq \top,\ \mathit{Int} \leq \mathit{Value},\ \mathit{Bool} \leq \mathit{Value},\ \mathit{Bit} \leq \mathit{Int} \wedge \mathit{Bool}.$$

Primitive subtype order structures that are not partially ordered, such as

$$\mathit{PolarComplex} \rightleftharpoons \mathit{RectComplex}$$

are still not expressible, though.

The encodability of primitive subtype relations can be stated formally as follows:

3.4.2.1. Definition: A *topological sort* of a finite collection of primitive types P is a bijective mapping $\mathit{index}_P \in P \to \{1\ ..\ |P|\}$ such that $\alpha \leq_P \beta$ implies $\mathit{index}_P(\alpha) \geq \mathit{index}_P(\beta)$.

3.4.2.2. Definition: Let P be a collection of primitive types topologically sorted by index_P. Let $\alpha_i \equiv \mathit{index}_P^{-1}(i)$. Then P is *encoded* by the following $F_\wedge$ context:

$$\Gamma_P = \ldots, \alpha_i \leq \bigwedge[\gamma \in P \mid \alpha_i \leq_P \gamma],\ \ldots$$

3.4.2.3. Remark: Note that every finite partial order can be topologically sorted.

3.4.2.4. Lemma: Let Γ be an $\lambda_\wedge$ context and assume that the primitive subtype relation $\leq_P$ can be topologically sorted by some function index_P. If $\Gamma \vdash^{\lambda\wedge} \sigma \leq \tau$, then $\Gamma_P, \Gamma \vdash^{\wedge} \sigma \leq \tau$.

Proof: By induction on the structure of the given derivation. All of the $\lambda_\wedge$ rules translate directly into $F_\wedge$ rules, except for SUB-PRIM; an instance of this rule with conclusion $\Gamma \vdash \alpha \leq \beta$ is translated into the following $F_\wedge$ derivation:

$$\dfrac{\dfrac{}{\Gamma_P, \Gamma \vdash^{\wedge} \alpha \leq \bigwedge[\gamma \in P \mid \alpha \leq_P \gamma]}\ (\text{SUB-TVAR}) \qquad \dfrac{}{\Gamma_P, \Gamma \vdash^{\wedge} \bigwedge[\gamma \in P \mid \alpha \leq_P \gamma] \leq \beta}\ (\text{SUB-INTER-LB})}{\Gamma_P, \Gamma \vdash^{\wedge} \alpha \leq \beta.}\ (\text{SUB-TRANS})$$

□

3.4.2.5. Lemma: Let Γ by a $\lambda_\wedge$ context and assume that the primitive subtype relation $\leq_P$ can be topologically sorted by some function index_P. Then $\Gamma \vdash^{\lambda\wedge} e \in \tau$ only if $\Gamma_P, \Gamma \vdash^{\wedge} e \in \tau$.

(In fact, the derivation-normalization results of the following chapter can be used to show the converse, so $\Gamma \vdash^{\lambda\wedge} e \in \tau$ iff $\Gamma_P, \Gamma \vdash^{\wedge} e \in \tau$.)

3.5 Alternative Formulations

The formulation of $F_\wedge$ presented in Section 3.2 represents a natural combination of the most elegant formulations of pure bounded quantification and first-order intersection types. However, a few alternative formulations are worthy of mention.

3.5.1 Unbounded Quantifiers

Probably the most important alternative is a system based on pure *un*bounded quantification, with the same formulation of intersection types. It would appear, at first sight, that this system is much less expressive than the one with bounded quantifiers. While this is certainly true of the quantification-only fragments, when intersection types are added it becomes possible to "encode" bounded quantification by reading a bounded quantifier as an abbreviation for an unbounded one with a slightly modified body:

$$\forall\alpha{\leq}\sigma.\ \tau \quad \stackrel{\text{def}}{=} \quad \forall\alpha.\ \{\sigma{\wedge}\alpha/\alpha\}\tau$$

Intuitively, the type $\forall\alpha{\leq}\sigma.\ \tau$ takes an argument that is forced, in advance, to fall beneath σ. On the other hand, $\forall\alpha.\ \{\sigma{\wedge}\alpha/\alpha\}\tau$ takes an argument that may be any type whatsoever, but at each point where this type is used it squeezes it down to below σ using a $\wedge$. (This abbreviation was suggested by John Mitchell.)

According to a typed view of the semantics of quantification and intersections, where "$\leq\sigma$" signals the existence of a coercion into σ rather than simply a proof that something already given falls within σ, this transformation makes little sense. In the simpler, untyped view, however, it is fairly appealing.

This is not, however, an *encoding* of bounded quantification in a full sense. For example, it does not validate the Sub-All rule. The derivable $F_\wedge$ statement

$$\Gamma \vdash \forall\alpha{\leq}\mathit{Real}.\ \alpha{\to}\alpha \leq \forall\alpha{\leq}\mathit{Int}.\ \alpha{\to}\alpha$$

translates to the non-derivable statement

$$\Gamma \vdash \forall\alpha.\ (\alpha{\wedge}\mathit{Real}){\to}(\alpha{\wedge}\mathit{Real}) \leq \forall\alpha.\ (\alpha{\wedge}\mathit{Int}){\to}(\alpha{\wedge}\mathit{Int}).$$

It is somewhat surprising, in view of this weakness, that many programming examples using bounded quantification still behave as expected under the translation. This point is explored at more length in Section 7.9.

3.5.2 Additional Subtyping Rules

It is also possible to *strengthen* $F_\wedge$ in various ways, the most obvious being the addition of rules allowing quantifiers to be introduced and eliminated independent of the syntactic forms $\Lambda\alpha{\leq}\sigma.\ e$ and $e[\theta]$:

$$\frac{\Gamma \vdash \sigma \leq \tau_1}{\Gamma \vdash \forall\alpha{\leq}\tau_1.\ \tau_2 \leq \{\sigma/\alpha\}\tau} \qquad \text{(Sub-All-E)}$$

$$\frac{\Gamma, \alpha{\leq}\tau_1 \vdash M \in \tau_2}{\Gamma \vdash M \in \forall\alpha{\leq}\tau_1.\ \tau_2} \qquad \text{(I-All-I)}$$

The resulting system is almost certainly too strong to form a suitable foundation for a programming language. For example, the problem of typechecking for the quantification-only fragment of this calculus is not known to be decidable; it is similar to the full type inference problem for

polymorphic lambda-calculus, which is also open, but known to be of at least exponential complexity [74].

3.5.3 Bounded Existential Types

Another natural extension of $F_\wedge$ would be to allow bounded existential types in addition to its bounded universal types. Bounded existentials are discussed by Cardelli and Wegner [33], who use them to obtain a notion of *partially abstract type* based on Mitchell and Plotkin's correspondence between abstract types (modules or packages) and existential types [97].

This extension seems straightforward. However, since bounded existential types can be encoded as bounded polymorphic types using the abbreviation

$$\exists\alpha{\leq}\sigma.\ \tau \quad \stackrel{\text{def}}{=} \quad \forall\beta.\ (\forall\alpha{\leq}\sigma.\ \tau{\rightarrow}\beta)\ \rightarrow\ \beta,$$

we can forgo their extra complication for the purposes of the present study.

Chapter 4

Typechecking

This chapter develops the proof theory of the $F_\wedge$ calculus, leading up to the definition and correctness proof of an algorithm for synthesizing minimal types of $F_\wedge$ terms. The major results we establish are as follows:

- We give an alternative formulation of the subtype relation in terms of "canonical types," where intersections appear only on the left of arrows and quantifiers. This formulation is equivalent to the original, in the sense that there is some canonical type in the equivalence class of each ordinary type. More formally, the "flattening" map from ordinary to canonical types both preserves and reflects derivability of subtyping statements.
- A proof-normalization argument based on the one used by Curien and Ghelli [50] shows that every derivable canonical subtyping statement has a "normal form" derivation with a particular, restricted shape.
- The existence of normal-form canonical derivations is used to prove the semi-completeness of a syntax-directed algorithm for checking the subtype relation.
- The soundness of a syntax-directed type synthesis algorithm for $F_\wedge$ terms is established by showing that there exist finite bases for the collections of arrow types and quantified types lying above a given type. This argument also shows that the subroutine for checking the subtype relation is the only source of possible nontermination in the typechecking algorithm.
- The shapes of the typing derivations discovered by this algorithm are used to prove that $F_\wedge$ is a conservative extension of the first-order intersection calculus $\lambda_\wedge$. Because of the different behavior of *Top* and $\top$, however, $F_\leq$ cannot similarly be embedded in $F_\wedge$.

4.1 Basic Properties

We begin by establishing some basic proof-theoretic properties of the subtype relation $\Gamma \vdash \sigma \leq \tau$ and the typing relation $\Gamma \vdash e \in \tau$.

First, we state several useful derived rules of inference.

4.1.1. Lemma:

$$\Gamma \vdash \bigwedge[\sigma \to \tau_1 \,..\, \sigma \to \tau_n] \sim \sigma \to \bigwedge[\tau_1 \,..\, \tau_n] \qquad \text{(D-Dist-IA)}$$

$$\Gamma \vdash \bigwedge[\forall \alpha \leq \sigma.\ \tau_1 \,..\, \forall \alpha \leq \sigma.\ \tau_n] \sim \forall \alpha \leq \sigma.\ \bigwedge[\tau_1 \,..\, \tau_n] \qquad \text{(D-Dist-IQ)}$$

Proof: Straightforward. □

4.1.2. Lemma:

$$\Gamma \vdash \bigwedge[\bigwedge T_1 .. \bigwedge T_n] \sim \bigwedge(T_1 * \cdots * T_n) \qquad \text{(D-Absorb)}$$

$$\frac{T \text{ and } T' \text{ enumerate the same finite set}}{\Gamma \vdash \bigwedge T \sim \bigwedge T'} \qquad \text{(D-Reindex)}$$

$$\frac{\text{for all } \tau_j \text{ there is some } \sigma_i \text{ such that } \Gamma \vdash \sigma_i \leq \tau_j}{\Gamma \vdash \bigwedge[\sigma_1 .. \sigma_m] \leq \bigwedge[\tau_1 .. \tau_n]} \qquad \text{(D-All-Some)}$$

Proof: Straightforward. □

4.1.3. Lemma: [Permutation] If Γ' is a permutation of Γ and both Γ and Γ' are closed, then

1. $\Gamma \vdash \sigma \leq \tau$ iff $\Gamma' \vdash \sigma \leq \tau$
2. $\Gamma \vdash e \in \tau$ iff $\Gamma' \vdash e \in \tau$.

Proof: By induction on derivations. □

4.1.4. Convention: Lemma 4.1.3 justifies a notational simplification: two closed contexts Γ and Γ' that differ only in the order of their bindings will be considered *identical* from here on, so that, for example, a derivation of $\Gamma \vdash \sigma \leq \tau$ is taken to *be* a derivation of $\Gamma' \vdash \sigma \leq \tau$.

Whenever a subtyping statement can be derived from a given context, it can also be derived from any larger context. In logic, this property is called *weakening*.

4.1.5. Lemma: [Weakening] If Γ_1, Γ_2 is closed and $\Gamma_1 \vdash \sigma \leq \tau$, then $\Gamma_1, \Gamma_2 \vdash \sigma \leq \tau$.

Proof: By induction on the structure of a derivation of $\Gamma_1 \vdash \sigma \leq \tau$. At each stage of the induction, we proceed by a case analysis on the rule used in the last step of the derivation.

Case Sub-Refl: $\sigma \equiv \tau$

Immediate by Sub-Refl.

Case Sub-Trans:

By the induction hypothesis and Sub-Trans.

Case Sub-TVar: $\sigma \equiv \alpha \qquad \tau \equiv \Gamma_1(\alpha)$

Since $\Gamma_1(\alpha) = (\Gamma_1, \Gamma_2)(\alpha)$, Sub-TVar immediately gives $\Gamma_1, \Gamma_2 \vdash \alpha \leq \Gamma_1(\alpha)$.

Case Sub-Arrow: $\sigma \equiv \sigma_1 \to \sigma_2 \qquad \tau \equiv \tau_1 \to \tau_2$

By the induction hypothesis and Sub-Arrow.

Case Sub-All: $\sigma \equiv \forall \alpha \leq \sigma_1.\ \sigma_2 \qquad \tau \equiv \forall \alpha \leq \tau_1.\ \tau_2$

By assumption, $\Gamma_1 \vdash \tau_1 \leq \sigma_1$ and $\Gamma_1, \alpha \leq \tau_1 \vdash \sigma_2 \leq \tau_2$. We may also assume that $\alpha \notin dom(\Gamma_1, \Gamma_2)$. Then $\Gamma_1, \Gamma_2, \alpha \leq \tau_1$ is closed, and, by the induction hypothesis, $\Gamma_1, \Gamma_2 \vdash \tau_1 \leq \sigma_1$ and (using Convention 4.1.4) $\Gamma_1, \Gamma_2, \alpha \leq \tau_1 \vdash \sigma_2 \leq \tau_2$. By Sub-All, $\Gamma_1, \Gamma_2 \vdash \forall \alpha \leq \sigma_1.\ \sigma_2 \leq \forall \alpha \leq \tau_1.\ \tau_2$.

Case Sub-Inter-G: $\tau \equiv \bigwedge[\tau_1 .. \tau_n]$

By the induction hypothesis and Sub-Inter-G.

Case Sub-Inter-LB: $\sigma \equiv \bigwedge[\sigma_1 .. \sigma_n] \qquad \tau \equiv \sigma_i$

Immediate.

Case Sub-Dist-IA: $\sigma \equiv \bigwedge[\sigma' \to \tau_1 \,..\, \sigma' \to \tau_n] \qquad \tau \equiv \sigma' \to \bigwedge[\tau_1 .. \tau_n]$

Immediate.

Case SUB-DIST-IQ: $\sigma \equiv \bigwedge[\forall\alpha{\leq}\sigma'.\ \tau_1\,..\forall\alpha{\leq}\sigma'.\ \tau_n] \qquad \tau \equiv \forall\alpha{\leq}\sigma'.\ \bigwedge[\tau_1..\tau_n]$

Immediate. □

A different kind of weakening lemma will also be needed. Rather than adding a new variable to the context, this one states that a derivable subtyping statement remains derivable when the bounds of some of the existing type variables are replaced by "narrower" bounds.

4.1.6. Lemma: [Narrowing] Let Γ and Γ' be closed contexts such that, for each $\alpha_i \in dom(\Gamma)$, $\Gamma' \vdash \Gamma'(\alpha_i) \leq \Gamma(\alpha_i)$. Then $\Gamma \vdash \sigma \leq \tau$ implies $\Gamma' \vdash \sigma \leq \tau$.

Proof: By induction on a derivation of $\Gamma \vdash \sigma \leq \tau$. Proceed by cases on the final rule.

Cases SUB-REFL, SUB-TRANS, SUB-ARROW, SUB-INTER-G, SUB-INTER-LB, SUB-DIST-IA, and SUB-DIST-IQ:

Either immediate or by straightforward use of the induction hypothesis.

Case SUB-TVAR: $\sigma \equiv \alpha \qquad \tau \equiv \Gamma(\alpha)$

By SUB-TVAR, $\Gamma' \vdash \alpha \leq \Gamma'(\alpha)$. By assumption, $\Gamma' \vdash \Gamma'(\alpha) \leq \Gamma(\alpha)$. By SUB-TRANS, $\Gamma' \vdash \alpha \leq \Gamma(\alpha)$.

Case SUB-ALL: $\sigma \equiv \forall\alpha{\leq}\sigma_1.\ \sigma_2 \qquad \tau \equiv \forall\alpha{\leq}\tau_1.\ \tau_2$

By assumption, $\Gamma \vdash \tau_1 \leq \sigma_1$ and $\Gamma, \alpha{\leq}\tau_1 \vdash \sigma_2 \leq \tau_2$. By the induction hypothesis, $\Gamma' \vdash \tau_1 \leq \sigma_1$. By assumption and weakening (Lemma 4.1.5), $\Gamma', \alpha{\leq}\tau_1 \vdash \Gamma'(\alpha_i) \leq \Gamma(\alpha_i)$ for each $\alpha_i \in dom(\Gamma)$. By SUB-REFL, $\Gamma', \alpha{\leq}\tau_1 \vdash (\Gamma', \alpha{\leq}\tau_1)(\alpha) \leq (\Gamma, \alpha{\leq}\tau_1)(\alpha)$. The induction hypothesis then gives $\Gamma', \alpha{\leq}\tau_1 \vdash \sigma_2 \leq \tau_2$. By SUB-ALL, $\Gamma' \vdash \forall\alpha{\leq}\sigma_1.\ \sigma_2 \leq \forall\alpha{\leq}\tau_1.\ \tau_2$. □

Using narrowing, we can show that the equivalence relation induced by the subtype relation is a congruence:

4.1.7. Lemma:

$$\frac{\Gamma \vdash \tau_1 \sim \tau_1' \qquad \Gamma \vdash \tau_2 \sim \tau_2'}{\Gamma \vdash \tau_1{\to}\tau_2 \sim \tau_1'{\to}\tau_2'} \qquad \text{(D-CONG-ARROW)}$$

$$\frac{\Gamma \vdash \tau_1 \sim \tau_1' \qquad \Gamma, \alpha{\leq}\tau_1' \vdash \tau_2 \sim \tau_2'}{\Gamma \vdash \forall\alpha{\leq}\tau_1.\ \tau_2 \sim \forall\alpha{\leq}\tau_1'.\ \tau_2'} \qquad \text{(D-CONG-ALL)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash \tau_i \sim \tau_i'}{\Gamma \vdash \bigwedge[\tau_1..\tau_n] \sim \bigwedge[\tau_1'..\tau_n']} \qquad \text{(D-CONG-INTER)}$$

Proof: Easy except for D-CONG-ALL, which requires Lemma 4.1.6. □

4.1.8. Lemma: [Subtyping Substitution]
If

$\Gamma_1, \alpha{\leq}\psi, \Gamma_2 \vdash \sigma \leq \tau$
$\Gamma_1 \vdash \phi \leq \psi,$

then

$\Gamma_1, \{\phi/\alpha\}\Gamma_2 \vdash \{\phi/\alpha\}\sigma \leq \{\phi/\alpha\}\tau.$

Proof: By induction on a derivation of $\Gamma_1, \alpha{\leq}\psi, \Gamma_2 \vdash \sigma \leq \tau$.

Cases SUB-REFL, SUB-TRANS, SUB-ARROW, SUB-INTER-G, SUB-INTER-LB, SUB-DIST-IA, and SUB-DIST-IQ:

Either immediate or by straightforward use of the induction hypothesis.

Case SUB-TVAR: $\sigma \equiv \beta \qquad \tau \equiv (\Gamma_1, \alpha{\leq}\psi, \Gamma_2)(\beta)$

Subcase: $\alpha \equiv \beta$

By assumption,

$$\Gamma_1 \vdash \phi \leq \psi.$$

By weakening (4.1.5),

$$\Gamma_1, \{\phi/\alpha\}\Gamma_2 \vdash \phi \leq \psi,$$

that is,

$$\Gamma_1, \{\phi/\alpha\}\Gamma_2 \vdash \{\phi/\alpha\}\alpha \leq \{\phi/\alpha\}\psi,$$

as required.

Subcase: $\alpha \not\equiv \beta$

By Sub-Tvar,

$$\Gamma_1, \{\phi/\alpha\}\Gamma_2 \vdash \beta \leq (\Gamma_1, \{\phi/\alpha\}\Gamma_2)(\beta),$$

that is,

$$\Gamma_1, \{\phi/\alpha\}\Gamma_2 \vdash \{\phi/\alpha\}\beta \leq \{\phi/\alpha\}(\Gamma_1, \alpha{\leq}\psi, \Gamma_2)(\beta).$$

Case Sub-All: $\sigma \equiv \forall\beta{\leq}\sigma_1.\ \sigma_2 \qquad \tau \equiv \forall\beta{\leq}\tau_1.\ \tau_2$

By assumption,

$$\Gamma_1, \alpha{\leq}\psi, \Gamma_2 \vdash \tau_1 \leq \sigma_1$$
$$\Gamma_1, \alpha{\leq}\psi, \Gamma_2, \beta \leq \tau_1 \vdash \sigma_2{\leq}\tau_2.$$

By the induction hypothesis,

$$\Gamma_1, \{\phi/\alpha\}\Gamma_2 \vdash \{\phi/\alpha\}\tau_1 \leq \{\phi/\alpha\}\sigma_1$$
$$\Gamma_1, \{\phi/\alpha\}\Gamma_2, \beta{\leq}\{\phi/\alpha\}\tau_1 \vdash \{\phi/\alpha\}\sigma_2 \leq \{\phi/\alpha\}\tau_2.$$

By Sub-All and the definition of substitution,

$$\Gamma_1, \{\phi/\alpha\}\Gamma_2 \vdash \{\phi/\alpha\}(\forall\beta{\leq}\sigma_1.\ \sigma_2) \leq \{\phi/\alpha\}(\forall\beta{\leq}\tau_1.\ \tau_2).$$ □

We would also like to show that whenever a variable α or x in $dom(\Gamma)$ is unused in a statement $\Gamma \vdash \sigma \leq \tau$ or $\Gamma \vdash e \in \tau$, we may drop it from Γ without affecting derivability. Because the rule Sub-Trans allows subtyping derivations that can contain internal uses of arbitrary types, we cannot prove this property for type variables with the machinery we have developed so far (it is an easy corollary of Theorem 4.2.8.12). But for term variables it is straightforward:

4.1.9. Lemma: [Term variable strengthening]

1. If $\Gamma, x{:}\theta \vdash \sigma \leq \tau$, then $\Gamma \vdash \sigma \leq \tau$.
2. If $\Gamma, x{:}\theta \vdash e \in \tau$ and $x \notin FV(e)$, then $\Gamma \vdash e \in \tau$.

Proof: By induction on derivations. □

4.1.10. Lemma: [Term substitution in terms] If $\Gamma, x{:}\sigma \vdash f \in \tau$ and $\Gamma \vdash v \in \sigma$, then $\Gamma \vdash \{v/x\}f \in \tau$.

Proof: By induction on a derivation of $\Gamma, x{:}\sigma \vdash f \in \tau$, using weakening (4.1.5) for the Arrow-I and All-I cases and term strengthening (4.1.9) for the All-E and Sub cases. □

4.1.11. Lemma: [Type substitution in terms] If $\Gamma_1, \alpha{\leq}\sigma, \Gamma_2 \vdash f \in \tau$, then $\Gamma_1, \{\sigma/\alpha\}\Gamma_2 \vdash \{\sigma/\alpha\}f \in \{\sigma/\alpha\}\tau$.

Proof: By induction on a derivation of $\Gamma_1, \alpha{\leq}\sigma, \Gamma_2 \vdash f \in \tau$, using the subtyping substitution lemma (4.1.8) for the All-E and Sub cases. □

4.2 Subtyping

In this section, we give a straightforward semi-decision procedure for the $F_\wedge$ subtype relation. This relation can be shown to be undecidable (see Chapter 6), so a semi-decision procedure is the best we can hope for; however, the same algorithm forms a decision procedure for several useful fragments of $F_\wedge$ (see Section 6.12).

We present the algorithm as an alternative collection of syntax-directed rules and then show that the relation defined by these rules coincides with $F_\wedge$. It is technically convenient to make this argument using an intermediate representation called *canonical types* (roughly analogous to conjunctive-normal-form formulas in logic):

- We identify the set of canonical types (Section 4.2.1) and define a canonical subtyping relation (marked $\vdash^\flat$) over this set (Section 4.2.2).
- We show (Sections 4.2.3 and 4.2.4) that derivations of canonical subtyping statements can be transformed into *normal form* derivations of a certain restricted shape (Section 4.2.6).
- We give a *flattening transformation* $\flat$ mapping $F_\wedge$ types into canonical types (Section 4.2.7) and show that

  $$\Gamma \vdash \sigma \le \tau \quad \text{iff} \quad \Gamma^\flat \vdash^\flat \sigma^\flat \le \tau^\flat.$$

- Finally, we define a syntax-directed subtyping relation on ordinary types (marked $\vdash$) and show (Section 4.2.8) that it coincides with canonical subtyping after flattening:

  $$\Gamma^\flat \vdash^\flat \sigma^\flat \le \tau^\flat \quad \text{iff} \quad \Gamma \vdash \sigma \le \tau$$

4.2.1 Canonical Types

We might naively hope to develop an algorithm for checking $F_\wedge$ subtyping simply by extending the $F_\le$ subtyping algorithm $F^N_\le$ (2.6.10) to include a case for intersections. Including cases *1* to *5* as before, the complete algorithm would then be:

$check(\Gamma \vdash \sigma \le \tau) =$

1. *if* $\tau \equiv Top$
  then true

2. *else if* $\sigma \equiv \sigma_1 \to \sigma_2$ *and* $\tau \equiv \tau_1 \to \tau_2$
  then $check(\Gamma \vdash \tau_1 \le \sigma_1)$
    and $check(\Gamma \vdash \sigma_2 \le \tau_2)$

3. *else if* $\sigma \equiv \forall \alpha \le \sigma_1.\ \sigma_2$ *and* $\tau \equiv \forall \alpha \le \tau_1.\ \tau_2$
  then $check(\Gamma \vdash \tau_1 \le \sigma_1)$
    and $check(\Gamma, \alpha \le \tau_1 \vdash \sigma_2 \le \tau_2)$

4. *else if* $\sigma \equiv \alpha$ *and* $\tau \equiv \alpha$
  then true

5. *else if* $\sigma \equiv \alpha$
  then $check(\Gamma \vdash \Gamma(\alpha) \le \tau)$

6. *else if* $\sigma \equiv \bigwedge[\sigma_1..\sigma_m]$ *and* $\tau \equiv \bigwedge[\tau_1..\tau_n]$
  then for each τ_i
    choose some σ_j
     such that $check(\Gamma \vdash \sigma_j \le \tau_i)$

n. *else*
  false.

That is, to check whether $\bigwedge[\sigma_1..\sigma_m]$ is a subtype of $\bigwedge[\tau_1..\tau_n]$, check that for each τ_i there is some σ_j such that $\sigma_j \le \tau_i$. (The selection of σ_j is expressed here as a nondeterministic choice; in practice, this is implemented using backtracking.)

But this rule, though clearly sound (by Lemma 4.1.2), is not complete, since there are many cases where a meet lies above or below a type that does not have the form of a meet. For example,

$$\alpha{\le}\top \vdash \alpha \le \bigwedge[\alpha]$$
$$\alpha{\le}\top \vdash \bigwedge[\alpha] \le \alpha.$$

These two cases can be handled by splitting the proposed rule into two,

6a. *else if* $\tau \equiv \bigwedge[\tau_1..\tau_n]$
then for each τ_i
check$(\Gamma \vdash \sigma \le \tau_i)$

6b. *else if* $\sigma \equiv \bigwedge[\sigma_1..\sigma_m]$
then choose some σ_j
such that check$(\Gamma \vdash \sigma_j \le \tau)$,

but we must be careful to apply *6a* before *6b* in order to respond correctly to the input

$$\alpha{\le}\top, \beta{\le}\top \vdash \bigwedge[\alpha, \beta] \le \bigwedge[\alpha, \beta].$$

Unfortunately, the real problem is more subtle than this: in the presence of the distributivity axioms Sub-Dist-IA and Sub-Dist-IQ, steps *2* and *3* of the $F^N_{\le}$ algorithm are also incomplete. For example,

$$\alpha{\le}\top, \beta{\le}\top \vdash \alpha{\to}\alpha \le \beta{\to}\top,$$

is derivable (using Sub-Dist-IA and Sub-Trans, with intermediate type $\top$), although

$$\alpha{\le}\top, \beta{\le}\top \not\vdash \beta \le \alpha.$$

To get things under control, we need to deal with the distributivity laws before doing anything else. We take the inverses of these laws (i.e., the other half of the equivalences D-Dist-IA and D-Dist-IQ given in 4.1.1) as rewrite rules

$$\sigma \to \bigwedge[\tau_1..\tau_n] \longrightarrow_\flat \bigwedge[\sigma{\to}\tau_1 .. \sigma{\to}\tau_n]$$
$$\forall\alpha{\le}\sigma.\ \bigwedge[\tau_1..\tau_n] \longrightarrow_\flat \bigwedge[\forall\alpha{\le}\sigma.\ \tau_1 .. \forall\alpha{\le}\sigma.\ \tau_n],$$

and apply them to a given $F_\wedge$ type as many times as possible to obtain an equivalent type with no intersections on the right-hand sides of arrows or quantifiers. For example, the type

$$\bigwedge[\alpha_1, \alpha_2] \to \bigwedge[\alpha_3, (\forall\beta{\le}\alpha_4.\ \bigwedge[\alpha_5, \alpha_6])]$$

becomes

$$\bigwedge[\bigwedge[\alpha_1, \alpha_2] \to \alpha_3,$$
$$\quad \bigwedge[\bigwedge[\alpha_1, \alpha_2] \to \forall\beta{\le}\alpha_4.\ \alpha_5,$$
$$\quad\quad \bigwedge[\alpha_1, \alpha_2] \to \forall\beta{\le}\alpha_4.\ \alpha_6]].$$

Since this type contains no meets on the right-hand sides of arrows or quantifiers, there is no way to use it (or any subphrase of it) in an instance of either of the distributivity rules.

Once the distributivity rules are eliminated, there remain just four subtyping rules in Definition 3.2.3 that can be used to prove subtyping statements where both sides of the conclusion are meets: Sub-Refl, Sub-Trans, Sub-Inter-G, and Sub-Inter-LB. The reflexivity and transitivity rules can be eliminated by a normalization argument that shows how to transform any derivation into one that applies transitivity and reflexivity only to variables (Sections 4.2.4 to 4.2.6). This leaves just Sub-Inter-G and Sub-Inter-LB, the behavior of which is completely captured by our rules *6a* and *6b*.

For the proof-theoretic analysis in the following sections, it is convenient to work with types in an even more restricted form, where every type has a single $\bigwedge$ as its outermost constructor and as the outermost constructor on the left-hand sides of arrows and quantifiers, and where no immediate component of a $\bigwedge$ is another $\bigwedge$. Our example, in this form, is:

$$\begin{aligned}\bigwedge[\bigwedge[\alpha_1, \alpha_2] &\rightarrow \alpha_3,\\ \bigwedge[\alpha_1, \alpha_2] &\rightarrow \forall\beta{\leq}\bigwedge[\alpha_4].\ \alpha_5,\\ \bigwedge[\alpha_1, \alpha_2] &\rightarrow \forall\beta{\leq}\bigwedge[\alpha_4].\ \alpha_6].\end{aligned}$$

This transformation effectively separates the set of types into two syntactic sorts: those whose outermost constructor is $\bigwedge$ (called "composite canonical types") and those whose outermost constructor is $\rightarrow$, $\forall$, or a variable (called "individual canonical types"). This separation is useful because our proposed rule *6* now captures *all* of the valid subtyping statements involving pairs of composite canonical types, while the original rules *2–5* are valid and complete for pairs of individual canonical types.

4.2.1.1. Definition: The sets of *composite canonical types* K and *individual canonical types* κ are defined by the following abstract grammar:

$$\begin{array}{lcl} K & ::= & \bigwedge[\kappa_1..\kappa_n] \\ \kappa & ::= & \alpha \ \mid\ K{\rightarrow}\kappa \ \mid\ \forall\alpha{\leq}K.\ \kappa \end{array}$$

4.2.1.2. Convention: The metavariables K and I range over composite canonical types; κ and ι range over individual canonical types; k and i range over both sorts of canonical types.

4.2.1.3. Notation: It will often be convenient in the remainder of this chapter to treat a composite canonical type $\bigwedge[\kappa_1..\kappa_n]$ as a finite set whose elements are the individual canonical types κ_1 through κ_n. The following notational conventions support this point of view:

$$\begin{array}{lcl} \iota \in K & \stackrel{\text{def}}{=} & K \equiv \bigwedge[\kappa_1..\kappa_n] \text{ and } \iota{\equiv}\kappa_i \text{ for some } i \\ \bigwedge[F(\iota) \mid \iota \in K] & \stackrel{\text{def}}{=} & \bigwedge[F(\kappa_1)\,..\,F(\kappa_n)], \text{ where } K \equiv \bigwedge[\kappa_1..\kappa_n] \\ K \cup I & \stackrel{\text{def}}{=} & \bigwedge[\kappa_1\,..\,\kappa_m, \iota_1\,..\,\iota_n], \text{ where } K \equiv \bigwedge[\kappa_1..\kappa_m] \text{ and } I \equiv \bigwedge[\iota_1..\iota_n]. \end{array}$$

4.2.1.4. Remark: Our canonical types are not as sparse as one could imagine. For example, we could try to define a notion of *fully canonical types* such that each $\sim$-equivalence class of types contains exactly one fully canonical type. For present purposes, though, the canonical types we have defined are refined enough.

The idea of canonical types comes from a proof by Reynolds [personal communication, 1988] of the soundness and completeness of a decision procedure for the subtype relation of Forsythe [121]. (Related formulations of intersection types are studied in [41, 43, 44, 126, 131, 132].) In fact, the entire type system of Forsythe can be reformulated in terms of canonical types, making this proof trivial. Such is not the case here, unfortunately, because the operation of substituting a canonical type for a variable in another canonical type yields a non-canonical type. We could, of course, change the definition of substitution so that it re-canonicalized its result, but it would be unfortunate to place such a complicated mechanism in such a basic piece of technical machinery. Also, an implementation based directly on canonical types would be less efficient than the one described in Section 4.2.8, which uses a data structure based on ordinary types. We therefore take the original system unchanged and make the translation to canonical form explicitly for purposes of analysis.

4.2.2 Canonical Subtyping

We now define the canonical subtype relation formally.

4.2.2.1. Definition: A *canonical context* Δ is a context whose range contains only canonical types.

4.2.2.2. Definition: The subtype relation on canonical types is defined as follows:

$$\frac{\forall i.\ \exists j.\quad \Delta \vdash \kappa_j \leq \iota_i}{\Delta \vdash \bigwedge[\kappa_1..\kappa_m] \leq \bigwedge[\iota_1..\iota_n]} \qquad \text{(CSub-AE)}$$

$$\Delta \vdash k \leq k \qquad \text{(CSub-Refl)}$$

$$\frac{\Delta \vdash k_1 \leq k_2 \qquad \Delta \vdash k_2 \leq k_3}{\Delta \vdash k_1 \leq k_3} \qquad \text{(CSub-Trans)}$$

$$\frac{\Delta \vdash \Delta(\alpha) \leq \bigwedge[\iota]}{\Delta \vdash \alpha \leq \iota} \qquad \text{(CSub-TVar)}$$

$$\frac{\Delta \vdash I_1 \leq K_1 \qquad \Delta \vdash \kappa_2 \leq \iota_2}{\Delta \vdash K_1 \to \kappa_2 \leq I_1 \to \iota_2} \qquad \text{(CSub-Arrow)}$$

$$\frac{\Delta \vdash I_1 \leq K_1 \qquad \Delta, \alpha \leq I_1 \vdash \kappa_2 \leq \iota_2}{\Delta \vdash \forall \alpha \leq K_1.\ \kappa_2 \leq \forall \alpha \leq I_1.\ \iota_2} \qquad \text{(CSub-All)}$$

4.2.2.3. Remark: Note that individual and composite canonical types cannot be mixed in canonical subtyping statements: we have statements of the form $\Delta \vdash K \leq I$ and $\Delta \vdash \kappa \leq \iota$, but never $\Delta \vdash \kappa \leq I$ or $\Delta \vdash K \leq \iota$.

4.2.2.4. Notation: The turnstile symbol is sometimes decorated $\vdash^{\flat}$ to distinguish canonical subtyping derivations from derivations in other calculi.

4.2.2.5. Remark: Anticipating the requirements of the normal-form derivations to be defined in the following section, we have slightly generalized the type variable rule, in effect embedding an instance of CSub-Trans and an instance of CSub-AE in each instance of CSub-TVar.

Again, it is convenient to have a linear notation for canonical subtyping derivations.

4.2.2.6. Definition: The set of *canonical subtyping derivation abbreviations* is defined by the following abstract grammar:

$$\begin{array}{lll} C & ::= & AE[\xi_1..\xi_n] \\ & | & id \\ & | & C\ ;C \\ \\ \xi & ::= & id \\ & | & \xi_1\ ;\xi_2 \\ & | & V_\alpha(C) \\ & | & C \to \xi \\ & | & \forall \alpha \leq C.\ \xi \end{array}$$

4.2.2.7. Notation: The metavariables C and D range over derivations of subtyping statements between composite canonical types; ξ and δ range over derivations of subtyping statements between individual canonical types; c and d range over both sorts of derivations.

As in Section 3.3, we can define a mapping from canonical derivation trees to linear abbreviations, which, though not injective, could easily be extended to an injective map at some cost in readability. Again, we will prefer readability over rigor and impose on the reader's imagination to supply the evident annotations on our linear abbreviations.

4.2.2.8. Definition: The translation function $(\text{—})^\dagger$ from canonical derivation trees to their abbreviated forms is defined as follows.

$$\left(\frac{\forall i.\ \exists j.\quad \xi_i :: \Delta \vdash \kappa_j \leq \iota_i}{\Delta \vdash \bigwedge[\kappa_1..\kappa_m] \leq \bigwedge[\iota_1..\iota_n]}\,{}_{\text{(CSUB-AE)}}\right)^\dagger \quad = \quad AE[\xi_1^\dagger..\xi_n^\dagger]$$

$$\left(\frac{}{\Delta \vdash k \leq k}\,{}_{\text{(CSUB-REFL)}}\right)^\dagger \quad = \quad id$$

$$\left(\frac{c_1 :: \Delta \vdash k_1 \leq k_2 \qquad c_2 :: \Delta \vdash k_2 \leq k_3}{\Delta \vdash k_1 \leq k_3}\,{}_{\text{(CSUB-TRANS)}}\right)^\dagger \quad = \quad c_1^\dagger\ ;\ c_2^\dagger$$

$$\left(\frac{C :: \Delta \vdash \Delta(\alpha) \leq \bigwedge[\iota]}{\Delta \vdash \alpha \leq \iota}\,{}_{\text{(CSUB-TVAR)}}\right)^\dagger \quad = \quad V_\alpha(C^\dagger)$$

$$\left(\frac{C_1 :: \Delta \vdash I_1 \leq K_1 \qquad \xi_2 :: \Delta \vdash \kappa_2 \leq \iota_2}{\Delta \vdash K_1 \to \kappa_2 \leq I_1 \to \iota_2}\,{}_{\text{(CSUB-ARROW)}}\right)^\dagger \quad = \quad C_1^\dagger \to \xi_2^\dagger$$

$$\left(\frac{C_1 :: \Delta \vdash I_1 \leq K_1 \qquad \xi_2 :: \Delta, \alpha \leq I_1 \vdash \kappa_2 \leq \iota_2}{\Delta \vdash \forall \alpha{\leq}K_1.\ \kappa_2 \leq \forall \alpha{\leq}I_1.\ \iota_2}\,{}_{\text{(CSUB-ALL)}}\right)^\dagger \quad = \quad \forall \alpha{\leq}C_1^\dagger.\ \xi_2^\dagger$$

4.2.2.9. Lemma: [Canonical context permutation] If Δ' is a permutation of Δ and both Δ' and Δ are closed, and if $c :: \Delta \vdash k \leq i$, then there is a subtyping derivation c', identical to c except for the ordering of contexts, such that $c' :: \Delta \vdash k \leq i$.

If $\alpha \notin FTV(I)$ and $c :: \Delta_1,\ \alpha{\leq}K,\ \beta{\leq}I,\ \Delta_2 \vdash k \leq i$, then there is a subtyping derivation c', identical to c except for the ordering of contexts, such that $c' :: \Delta_1,\ \beta{\leq}I,\ \alpha{\leq}K,\ \Delta_2 \vdash k \leq i$.

Proof: By induction on the structure of c. □

4.2.2.10. Convention: As for ordinary subtyping (c.f. Convention 4.1.4), this lemma justifies the convention that closed canonical contexts differing only in the ordering of their bindings are regarded as identical.

4.2.3 Weakening and Narrowing

The proof transformations used to normalize canonical subtyping derivations rely on analogues of the weakening and narrowing lemmas proved in Section 4.1. These properties need to be formulated here as explicit operations on subtyping derivations.

By analogy with extending a context, our linear notation for weakening is "$c,\ \alpha{\leq}K$." If c is a derivation term whose conclusion is $\Delta \vdash k \leq i$, then $c,\ \alpha{\leq}K$ is a derivation with conclusion $\Delta, \alpha{\leq}K \vdash k \leq i$.

4.2.3.1. Remark: For the same of explicitness in what follows, this definition is more concrete than the definition of weakening for ordinary types (4.1.5), which allowed many variables to be added at once.

4.2.3.2. Definition: The *weakening* of a derivation c with a new binding $\alpha{\leq}K$, written $c, \alpha{\leq}K$, is defined as follows:

$$\begin{array}{lcl}
(AE[\xi_1..\xi_n]), \alpha{\leq}K & = & AE[(\xi_1, \alpha{\leq}K)\,..\,(\xi_n, \alpha{\leq}K)] \\
id, \alpha{\leq}K & = & id \\
(c_1\,;\,c_2), \alpha{\leq}K & = & (c_1, \alpha{\leq}K)\,;\,(c_2, \alpha{\leq}K) \\
(V_\beta(C)), \alpha{\leq}K & = & V_\beta(C, \alpha{\leq}K) \\
(C_1{\rightarrow}\xi_2), \alpha{\leq}K & = & (C_1, \alpha{\leq}K){\rightarrow}(\xi_2, \alpha{\leq}K) \\
(\forall\beta{\leq}C_1.\ \xi_2), \alpha{\leq}K & = & \forall\beta{\leq}(C_1, \alpha{\leq}K).\ (\xi_2, \alpha{\leq}K).
\end{array}$$

4.2.3.3. Remark: Since weakening only changes contexts, this operation is an identity on our abbreviated linear forms, where contexts are always elided.

4.2.3.4. Lemma: [Weakening for canonical subtyping] If

$$c :: \Delta \vdash k \leq i,$$

then

$$c, \alpha{\leq}K \ :: \Delta, \alpha{\leq}K \vdash k \leq i.$$

Proof: By induction on the structure of c.

Cases CSub-AE, CSub-Trans, CSub-Refl, CSub-Arrow:

Either immediate or by straightforward use of the induction hypothesis.

Case CSub-TVar: $k \equiv \beta \quad (\beta{\not\equiv}\alpha) \qquad i \equiv \iota$
$c \equiv V_\beta(C) \qquad C :: \Delta \vdash \Delta(\beta) \leq \bigwedge[\iota]$

By the induction hypothesis,

$$C, \alpha{\leq}K :: \Delta, \alpha{\leq}K \ \vdash\ \Delta(\beta)\ \leq\ \bigwedge[\iota],$$

that is,

$$C, \alpha{\leq}K :: \Delta, \alpha{\leq}K \ \vdash\ (\Delta, \alpha{\leq}K)(\beta)\ \leq\ \bigwedge[\iota].$$

By CSub-TVar,

$$V_\beta(C, \alpha{\leq}K) :: \Delta, \alpha{\leq}K \ \vdash\ \beta\ \leq\ \iota,$$

that is,

$$(V_\beta(C)), \alpha{\leq}K :: \Delta, \alpha{\leq}K \ \vdash\ \beta\ \leq\ \iota.$$

Case CSub-All: $k \equiv \forall\beta{\leq}K_1.\ \kappa_2 \qquad i \equiv \forall\beta{\leq}I_1.\ \iota_2$
$c \equiv \forall\beta{\leq}C_1.\ \xi_2 \qquad C_1 :: \Delta \vdash I_1 \leq K_1 \qquad \xi_2 :: \Delta, \beta{\leq}I_1 \vdash \kappa_2 \leq \iota_2$

(Also, since β is bound, we may assume that $\alpha \not\equiv \beta$.) By the induction hypothesis,

$$C_1, \alpha{\leq}K :: \Delta, \alpha{\leq}K \ \vdash\ I_1\ \leq\ K_1$$

and

$$\xi_2, \alpha{\leq}K :: \Delta, \alpha{\leq}K, \beta{\leq}I_1 \ \vdash\ \kappa_2\ \leq\ \iota_2.$$

By CSub-All,

$$(\forall\beta{\leq}(C_1, \alpha{\leq}K).\ (\xi_2, \alpha{\leq}K)) :: \Delta, \alpha{\leq}K \ \vdash\ \forall\beta{\leq}K_1.\ \kappa_2\ \leq\ \forall\beta{\leq}I_1.\ \iota_2,$$

that is,

$$((\forall\beta{\leq}C_1.\ \xi_2), \alpha{\leq}K) :: \Delta, \alpha{\leq}K \vdash \forall\beta{\leq}K_1.\ \kappa_2 \leq \forall\beta{\leq}I_1.\ \iota_2.$$

□

As for ordinary types, the narrowing transformation allows the context in the conclusion of a derivation to be modified by replacing the bound of a variable by a subtype of the current bound. For example,

$$\alpha_1{\leq}\top, \alpha_2{\leq}\bigwedge[\alpha_1], \alpha_3{\leq}\bigwedge[\alpha_1] \vdash \bigwedge[\alpha_1]{\to}\alpha_3 \leq \bigwedge[\alpha_1]{\to}\alpha_1$$

can be narrowed to

$$\alpha_1{\leq}\top, \alpha_2{\leq}\bigwedge[\alpha_1], \alpha_3{\leq}\bigwedge[\alpha_2] \vdash \bigwedge[\alpha_1]{\to}\alpha_3 \leq \bigwedge[\alpha_1]{\to}\alpha_1.$$

Note, however, that the structure of the old derivation must be altered in order for the result of narrowing to be valid: wherever the assumption $\alpha_3{\leq}\alpha_1$ was used in the original, we must use the new assumption $\alpha_3{\leq}\bigwedge[\alpha_2]$ plus the fact that $\alpha_2{\leq}\bigwedge[\alpha_1]$. To take this into account, we use the linear notation $c[D@\alpha{\leq}K]$, where c is a subtyping derivation, α is a variable in $dom(\Delta)$, K is its new bound, and D is a derivation establishing that the new bound is a subtype of the existing one.

4.2.3.5. Definition: Let c and D be subtyping derivations such that

$$c :: \Delta, \alpha{\leq}I \vdash k \leq i$$
$$D :: \Delta \vdash K \leq I.$$

The *narrowing* of c with the binding $\alpha{\leq}K$ (justified by D) is written $c[D@\alpha{\leq}K]$ and defined as follows:

$$\begin{array}{lll}
(AE[\xi_1..\xi_n])[D@\alpha{\leq}K] & = & AE[(\xi_1[D@\alpha{\leq}K]) .. (\xi_n[D@\alpha{\leq}K])] \\
id[D@\alpha{\leq}K] & = & id \\
(c_1 \,;\, c_2)[D@\alpha{\leq}K] & = & (c_1[D@\alpha{\leq}K]) \,;\, (c_2[D@\alpha{\leq}K]) \\
(V_\beta(C))[D@\alpha{\leq}K] \quad \text{where } \beta{\not\equiv}\alpha & = & V_\beta(C[D@\alpha{\leq}K]) \\
(V_\alpha(C))[D@\alpha{\leq}K] & = & V_\alpha((D, \alpha{\leq}K) \,;\, (C[D@\alpha{\leq}K])) \\
(C_1{\to}\xi_2)[D@\alpha{\leq}K] & = & (C_1[D@\alpha{\leq}K]){\to}(\xi_2[D@\alpha{\leq}K]) \\
(\forall\beta{\leq}C_1.\ \xi_2)[D@\alpha{\leq}K] & = & \forall\beta{\leq}(C_1[D@\alpha{\leq}K]).\ (\xi_2[(D, \beta{\leq}I_1)@\alpha{\leq}K]) \\
 & & \text{where } C_1, \alpha{\leq}I :: \Delta \vdash I_1 \leq K_1.
\end{array}$$

4.2.3.6. Lemma: [Narrowing for canonical subtyping] If

$$c :: \Delta, \alpha{\leq}I \vdash k \leq i$$
$$D :: \Delta \vdash K \leq I,$$

then

$$(c[D@\alpha{\leq}K]) :: \Delta, \alpha{\leq}K \vdash k \leq i.$$

Proof: By induction on c.

Cases CSub-AE, CSub-Refl, CSub-Trans, CSub-Arrow:

Either immediate or by straightforward induction.

Case CSub-TVar: $k \equiv \beta \quad (\beta{\not\equiv}\alpha) \qquad i \equiv \iota$

By the induction hypothesis and CSub-TVar.

Case CSub-TVar: $k \equiv \alpha \qquad i \equiv \iota$
$c \equiv V_\alpha(C) \qquad C :: \Delta, \alpha{\leq}I \vdash I \leq \bigwedge[\iota]$

By the induction hypothesis,

$$(C[D@\alpha{\leq}K]) :: \Delta, \alpha{\leq}K \vdash I \leq \bigwedge[\iota].$$

By weakening (4.2.3.4),

$$(D, \alpha{\leq}K) :: \Delta, \alpha{\leq}K \vdash K \leq I.$$

By CSub-Trans,

$((D, \alpha{\leq}K)\,;(C[D@\alpha{\leq}K])) :: \Delta, \alpha{\leq}K \vdash K \leq \bigwedge[\iota]$.

By CSub-TVar,

$V_\alpha((D, \alpha{\leq}K)\,;(C[D@\alpha{\leq}K])) :: \Delta, \alpha{\leq}K \vdash \alpha \leq \iota$,

that is,

$(V_\alpha(C))[D@\alpha{\leq}K] :: \Delta, \alpha{\leq}K \vdash \alpha \leq \iota$.

Case CSub-All: $k \equiv \forall\beta{\leq}K_1.\ \kappa_2 \qquad i \equiv \forall\beta{\leq}I_1.\ \iota_2$
$c \equiv \forall\beta{\leq}C_1.\ \xi_2$
$C_1 :: \Delta, \alpha{\leq}I \vdash I_1 \leq K_1 \qquad \xi_2 :: \Delta, \alpha{\leq}I, \beta{\leq}I_1 \vdash \kappa_2 \leq \iota_2$

By the induction hypothesis on the first subderivation,

$(C_1[D@\alpha{\leq}K]) :: \Delta, \alpha{\leq}K \vdash I_1 \leq K_1$.

By weakening (4.2.3.4) on the second main hypothesis,

$(D, \beta{\leq}I_1) :: \Delta, \beta{\leq}I_1 \vdash K \leq I$.

By the induction hypothesis on the second subderivation,

$\xi_2[(D, \beta{\leq}I_1)@\alpha{\leq}K] :: \Delta, \beta{\leq}I_1, \alpha{\leq}K \vdash \kappa_2 \leq \iota_2$.

By CSub-All,

$(\forall\beta{\leq}(C_1[D@\alpha{\leq}K]).\ (\xi_2[(D, \beta{\leq}I_1)@\alpha{\leq}K])) :: \Delta, \alpha{\leq}K \vdash \forall\beta{\leq}K_1.\ \kappa_2 \leq \forall\beta{\leq}I_1.\ \iota_2$,

that is,

$(\forall\beta{\leq}C_1.\ \xi_2)[D@\alpha{\leq}K] :: \Delta, \alpha{\leq}K \vdash \forall\beta{\leq}K_1.\ \kappa_2 \leq \forall\beta{\leq}I_1.\ \iota_2$. □

4.2.4 Subtyping Derivation Normalization Rules

To construct an algorithm for checking the canonical subtype relation, we need a notion of *normal form* derivations (similar to the one described in Section 2.6 for $F_{\leq}$ subtyping) and an effective procedure for transforming arbitrary derivations into this form. The main task of this *normalization procedure* is to push instances of CSub-Trans toward the leaves of the derivation, until they eventually disappear into instances of the CSub-TVar rule. For example, if $\Delta \equiv \alpha_1{\leq}\top, \alpha_2{\leq}\bigwedge[\alpha_1], \alpha_3{\leq}\bigwedge[\alpha_2]$, then the derivation

$$\frac{\Delta \vdash \alpha_3 \leq \alpha_2 \qquad \Delta \vdash \alpha_2 \leq \alpha_1}{\Delta \vdash \alpha_3 \leq \alpha_1}\ (\text{CSUB-TRANS})$$

becomes the following normal-form derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta \vdash \alpha_1 \leq \alpha_1}\ (\text{CSUB-REFL})}{\Delta \vdash \bigwedge[\alpha_1] \leq \bigwedge[\alpha_1]}\ (\text{CSUB-AE})}{\Delta \vdash \alpha_2 \leq \alpha_1}\ (\text{CSUB-TVAR})}{\dfrac{\Delta \vdash \bigwedge[\alpha_2] \leq \bigwedge[\alpha_1]}{\Delta \vdash \alpha_3 \leq \alpha_1.}\ (\text{CSUB-TVAR})}\ (\text{CSUB-AE})$$

The normalization procedure is presented as a collection of rewrite rules on subtyping derivations. These rules are separated into two groups to simplify the presentation of a *rewriting strategy* later on (Section 4.2.5) and the proof that this strategy always terminates.

We first list the abbreviated linear forms of the rules, then justify them by discussing how they operate on proof trees.

4.2.4.1. Definition: The *one-step, outermost simplification relation* on subtyping derivations, $\longrightarrow_1^t$, is defined by the following rewrite rules:

I. Reflexivity simplification

1. $id \;::\; \Delta \vdash K_1 \to \kappa_2 \;\leq\; K_1 \to \kappa_2$
   $\longrightarrow_1^t \;\; id \to id$
2. $id \;::\; \Delta \vdash \forall \alpha \leq K_1.\ \kappa_2 \;\leq\; \forall \alpha \leq K_1.\ \kappa_2$
   $\longrightarrow_1^t \;\; \forall \alpha \leq id.\ id$
3. $id \;::\; \Delta \vdash \bigwedge[\kappa_1 .. \kappa_n] \;\leq\; \bigwedge[\kappa_1 .. \kappa_n]$
   $\longrightarrow_1^t \;\; AE[id \,..\, id]$

II. Cut simplification

1. $id \,;\, c$
   $\longrightarrow_1^t \;\; c$
2. $c \,;\, id$
   $\longrightarrow_1^t \;\; c$
3. $AE[\xi_1 .. \xi_m] \,;\, AE[\delta_1 .. \delta_n]$ where each $\delta_i :: \Delta \vdash \kappa_{j_i} \;\leq\; \iota_i$
   $\longrightarrow_1^t \;\; AE[(\xi_{j_1} \,;\, \delta_1) \,..\, (\xi_{j_n} \,;\, \delta_n)]$
4. $V_\alpha(AE[\xi]) \,;\, \delta$
   $\longrightarrow_1^t \;\; V_\alpha(AE[\xi \,;\, \delta])$
5. $V_\alpha(id) \,;\, \delta$
   $\longrightarrow_1^t \;\; V_\alpha(AE[\delta])$
6. $C \to \xi \,;\, D \to \delta$
   $\longrightarrow_1^t \;\; (D \,;\, C) \to (\xi \,;\, \delta)$
7. $(\forall \alpha \leq C.\ \xi) \,;\, (\forall \alpha \leq D.\ \delta)$ where $D :: \Delta \vdash K_3 \;\leq\; K_2$
   $\longrightarrow_1^t \;\; \forall \alpha \leq (D \,;\, C).\ (\xi[D @ \alpha \leq K_3] \;\;;\;\; \delta)$

Rules I.1, I.2, and I.3 together restrict the form of instances of reflexivity in normal form derivations to reflexivity between variables.

Rules II.1 and II.2 eliminate instances of transitivity with an instance of reflexivity as one of their immediate subderivations.

Rule II.3 simplifies instances of transitivity whose subderivations are both instances of CSub-AE by pushing the application of transitivity toward the leaves of the derivation. Similarly, rule II.4 simplifies instances of transitivity whose left-hand subderivation ends with the rule CSub-TVar by pushing the application of transitivity deeper, toward the leaves of the derivation. (Rule II.5 handles the simpler situation where the premise of CSub-TVar is an instance of CSub-Id, in which case no new instance of transitivity need be created.) Rule II.6 simplifies instances of transitivity whose subderivations are both instances of CSub-Arrow by pushing the applications of transitivity toward the leaves of the derivation.

Rule II.7 — the keystone of the definition — simplifies instances of transitivity whose subderivations are both instances of CSub-All. Pictorially, it rewrites the derivation

$$\dfrac{\dfrac{\dfrac{C}{\Delta \vdash K_2 \leq K_1} \quad \dfrac{\xi}{\Delta, \alpha \leq K_2 \vdash \kappa_1 \leq \kappa_2}}{\Delta \vdash \forall \alpha \leq K_1.\ \kappa_1 \leq \forall \alpha \leq K_2.\ \kappa_2} \quad \dfrac{\dfrac{D}{\Delta \vdash K_3 \leq K_2} \quad \dfrac{\delta}{\Delta, \alpha \leq K_3 \vdash \kappa_2 \leq \kappa_3}}{\Delta \vdash \forall \alpha \leq K_2.\ \kappa_2 \leq \forall \alpha \leq K_3.\ \kappa_3}}{\Delta \vdash \forall \alpha \leq K_1.\ \kappa_1 \leq \forall \alpha \leq K_3.\ \kappa_3}$$

as

$$\dfrac{\dfrac{D}{\Delta \vdash K_3 \leq K_2} \quad \dfrac{C}{\Delta \vdash K_2 \leq K_1}}{\Delta \vdash K_3 \leq K_1} \qquad \dfrac{\dfrac{\xi[D@\alpha{\leq}K_3]}{\Delta, \alpha{\leq}K_3 \vdash \kappa_1 \leq \kappa_2} \quad \dfrac{\delta}{\Delta, \alpha{\leq}K_3 \vdash \kappa_2 \leq \kappa_3}}{\Delta, \alpha{\leq}K_3 \vdash \kappa_1 \leq \kappa_3}$$

$$\Delta \vdash \forall\alpha{\leq}K_1.\ \kappa_1 \leq \forall\alpha{\leq}K_3.\ \kappa_3.$$

4.2.4.2. Definition: Let $\longrightarrow_1$ be the compatible closure of $\longrightarrow_1^t$, that is:

if $c \longrightarrow_1^t c'$, then $c \longrightarrow_1 c'$
if $\xi_i \longrightarrow_1 \xi_i'$, then $AE[\xi_1..\xi_i..\xi_n] \longrightarrow_1 AE[\xi_1..\xi_i'..\xi_n]$
if $c \longrightarrow_1 c'$, then $(c\ ;\ d) \longrightarrow_1 (c'\ ;\ d)$
if $d \longrightarrow_1 d'$, then $(c\ ;\ d) \longrightarrow_1 (c\ ;\ d')$
if $C \longrightarrow_1 C'$, then $V_\alpha(C) \longrightarrow_1 V_\alpha(C')$
if $C \longrightarrow_1 C'$, then $(\forall\alpha{\leq}C.\ \delta) \longrightarrow_1 (\forall\alpha{\leq}C'.\ \delta)$
if $\delta \longrightarrow_1 \delta'$, then $(\forall\alpha{\leq}C.\ \delta) \longrightarrow_1 (\forall\alpha{\leq}C.\ \delta')$
if $C \longrightarrow_1 C'$, then $(C{\rightarrow}\delta) \longrightarrow_1 (C'{\rightarrow}\delta)$
if $\delta \longrightarrow_1 \delta'$, then $(C{\rightarrow}\delta) \longrightarrow_1 (C{\rightarrow}\delta')$.

4.2.4.3. Remark: A *redex* in a subtyping derivation c is a subderivation d that matches the left-hand side of one of the simplification rules. The corresponding right-hand side, d', is the *contractum* of d. When $c \longrightarrow_1 c'$ by replacing d with d', we say that c *reduces* to c' in one step.

4.2.4.4. Definition: A *normal-form* subtyping derivation is one that contains no redices. Similarly, a *I-normal-form* subtyping derivation is one that contains no I-redices (instances of the left hand sides of any of the rules in group I).

4.2.4.5. Definition: Let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow_1$.

Having shown already that weakening and narrowing preserve validity, it is a simple matter to check that the rewriting rules do too.

4.2.4.6. Lemma: [Replacement] If c is a valid subtyping derivation and c' is formed from c by replacing some subderivation d by a valid derivation d' with the same conclusion as d, then c' is valid and has the same conclusion as c.

Proof: Immediate, since none of the CSub rules place any requirements on the shape of the derivations of their hypotheses. □

4.2.4.7. Remark: In type assignment systems (c.f. 2.4), a typing statement $\Gamma \vdash e \in \tau$ is often thought of as a kind of sentence, where e is the *subject* and τ is the *predicate*. When $e \longrightarrow^* e'$ according to some system of rewrite rules (usually read "e evaluates to e'") we speak of "reducing the subject." If the type assignment system in question has the property that whenever $\Gamma \vdash e \in \tau$ is derivable and $e \longrightarrow^* e'$ the statement $\Gamma \vdash e' \in \tau$ is also derivable, we say that the typing relation is closed under subject reduction, or that the system in question has the *subject reduction property.*

The same terminology can conveniently be applied to statements of the form $c :: \Delta \vdash k \leq i$ if we think of $\Delta \vdash k \leq i$ as a kind of predicate — true of valid subtyping derivations with this conclusion — and derivation simplification as a kind of evaluation.

4.2.4.8. Lemma: [Subject reduction for the simplification rules] If c is a valid subtyping derivation such that $c :: \Delta \vdash k \leq i$ and $c \longrightarrow^* d$, then $d :: \Delta \vdash k \leq i$.

Proof: First, note that all of the rewrite rules map valid subtyping derivations to valid derivations with the same conclusions (except for rule II.7 the argument is straightforward; the case for II.7

follows from the narrowing lemma for canonical types (4.2.3.6)). This observation extends to $\longrightarrow_1$ by the replacement lemma (4.2.4.6) and to $\longrightarrow^*$ by induction. □

4.2.5 Termination of the Normalization Rules

Now we must show that every canonical subtyping derivation can be rewritten into one in normal form. The crux of the argument will be that whenever an instance of transitivity is reduced, the size of the intermediate type — or *cut type* — is decreased in any new instances of transitivity in the result. The argument is slightly delicate, because both rule II.3 and the narrowing operation in rule II.7 can create copies of subderivations that may themselves contain redices. This is handled by reducing redices with the largest cut types first.

The argument is also slightly complicated by the fact that rule II.4 creates a new instance of transitivity whose intermediate type is exactly the same as that of the original. In this case, we argue that the resulting derivation is simpler because an instance of transitivity has been pushed toward the leaves of the derivation.

4.2.5.1. Definition: A subtyping derivation of the form $c\ ;d$ is called a *compound derivation.*

4.2.5.2. Definition: The *cut type* of a compound derivation $(c :: \Delta \vdash k_1 \leq k_2)\ ;(d :: \Delta \vdash k_2 \leq k_3)$, written $\mathit{cut\text{-}type}(c\ ;d)$, is k_2.

4.2.5.3. Definition: The *cut size* of a compound derivation $c\ ;d$ is

$$\mathit{cut\text{-}size}(c\ ;d) = \mathit{size}(\mathit{cut\text{-}type}(c\ ;d)).$$

4.2.5.4. Definition: The *complexity* of a compound derivation c, written $\mathit{complexity}(c)$, is the pair $\langle \mathit{cut\text{-}size}(c), \mathit{size}(c)\rangle$, ordered lexicographically.

4.2.5.5. Definition: A II-redex c is an *innermost* redex of complexity m if no proper subderivation of c is a II-redex of complexity m.

4.2.5.6. Definition: The *rewriting strategy* for normalizing subtyping derivations comprises the following steps:

1. *Perform I-reductions in any order until a I-normal form is reached.*
2. *If there are any remaining II-redexes, let m be the largest complexity of any remaining II-redex. Choose an innermost redex of complexity m, reduce it, and return to step 2.*

Since each I-reduction decreases the size of the types in the conclusions of any new I-redices it creates, the I-rules are clearly strongly normalizing. Moreover, none of the II-rules can create new I-redices when applied to I-normal forms, so we need worry no further about the I-rules.

Progress in normalizing the II-redices of a derivation c is measured as follows:

4.2.5.7. Definition: The *total complexity* of a derivation c, written $\mathit{total}(c)$, is the pair $\langle m, n\rangle$, where m is the maximum complexity of any compound subderivation of c and n is the number of compound subderivations of this size, ordered lexicographically.

4.2.5.8. Lemma: For any derivation c, the weakening $(c, \alpha{\leq}K)$ contains no II-redices not already present in c.

Proof: Immediate from the definition of weakening (4.2.3.2). □

4.2.5.9. Lemma: Let $c :: \Delta, \alpha{\leq}I \vdash k \leq k'$ and $D :: \Delta \vdash K \leq I$. Then the cut type of every new II-redex in the narrowing $c[D@\alpha{\leq}K]$ is I.

Proof: By induction on the structure of c.

Case: $c \equiv AE[\xi_1..\xi_n]$
By the induction hypothesis and CSUB-AE.

Case: $c \equiv id$
Immediate.

Case: $c \equiv c_1 \,;\, c_2$
Can't happen.

Case: $c \equiv V_\beta(C)$

Subcase: $\beta \neq \alpha$
By the induction hypothesis and CSUB-TVAR.

Subcase: $\beta = \alpha$
By the definition of narrowing (4.2.3.5),
$$(V_\alpha(C))[D@\alpha{\leq}K] = V_\alpha((D, \alpha{\leq}K) \,;\, (C[D@\alpha{\leq}K])).$$
By the induction hypothesis, the cut type of every new II-redex in $C[D@\alpha{\leq}K]$ is I. By Lemma 4.2.5.8, there are no new redices in $(D, \alpha{\leq}K)$. The cut type of the new redex $((D, \alpha{\leq}K) \,;\, (C[D@\alpha{\leq}K]))$ is I.

Case: $c \equiv C{\rightarrow}\xi$
By the induction hypothesis and CSUB-ARROW.

Case: $c \equiv \forall\beta{\leq}C.\ \xi$
By the definition of narrowing (4.2.3.5),
$$(\forall\beta{\leq}C.\ \xi)[D@\alpha{\leq}K] = \forall\beta{\leq}(C[D@\alpha{\leq}K]).\ (\xi[(D, \beta{\leq}I_1)@\alpha{\leq}K]),$$
where $C :: \Delta, \alpha{\leq}I \vdash I_1 \leq K_1$. By the induction hypothesis, every new II-redex in $C[D@\alpha{\leq}K]$ has cut type I. By Lemma 4.2.5.8, $(D, \beta{\leq}I_1) :: \Delta, \beta{\leq}I_1 \vdash K \leq I$ has no new redices. By the induction hypothesis, every new II-redex in $\xi_2[(D, \beta{\leq}I_1)@\alpha{\leq}K]$ has cut type I. □

4.2.5.10. Lemma: Let o be a derivation and $(c \,;\, d)$ an innermost II-redex of maximum complexity in o. Then $total(o) > total(o')$, where o' is the result of replacing $(c \,;\, d)$ by its contractum, e.

Proof: By cases on the rule used to reduce $c \,;\, d$ to e.

Case II.1: $c \equiv id \qquad e \equiv d$
This reduction removes one II-redex of maximum complexity from o.

Case II.2: $d \equiv id \qquad e \equiv c$
Similar.

Case II.3:
$c \equiv AE[\xi_1..\xi_m] \qquad d \equiv AE[\delta_1..\delta_n] \qquad \delta_i :: \Delta \vdash \kappa_{j_i} \leq \iota_i \qquad e \equiv AE[(\xi_{j_1} \,;\, \delta_1) \,..\, (\xi_{j_n} \,;\, \delta_n)]$
This reduction removes one redex of maximum complexity with $cut\text{-}size = size(\bigwedge[\kappa_1..\kappa_n])$ and creates n new redices of cut sizes $size(\kappa_{j_1})$, ..., $size(\kappa_{j_n})$, all of which are strictly smaller. In addition, some redices in the ξ_i's may be copied, but these must also have smaller cut size because $c \,;\, d$ is an innermost redex of maximal complexity.

Case II.4: $c \equiv V_\alpha(AE[\xi]) \qquad d \equiv \delta \qquad e \equiv V_\alpha(AE[\xi \,;\, \delta])$
This reduction removes one redex of maximum complexity and creates one new one with exactly the same cut type. However, the size of the new redex is $size(\xi \,;\, \delta) < size(c \,;\, d)$, so its complexity is smaller. The complexities of other redices in o are unchanged.

Case II.5: $c \equiv V_\alpha(id) \qquad d \equiv \delta \qquad e \equiv V_\alpha(AE[\delta])$
This reduction removes one redex of maximum complexity and creates no new redices.

Case II.6: $c\,;d \equiv (C{\to}\xi)\,;(D{\to}\delta)$
$e \quad \equiv (D\,;C){\to}(\xi\,;\delta)$
where $C{::}\Delta \vdash K_2 \le K_1$
$D{::}\Delta \vdash K_3 \le K_2$
$\xi\ {::}\Delta \vdash \kappa_1 \le \kappa_2$
$\delta\ {::}\Delta \vdash \kappa_2 \le \kappa_3$.

This reduction removes one redex of maximal complexity with cut type $K_2{\to}\kappa_2$ and creates two new redices, one with cut type K_2 and one with cut type κ_2.

Case II.7: $c\,;d \equiv (\forall\alpha{\le}C.\ \xi)\,;(\forall\alpha{\le}D.\ \delta)$
$e \quad \equiv \forall\alpha{\le}(D\,;C).\ ((\xi[D@\alpha{\le}K_3])\,;\delta)$
where $C{::}\Delta \vdash K_2 \le K_1$
$D{::}\Delta \vdash K_3 \le K_2$
$\xi\ {::}\Delta, \alpha{\le}K_2 \vdash \kappa_1 \le \kappa_2$
$\delta\ {::}\Delta, \alpha{\le}K_3 \vdash \kappa_2 \le \kappa_3$.

The cut type of $(c\,;d)$ is $\forall\alpha{\le}K_2.\ \kappa_2$. The new II-redices in e are:

- $(D\,;C)$, with cut type K_2;
- $(\xi[D_1@\alpha{\le}K_3]\,;\delta)$, with cut type κ_2; and
- new redices in $\xi[D@\alpha{\le}K_3]$, with cut type K_2 (by Lemma 4.2.5.9).

Thus, rewriting $(c\,;d)$ as e removes one redex of maximal complexity with cut type $\forall\alpha{\le}K_2.\ \kappa_2$ and creates some new redices with strictly smaller cut types K_2 and κ_2. □

Since steps 1 and 2 of our rewriting stretegy each terminate, and since step 2 introduces no new I-redices, we are assured that every derivation is eventually rewritten by this strategy to one in I/II-normal form:

4.2.5.11. Theorem: The given rewriting strategy is normalizing.

4.2.5.12. Corollary: If there is any derivation of a canonical subtyping statement $\Delta \vdash k \le i$, then there is one in normal form.

4.2.6 Shapes of Normal-Form Subtyping Derivations

4.2.6.1. Lemma: If $id :: \Delta \vdash k \le i$ is a normal-form subtyping derivation, then $k \equiv i \equiv \alpha$ for some variable α.

Proof: By the form of the I-rules (4.2.4.1). □

4.2.6.2. Lemma: If e is a normal-form subtyping derivation, then $e \not\equiv (c\,;d)$.

Proof: By induction on the size of e, with a case analysis on the possible forms of c and d. In each case, either the induction hypothesis or one of the II-rules guarantees that a well-formed derivation of the form $c\,;d$ is not in normal form:

c	d	Reason
id	any	II.1
$c_1 \,;\, c_2$	any	induction hypothesis
any	id	II.2
any	$d_1 \,;\, d_2$	induction hypothesis
$AE[\xi_1..\xi_m]$	$AE[\delta_1..\delta_n]$	II.3
$V_\alpha(c_1 \,;\, c_2)$	δ	induction hypothesis
$V_\alpha(AE[\xi])$	δ	II.4
$V_\alpha(id)$	δ	II.5
$C \rightarrow \xi$	$V_\beta(D)$	ill formed
$C \rightarrow \xi$	$D \rightarrow \delta$	II.6
$C \rightarrow \xi$	$\forall \beta \leq D.\ \delta$	ill formed
$\forall \alpha \leq C.\ \xi$	$V_\beta(D)$	ill formed
$\forall \alpha \leq C.\ \xi$	$D \rightarrow \delta$	ill formed
$\forall \alpha \leq C.\ \xi$	$\forall \alpha \leq D.\ \delta$	II.7

□

4.2.6.3. Lemma: [Syntax-directedness of canonical subtyping]

1. If $\Delta \vdash K \leq I$, then for every $\iota \in I$ there is some $\kappa \in K$ such that $\Delta \vdash \kappa \leq \iota$.
2. If $\Delta \vdash K_1 \rightarrow \kappa_2 \leq \iota$, then $\iota \equiv I_1 \rightarrow \iota_2$ with $\Delta \vdash I_1 \leq K_1$ and $\Delta \vdash \kappa_2 \leq \iota_2$.
3. If $\Delta \vdash \forall \alpha \leq K_1.\ \kappa_2 \leq \iota$, then $\iota \equiv \forall \alpha \leq I_1.\ \iota_2$ with $\Delta \vdash I_1 \leq K_1$ and $\Delta, \alpha \leq I_1 \vdash \kappa_2 \leq \iota_2$.
4. If $\Delta \vdash \alpha \leq \kappa$, then either $\kappa \equiv \alpha$ or else $\Delta \vdash \Delta(\alpha) \leq \bigwedge[\kappa]$.

Moreover, when the given derivation is in normal form, the derivations promised in each clause are proper subderivations of the original.

Proof: In each case, we are given a derivation of $\Delta \vdash k \leq i$. By Corollary 4.2.5.12, there exists a normal-form derivation of the same statement. By Lemmas 4.2.6.1 and 4.2.6.2, this normal derivation does not end with an instance of CSub-Refl (except CSub-Refl applied to variables) or CSub-Trans. The desired result follows by inspection of the remainder of the CSub rules. □

4.2.6.4. Corollary: If $\Delta \vdash \kappa \leq \alpha$, then either $\kappa \equiv \alpha$ or else $\kappa \equiv \beta$ for some β with $\Delta \vdash \Delta(\beta) \leq \bigwedge[\alpha]$.

4.2.7 Equivalence of Ordinary and Canonical Subtyping

Our next task is to establish that the subtyping relations on ordinary types and on the corresponding canonical types are equivalent in an appropriate sense. We accomplish this by defining a *flattening mapping* $\flat$ from ordinary types to canonical types with the following properties:

1. Flattening preserves subtyping: if $\Gamma \vDash^{\wedge} \sigma \leq \tau$, then $\Gamma^\flat \vDash^\flat \sigma^\flat \leq \tau^\flat$.
2. Flattening yields a type equivalent to the original: $\Gamma \vDash^{\wedge} \tau^\flat \sim \tau$.

The second observation (plus narrowing and the fact that the identity injection from the set of canonical types into the set of ordinary types preserves subtyping — if $\Delta \vDash^\flat \kappa \leq \iota$, then $\Delta \vDash^{\wedge} \kappa \leq \iota$) implies the converse of the first: if the translation of a statement is provable in the canonical system then the original statement is also provable. Thus, the flattening mapping also *reflects* subtyping.

4.2.7.1. Definition: The *flattening mapping* $\flat$ from ordinary types to composite canonical types is defined as follows:

$$\begin{array}{lcl}
\alpha^\flat & = & \bigwedge[\alpha] \\
(\sigma{\to}\tau)^\flat & = & \bigwedge[\sigma^\flat{\to}\kappa \mid \kappa \in \tau^\flat] \\
(\forall\alpha{\leq}\sigma.\ \tau)^\flat & = & \bigwedge[\forall\alpha{\leq}\sigma^\flat.\ \kappa \mid \kappa \in \tau^\flat] \\
\bigwedge[\tau_1..\tau_n]^\flat & = & \bigcup_i \tau_i^\flat
\end{array}$$

This mapping is extended pointwise to contexts:

$$\begin{array}{lcl}
\{\}^\flat & = & \{\} \\
(\Gamma, \alpha{\leq}\tau)^\flat & = & \Gamma^\flat, \alpha{\leq}\tau^\flat \\
(\Gamma, x{:}\tau)^\flat & = & \Gamma^\flat, x{:}\tau^\flat
\end{array}$$

4.2.7.2. Lemma: [$\flat$ preserves subtyping] If $\Gamma \vdash^\wedge \sigma \leq \tau$ then $\Gamma^\flat \vdash^\flat \sigma^\flat \leq \tau^\flat$.

Proof: By induction on a derivation of $\Gamma \vdash^\wedge \sigma \leq \tau$. Proceed by cases on the final step of the derivation.

Case Sub-Refl: $\sigma \equiv \tau$

By CSub-Refl.

Case Sub-Trans:

By the induction hypothesis and CSub-Trans.

Case Sub-TVar: $\sigma \equiv \alpha \qquad \tau \equiv \Gamma(\alpha)$

By the definition of canonical subtyping, $(\Gamma(\alpha))^\flat \equiv \Gamma^\flat(\alpha) \equiv \bigwedge[\kappa_1..\kappa_n]$. The desired result is checked by constructing the following derivation:

$$\dfrac{\dfrac{\dfrac{\Gamma^\flat \vdash^\flat \kappa_1 \leq \kappa_1}{\Gamma^\flat \vdash^\flat \Gamma^\flat(\alpha) \leq \bigwedge[\kappa_1]}\,\text{(CSUB-AE)}}{\Gamma^\flat \vdash^\flat \alpha \leq \kappa_1}\,\text{(CSUB-TVAR)} \quad \cdots \quad \dfrac{\dfrac{\dfrac{\Gamma^\flat \vdash^\flat \kappa_n \leq \kappa_n}{\Gamma^\flat \vdash^\flat \Gamma^\flat(\alpha) \leq \bigwedge[\kappa_n]}\,\text{(CSUB-AE)}}{\Gamma^\flat \vdash^\flat \alpha \leq \kappa_n}\,\text{(CSUB-TVAR)}}{\Gamma^\flat \vdash^\flat \bigwedge[\alpha] \leq \bigwedge[\kappa_1..\kappa_n]}\,\text{(CSUB-AE)}$$

Case Sub-Arrow: $\sigma \equiv \sigma_1{\to}\sigma_2 \qquad \tau \equiv \tau_1{\to}\tau_2$
$\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq \tau_2$

By the induction hypothesis, $\Gamma^\flat \vdash^\flat \tau_1^\flat \leq \sigma_1^\flat$ and $\Gamma^\flat \vdash^\flat \sigma_2^\flat \leq \tau_2^\flat$. By the syntax-directedness of canonical subtyping (Lemma 4.2.6.3(2)), for every $\iota_j \in \tau_2^\flat$ there is some $\kappa_j \in \sigma_2^\flat$ such that $\Gamma^\flat \vdash^\flat \kappa_j \leq \iota_j$; in each case, CSub-Arrow gives $\Gamma^\flat \vdash^\flat \sigma_1^\flat{\to}\kappa_j \leq \tau_1^\flat{\to}\iota_j$. By CSub-AE, $\Gamma^\flat \vdash^\flat \bigwedge[\sigma_1^\flat{\to}\kappa \mid \kappa \in \sigma_2^\flat] \leq \bigwedge[\tau_1^\flat{\to}\iota \mid \iota \in \tau_2^\flat]$, as required.

Case Sub-All: $\sigma \equiv \forall\alpha{\leq}\sigma_1.\ \sigma_2 \qquad \tau \equiv \forall\alpha{\leq}\tau_1.\ \tau_2$

Similar.

Case Sub-Inter-G: $\tau \equiv \bigwedge[\tau_1..\tau_n] \qquad \Gamma \vdash \sigma \leq \tau_i \quad \text{for each } \tau_i$

For each τ_i, the induction hypothesis gives $\Gamma^\flat \vdash^\flat \sigma^\flat \leq \tau_i^\flat$. By the syntax-directedness of canonical subtyping (Lemma 4.2.6.3(1)), for each $\iota_{ij} \in \tau_i^\flat$ there is some $\kappa_{ij} \in \sigma^\flat$ such that $\Gamma^\flat \vdash^\flat \kappa_{ij} \leq \iota_{ij}$. Combining these derivations for all the τ_i's, we have $\Gamma^\flat \vdash^\flat \sigma^\flat \leq \bigcup_i \tau_i^\flat$ by CSub-AE.

Case SUB-INTER-LB: $\tau \equiv \sigma_i \qquad \sigma \equiv \bigwedge[\sigma_1..\sigma_n]$

Let $\sigma_i{}^\flat \equiv \bigwedge[\sigma_{i1} .. \sigma_{im}]$. Then the derivation

$$\frac{\Gamma^\flat \stackrel{\flat}{\vdash} \sigma^\flat_{i1} \le \sigma^\flat_{i1} \qquad \cdots \qquad \Gamma^\flat \stackrel{\flat}{\vdash} \sigma^\flat_{im} \le \sigma^\flat_{im}}{\Gamma^\flat \stackrel{\flat}{\vdash} \bigcup_i \sigma^\flat_i \le \sigma^\flat_i}\ (\text{CSUB-AE})$$

establishes the desired result.

Case SUB-DIST-IA: $\sigma \equiv \bigwedge[\sigma' \to \tau_1 .. \sigma' \to \tau_n] \qquad \tau \equiv \sigma' \to \bigwedge[\tau_1..\tau_n]$

By the definition of flattening, $\sigma^\flat \equiv \tau^\flat$. The result follows by CSUB-REFL.

Case SUB-DIST-IQ: $\sigma \equiv \bigwedge[\forall\alpha{\le}\sigma'.\ \tau_1 .. \forall\alpha{\le}\sigma'.\ \tau_n] \qquad \tau \equiv \forall\alpha{\le}\sigma'.\ \bigwedge[\tau_1..\tau_n]$

Similar. □

Next, we remark that subtyping on canonical types is preserved when canonical types are read as ordinary types.

4.2.7.3. Lemma: If $\Delta \stackrel{\flat}{\vdash} k \le i$, then $\Delta \stackrel{\wedge}{\vdash} k \le i$.

Proof: By induction on the structure of a canonical subtyping derivation. □

The last fact needed to establish the equivalence of subtyping on ordinary and canonical types is that the flattening transformation always yields a type equivalent to the original.

4.2.7.4. Lemma: $\Gamma \stackrel{\wedge}{\vdash} \tau^\flat \sim \tau$ for all $F_\wedge$ types τ and contexts Γ.

Proof: By induction on the structure of τ.

Case: $\tau \equiv \alpha$

By SUB-INTER-LB and SUB-INTER-G.

Case: $\tau \equiv \tau_1 \to \tau_2$

By the definition of flattening (4.2.7.1), $\tau^\flat \equiv \bigwedge[\tau_1^\flat \to \kappa \mid \kappa \in \tau_2^\flat]$. By derived rule D-DIST-IA (4.1.1), $\Gamma \vdash \tau^\flat \sim (\tau_1^\flat \to \bigwedge[\kappa \mid \kappa \in \tau_2^\flat])$, i.e., $\Gamma \vdash \tau^\flat \sim \tau_1^\flat \to \tau_2^\flat$. By the induction hypothesis, $\Gamma \vdash \tau_1^\flat \sim \tau_1$ and $\Gamma \vdash \tau_2^\flat \sim \tau_2$. The desired result follows by D-CONG-ARROW (4.1.7) and SUB-TRANS.

Case: $\tau \equiv \forall\alpha{\le}\tau_1.\ \tau_2$

Similar.

Case: $\tau \equiv \bigwedge[\tau_1..\tau_n]$

By the induction hypothesis, $\Gamma \vdash \tau_i^\flat \sim \tau_i$ for each i. By D-CONG-INTER (4.1.7), $\Gamma \vdash \bigwedge[\tau_1^\flat .. \tau_n^\flat] \sim \bigwedge[\tau_1..\tau_n]$. The result then follows by D-ABSORB (4.1.2) and SUB-TRANS. □

Combining this with the previous lemma, we can show that the translation from ordinary to canonical types reflects subtyping in $F_\wedge$.

4.2.7.5. Lemma: [$\flat$ reflects subtyping] If $\Gamma^\flat \stackrel{\flat}{\vdash} \sigma^\flat \le \tau^\flat$, then $\Gamma \stackrel{\wedge}{\vdash} \sigma \le \tau$.

Proof: By Lemma 4.2.7.3, $\Gamma^\flat \stackrel{\wedge}{\vdash} \sigma^\flat \le \tau^\flat$. By Lemma 4.2.7.4, $\Gamma \stackrel{\wedge}{\vdash} \Gamma(\alpha) \le \Gamma^\flat(\alpha)$ for each $\alpha \in dom(\Gamma^\flat)$. By narrowing (4.2.3.6), $\Gamma \stackrel{\wedge}{\vdash} \sigma^\flat \le \tau^\flat$. By Lemma 4.2.7.4 again, $\Gamma \stackrel{\wedge}{\vdash} \sigma \le \sigma^\flat$ and $\Gamma \stackrel{\wedge}{\vdash} \tau^\flat \le \tau$. By two applications of SUB-TRANS, $\Gamma \stackrel{\wedge}{\vdash} \sigma \le \tau$. □

Lemmas 4.2.7.2 and 4.2.7.5 together show that the subtype relations on ordinary and canonical types correspond appropriately:

4.2.7.6. Theorem: [Equivalence of ordinary and canonical subtyping]

$\Gamma \stackrel{\wedge}{\vdash} \sigma \le \tau$ iff $\Gamma^\flat \stackrel{\flat}{\vdash} \sigma^\flat \le \tau^\flat$.

4.2.8 Subtyping Algorithm

The definition of canonical subtyping leads directly to one algorithm for deciding the subtype relation on $F_\wedge$ types: to check whether $\Gamma \vdash \sigma \leq \tau$, flatten Γ, σ, and τ and check whether $\Gamma^\flat \vdash \sigma^\flat \leq \tau^\flat$. In this section we describe an alternative algorithm that operates directly on $F_\wedge$ types, effectively performing the flattening translation on the fly.

Given Γ, σ, and τ, the new algorithm first performs a complete analysis of the structure of τ. Whenever τ has the form $\tau_1 \to \tau_2$ or $\forall \alpha \leq \tau_1.\ \tau_2$, it pushes the left-hand side — τ_1 or $\alpha \leq \tau_1$ — onto a queue of pending left-hand sides and proceeds recursively with the analysis of τ_2. When τ has the form of an intersection, it calls itself recursively on each of the elements. When τ has finally been reduced to a type variable, the algorithm begins analyzing σ, matching left-hand sides of arrow and polymorphic types against the queue of pending left-hand sides from τ. In the base case, when both σ and τ have been reduced to variables, the algorithm first checks whether they are identical; if so, and if the queue of pending left-hand sides is empty, the algorithm immediately returns *true*. Otherwise, the variable σ is replaced by its upper bound from Γ and the analysis continues as before.

The algorithm presented here generalizes one described by Reynolds for deciding the subtype relation of Forsythe [personal communication, 1988].

4.2.8.1. Definition: Let X be a finite sequence of elements of the set

$$\{\tau \mid \tau \text{ a type}\} \cup \{\alpha \leq \tau \mid \alpha \text{ a type variable and } \tau \text{ a type}\}.$$

Define the type $X \Rightarrow \tau$ as follows:

$$\begin{array}{lcl} [\,] \Rightarrow \tau & = & \tau \\ [\sigma, X] \Rightarrow \tau & = & \sigma \to (X \Rightarrow \tau) \\ [\alpha \leq \sigma, X] \Rightarrow \tau & = & \forall \alpha \leq \sigma.\ (X \Rightarrow \tau). \end{array}$$

From the definitions of $\flat$ and $X \Rightarrow \tau$, the following facts are immediate:

4.2.8.2. Lemma:

1. $(X \Rightarrow (\bigwedge[\tau_1..\tau_n]))^\flat = \bigcup_i (X \Rightarrow \tau_i)^\flat$.
2. $(X \Rightarrow (\tau_1 \to \tau_2))^\flat = ([X, \tau_1] \Rightarrow \tau_2))^\flat$.
3. $(X \Rightarrow (\forall \alpha \leq \tau_1.\ \tau_2))^\flat = ([X, \alpha \leq \tau_1] \Rightarrow \tau_2)^\flat$.

4.2.8.3. Remark: Note that every type τ has either the form $X \Rightarrow \alpha$ or the form $X \Rightarrow \bigwedge[\tau_1..\tau_n]$ for a unique X.

4.2.8.4. Definition: The four-place algorithmic subtyping relation $\Gamma \stackrel{\flat}{\vdash} \sigma \leq X \Rightarrow \tau$ is the least relation closed under the following rules:

$$\frac{\text{for all } i,\ \Gamma \vdash \sigma \leq X \Rightarrow \tau_i}{\Gamma \vdash \sigma \leq X \Rightarrow \bigwedge[\tau_1..\tau_n]} \qquad \text{(ASubR-Inter)}$$

$$\frac{\text{for some } i,\ \Gamma \vdash \sigma_i \leq X \Rightarrow \alpha}{\Gamma \vdash \bigwedge[\sigma_1..\sigma_n] \leq X \Rightarrow \alpha} \qquad \text{(ASubL-Inter)}$$

$$\frac{\Gamma \vdash \tau_1 \leq [\,] \Rightarrow \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq X_2 \Rightarrow \alpha}{\Gamma \vdash \sigma_1 \to \sigma_2 \leq [\tau_1, X_2] \Rightarrow \alpha} \qquad \text{(ASubL-Arrow)}$$

$$\frac{\Gamma \vdash \tau_1 \leq [\,] \Rightarrow \sigma_1 \qquad \Gamma, \beta \leq \tau_1 \vdash \sigma_2 \leq X_2 \Rightarrow \alpha}{\Gamma \vdash \forall \beta \leq \sigma_1.\ \sigma_2 \leq [\beta \leq \tau_1, X_2] \Rightarrow \alpha} \qquad \text{(ASubL-All)}$$

$$\Gamma \vdash \alpha \leq [\,] \Rightarrow \alpha \qquad \text{(ASubL-Refl)}$$

$$\frac{\Gamma \vdash \Gamma(\beta) \leq X \Rightarrow \alpha}{\Gamma \vdash \beta \leq X \Rightarrow \alpha} \qquad \text{(ASubL-TVar)}$$

4.2.8.5. Notation: We sometimes decorate the turnstile symbol $\vdash$ to distinguish algorithmic derivations from derivations in other calculi.

4.2.8.6. Definition: We write $\textsc{Dist}^*_{\Gamma,\bigwedge[X\Rightarrow\tau_1..X\Rightarrow\tau_n]}$ (or just $\textsc{Dist}^*$ when the appropriate subscript is clear) for the following compound derivation:

$$\textsc{Dist}^*_{\Gamma,\bigwedge[[\,]\Rightarrow\tau_1..[\,]\Rightarrow\tau_n]} \quad = \quad \frac{}{\Gamma \vdash^\wedge \bigwedge[[\,]\Rightarrow\tau_1 \,..\, [\,]\Rightarrow\tau_n] \leq [\,]\Rightarrow\bigwedge[\tau_1..\tau_n]}\,\text{(SUB-REFL)}$$

$\textsc{Dist}^*_{\Gamma,\bigwedge[\sigma\to(X'\Rightarrow\tau_1)..\sigma\to(X'\Rightarrow\tau_n)]}$ =

$$\frac{\dfrac{}{\begin{array}{lll}\Gamma & \vdash^\wedge & \bigwedge[\sigma\to(X'\Rightarrow\tau_1) \,..\, \sigma\to(X'\Rightarrow\tau_n)] \\ & \leq & \sigma\to\bigwedge[X'\Rightarrow\tau_1 \,..\, X'\Rightarrow\tau_n]\end{array}}\,\text{(SUB-DIST-IA)} \qquad \dfrac{\dfrac{}{\Gamma \vdash \sigma \leq \sigma} \quad \textsc{Dist}^*_{\Gamma,\bigwedge[X'\Rightarrow\tau_1..X'\Rightarrow\tau_n]}}{\begin{array}{lll}\Gamma & \vdash^\wedge & \sigma\to\bigwedge[X'\Rightarrow\tau_1 \,..\, X'\Rightarrow\tau_n] \\ & \leq & \sigma\to(X'\Rightarrow\bigwedge[\tau_1..\tau_n])\end{array}}\,\text{(SUB-ARROW)}}{\Gamma \vdash^\wedge \bigwedge[\sigma\to(X'\Rightarrow\tau_1) \,..\, \sigma\to(X'\Rightarrow\tau_n)] \leq \sigma\to(X'\Rightarrow\bigwedge[\tau_1..\tau_n])}\,\text{(SUB-TRANS)}$$

$\textsc{Dist}^*_{\Gamma,\bigwedge[\forall\alpha\leq\sigma.\,(X'\Rightarrow\tau_1)..\forall\alpha\leq\sigma.\,(X'\Rightarrow\tau_n)]}$ =

$$\frac{\dfrac{}{\begin{array}{lll}\Gamma & \vdash^\wedge & \bigwedge[\forall\alpha\leq\sigma.\,(X'\Rightarrow\tau_1) \,..\, \forall\alpha\leq\sigma.\,(X'\Rightarrow\tau_n)] \\ & \leq & \forall\alpha\leq\sigma.\,\bigwedge[X'\Rightarrow\tau_1 \,..\, X'\Rightarrow\tau_n]\end{array}}\,\text{(SUB-DIST-IQ)} \qquad \dfrac{\dfrac{}{\Gamma \vdash \sigma \leq \sigma} \quad \textsc{Dist}^*_{(\Gamma,\alpha\leq\sigma),\bigwedge[X'\Rightarrow\tau_1..X'\Rightarrow\tau_n]}}{\begin{array}{lll}\Gamma & \vdash^\wedge & \forall\alpha\leq\sigma.\,\bigwedge[X'\Rightarrow\tau_1 \,..\, X'\Rightarrow\tau_n] \\ & \leq & \forall\alpha\leq\sigma.\,(X'\Rightarrow\bigwedge[\tau_1..\tau_n])\end{array}}\,\text{(SUB-ALL)}}{\Gamma \vdash^\wedge \bigwedge[\forall\alpha\leq\sigma.\,(X'\Rightarrow\tau_1) \,..\, \forall\alpha\leq\sigma.\,(X'\Rightarrow\tau_n)] \leq \forall\alpha\leq\sigma.\,(X'\Rightarrow\bigwedge[\tau_1..\tau_n])}\,\text{(SUB-TRANS)}$$

4.2.8.7. Definition: Let $c :: \Gamma \vdash^! \sigma \leq \tau$ be an algorithmic subtyping derivation. Then $c^\wedge :: \Gamma \vdash^\wedge \sigma \leq \tau$ is the following ordinary derivation:

$$\left(\frac{\text{for all } i,\ c_i :: \Gamma \vdash^! \sigma \leq X\Rightarrow\tau_i}{\Gamma \vdash^! \sigma \leq X\Rightarrow\bigwedge[\tau_1..\tau_n]}\,\text{(ASUBR-INTER)}\right)^\wedge \quad =$$

$$\frac{\dfrac{c_1^\wedge \cdots c_n^\wedge}{\Gamma \vdash^\wedge \sigma \leq \bigwedge[X\Rightarrow\tau_1 \,..\, X\Rightarrow\tau_n]}\,\text{(SUB-INTER-G)} \qquad \begin{array}{l}\textsc{Dist}^* :: \Gamma \vdash^\wedge \bigwedge[X\Rightarrow\tau_1 \,..\, X\Rightarrow\tau_n] \\ \qquad\qquad \leq X\Rightarrow\bigwedge[\tau_1..\tau_n]\end{array}}{\Gamma \vdash^\wedge \sigma \leq X\Rightarrow\bigwedge[\tau_1..\tau_n]}$$

$$\left(\frac{\text{for some } i,\ c_i :: \Gamma \vdash^! \sigma_i \leq X\Rightarrow\alpha}{\Gamma \vdash^! \bigwedge[\sigma_1..\sigma_n] \leq X\Rightarrow\alpha}\,\text{(ASUBL-INTER)}\right)^\wedge \quad =$$

$$\frac{\dfrac{}{\Gamma \vdash^\wedge \bigwedge[\sigma_1..\sigma_n] \leq \sigma_i}\,\text{(SUB-INTER-LB)} \qquad c_i^\wedge}{\Gamma \vdash^\wedge \bigwedge[\sigma_1..\sigma_n] \leq X\Rightarrow\alpha}$$

$$\left(\frac{c_1 :: \Gamma \vdash^{\downarrow} \tau_1 \leq [\,]\Rightarrow\sigma_1 \qquad c_2 :: \Gamma \vdash^{\downarrow} \sigma_2 \leq X\Rightarrow\alpha}{\Gamma \vdash^{\downarrow} \sigma_1\to\sigma_2 \leq [\tau_1, X]\Rightarrow\alpha}\ \text{(ASUBL-ARROW)}\right)^{\wedge} =$$

$$\frac{c_1^{\wedge} \qquad c_2^{\wedge}}{\Gamma \vdash^{\wedge} \sigma_1\to\sigma_2 \leq [\tau_1, X]\Rightarrow\alpha}\ \text{(SUB-ARROW)}$$

$$\left(\frac{c_1 :: \Gamma \vdash^{\downarrow} \tau_1 \leq [\,]\Rightarrow\sigma_1 \qquad c_2 :: \Gamma, \beta{\leq}\tau_1 \vdash^{\downarrow} \sigma_2 \leq X\Rightarrow\alpha}{\Gamma \vdash^{\downarrow} \forall\beta{\leq}\sigma_1.\ \sigma_2 \leq [\beta{\leq}\tau_1, X]\Rightarrow\alpha}\ \text{(ASUBL-ALL)}\right)^{\wedge} =$$

$$\frac{c_1^{\wedge} \qquad c_2^{\wedge}}{\Gamma \vdash^{\wedge} \forall\beta{\leq}\sigma_1.\ \sigma_2 \leq [\beta{\leq}\tau_1, X]\Rightarrow\alpha}\ \text{(SUB-ALL)}$$

$$\left(\frac{}{\Gamma \vdash^{\downarrow} \alpha \leq [\,]\Rightarrow\alpha}\ \text{(ASUBL-REFL)}\right)^{\wedge} = \frac{}{\Gamma \vdash^{\wedge} \alpha \leq [\,]\Rightarrow\alpha}\ \text{(SUB-REFL)}$$

$$\left(\frac{c_1 :: \Gamma \vdash^{\downarrow} \Gamma(\beta) \leq X\Rightarrow\alpha}{\Gamma \vdash^{\downarrow} \beta \leq X\Rightarrow\alpha}\ \text{(ASUBL-TVAR)}\right)^{\wedge} = \frac{\dfrac{}{\Gamma \vdash^{\wedge} \beta \leq \Gamma(\beta)}\ \text{(SUB-TVAR)} \qquad c_1^{\wedge}}{\Gamma \vdash^{\wedge} \beta \leq X\Rightarrow\alpha}\ \text{(SUB-TRANS)}$$

4.2.8.8. Theorem: [Soundness of the algorithm] If $\Gamma \vdash^{\downarrow} \sigma \leq X\Rightarrow\tau$ then $\Gamma \vdash^{\wedge} \sigma \leq X\Rightarrow\tau$.

Proof: By the well-formedness of the translation in Definition 4.2.8.7. □

We must now check that the relation defined by these rules coincides with the subtype relation on canonical types, from which it follows, by Theorem 4.2.7.6, that the algorithm gives a semi-decision procedure for the $F_\wedge$ subtype relation.

4.2.8.9. Lemma: [Completeness of the algorithm with respect to canonical subtyping] If $\Gamma^\flat \vdash^{\flat} \sigma^\flat \leq (X\Rightarrow\tau)^\flat$, then $\Gamma \vdash^{\downarrow} \sigma \leq X\Rightarrow\tau$.

Proof: By induction on the size of a normal-form derivation of $\Gamma^\flat \vdash^{\flat} \sigma^\flat \leq (X\Rightarrow\tau)^\flat$, with a sub-induction on the form of τ and, when $\tau \equiv \alpha$, a sub-sub-induction on the form of σ. Proceed by cases on the form of τ and σ.

Case: $\tau \equiv \bigwedge[\tau_1..\tau_n] \qquad \Gamma^\flat \vdash^{\flat} \sigma^\flat \leq (X\Rightarrow\bigwedge[\tau_1..\tau_n])^\flat$

By Lemma 4.2.8.2(1), $\Gamma^\flat \vdash^{\flat} \sigma^\flat \leq \bigcup_i (X\Rightarrow\tau_i)^\flat$. By the syntax-directedness of canonical subtyping (4.2.6.3(1)), for every $\iota \in \bigcup_i (X\Rightarrow\tau_i)^\flat$ there is some $\kappa \in \sigma^\flat$ and a subderivation of the original whose conclusion is $\Gamma^\flat \vdash^{\flat} \kappa \leq \iota$. In particular, for each i and every $\iota \in (X\Rightarrow\tau_i)^\flat$ there is some $\kappa \in \sigma^\flat$ such that $\Gamma^\flat \vdash^{\flat} \kappa \leq \iota$. By CSub-AE, $\Gamma^\flat \vdash^{\flat} \sigma^\flat \leq (X\Rightarrow\tau_i)^\flat$. This derivation is no larger than the original and τ_i is smaller than τ, so, by the main or sub-induction hypothesis, $\Gamma \vdash^{\downarrow} \sigma \leq X\Rightarrow\tau_i$. Then by rule ASubR-Inter, $\Gamma \vdash^{\downarrow} \sigma \leq \bigwedge[\tau_1..\tau_n]$.

Case: $\tau \equiv \alpha \qquad \sigma \equiv \bigwedge[\sigma_1..\sigma_m] \qquad \Gamma^\flat \vdash^{\flat} \bigwedge[\sigma_1..\sigma_n]^\flat \leq (X\Rightarrow\alpha)^\flat$

Since $(X\Rightarrow\alpha)^\flat \equiv \bigwedge[\iota]$ is a singleton, the syntax-directedness of canonical subtyping (4.2.6.3(1)) implies that for some $\kappa \in \sigma^\flat$ we have $\Gamma^\flat \vdash^{\flat} \kappa \leq \iota$ as a subderivation of the original. Since $\sigma^\flat \equiv \bigcup_i \sigma_i^\flat$, there is some σ_i such that $\kappa \in \sigma_i^\flat$. CSub-AE then gives $\Gamma^\flat \vdash^{\flat} \sigma_i^\flat \leq (X\Rightarrow\alpha)^\flat$. This derivation is no larger than the original and σ_i is strictly smaller than σ, so by either the main or the sub-sub-induction hypothesis, $\Gamma \vdash^{\downarrow} \sigma_i \leq X\Rightarrow\alpha$. By rule ASubL-Inter, $\Gamma \vdash^{\downarrow} \sigma \leq X\Rightarrow\alpha$.

Case: $\tau \equiv \alpha \qquad \sigma \equiv \sigma_1 \to \sigma_2 \qquad \Gamma^\flat \stackrel{\flat}{\vdash} (\sigma_1 \to \sigma_2)^\flat \leq (X \Rightarrow \alpha)^\flat$

Since $(X \Rightarrow \alpha)^\flat \equiv \bigwedge[\iota]$ is a singleton, the syntax-directedness of canonical subtyping (4.2.6.3(1)) implies that for some $K_1 \to \kappa_2 \in \sigma^\flat$ we have $\Gamma^\flat \stackrel{\flat}{\vdash} K_1 \to \kappa_2 \leq \iota$ as a subderivation. By syntax-directedness again (4.2.6.3(2)), ι must have the form $I_1 \to \iota_2$, with $\Gamma^\flat \stackrel{\flat}{\vdash} I_1 \leq K_1$ and $\Gamma^\flat \stackrel{\flat}{\vdash} \kappa_2 \leq \iota_2$ as subderivations. By Definition 4.2.8.1, $X \equiv [\tau_1, X_2]$, where $I_1 \equiv \tau_1^\flat$ and $\bigwedge[\iota_2] \equiv (X_2 \Rightarrow \alpha)^\flat$. Since $\kappa_2 \in \sigma_2{}^\flat$, we have $\Gamma^\flat \stackrel{\flat}{\vdash} \sigma_2{}^\flat \leq (X_2 \Rightarrow \alpha)^\flat$ by CSub-AE. This derivation is no larger than the original, and σ is strictly smaller, so by either the main or the sub-sub-induction hypothesis, $\Gamma \stackrel{\flat}{\vdash} \sigma_2 \leq X_2 \Rightarrow \alpha$. Also, by the main induction hypothesis, $\Gamma \stackrel{\flat}{\vdash} \tau_1 \leq [] \Rightarrow \sigma_1$. By rule ASubL-Arrow, $\Gamma \stackrel{\flat}{\vdash} \sigma_1 \to \sigma_2 \leq \tau_1 \to (X_2 \Rightarrow \tau_2)$, i.e., $\Gamma \stackrel{\flat}{\vdash} \sigma_1 \to \sigma_2 \leq [\tau_1, X_2] \Rightarrow \tau_2$.

Case: $\tau \equiv \alpha \qquad \sigma \equiv \forall \alpha \leq \sigma_1.\ \sigma_2$

Similar.

Case: $\tau \equiv \alpha \qquad \sigma \equiv \beta \qquad \Gamma^\flat \stackrel{\flat}{\vdash} \beta^\flat \leq (X \Rightarrow \alpha)^\flat$

Since $\beta^\flat \equiv \bigwedge[\beta]$ and $(X \Rightarrow \alpha)^\flat \equiv \bigwedge[\iota]$ are both singletons, the syntax-directedness of canonical subtyping (4.2.6.3(1)) gives $\Gamma^\flat \stackrel{\flat}{\vdash} \beta \leq \iota$ as a subderivation. By syntax-directedness again (4.2.6.3(4)), either $\iota \equiv \beta$ or $\Gamma^\flat \stackrel{\flat}{\vdash} \Gamma^\flat(\beta) \leq \bigwedge[\iota]$. In the first case, rule ASubL-Refl gives the desired result immediately. In the second case, the main induction hypothesis gives $\Gamma \stackrel{\flat}{\vdash} \Gamma(\beta) \leq X \Rightarrow \alpha$, from which rule ASubL-TVar again yields $\Gamma \stackrel{\flat}{\vdash} \beta \leq X \Rightarrow \alpha$. □

4.2.8.10. Theorem: [Completeness of the algorithm with respect to ordinary subtyping] If $\Gamma \stackrel{\wedge}{\vdash} \sigma \leq X \Rightarrow \tau$ then $\Gamma \stackrel{\flat}{\vdash} \sigma \leq X \Rightarrow \tau$.

Proof: By the equivalence of ordinary and canonical subtyping (Theorem 4.2.7.6) and the completeness of the algorithm with respect to canonical subtyping (Lemma 4.2.8.9). □

4.2.8.11. Definition: The more convenient three-place relation $\Gamma \stackrel{\flat}{\vdash} \sigma \leq \tau$ may be defined as

$$\Gamma \stackrel{\flat}{\vdash} \sigma \leq \tau \quad \text{iff} \quad \Gamma \stackrel{\flat}{\vdash} \sigma \leq X \Rightarrow \phi,$$

where $\tau \equiv X \Rightarrow \phi$ and either $\phi \equiv \bigwedge[\phi_1 .. \phi_n]$ or $\phi \equiv \alpha$.

4.2.8.12. Theorem: [Equivalence of ordinary and syntax-directed subtyping]

$$\Gamma \stackrel{\wedge}{\vdash} \sigma \leq \tau \quad \text{iff} \quad \Gamma \stackrel{\flat}{\vdash} \sigma \leq \tau.$$

4.3 Typechecking

We now present an algorithm for synthesizing types for $F_\wedge$ terms. Given a term e and a context Γ (where e is closed in Γ), the algorithm constructs a *minimal type* σ for e under Γ — that is, a type σ such that $\Gamma \vdash e \in \sigma$, and such that any other type that can be derived for e from these rules is a supertype of σ.

The algorithm can be explained by separating the typing rules of Definition 3.2.3 into two sets: the *structural* or *syntax-directed* rules (Var, Arrow-E, All-I, All-E, and For), whose applicability depends on the form of e, and the non-structural rules (Inter-I and Sub), which can be applied without regard to the form of e. The non-structural rules are then removed from the system and their possible effects accounted for by modifying the structural rules Var, Arrow-E, All-E, and For appropriately.

The main novel source of difficulty here is the application rules Arrow-E and All-E. An application $(e_1\ e_2)$ in the original system has *every* type τ_2 such that e_1 can be shown to have some

type $\tau_1 \to \tau_2$ and e_2 can be shown to have type τ_1, where the rule SUB may be used on both sides to promote the types of e_1 and e_2 to supertypes with appropriate shapes. For example, if

$$e_1 \in (\sigma_1 \to \sigma_2) \wedge (\forall \alpha \leq \sigma_3.\ \sigma_4) \wedge (\sigma_5 \to \sigma_6) \wedge (\sigma_7 \to \sigma_8)$$
$$e_2 \in \sigma_1 \wedge (\forall \alpha \leq \sigma_3.\ \sigma_4) \wedge \sigma_5,$$

then

$$(e_1\ e_2)$$

has both types σ_2 and σ_6, and hence (by INTER-I) also type $\sigma_2 \wedge \sigma_6$.

To deal with this flexibility deterministically, we observe that the set of supertypes of $(\sigma_1 \to \sigma_2) \wedge (\forall \alpha \leq \sigma_3.\ \sigma_4) \wedge (\sigma_5 \to \sigma_6) \wedge (\sigma_7 \to \sigma_8)$ that have the appropriate shape to appear as the type of e_1 in an instance of ARROW-E can be characterized finitely:

$$\begin{aligned} &\mathit{arrowbasis}((\sigma_1 \to \sigma_2) \wedge (\forall \alpha \leq \sigma_3.\ \sigma_4) \wedge (\sigma_5 \to \sigma_6) \wedge (\sigma_7 \to \sigma_8)) \\ &\quad = [\sigma_1 \to \sigma_2,\ \sigma_5 \to \sigma_6,\ \sigma_7 \to \sigma_8]. \end{aligned}$$

It is then a simple matter to characterize the possible types for $(e_1\ e_2)$ by checking whether the minimal type of e_2 is a subtype of each domain type in the finite arrow basis of the minimal type of e_1. Type applications are handled similarly.

4.3.1 Finite Bases for Applications

4.3.1.1. Definition: The functions $\mathit{arrowbasis}_\Gamma$ and $\mathit{allbasis}_\Gamma$ are defined as follows:

$$\begin{aligned} \mathit{arrowbasis}_\Gamma(\alpha) &= \mathit{arrowbasis}_\Gamma(\Gamma(\alpha)) \\ \mathit{arrowbasis}_\Gamma(\tau_1 \to \tau_2) &= [\tau_1 \to \tau_2] \\ \mathit{arrowbasis}_\Gamma(\forall \alpha \leq \tau_1.\ \tau_2) &= [\,] \\ \mathit{arrowbasis}_\Gamma(\textstyle\bigwedge[\tau_1..\tau_n]) &= \mathit{arrowbasis}_\Gamma(\tau_1) * \cdots * \mathit{arrowbasis}_\Gamma(\tau_n) \\ \\ \mathit{allbasis}_\Gamma(\alpha) &= \mathit{allbasis}_\Gamma(\Gamma(\alpha)) \\ \mathit{allbasis}_\Gamma(\tau_1 \to \tau_2) &= [\,] \\ \mathit{allbasis}_\Gamma(\forall \alpha \leq \tau_1.\ \tau_2) &= [\forall \alpha \leq \tau_1.\ \tau_2] \\ \mathit{allbasis}_\Gamma(\textstyle\bigwedge[\tau_1..\tau_n]) &= \mathit{allbasis}_\Gamma(\tau_1) * \cdots * \mathit{allbasis}_\Gamma(\tau_n). \end{aligned}$$

4.3.1.2. Remark: To check that these definitions are proper, note that a closed context cannot contain cyclic chains of variable references where $\alpha_0 \in FTV(\Gamma(\alpha_1))$, $\alpha_1 \in FTV(\Gamma(\alpha_2))$, ..., $\alpha_n \in FTV(\Gamma(\alpha_0))$.

The next two lemmas verify that $\mathit{arrowbasis}_\Gamma$ and $\mathit{allbasis}_\Gamma$ compute finite bases for the sets of arrow types and polymorphic types above a given type.

4.3.1.3. Lemma: [Finite $\to$ basis computed by $\mathit{arrowbasis}_\Gamma$]

1. $\Gamma \vdash^\wedge \sigma \leq \bigwedge(\mathit{arrowbasis}_\Gamma(\sigma))$.
2. If $\Gamma \vdash^\wedge \sigma \leq \tau_1 \to \tau_2$, then $\Gamma \vdash^\wedge \bigwedge(\mathit{arrowbasis}_\Gamma(\sigma)) \leq \tau_1 \to \tau_2$.

Proof:

1. By induction on the definition of $\mathit{arrowbasis}_\Gamma$.
2. By the completeness of the subtyping algorithm (4.2.8.10), $\Gamma \vdash^\wedge \sigma \leq \tau_1 \to \tau_2$ implies $\Gamma \vdash \sigma \leq [\tau_1, [\,]] \Rightarrow \tau_2$. We show, by induction on derivations, that $\Gamma \vdash \sigma \leq [\tau_1, X_a] \Rightarrow \tau_b$ implies $\Gamma \vdash^\wedge \bigwedge(\mathit{arrowbasis}_\Gamma(\sigma)) \leq [\tau_1, X_a] \Rightarrow \tau_b$, from which the desired result follows as a special case, since $\tau_1 \to \tau_2$ can always be written in the form $[\tau_1, X_a] \Rightarrow \tau_b$, where the outermost constructor of τ_b is $\wedge$ or a variable.

   Proceed by cases on the final step of a derivation of $\Gamma \vdash \sigma \leq [\tau_1, X_a] \Rightarrow \tau_b$.

Case ASubR-Inter: $\tau_b \equiv \bigwedge[\tau_{b1}..\tau_{bn}]$

By assumption,

$$\Gamma \stackrel{\downarrow}{\vdash} \sigma \leq [\tau_1, X_a] \Rightarrow \tau_{bi}$$

for each i; by the induction hypothesis,

$$\Gamma \stackrel{\wedge}{\vdash} \bigwedge(\mathit{arrowbasis}_\Gamma(\sigma)) \leq [\tau_1, X_a] \Rightarrow \tau_{bi}.$$

By derived rule D-Cong-Inter (4.1.7),

$$\Gamma \stackrel{\wedge}{\vdash} \bigwedge[\bigwedge(\mathit{arrowbasis}_\Gamma(\sigma)) .. \bigwedge(\mathit{arrowbasis}_\Gamma(\sigma))] \leq \bigwedge[([\tau_1, X_a] \Rightarrow \tau_{b1}) .. ([\tau_1, X_a] \Rightarrow \tau_{bn})].$$

By D-Absorb and D-Reindex (4.1.2) and Sub-Trans,

$$\Gamma \stackrel{\wedge}{\vdash} \bigwedge(\mathit{arrowbasis}_\Gamma(\sigma)) \leq \bigwedge[([\tau_1, X_a] \Rightarrow \tau_{b1}) .. ([\tau_1, X_a] \Rightarrow \tau_{bn})].$$

By $\mathit{len}([\tau_1, X_a])$ applications of Sub-Dist-Ia and Sub-Dist-IQ (as appropriate) and Sub-Trans,

$$\Gamma \stackrel{\wedge}{\vdash} \bigwedge(\mathit{arrowbasis}_\Gamma(\sigma)) \leq [\tau_1, X_a] \Rightarrow \bigwedge[\tau_{b1}..\tau_{bn}].$$

Case ASubL-Inter: $\sigma \equiv \bigwedge[\sigma_1..\sigma_n] \qquad \tau_b \equiv \alpha$

By assumption, $\Gamma \stackrel{\downarrow}{\vdash} \sigma_i \leq [\tau_1, X_a] \Rightarrow \alpha$ for some i. By the induction hypothesis,

$$\Gamma \stackrel{\wedge}{\vdash} \bigwedge(\mathit{arrowbasis}_\Gamma(\sigma_i)) \leq [\tau_1, X_a] \Rightarrow \alpha.$$

Since $\mathit{arrowbasis}_\Gamma(\sigma_i) \subseteq \mathit{arrowbasis}_\Gamma(\sigma)$, Sub-Refl, D-All-Some (4.1.2), and Sub-Trans give

$$\Gamma \stackrel{\wedge}{\vdash} \bigwedge(\mathit{arrowbasis}_\Gamma(\sigma)) \leq [\tau_1, X_a] \Rightarrow \alpha.$$

Case ASubL-Arrow: $\sigma \equiv \sigma_1 \rightarrow \sigma_2 \qquad \tau_b \equiv \alpha$

By the definition of $\mathit{arrowbasis}_\Gamma$ and the equivalence of ordinary and syntax-directed subtyping (4.2.8.12).

Case ASubL-All: $\sigma \equiv \forall\alpha{\leq}\sigma_1.\ \sigma_2 \qquad \tau_b \equiv \alpha$

Can't happen ($[\tau_1, X_b]$ has the wrong form).

Case ASubL-Refl: $\sigma \equiv \beta \qquad \tau_b \equiv \beta \qquad [\tau_1, X_a] \equiv [\,]$

Can't happen.

Case ASubL-TVar: $\sigma \equiv \beta \qquad \tau_b \equiv \alpha \qquad \Gamma \stackrel{\downarrow}{\vdash} \Gamma(\beta) \leq [\tau_1, X_a] \Rightarrow \alpha$

By the induction hypothesis,

$$\Gamma \stackrel{\wedge}{\vdash} \bigwedge(\mathit{arrowbasis}(\Gamma(\beta))) \leq [\tau_1, X_a] \Rightarrow \alpha.$$

By the definition of $\mathit{arrowbasis}_\Gamma$,

$$\Gamma \stackrel{\wedge}{\vdash} \bigwedge(\mathit{arrowbasis}(\beta)) \leq [\tau_1, X_a] \Rightarrow \alpha.$$

□

4.3.1.4. Lemma: [Finite $\forall$ basis computed by $\mathit{allbasis}_\Gamma$]

1. $\Gamma \vdash \sigma \leq \bigwedge(\mathit{allbasis}_\Gamma(\sigma))$.
2. If $\Gamma \vdash \sigma \leq (\forall\alpha{\leq}\tau_1.\ \tau_2)$, then $\Gamma \vdash \bigwedge(\mathit{allbasis}_\Gamma(\sigma)) \leq (\forall\alpha{\leq}\tau_1.\ \tau_2)$.

Proof: Similar. □

The crucial step in the correctness proof for the type synthesis algorithm is showing that application and type application are correctly characterized by the sets computed by *arrowbasis* and *allbasis*.

4.3.1.5. Lemma: [Application]
If

$M \equiv [\phi_1 \to \psi_1 \,..\, \phi_n \to \psi_n]$
$V \equiv [\psi_i \mid \Gamma \vdash \sigma \leq \phi_i]$
$\Gamma \vdash \bigwedge M \leq \tau_1 \to \tau_2$
$\Gamma \vdash \sigma \leq \tau_1$,

then

$\Gamma \vdash \bigwedge V \leq \tau_2$.

Proof: By the equivalence of ordinary and canonical subtyping (4.2.7.6),

$\Gamma^\flat \vdash^\flat (\bigwedge M)^\flat \leq (\tau_1 \to \tau_2)^\flat$
$\Gamma^\flat \vdash^\flat \sigma^\flat \leq \tau_1^\flat$
$V \equiv [\psi_i \mid \Gamma^\flat \vdash^\flat \sigma^\flat \leq {\phi_i}^\flat]$.

By the definition of $\flat$ (4.2.7.1),

$\Gamma^\flat \vdash^\flat \bigcup_i (\bigwedge [{\phi_i}^\flat \to \kappa \mid \kappa \in {\psi_i}^\flat]) \leq \bigwedge [\tau_1^\flat \to \iota \mid \iota \in \tau_2^\flat]$.

By the syntax-directedness of canonical subtyping (4.2.6.3(1)),

for all $\iota \in \tau_2^\flat$
  there is some i and some $\kappa \in {\psi_i}^\flat$ such that
    $\Gamma^\flat \vdash^\flat {\phi_i}^\flat \to \kappa \leq \tau_1^\flat \to \iota$.

By syntax-directedness again (4.2.6.3(2)),

for all $\iota \in \tau_2^\flat$
  there is some i and some $\kappa \in {\psi_i}^\flat$ such that
    $\Gamma^\flat \vdash^\flat \tau_1^\flat \leq {\phi_i}^\flat$ and
    $\Gamma^\flat \vdash^\flat \kappa \leq \iota$,

that is,

for all $\iota \in \tau_2^\flat$
  there is some i such that
    $\Gamma^\flat \vdash^\flat \tau_1^\flat \leq {\phi_i}^\flat$ and
    there is some $\kappa \in {\psi_\iota}^\flat$ such that
      $\Gamma^\flat \vdash^\flat \kappa \leq \iota$.

By CSub-Trans,

for all $\iota \in \tau_2^\flat$
  there is some i such that
    $\Gamma^\flat \vdash^\flat \sigma^\flat \leq {\phi_i}^\flat$ and
    there is some $\kappa \in {\psi_\iota}^\flat$ such that
      $\Gamma^\flat \vdash^\flat \kappa \leq \iota$.

By CSub-AE,

$\Gamma^\flat \vdash^\flat \bigcup_i (\bigwedge [\kappa \mid \kappa \in {\psi_i}^\flat \text{ and } \Gamma^\flat \vdash^\flat \sigma^\flat \leq {\phi_i}^\flat]) \leq \tau_2^\flat$,

that is,

$\Gamma^\flat \vdash^\flat (\bigwedge V)^\flat \leq \tau_2^\flat$.

By the equivalence of ordinary and canonical subtyping (4.2.7.6),

$\Gamma \vdash^{\wedge} \bigwedge V \leq \tau_2$. □

4.3.1.6. Lemma: [Type application]
If

$M \equiv [(\forall\alpha{\leq}\phi_1.\ \psi_1)\,..\,(\forall\alpha{\leq}\phi_n.\ \psi_n)]$
$V \equiv [\{\sigma/\alpha\}\psi_i \mid \Gamma \vdash \sigma \leq \phi_i]$
$\Gamma \vdash \bigwedge M \leq (\forall\alpha{\leq}\tau_1.\ \tau_2)$
$\Gamma \vdash \sigma \leq \tau_1$,

then

$\Gamma \vdash \bigwedge V \leq \{\sigma/\alpha\}\tau_2$.

Proof: By the equivalence of ordinary and canonical subtyping (Theorem 4.2.7.6),

$\Gamma^\flat \vdash^\flat (\bigwedge M)^\flat \leq (\forall\alpha{\leq}\tau_1.\ \tau_2)^\flat$
$\Gamma^\flat \vdash^\flat \sigma^\flat \leq \tau_1^\flat$
$V \equiv [\{\sigma/\alpha\}\psi_i \mid \Gamma^\flat \vdash^\flat \sigma^\flat \leq {\phi_i}^\flat]$.

By the definition of $\flat$,

$\Gamma^\flat \vdash^\flat \bigcup_i(\bigwedge[\forall\alpha{\leq}{\phi_i}^\flat.\ \kappa \mid \kappa \in {\psi_i}^\flat]) \leq \bigwedge[\forall\alpha{\leq}\tau_1^\flat.\ \iota \mid \iota \in \tau_2^\flat]$.

By the syntax-directedness of canonical subtyping (4.2.6.3(1)),

for all $\iota \in \tau_2^\flat$
  there is some i and some $\kappa \in {\psi_i}^\flat$ such that
    $\Gamma^\flat \vdash^\flat (\forall\alpha{\leq}{\phi_i}^\flat.\ \kappa) \leq (\forall\alpha{\leq}\tau_1^\flat.\ \iota)$.

By syntax-directedness again (4.2.6.3(3)),

for all $\iota \in \tau_2^\flat$
  there is some i and some $\kappa \in {\psi_i}^\flat$ such that
    $\Gamma^\flat \vdash^\flat \tau_1^\flat \leq {\phi_i}^\flat$ and
    $\Gamma^\flat, \alpha{\leq}\tau_1^\flat \vdash \kappa \leq \iota$,

that is,

for all $\iota \in \tau_2^\flat$
  there is some i such that
    $\Gamma^\flat \vdash^\flat \tau_1^\flat \leq {\phi_i}^\flat$ and
    there is some $\kappa \in {\psi_i}^\flat$ such that
      $\Gamma^\flat, \alpha{\leq}\tau_1^\flat \vdash \kappa \leq \iota$.

By CSub-Trans,

for all $\iota \in \tau_2^\flat$
  there is some i such that
    $\Gamma^\flat \vdash^\flat \sigma^\flat \leq {\phi_i}^\flat$ and
    there is some $\kappa \in {\psi_i}^\flat$ such that
      $\Gamma^\flat, \alpha{\leq}\tau_1^\flat \vdash \kappa \leq \iota$.

By CSUB-AE,

$$\Gamma^\flat, \alpha{\leq}\tau_1^\flat \vdash^\flat \bigcup_i(\bigwedge[\kappa \mid \kappa \in \psi_i \text{ and } \Gamma^\flat \vdash^\flat \sigma^\flat \leq {\phi_i}^\flat]) \leq \tau_2^\flat,$$

that is,

$$\Gamma^\flat, \alpha{\leq}\tau_1^\flat \vdash^\flat (\bigwedge[\psi_i \mid \Gamma^\flat \vdash^\flat \sigma^\flat \leq {\phi_i}^\flat])^\flat \leq \tau_2^\flat.$$

By the equivalence of ordinary and canonical subtyping (4.2.7.6),

$$\Gamma, \alpha{\leq}\tau_1 \vdash^\wedge \bigwedge[\psi_i \mid \Gamma \vdash \sigma \leq \phi_i] \leq \tau_2.$$

Then by the substitution property (4.1.8),

$$\Gamma \vdash^\wedge \{\sigma/\alpha\}(\bigwedge[\psi_i \mid \Gamma \vdash \sigma \leq \phi_i]) \leq \{\sigma/\alpha\}\tau_2,$$

that is,

$$\Gamma \vdash^\wedge \bigwedge V \leq \{\sigma/\alpha\}\tau_2.$$ □

4.3.2 Type Synthesis

4.3.2.1. Definition: The three-place *type synthesis relation* $\Gamma \vdash^{\cdot} e \in \tau$ is the least relation closed under the following rules:

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(A-VAR)}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ e \in \tau_1{\rightarrow}\tau_2} \qquad \text{(A-ARROW-I)}$$

$$\frac{\Gamma \vdash e_1 \in \sigma_1 \qquad \Gamma \vdash e_2 \in \sigma_2}{\Gamma \vdash e_1\ e_2 \in \bigwedge[\psi_i \mid (\phi_i{\rightarrow}\psi_i) \in \mathit{arrowbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \sigma_2 \leq \phi_i]} \qquad \text{(A-ARROW-E)}$$

$$\frac{\Gamma, \alpha{\leq}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \Lambda\alpha{\leq}\tau_1.\ e \in \forall\alpha{\leq}\tau_1.\ \tau_2} \qquad \text{(A-ALL-I)}$$

$$\frac{\Gamma \vdash e \in \sigma_1}{\Gamma \vdash e[\tau] \in \bigwedge[\{\tau/\alpha\}\psi_i \mid (\forall\alpha{\leq}\phi_i.\ \psi_i) \in \mathit{allbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \tau \leq \phi_i]} \qquad \text{(A-ALL-E)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash \{\sigma_i/\alpha\}e \in \tau_i}{\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e \in \bigwedge[\tau_1..\tau_n]} \qquad \text{(A-FOR)}$$

4.3.2.2. Notation: Again, turnstiles in type synthesis derivations are sometimes marked $\vdash^{\cdot}$ to distinguish them from derivations in other calculi.

4.3.2.3. Lemma: [Syntax-directedness of the type synthesis rules] For given Γ and e, there is at most one rule that can be used to establish $\Gamma \vdash^{\cdot} e \in \tau$ for some τ. Moreover, the existence of such a derivation can be established from the form of e and the results of applying the type synthesis procedure to proper subphrases of e plus a finite number of applications of the subroutine for checking the subtyping relation. In particular:

1. If $\Gamma \vdash^{\cdot} x \in \theta$, then $\theta \equiv \Gamma(x)$.
2. If $\Gamma \vdash^{\cdot} \lambda x{:}\tau_1.\ e \in \theta$, then $\theta \equiv \tau_1{\rightarrow}\tau_2$, where $\Gamma, x{:}\tau_1 \vdash^{\cdot} e \in \tau_2$ as a subderivation.
3. If $\Gamma \vdash^{\cdot} e_1 e_2 \in \theta$, then $\theta \equiv \bigwedge[\psi_i \mid (\phi_i{\rightarrow}\psi_i) \in \mathit{arrowbasis}_\Gamma(\sigma_1)$ and $\Gamma \vdash \sigma_2 \leq \phi_i]$, where $\Gamma \vdash^{\cdot} e_1 \in \sigma_1$ and $\Gamma \vdash^{\cdot} e_2 \in \sigma_2$ as subderivations.
4. If $\Gamma \vdash^{\cdot} \Lambda\alpha{\leq}\tau_1.\ e \in \theta$, then $\theta \equiv \forall\alpha{\leq}\tau_1.\ \tau_2$, where $\Gamma, \alpha{\leq}\tau_1 \vdash^{\cdot} e \in \tau_2$ as a subderivation.

5. If $\Gamma \stackrel{\cdot}{\vdash} e[\tau] \in \theta$, then $\theta \equiv \bigwedge[\{\tau/\alpha\}\psi_i \mid (\forall\alpha{\leq}\phi_i.\ \psi_i) \in \mathit{allbasis}_\Gamma(\sigma_1)$ and $\Gamma \vdash \tau \leq \phi_i]$, where $\Gamma \stackrel{\cdot}{\vdash} e \in \sigma_1$ as a subderivation.
6. If $\Gamma \stackrel{\cdot}{\vdash} \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e \in \theta$, then $\theta \equiv \bigwedge[\tau_1..\tau_n]$, where $\Gamma \stackrel{\cdot}{\vdash} \{\sigma_i/\alpha\}e \in \tau_i$, for each i, as a subderivation.

Proof: By inspection. □

4.3.2.4. Remark: The syntax-directedness of algorithmic derivations permits us to skip introducing a linear shorthand, as we did for ordinary and canonical derivations, since terms themselves are essentially the shorthand we need.

4.3.2.5. Definition: Let $c :: \Gamma \stackrel{\cdot}{\vdash} e \in \tau$ be a typing derivation from the algorithmic rules (4.3.2.1). Then $c^\wedge :: \Gamma \stackrel{\wedge}{\vdash} e \in \tau$ is the following derivation from the ordinary typing rules:

$$\left(\frac{}{\Gamma \stackrel{\cdot}{\vdash} x \in \Gamma(x)}\ \text{(A-VAR)}\right)^\wedge \quad = \quad \frac{}{\Gamma \stackrel{\wedge}{\vdash} x \in \Gamma(x)}\ \text{(VAR)}$$

$$\left(\frac{c_1 :: \Gamma, x{:}\tau_1 \stackrel{\cdot}{\vdash} e \in \tau_2}{\Gamma \stackrel{\cdot}{\vdash} (\lambda x{:}\tau_1.\ e) \in \tau_1{\rightarrow}\tau_2}\ \text{(A-ARROW-I)}\right)^\wedge \quad = \quad \frac{c_1^\wedge}{\Gamma \stackrel{\wedge}{\vdash} (\lambda x{:}\tau_1.\ e) \in \tau_1{\rightarrow}\tau_2}\ \text{(ARROW-I)}$$

$$\left(\frac{c_1 :: \Gamma \stackrel{\cdot}{\vdash} e_1 \in \sigma_1 \qquad c_2 :: \Gamma \stackrel{\cdot}{\vdash} e_2 \in \sigma_2}{\Gamma \stackrel{\cdot}{\vdash} (e_1\ e_2) \in \bigwedge[\psi_i \mid (\phi_i{\rightarrow}\psi_i) \in \mathit{arrowbasis}_\Gamma(\sigma_1) \text{ and } d_i :: \Gamma \vdash \sigma_2 \leq \phi_i]}\ \text{(A-ARROW-E)}\right)^\wedge \quad =$$

$$\cdots \quad \frac{\dfrac{c_1^\wedge \qquad 4.3.1.3 :: \Gamma \stackrel{\wedge}{\vdash} \sigma_1 \leq \phi_i{\rightarrow}\psi_i}{\Gamma \stackrel{\wedge}{\vdash} e_1 \in \phi_i{\rightarrow}\psi_i}\ \text{(SUB)} \qquad \dfrac{c_2^\wedge \qquad d_i^\wedge}{\Gamma \stackrel{\wedge}{\vdash} e_2 \in \phi_i}\ \text{(SUB)}}{\Gamma \stackrel{\wedge}{\vdash} (e_1\ e_2) \in \psi_i}\ \text{(ARROW-E)} \quad \cdots$$

$$\frac{}{\Gamma \stackrel{\wedge}{\vdash} (e_1\ e_2) \in \bigwedge[\psi_i \mid (\phi_i{\rightarrow}\psi_i) \in \mathit{arrowbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \sigma_2 \leq \phi_i]}\ \text{(INTER-I)}$$

$$\left(\frac{c_1 :: \Gamma, \alpha{\leq}\tau_1 \stackrel{\cdot}{\vdash} e \in \tau_2}{\Gamma \stackrel{\cdot}{\vdash} (\Lambda\alpha{\leq}\tau_1.\ e) \in \forall\alpha{\leq}\tau_1.\ \tau_2}\ \text{(A-ALL-I)}\right)^\wedge \quad = \quad \frac{c_1^\wedge}{\Gamma \stackrel{\wedge}{\vdash} (\Lambda\alpha{\leq}\tau_1.\ e) \in \forall\alpha{\leq}\tau_1.\ \tau_2}\ \text{(ALL-I)}$$

$$\left(\frac{c_1 :: \Gamma \stackrel{\cdot}{\vdash} e \in \sigma_1}{\Gamma \stackrel{\cdot}{\vdash} e\ [\tau] \in \bigwedge[\{\tau/\alpha\}\psi_i \mid (\forall\alpha{\leq}\phi_i.\ \psi_i) \in \mathit{allbasis}_\Gamma(\sigma_1) \text{ and } d_i :: \Gamma \vdash \tau \leq \phi_i]}\ \text{(A-ALL-E)}\right)^\wedge \quad =$$

$$\cdots \quad \frac{\dfrac{c_1^\wedge \qquad 4.3.1.4 :: \Gamma \stackrel{\wedge}{\vdash} \sigma_1 \leq \forall\alpha{\leq}\phi_i.\ \psi_i}{\Gamma \stackrel{\wedge}{\vdash} e \in \forall\alpha{\leq}\phi_i.\ \psi_i}\ \text{(SUB)} \qquad d_i^\wedge}{\Gamma \stackrel{\wedge}{\vdash} e\ [\tau] \in \{\tau/\alpha\}\psi_i}\ \text{(ALL-E)} \quad \cdots$$

$$\frac{}{\Gamma \stackrel{\wedge}{\vdash} e\ [\tau] \in \bigwedge[\{\tau/\alpha\}\psi_i \mid (\forall\alpha{\leq}\phi_i.\ \psi_i) \in \mathit{allbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \tau \leq \phi_i]}\ \text{(INTER-I)}$$

$$\left(\frac{\text{for all } i,\ c_i :: \Gamma \stackrel{\cdot}{\vdash} \{\sigma_i/\alpha\}e \in \tau_i}{\Gamma \stackrel{\cdot}{\vdash} (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e) \in \bigwedge[\tau_1..\tau_n]}\ \text{(A-FOR)}\right)^\wedge \quad =$$

$$\cdots \quad \frac{c_i^\wedge :: \Gamma \stackrel{\wedge}{\vdash} \{\sigma_i/\alpha\}e \in \tau_i}{\Gamma \stackrel{\wedge}{\vdash} (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e) \in \tau_i}\ \text{(FOR)} \quad \cdots$$

$$\frac{}{\Gamma \stackrel{\wedge}{\vdash} (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e) \in \bigwedge[\tau_1..\tau_n]}\ \text{(INTER-I)}$$

4.3.2.6. Theorem: If $s :: \Gamma \stackrel{\cdot}{\vdash} e \in \tau$, then $s^\wedge :: \Gamma \stackrel{\wedge}{\vdash} e \in \tau$.

Proof: By induction on the structure of s. □

4.3.2.7. Theorem: [Minimal typing] If $\Gamma \stackrel{\cdot}{\vdash} e \in \sigma$ and $\Gamma \stackrel{\wedge}{\vdash} e \in \tau$, then $\Gamma \vdash \sigma \leq \tau$.

Proof: By induction on a derivation of $\Gamma \stackrel{\wedge}{\vdash} e \in \tau$. Proceed by cases on the final rule.

Case VAR: $e \equiv x \qquad \tau \equiv \Gamma(x)$

Immediate by A-VAR.

Case ARROW-I: $e \equiv \lambda x{:}\tau_1.\ e' \qquad \Gamma, x{:}\tau_1 \stackrel{\wedge}{\vdash} e' \in \tau_2 \qquad \tau \equiv \tau_1 \rightarrow \tau_2$

By the syntax-directedness of the type synthesis rules (4.3.2.3), the last rule in the derivation of $\Gamma \stackrel{\vdash}{} e \in \sigma$ must be A-ARROW-I, so

$\Gamma, x{:}\tau_1 \vdash e' \in \sigma_2$

$\sigma \equiv \tau_1 \rightarrow \sigma_2$.

By the induction hypothesis, $\Gamma \stackrel{\wedge}{\vdash} \sigma_2 \leq \tau_2$. By SUB-REFL and SUB-ARROW,

$\Gamma \stackrel{\wedge}{\vdash} \tau_1 \rightarrow \sigma_2 \leq \tau_1 \rightarrow \tau_2$.

Case ARROW-E: $e \equiv e_1\ e_2 \qquad \Gamma \stackrel{\wedge}{\vdash} e_1 \in \tau_1 \rightarrow \tau_2 \qquad \Gamma \stackrel{\wedge}{\vdash} e_2 \in \tau_1 \qquad \tau \equiv \tau_2$

By the syntax-directedness of the type synthesis rules (4.3.2.3),

$\Gamma \stackrel{\vdash}{} e_1 \in \sigma_1$

$\Gamma \stackrel{\vdash}{} e_2 \in \sigma_2$

$\sigma \equiv \bigwedge[\psi_i \mid (\phi_i \rightarrow \psi_i) \in \mathit{arrowbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \stackrel{\vdash}{} \sigma_2 \leq \phi_i]$.

By the equivalence of ordinary and syntax-directed subtyping (4.2.8.12),

$\sigma \equiv \bigwedge[\psi_i \mid (\phi_i \rightarrow \psi_i) \in \mathit{arrowbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \stackrel{\wedge}{\vdash} \sigma_2 \leq \phi_i]$.

By the induction hypothesis,

$\Gamma \stackrel{\wedge}{\vdash} \sigma_1 \leq \tau_1 \rightarrow \tau_2$

$\Gamma \stackrel{\wedge}{\vdash} \sigma_2 \leq \tau_1$.

Since $\mathit{arrowbasis}_\Gamma(\sigma_1)$ is a finite basis for the arrow types above σ_1 (4.3.1.3),

$\Gamma \stackrel{\wedge}{\vdash} \bigwedge(\mathit{arrowbasis}_\Gamma(\sigma_1)) \leq \tau_1 \rightarrow \tau_2$.

By the application lemma (4.3.1.5),

$\Gamma \stackrel{\wedge}{\vdash} \bigwedge[\psi_i \mid (\phi_i \rightarrow \psi_i) \in \mathit{arrowbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \stackrel{\wedge}{\vdash} \sigma_2 \leq \phi_i] \leq \tau_2$.

Case ALL-I: $e \equiv \Lambda\alpha{\leq}\tau_1.\ e' \qquad \Gamma, \alpha{\leq}\tau_1 \stackrel{\wedge}{\vdash} e' \in \tau_2 \qquad \tau \equiv \forall\alpha{\leq}\tau_1.\ \tau_2$

By the syntax-directedness of the type synthesis rules (4.3.2.3),

$\Gamma, \alpha{\leq}\tau_1 \stackrel{\vdash}{} e' \in \sigma_2$

$\sigma \equiv \forall\alpha{\leq}\tau_1.\ \sigma_2$.

By the induction hypothesis,

$\Gamma, \alpha{\leq}\tau_1 \stackrel{\wedge}{\vdash} \sigma_2 \leq \tau_2$.

By SUB-REFL and SUB-ALL,

$\Gamma \stackrel{\wedge}{\vdash} (\forall\alpha{\leq}\tau_1.\ \sigma_2) \leq (\forall\alpha{\leq}\tau_1.\ \tau_2)$.

Case ALL-E: $e \equiv e'[\tau'] \qquad \Gamma \stackrel{\wedge}{\vdash} e' \in \forall\alpha{\leq}\tau_1.\ \tau_2 \qquad \Gamma \stackrel{\wedge}{\vdash} \tau' \leq \tau_1 \qquad \tau \equiv \{\tau'/\alpha\}\tau_2$

By the syntax-directedness of the type synthesis rules (4.3.2.3),

$\Gamma \stackrel{\vdash}{} e' \in \sigma_1$

$\sigma \equiv \bigwedge[\{\tau'/\alpha\}\psi_i \mid (\forall\alpha{\leq}\phi_i.\ \psi_i) \in \mathit{allbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \stackrel{\vdash}{} \tau' \leq \phi_i]$.

By the equivalence of ordinary and syntax-directed subtyping (4.2.8.12),

$\sigma \equiv \bigwedge[\{\tau'/\alpha\}\psi_i \mid (\forall\alpha{\leq}\phi_i.\ \psi_i) \in \mathit{allbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \stackrel{\wedge}{\vdash} \tau' \leq \phi_i]$.

By the induction hypothesis,

$\Gamma \stackrel{\wedge}{\vdash} \sigma_1 \leq \forall\alpha{\leq}\tau_1.\ \tau_2$.

Since $\mathit{allbasis}_\Gamma(\sigma_1)$ is a finite basis for the polymorphic types above σ_1 (4.3.1.4),

$\Gamma \stackrel{\wedge}{\vdash} \bigwedge(\mathit{allbasis}_\Gamma(\sigma_1)) \leq \forall\alpha{\leq}\tau_1.\ \tau_2$.

By the type application lemma (4.3.1.6),

$\Gamma \stackrel{\wedge}{\vdash} \bigwedge[\{\tau'/\alpha\}\psi_i \mid (\forall\alpha{\leq}\phi_i.\ \psi_i) \in \mathit{allbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \tau' \leq \phi_i] \leq \{\tau'/\alpha\}\tau_2$.

Case FOR: $e \equiv \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e' \qquad \Gamma \vdash^{\wedge} \{\sigma_i/\alpha\}e \in \tau_i \qquad \tau \equiv \tau_i$

By the syntax-directedness of the type synthesis rules (4.3.2.3),

for all j, $\Gamma \vdash^{\downarrow} \{\sigma_j/\alpha\}e \in \phi_j$

$\sigma \equiv \bigwedge[\phi_1..\phi_n]$.

By the induction hypothesis,

$\Gamma \vdash^{\wedge} \phi_i \leq \tau_i$.

By SUB-INTER-LB and SUB-TRANS,

$\Gamma \vdash^{\wedge} \bigwedge[\phi_1..\phi_n] \leq \tau_i$.

Case INTER-I: for all i, $\Gamma \vdash^{\wedge} e \in \tau_i \qquad \tau \equiv \bigwedge[\tau_1..\tau_n]$

By the induction hypothesis,

for all i, $\Gamma \vdash^{\wedge} \sigma \leq \tau_i$.

By SUB-INTER-G,

$\Gamma \vdash^{\wedge} \sigma \leq \bigwedge[\tau_1..\tau_n]$.

Case SUB: $\Gamma \vdash^{\wedge} e \in \tau_1 \qquad \Gamma \vdash^{\wedge} \tau_1 \leq \tau_2 \qquad \tau \equiv \tau_2$

By the induction hypothesis,

$\Gamma \vdash^{\wedge} \sigma \leq \tau_1$.

By SUB-TRANS,

$\Gamma \vdash^{\wedge} \sigma \leq \tau_2$. □

4.3.3 Conservativity

$F_\wedge$ was described as essentially the union of the two simpler calculi $\lambda_\wedge$ and $F_\leq$. We can gauge the accuracy of this characterization by checking whether the features of the component calculi operate "orthogonally," so that each component system can be thought of as a restriction of $F_\wedge$— i.e., by asking whether $F_\wedge$ is a conservative extension of $\lambda_\wedge$ and of $F_\leq$.

4.3.3.1. Definition: Let C and E be two calculi and $E(\text{—}) \in C \to E$ an injective mapping from C statements to E statements. $E(\text{—})$ is said to be an *embedding* of C into E if, for every C statement J, J is derivable in C iff $E(J)$ is derivable in E.

Typically, $E(\text{—})$ is just an identity injection. For instance, this is the case for the embedding of $\lambda_\wedge$ into $F_\wedge$.

4.3.3.2. Definition: If the identity injection is an embedding of C into E, then E is said to be a *conservative extension* of C.

4.3.3.3. Theorem: Let σ and τ be $\lambda_\wedge$ types, Γ an $\lambda_\wedge$ context, and e an $\lambda_\wedge$ expression. Assume that the primitive subtype relation of $\lambda_\wedge$ is encoded as a context Γ_P (c.f. 3.4.2.2). Then:

1. $\Gamma_P, \Gamma \vdash^{\wedge} \sigma \leq \tau$ iff $\Gamma \vdash^{\lambda\wedge} \sigma \leq \tau$.
2. $\Gamma_P, \Gamma \vdash^{\wedge} e \in \tau$ iff $\Gamma \vdash^{\lambda\wedge} e \in \tau$.

Proof:

1. ($\Longleftarrow$) Lemma 3.4.2.4.

   ($\Longrightarrow$) If $\Gamma_P, \Gamma \vdash^{\wedge} \sigma \leq \tau$, then by the completeness of the subtyping algorithm, $\Gamma_P, \Gamma \vdash^{\downarrow} \sigma \leq \tau$. By the syntax-directedness of the subtyping algorithm and the fact that σ, τ, and Γ contain no quantified types, this derivation will not contain any instances of ASUBL-ALL, the rule that deals with quantified types. It may be therefore be rewritten as a derivation from the $\lambda_\wedge$ rules by a translation similar to the one in the proof of Theorem 4.2.8.8, dropping

the bindings Γ_P and translating instances of ASubL-Refl as instances of Sub-Refl and instances of ASubL-TVar as derivations of the following form:

$$\dfrac{\dfrac{\forall\gamma \in P \text{ with } \beta{\leq_P}\gamma. \quad \dfrac{\beta \leq_P \gamma}{\Gamma \stackrel{\lambda\wedge}{\vdash} \beta \leq \gamma}\,\text{(SUB-PRIM)}}{\Gamma \stackrel{\lambda\wedge}{\vdash} \beta \leq \bigwedge[\gamma \in P \mid \beta{\leq_P}\gamma]}\,\text{(SUB-INTER-G)} \qquad \dfrac{\text{(induction hypothesis)}}{\Gamma \stackrel{\lambda\wedge}{\vdash} \bigwedge[\gamma \in P \mid \beta{\leq_P}\gamma] \leq X{\Rightarrow}\alpha}}{\Gamma \stackrel{\lambda\wedge}{\vdash} \beta \leq X{\Rightarrow}\alpha.}\,\text{(SUB-TRANS)}$$

2. ($\Longleftarrow$) Lemma 3.4.2.5.
   ($\Longrightarrow$) If Γ, $\Gamma_P \stackrel{\wedge}{\vdash} e \in \tau$, then by the completeness of the subtyping algorithm, Γ, $\Gamma_P \stackrel{\perp}{\vdash} e \in \tau$. By the syntax-directedness of the subtyping algorithm and the fact that e, τ, and Γ contain no type abstractions, type applications, or quantified types, this derivation will not contain any instances of A-All-I or A-All-E. It may be therefore be rewritten straightforwardly as a derivation from the $\lambda_\wedge$ rules, using the previous case to handle the translation of subtyping derivations. □

The mapping from the other subsystem, $F_\leq$, into $F_\wedge$ must take the type *Top* into $\top$, and it is here that it fails to be an embedding (as we might expect from the discussion in Section 3.4):

4.3.3.4. Example: The subtyping statement

$$\vdash \mathit{Top} \leq \forall\alpha{\leq}\mathit{Top}.\ \mathit{Top}$$

is derivable in $F_\wedge$(reading *Top* as $\top$), but not in $F_\leq$.

4.3.3.5. Conjecture: Since *Top* must be mapped to a maximal type by any embedding function from $F_\leq$ to $F_\wedge$ and $\top$ is the only such type (up to equivalence), there is probably *no* embedding of $F_\leq$ into $F_\wedge$.

4.3.3.6. Conjecture: By replacing *Top* with $\top$ in $F_\leq$ and adding appropriate distributivity laws to the subtyping relation and a $\top$-introduction rule to the typing relation, we can construct a system that can be embedded into $F_\wedge$ (indeed, such that $F_\wedge$ extends it conservatively), but this system is only a technical curiosity: it has most of the problematic features of $F_\wedge$ (the distributivity laws in particular) but is much less expressive.

Chapter 5

Semantics

This chapter surveys a collection of preliminary results concerning the semantics of $F_\wedge$.

Section 5.1 gives a simple untyped semantics for $F_\wedge$ based on Bruce and Longo's partial equivalence relation model for $F_\leq$ [12].

Section 5.2 discusses a negative technical result — the nonexistence of syntactic least upper bounds — with some serious implications for the difficulty of constructing a typed model for $F_\wedge$ in which the subtype relation is interpreted by semantic coercion functions.

The remainder of the chapter presents two different partial accounts of the typed semantics of $F_\wedge$. Section 5.3 defines a semantics for $F_\wedge$ by translating $F_\wedge$ typing derivations into the pure second-order λ-calculus with surjective pairing, system $F_\times$. This style of presentation avoids some of the subtleties involved in giving a direct denotational semantics for $F_\wedge$, since $F_\times$ itself has many well-studied models, but it still yields a useful soundness theorem relating the semantics to the $F_\wedge$ type system: valid $F_\wedge$ typing derivations are translated to well-typed (and hence well-behaved) $F_\times$ terms. We then (Section 5.5) define an equational theory of provable equivalences between terms of pure $F_\wedge$. The equational theory is shown to be sound for both the untyped semantics and the translation semantics (the latter in the sense that provably equal $F_\wedge$ terms are translated into equal terms in the target calculus, assuming that the translation is coherent).

5.1 Untyped Semantics

One of the simplest styles of semantics for typed λ-calculi is based on partial equivalence relations (PERs). A model in this style is essentially untyped (c.f. Section 2.4.1): terms are interpreted by erasing all type information and interpreting the resulting pure λ-term as an element of the model. A type, in this setting, is just a subset of the model along with an appropriate notion of equivalence of elements. Coercions between types are interpreted by inclusion of PERs.

The PER model given here for $F_\wedge$ is based on Bruce and Longo's model for $F_\leq$ [12]. However, the full generality of Bruce and Longo's construction, involving the category of ω-sets, is not required here.

The usual interpretation of a quantified type $\forall\alpha.\ \tau$ in a second-order PER model is the PER-indexed intersection of all possible instances of τ. Bruce and Longo showed how to extend this definition to interpret a bounded quantifier $\forall\alpha{\leq}\sigma.\ \tau$ as the intersection of all the instances of τ where α is interpreted as a sub-PER of the interpretation of σ. This intuition also serves for intersection types: $\bigwedge[\tau_1..\tau_n]$ is interpreted as the intersection of the PERs interpreting each of the τ_i's.

We need to make one significant departure here from PER models of $F_{\leq}$: instead of allowing the elements of our PERs to be drawn from the carrier of an arbitrary partial combinatory algebra $\mathcal{D}$, we require that $\mathcal{D}$ be a *total* combinatory algebra. This restriction is needed to validate nullary instances of the distributive law SUB-DIST-IA, which have the form $\Gamma \vdash \top \leq \sigma \to \top$. To see why, let $\sigma \equiv \top$. The empty intersection $\top$ is interpreted by the everywhere-defined PER, i.e., $[\![\top]\!]$ relates every m to itself. To validate the distributivity law, it must therefore be the case that $[\![\top \to \top]\!]$ relates every element to itself. But this will only be true if the application of any element to any other element is defined. This observation is due to QingMing Ma [personal communication, 1991].

The notation and fundamental definitions used in this section are based on papers of Bruce and Longo [12], Freyd, Mulry, Rosolini, and Scott [61], and others. A good basic reference for PER models of second-order λ-calculi is [95]; also see [13] for more general discussion of second-order models and [5, 77] for general discussion of combinatory models.

5.1.1 Total Combinatory Algebras

5.1.1.1. Definition: A *total combinatory algebra* is a tuple $\mathcal{D} = \langle D, \cdot, k, s \rangle$ comprising

- a set D of *elements*,
- an *application function* $\cdot$ with type $D \to (D \to D)$,
- distinguished elements $k, s \in D$,

such that, for all $d_1, d_2, d_3 \in D$,

$$k \cdot d_1 \cdot d_2 = d_1$$
$$s \cdot d_1 \cdot d_2 \cdot d_3 = (d_1 \cdot d_3) \cdot (d_2 \cdot d_3).$$

5.1.1.2. Remark: Throughout this section, we work with a fixed, but unspecified, total combinatory algebra $\mathcal{D}$. (For example, Scott's D_∞ or P_ω [128] model [128].)

5.1.1.3. Definition: The set of *pure λ-terms* is defined by the following abstract grammar:

$$M \quad ::= \quad x \quad | \quad \lambda x.\, M \quad | \quad M_1\, M_2$$

5.1.1.4. Definition: The set of *combinator terms* is defined by the following abstract grammar:

$$C \quad ::= \quad x \quad | \quad C_1\, C_2 \quad | \quad K \quad | \quad S$$

5.1.1.5. Definition: The *bracket abstraction* of a combinator term C with respect to a variable x, written $\lambda^\star x.\, C$, is defined as follows:

$$\begin{array}{lcll} \lambda^\star x.\, C & = & K\, C & \text{when } x \notin FV(C) \\ \lambda^\star x.\, x & = & S\, K\, K & \\ \lambda^\star x.\, C_1\, C_2 & = & S\, (\lambda^\star x.\, C_1)\, (\lambda^\star x.\, C_2) & \text{when } x \in FV(C_1\, C_2) \end{array}$$

5.1.1.6. Definition: The *combinator translation* of a pure λ-term M, written $|M|$, is defined as follows:

$$\begin{array}{lcl} |x| & = & x \\ |\lambda x.\, M| & = & \lambda^\star x.\, |M| \\ |M_1\, M_2| & = & |M_1|\, |M_2| \end{array}$$

5.1.1.7. Definition: An *environment* η is a finite function from type variables to PERs (defined below) and term variables to elements of D. When $x \notin dom(\eta)$, we write $\eta[x \leftarrow d]$ for the environment that maps x to d and agrees with η everywhere else; $\eta[\alpha \leftarrow A]$ is defined similarly. We write $\eta \backslash x$

for the environment like η except that $\eta(x)$ is undefined; $\eta\backslash\alpha$ similarly. We say that η' *extends* η when $dom(\eta) \subseteq dom(\eta')$ and η and η' agree on $dom(\eta)$.

5.1.1.8. Definition: Let C be a combinatory term and η an environment such that $FV(C) \subseteq dom(\eta)$. Then the *interpretation* of C under η, written $[\![C]\!]_\eta$, is defined as follows:

$$\begin{array}{lcl} [\![x]\!]_\eta & = & \eta(x) \\ [\![C_1\,C_2]\!]_\eta & = & [\![C_1]\!]_\eta \cdot [\![C_2]\!]_\eta \\ [\![K]\!]_\eta & = & k \\ [\![S]\!]_\eta & = & s \end{array}$$

5.1.1.9. Lemma: If η' extends η and $FV(C) \subseteq dom(\eta)$, then $[\![C]\!]_\eta = [\![C]\!]_{\eta'}$.

Proof: Straightforward induction on C. □

5.1.1.10. Lemma: $[\![\lambda^\star x.\ C]\!]_\eta \cdot m = [\![C]\!]_{\eta[x \leftarrow m]}$.

Proof: By induction on the form of C.

Case: $x \notin FV(C)$

$$\begin{array}{lcll} [\![\lambda^\star x.\ C]\!]_\eta \cdot m & = & [\![K\,C]\!]_\eta \cdot m & \\ & = & k \cdot [\![C]\!]_\eta \cdot m & \\ & = & [\![C]\!]_\eta & \\ & = & [\![C]\!]_{\eta[x \leftarrow m]} & \text{by Lemma 5.1.1.9.} \end{array}$$

Case: $C \equiv x$

$$\begin{array}{lcl} [\![\lambda^\star x.\ C]\!]_\eta \cdot m & = & [\![S\,K\,K]\!]_\eta \cdot m \\ & = & s \cdot k \cdot k \cdot m \\ & = & m \\ & = & (\eta[x \leftarrow m])(x) \\ & = & [\![C]\!]_{\eta[x \leftarrow m]}. \end{array}$$

Case: $C \equiv C_1\,C_2 \qquad x \in FV(C_1\,C_2)$

$$\begin{array}{lcll} [\![\lambda^\star x.\ C]\!]_\eta \cdot m & = & [\![S\,(\lambda^\star x.\ C_1)\,(\lambda^\star x.\ C_2)]\!]_\eta \cdot m & \\ & = & s \cdot [\![\lambda^\star x.\ C_1]\!]_\eta \cdot [\![\lambda^\star x.\ C_2]\!]_\eta \cdot m & \\ & = & ([\![\lambda^\star x.\ C_1]\!]_\eta \cdot m) \cdot ([\![\lambda^\star x.\ C_2]\!]_\eta \cdot m) & \\ & = & ([\![C_1]\!]_{\eta[x \leftarrow m]}) \cdot ([\![C_2]\!]_{\eta[x \leftarrow m]}) & \text{by the induction hypothesis} \\ & = & ([\![C_1\,C_2]\!]_{\eta[x \leftarrow m]}). & \end{array}$$

□

5.1.2 Partial Equivalence Relations

5.1.2.1. Definition: A *partial equivalence relation* (PER) on $\mathcal{D}$ is a symmetric and transitive relation A on D. We write $m\ \{A\}\ n$ when A relates m and n. The *domain* of A, written $dom(A)$, is the set $\{n \mid n\ \{A\}\ n\}$. Note that $m\ \{A\}\ n$ implies $m \in dom(A)$.

5.1.2.2. Definition: Let A and B be relations. Then $A \rightarrow B$ is the relation defined by

$$m\ \{A \rightarrow B\}\ n \quad \text{iff} \quad \text{for all } p, q \in D,\ \ p\ \{A\}\ q \text{ implies } m \cdot p\ \{B\}\ n \cdot q.$$

5.1.2.3. Lemma: $A \rightarrow B$ is a PER when A and B are PERs.

Proof: (Symmetry) Let $m\ \{A \rightarrow B\}\ n$. Then

$$\text{for all } p \text{ and } q, \quad p\ \{A\}\ q \text{ implies } m \cdot p\ \{B\}\ n \cdot q,$$

which by the symmetry of A and B implies that

for all p and q, $\quad q\ \{A\}\ p$ implies $n{\cdot}q\ \{B\}\ m{\cdot}p$,

that is, $n\ \{A{\rightarrow}B\}\ m$.

(Transitivity) Let $m\ \{A{\rightarrow}B\}\ n$ and $n\ \{A{\rightarrow}B\}\ o$. Then

$$\begin{array}{lll} & \text{for all } p, q, \text{ and } r, & (p\ \{A\}\ q \text{ implies } m{\cdot}p\ \{B\}\ n{\cdot}q) \text{ and } (q\ \{A\}\ r \text{ implies } n{\cdot}q\ \{B\}\ o{\cdot}r) \\ \Rightarrow & \text{for all } p, q, \text{ and } r, & (p\ \{A\}\ q \text{ and } q\ \{A\}\ r) \text{ implies } (m{\cdot}p\ \{B\}\ n{\cdot}q \text{ and } n{\cdot}q\ \{B\}\ o{\cdot}r) \\ \Rightarrow & \text{for all } p, q, \text{ and } r, & (p\ \{A\}\ q \text{ and } q\ \{A\}\ r) \text{ implies } m{\cdot}p\ \{B\}\ o{\cdot}r \\ \Rightarrow & \text{for all } p \text{ and } r, & (p\ \{A\}\ p \text{ and } p\ \{A\}\ r) \text{ implies } m{\cdot}p\ \{B\}\ o{\cdot}r \\ \Rightarrow & \text{for all } p \text{ and } r, & p\ \{A\}\ r \text{ implies } m{\cdot}p\ \{B\}\ o{\cdot}r, \end{array}$$

that is, $m\ \{A{\rightarrow}B\}\ o$. □

5.1.2.4. Definition: A is a *subrelation* of B, written $A \subseteq B$, iff $m\ \{A\}\ n$ implies $m\ \{B\}\ n$ for all $m, n \in D$.

5.1.2.5. Definition: Let $\{A_i\}_{i\in I}$ be a set of relations indexed by a set I. Then $\bigcap_{i\in I} A_i$ is the relation defined by

$m\ \{\bigcap_{i\in I} A_i\}\ n$ iff for every i, $m\ \{A_i\}\ n$.

5.1.2.6. Lemma: $\bigcap_{i\in I} A_i$ is a PER when all the A_i's are PERs.

Proof: Straightforward. □

5.1.3 PER Interpretation of $F_\wedge$

5.1.3.1. Definition: The *erasure* of an $F_\wedge$ term e, written $erase(e)$, is the pure λ-term defined as follows:

$$\begin{array}{lcl} erase(x) & = & x \\ erase(\lambda x{:}\tau.\ e) & = & \lambda x.\ erase(e) \\ erase(e_1\ e_2) & = & erase(e_1)\ erase(e_2) \\ erase(\forall\alpha{\leq}\tau.\ e) & = & erase(e) \\ erase(e\ [\tau]) & = & erase(e) \\ erase(for\ \alpha\ in\ \sigma_1..\sigma_n.\ e) & = & erase(e) \end{array}$$

5.1.3.2. Definition: Let η be an environment and e an expression such that $FV(e) \subseteq dom(\eta)$. Then the *interpretation* of e under η, written $[\![e]\!]_\eta$, is $[\![|erase(e)|]\!]_\eta$.

5.1.3.3. Remark: Since this style of semantics interprets the erasures of terms rather than interpreting typing derivations, it is coherent in a trivial sense.

5.1.3.4. Lemma: $[\![\lambda x{:}\tau.\ e']\!]_\eta = [\![\lambda^\star x.\ |erase(e)|]\!]_\eta$.

Proof: Straightforward. □

5.1.3.5. Definition: Let η be an environment and τ a type expression such that $FTV(\tau) \subseteq dom(\eta)$. The *interpretation* of τ under η, written $[\![\tau]\!]_\eta$, is the PER defined as follows:

$$\begin{array}{lcll} [\![\alpha]\!]_\eta & = & \eta(\alpha) & \\ [\![\tau_1{\rightarrow}\tau_2]\!]_\eta & = & [\![\tau_1]\!]_\eta\ \rightarrow\ [\![\tau_2]\!]_\eta & \\ [\![\forall\alpha{\leq}\tau_1.\ \tau_2]\!]_\eta & = & \bigcap_{A\ \subseteq\ [\![\tau_1]\!]_\eta} [\![\tau_2]\!]_{\eta[\alpha\leftarrow A]} & \text{where } A \text{ is a PER} \\ [\![\bigwedge[\tau_1..\tau_n]]\!]_\eta & = & \bigcap_{1\leq i\leq n} [\![\tau_i]\!]_\eta & \end{array}$$

5.1.3.6. Definition: An environment η *satisfies* a context Γ, written $\eta \models \Gamma$, if $dom(\eta) = dom(\Gamma)$ and

1. $\Gamma \equiv \{\}$, or
2. $\Gamma \equiv \Gamma_1, x : \tau$, where $\eta\backslash x$ satisfies Γ_1 and $\eta(x) \in dom([\![\tau]\!]_{\eta\backslash x})$, or
3. $\Gamma \equiv \Gamma_1, \alpha{\leq}\tau$, where $\eta\backslash\alpha$ satisfies Γ_1 and $\eta(\alpha) \subseteq [\![\tau]\!]_{\eta\backslash\alpha}$.

5.1.3.7. Lemma: If η' extends η and $FV(\tau) \subseteq dom(\eta)$, then $[\![\tau]\!]_\eta = [\![\tau]\!]_{\eta'}$.

Proof: Straightforward. □

5.1.3.8. Lemma: (Soundness of subtyping) If $\Gamma \vdash \sigma \leq \tau$ and $\eta \models \Gamma$, then $[\![\sigma]\!]_\eta \subseteq [\![\tau]\!]_\eta$.

Proof: By induction on the structure of a derivation of $\Gamma \vdash \sigma \leq \tau$.

Case Sub-Refl: $\sigma \equiv \tau$

Immediate.

Case Sub-Trans: $\Gamma \vdash \sigma \leq \theta \qquad \Gamma \vdash \theta \leq \tau$

By the induction hypothesis.

Case Sub-TVar: $\sigma \equiv \alpha \qquad \tau \equiv \Gamma(\alpha)$

Immediate from 5.1.3.6.

Case Sub-Arrow: $\sigma \equiv \sigma_1{\to}\sigma_2 \qquad \tau \equiv \tau_1 \to \tau_2 \qquad \Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq \tau_2$

$$
\begin{array}{ll}
 & m\,\{[\![\sigma_1{\to}\sigma_2]\!]_\eta\}\,n \\
\Leftrightarrow & m\,\{[\![\sigma_1]\!]_\eta{\to}[\![\sigma_2]\!]_\eta\}\,n \\
\Leftrightarrow & \forall p,q.\quad p\,\{[\![\sigma_1]\!]_\eta\}\,q \text{ implies } m{\cdot}p\,\{[\![\sigma_2]\!]_\eta\}\,n{\cdot}q \\
\Rightarrow & \forall p,q.\quad p\,\{[\![\tau_1]\!]_\eta\}\,q \text{ implies } m{\cdot}p\,\{[\![\tau_2]\!]_\eta\}\,n{\cdot}q \quad \text{by the induction hypothesis} \\
\Leftrightarrow & m\,\{[\![\tau_1{\to}\tau_2]\!]_\eta\}\,n.
\end{array}
$$

Case Sub-All: $\sigma \equiv \forall\alpha{\leq}\sigma_1.\ \sigma_2 \qquad \tau \equiv \forall\alpha{\leq}\tau_1.\ \tau_2 \qquad \Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma, \alpha{\leq}\tau_1 \vdash \sigma_2 \leq \tau_2$

$$
\begin{array}{ll}
 & m\,\{[\![\forall\alpha{\leq}\sigma_1.\ \sigma_2]\!]_\eta\}\,n \\
\Leftrightarrow & m\,\{\bigcap_{A\subseteq[\![\sigma_1]\!]_\eta}[\![\sigma_2]\!]_{\eta[\alpha\leftarrow A]}\}\,n \\
\Leftrightarrow & \forall A{\subseteq}[\![\sigma_1]\!]_\eta.\quad m\,\{[\![\sigma_2]\!]_{\eta[\alpha\leftarrow A]}\}\,n \\
\Rightarrow & \forall A{\subseteq}[\![\tau_1]\!]_\eta.\quad m\,\{[\![\tau_2]\!]_{\eta[\alpha\leftarrow A]}\}\,n \quad \text{by the induction hypothesis} \\
\Leftrightarrow & m\,\{[\![\forall\alpha{\leq}\tau_1.\ \tau_2]\!]_\eta\}\,n.
\end{array}
$$

Case Sub-Inter-G: $\tau \equiv \bigwedge[\tau_1..\tau_n] \qquad$ for all i, $\Gamma \vdash \sigma \leq \tau_i$

By the induction hypothesis, $[\![\sigma]\!]_\eta \subseteq [\![\tau_i]\!]_\eta$ for each i; so $[\![\sigma]\!]_\eta \subseteq \bigcap_{1\leq i\leq n}[\![\tau_i]\!]_\eta = [\![\bigwedge[\tau_1..\tau_n]]\!]_\eta$.

Case Sub-Inter-LB: $\sigma \equiv \bigwedge[\tau_1..\tau_n] \qquad \tau \equiv \tau_i$

Immediate from the definition of $\bigwedge$.

Case Sub-Dist-IA: $\sigma \equiv \bigwedge[\sigma'{\to}\tau_1\ ..\ \sigma'{\to}\tau_n] \qquad \tau \equiv \sigma'{\to}\bigwedge[\tau_1..\tau_n]$

$$
\begin{array}{ll}
 & m\,\{[\![\bigwedge[\sigma'{\to}\tau_1\ ..\ \sigma'{\to}\tau_n]]\!]_\eta\}\,m' \\
\Leftrightarrow & m\,\{\bigcap_{1\leq i\leq n}[\![\sigma'{\to}\tau_i]\!]_\eta\}\,m' \\
\Leftrightarrow & \forall i.\quad m\,\{[\![\sigma']\!]_\eta{\to}[\![\tau_i]\!]_\eta\}\,m' \\
\Leftrightarrow & \forall i.\quad \forall p,q.\quad p\,\{[\![\sigma']\!]_\eta\}\,q \text{ implies } m{\cdot}p\,\{[\![\tau_i]\!]_\eta\}\,m'{\cdot}q \\
\Leftrightarrow & \forall p,q.\quad p\,\{[\![\sigma']\!]_\eta\}\,q \text{ implies } (\forall i.\ m{\cdot}p\,\{[\![\tau_i]\!]_\eta\}\,m'{\cdot}q) \\
\Leftrightarrow & \forall p,q.\quad p\,\{[\![\sigma']\!]_\eta\}\,q \text{ implies } m{\cdot}p\,\{[\![\bigwedge[\tau_1..\tau_n]]\!]_\eta\}\,m'{\cdot}q \\
\Leftrightarrow & m\,\{[\![\sigma'{\to}\bigwedge[\tau_1..\tau_n]]\!]_\eta\}\,m'.
\end{array}
$$

Case SUB-DIST-IQ: $\sigma \equiv \bigwedge[\forall\alpha{\leq}\sigma'.\ \tau_1\ ..\ \forall\alpha{\leq}\sigma'.\ \tau_n] \qquad \tau \equiv \forall\alpha{\leq}\sigma'.\ \bigwedge[\tau_1..\tau_n]$

$$\begin{aligned} & [\![\bigwedge[\forall\alpha{\leq}\sigma'.\ \tau_1\ ..\ \forall\alpha{\leq}\sigma'.\ \tau_n]]\!]_\eta \\ = \ & \bigcap_{1\leq i\leq n}\bigcap_{A\subseteq[\![\sigma']\!]_\eta}\tau_i \\ = \ & \bigcap_{A\subseteq[\![\sigma']\!]_\eta}\bigcap_{1\leq i\leq n}\tau_i \\ = \ & [\![\forall\alpha{\leq}\sigma.\ \bigwedge[\tau_1..\tau_n]]\!]_\eta. \end{aligned}$$

□

5.1.3.9. Lemma:

1. $[\![e_1\,e_2]\!]_\eta = [\![e_1]\!]_\eta \cdot [\![e_2]\!]_\eta$.
2. $erase(\{\sigma/\alpha\}e) = erase(e)$.
3. $[\![\tau]\!]_{\eta[\alpha\leftarrow[\![\sigma]\!]_\eta]} = [\![\{\sigma/\alpha\}\tau]\!]_\eta$.
4. $[\![C]\!]_{\eta[x\leftarrow[\![C']\!]_\eta]} = [\![\{C'/x\}C]\!]_\eta$.
5. $\{|erase(v)|/x\}|erase(f)| = |erase(\{v/x\}f)|$.

Proof: Straightforward. □

5.1.3.10. Lemma: If

$\eta_1 \models \Gamma$
$\eta_2 \models \Gamma$
$\forall\alpha \in dom(\Gamma).\ \eta_1(\alpha) = \eta_2(\alpha) = \eta(\alpha)$
$\forall x \in dom(\Gamma).\ \eta_1(x)\ \{[\![\Gamma(x)]\!]_\eta\}\ \eta_2(x)$
$\Gamma \vdash e \in \tau,$

then

$[\![e]\!]_{\eta_1}\ \{[\![\tau]\!]_\eta\}\ [\![e]\!]_{\eta_2}$.

(Here η is just a convenient name for the portions of η_1 and η_2 dealing with type variables, which must be identical.)

Proof: By induction on a derivation of $\Gamma \vdash e \in \tau$.

Case VAR: $e \equiv x \qquad \tau \equiv \Gamma(x)$

Immediate.

Case ARROW-I: $e \equiv \lambda x{:}\tau_1.\ e' \qquad \Gamma, x{:}\tau_1 \vdash e' \in \tau_2 \qquad \tau \equiv \tau_1{\rightarrow}\tau_2$

Choose m and n such that $m\ \{[\![\tau_1]\!]_\eta\}\ n$. Then $m\ \{[\![\tau_1]\!]_\eta\}\ m$ and $n\ \{[\![\tau_1]\!]_\eta\}\ n$, so $\eta_1[x\leftarrow m] \models \Gamma, x{:}\tau_1$ and $\eta_2[x\leftarrow n] \models \Gamma, x{:}\tau_1$. The induction hypothesis gives

$$[\![e']\!]_{\eta_1[x\leftarrow m]}\ \{[\![\tau_2]\!]_\eta\}\ [\![e']\!]_{\eta_2[x\leftarrow n]}.$$

But

$$\begin{aligned} [\![e]\!]_{\eta_1} \cdot m &= [\![\lambda^\star x.\ |erase(e')|]\!]_{\eta_1} \cdot m && \text{by definition} \\ &= [\![|erase(e')|]\!]_{\eta_1[x\leftarrow m]} && \text{by Lemma 5.1.1.10} \\ &= [\![e']\!]_{\eta_1[x\leftarrow m]} && \text{by definition,} \end{aligned}$$

and similarly $[\![e]\!]_{\eta_2} \cdot n = [\![e']\!]_{\eta_2[x\leftarrow n]}$. So

$$[\![e]\!]_{\eta_1} \cdot m\ \{[\![\tau_2]\!]_\eta\}\ [\![e]\!]_{\eta_2} \cdot n.$$

Since this holds for all m and n such that $m\ \{[\![\tau_1]\!]_\eta\}\ n$, the definition of $\rightarrow$ gives

$$[\![e]\!]_{\eta_1}\ \{[\![\tau_1]\!]_\eta{\rightarrow}[\![\tau_2]\!]_\eta\}\ [\![e]\!]_{\eta_2},$$

i.e.,

$$[\![e]\!]_{\eta_1}\ \{[\![\tau_1{\rightarrow}\tau_2]\!]_\eta\}\ [\![e]\!]_{\eta_2}.$$

Case ARROW-E: $e \equiv e_1\, e_2 \qquad \Gamma \vdash e_1 \in \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \in \tau_1 \qquad \tau \equiv \tau_2$

By the induction hypothesis,

$$[\![e_1]\!]_{\eta_1}\ \{[\![\tau_1 \to \tau_2]\!]_\eta\}\ [\![e_1]\!]_{\eta_2},$$

i.e.

$$[\![e_1]\!]_{\eta_1}\ \{[\![\tau_1]\!]_\eta \to [\![\tau_2]\!]_\eta\}\ [\![e_1]\!]_{\eta_2},$$

and

$$[\![e_2]\!]_{\eta_1}\ \{[\![\tau_1]\!]_\eta\}\ [\![e_2]\!]_{\eta_2}.$$

So, by the definition of $\to$,

$$[\![e_1]\!]_{\eta_1} \cdot [\![e_2]\!]_{\eta_1}\ \{[\![\tau_2]\!]_\eta\}\ [\![e_1]\!]_{\eta_2} \cdot [\![e_2]\!]_{\eta_2},$$

i.e. (by Lemma 5.1.3.9(1)),

$$[\![e_1\, e_2]\!]_{\eta_1}\ \{[\![\tau_2]\!]_\eta\}\ [\![e_1\, e_2]\!]_{\eta_2}.$$

Case ALL-I: $e \equiv \Lambda\alpha{\le}\tau.\ e' \qquad \Gamma, \alpha{\le}\tau_1 \vdash e' \in \tau_2 \qquad \tau \equiv \forall\alpha{\le}\tau_1.\ \tau_2$

Choose an arbitrary PER $A \subseteq [\![\tau_1]\!]_\eta$. By the induction hypothesis,

$$[\![e']\!]_{\eta_1[\alpha \leftarrow A]}\ \{[\![\tau_2]\!]_{\eta[\alpha \leftarrow A]}\}\ [\![e']\!]_{\eta_2[\alpha \leftarrow A]}.$$

By Lemma 5.1.1.9,

$$[\![e']\!]_{\eta_1}\ \{[\![\tau_2]\!]_{\eta[\alpha \leftarrow A]}\}\ [\![e']\!]_{\eta_2}.$$

Since this holds for every $A \subseteq [\![\tau_1]\!]_\eta$, the definition of $\bigcap$ yields,

$$[\![e']\!]_{\eta_1}\ \{\textstyle\bigcap_{A \subseteq [\![\tau_1]\!]_\eta} [\![\tau_2]\!]_{\eta[\alpha \leftarrow A]}\}\ [\![e']\!]_{\eta_2},$$

i.e.,

$$[\![e']\!]_{\eta_1}\ \{[\![\forall\alpha{\le}\tau_1.\ \tau_2]\!]_\eta\}\ [\![e']\!]_{\eta_2},$$

i.e. (by the definition of *erase*),

$$[\![e]\!]_{\eta_1}\ \{[\![\forall\alpha{\le}\tau_1.\ \tau_2]\!]_\eta\}\ [\![e]\!]_{\eta_2}.$$

Case ALL-E: $e \equiv e'\,[\sigma] \qquad \Gamma \vdash e' \in \forall\alpha{\le}\tau_1.\ \tau_2 \qquad \Gamma \vdash \sigma \le \tau_1 \qquad \tau \equiv \tau_2$

By the induction hypothesis,

$$[\![e']\!]_{\eta_1}\ \{[\![\forall\alpha{\le}\tau_1.\ \tau_2]\!]_\eta\}\ [\![e']\!]_{\eta_2},$$

i.e.,

$$[\![e']\!]_{\eta_1}\ \{\textstyle\bigcap_{A \subseteq [\![\tau_1]\!]_\eta} [\![\tau_2]\!]_{\eta[\alpha \leftarrow A]}\}\ [\![e']\!]_{\eta_2}.$$

Since, by Lemma 5.1.3.8, $[\![\sigma]\!]_\eta \subseteq [\![\tau_1]\!]_\eta$,

$$[\![e']\!]_{\eta_1}\ \{[\![\tau_2]\!]_{\eta[\alpha \leftarrow [\![\sigma]\!]_\eta]}\}\ [\![e']\!]_{\eta_2},$$

i.e. (by the definition of *erase*),

$$[\![e]\!]_{\eta_1}\ \{[\![\tau_2]\!]_{\eta[\alpha \leftarrow [\![\sigma]\!]_\eta]}\}\ [\![e]\!]_{\eta_2},$$

i.e. (by Lemma 5.1.3.9(3)

$$[\![e]\!]_{\eta_1}\ \{[\![\{\sigma/\alpha\}\tau_2]\!]_\eta\}\ [\![e]\!]_{\eta_2}.$$

Case FOR: $e \equiv \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ .e' \qquad \Gamma \vdash \{\sigma_i/\alpha\}e' \in \tau_i \qquad \tau \equiv \tau_i$

By the induction hypothesis,

$$[\![\{\sigma_i/\alpha\}e']\!]_{\eta_1}\ \{[\![\tau_i]\!]_\eta\}\ [\![\{\sigma_i/\alpha\}e']\!]_{\eta_2}.$$

By Lemma 5.1.3.9(2),

$$[\![e']\!]_{\eta_1}\ \{[\![\tau_i]\!]_\eta\}\ [\![e']\!]_{\eta_2}.$$

By the definition of *erase*,

$$[\![e]\!]_{\eta_1}\ \{[\![\tau_i]\!]_\eta\}\ [\![e]\!]_{\eta_2}.$$

Case INTER-I: $\Gamma \vdash e \in \tau_i$ for each i $\quad \tau \equiv \bigwedge[\tau_1..\tau_n]$

By the induction hypothesis,

$$[\![e]\!]_{\eta_1} \{[\![\tau_i]\!]_\eta\} [\![e]\!]_{\eta_2}$$

for each i; hence,

$$[\![e]\!]_{\eta_1} \{\bigcap_{1 \le i \le n} [\![\tau_i]\!]_\eta\} [\![e]\!]_{\eta_2},$$

i.e.,

$$[\![e]\!]_{\eta_1} \{[\![\bigwedge[\tau_1..\tau_n]]\!]_\eta\} [\![e]\!]_{\eta_2}.$$

Case SUB: $\Gamma \vdash e \in \sigma \quad \Gamma \vdash \sigma \le \tau$

By the induction hypothesis,

$$[\![e]\!]_{\eta_1} \{[\![\sigma]\!]_\eta\} [\![e]\!]_{\eta_2},$$

hence (by Lemma 5.1.3.8)

$$[\![e]\!]_{\eta_1} \{[\![\tau]\!]_\eta\} [\![e]\!]_{\eta_2}.$$ □

5.1.3.11. Corollary: (Soundness of typing) If $\Gamma \vdash e \in \tau$ and $\eta \models \Gamma$, then $[\![e]\!]_\eta \in dom([\![\tau]\!]_\eta)$.

Proof: Take $\eta_1 = \eta_2 = \eta$. □

5.2 Nonexistence of Least Upper Bounds

One important question about the order-theoretic properties of any calculus with subtyping is the existence or nonexistence of *least upper bounds* (lubs) for finite sets of types. When they are present, lubs often greatly simplify the presentations of both semantic and proof-theoretic arguments; for example, Reynolds' model construction for Forsythe depends on the existence and special properties of lubs. Unfortunately, like its component system $F_{\le}$ (though not for the same reason), $F_\wedge$ does *not* have a lub for every finite set of types.

To simplify the discussion, we consider only lubs of pairs of types. The fact that a calculus of intersection types may be formulated in terms of an n-ary meet constructor, as we have done here, or, equivalently, in terms of $\top$ and binary meets, implies that we may make this simplification without loss of generality.

5.2.1. Definition: Let σ and τ be types, both closed under a context Γ. Then a *least upper bound* of σ and τ under Γ is a supertype of both σ and τ and a subtype of every common supertype of σ and τ — that is, a type θ such that:

$$\Gamma \vdash \sigma \le \theta$$
$$\Gamma \vdash \tau \le \theta$$
$$\Gamma \vdash \sigma \le \phi \quad \text{and} \quad \Gamma \vdash \tau \le \phi \quad \text{imply} \quad \Gamma \vdash \theta \le \phi.$$

(Note that least upper bounds are unique only up to equivalence.)

In systems with intersection types, it is simplest to define least upper bounds for canonical types (c.f. Section 4.2.1) and then transfer the definition to ordinary types. Here is Reynolds' definition of lubs for the canonical formulation of first-order intersection types:

5.2.2. Definition: Assume that we are given a partial function $\sqcup_P$ that yielding a least upper bound for every pair of primitive types with any upper bound. That is:

if $(\rho_1 \sqcup_P \rho_2)\downarrow$	then	$\rho_1 \le_P (\rho_1 \sqcup_P \rho_2)$
		$\rho_2 \le_P (\rho_1 \sqcup_P \rho_2)$
		$\rho_1 \le_P \rho'$ and $\rho_2 \le_P \rho'$ imply $(\rho_1 \sqcup_P \rho_2) \le_P \rho'$
if $(\rho_1 \sqcup_P \rho_2)\uparrow$	then	there is no ρ' such that $\rho_1 \le_P \rho'$ and $\rho_2 \le_P \rho'$.

5.2.3. Definition: Let k and i be canonical $\lambda_\wedge$ types. Then the distinguished least upper bound of k and i, written $k \sqcup i$, is defined by the following function (partial on individual canonical types and total on composite canonical types):

$$\begin{array}{lcl} K \sqcup I & = & \bigwedge[\kappa \sqcup \iota \mid \kappa \in K \text{ and } \iota \in I \text{ and } (\kappa \sqcup \iota) \downarrow] \\ \rho_1 \sqcup \rho_2 & = & \rho_1 \sqcup_P \rho_2 \\ (K \to \kappa) \sqcup (I \to \iota) & = & (K \cup I) \to (\iota \sqcup \kappa) \\ (K \to \kappa) \sqcup \rho & = & \uparrow \\ \rho \sqcup (I \to \iota) & = & \uparrow . \end{array}$$

(Recall from 4.2.1.3 that $K \cup I$ is shorthand for the intersection of all the elements of K and I.)

5.2.4. Fact: [Reynolds]

1. If $\kappa \sqcup \iota$ is defined, then it is a least upper bound of κ and ι. If $\kappa \sqcup \iota$ is undefined, then κ and ι have no common upper bounds.
2. $K \sqcup I$ is a least upper bound of K and I.

The existence of lubs for canonical types is easily shown to be equivalent to the existence of lubs for ordinary types, using the first-order analog of Theorem 4.2.7.6.

In his Ph.D. thesis [63], Ghelli observed that $F_\leq$ possesses neither least upper bounds nor greatest lower bounds.

5.2.5. Definition: A pair of types σ and τ is *downward compatible* if there is some type that is a subtype of both σ and τ.

5.2.6. Fact: [63, p. 92] There exists a pair of downward-compatible $F_\leq$ types σ and τ with no greatest lower bound.

Proof: Consider the context

$$\Gamma = \alpha \leq \mathit{Top},\ \beta \leq \mathit{Top},\ \alpha' \leq \alpha,\ \beta' \leq \beta$$

and the types

$$\begin{array}{lcl} \sigma & = & \forall\gamma \leq \alpha \to \beta.\ \alpha \to \beta \\ \tau & = & \forall\gamma \leq \alpha' \to \beta'.\ \alpha' \to \beta'. \end{array}$$

Then both

$$\forall\gamma \leq \alpha' \to \beta.\ \alpha \to \beta'$$

and

$$\forall\gamma \leq \alpha' \to \beta.\ \gamma$$

are lower bounds for σ and τ, but these two types have no common supertype that is also a subtype of σ and τ. □

5.2.7. Fact: [Ghelli] There is a pair of $F_\leq$ types with no least upper bound.

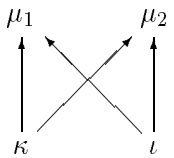
Proof: Consider $\sigma \to \mathit{Top}$ and $\tau \to \mathit{Top}$. □

Since $F_\wedge$, by definition, possesses greatest lower bounds for every pair of types, we might hope that lubs would also be recovered in $F_\wedge$. Unfortunately, this is not the case.

For example, consider the individual canonical types

$$\begin{array}{lcl} \kappa & \equiv & \forall\alpha \leq \bigwedge[].\ \forall\beta \leq \bigwedge[].\ \alpha \\ \iota & \equiv & \forall\alpha \leq \bigwedge[].\ \forall\beta \leq \bigwedge[].\ \beta \\ \mu_1 & \equiv & \forall\alpha \leq \bigwedge[].\ \forall\beta \leq \bigwedge[\alpha].\ \alpha \\ \mu_2 & \equiv & \forall\alpha \leq \bigwedge[\nu].\ \forall\beta \leq \bigwedge[\nu].\ \nu, \end{array}$$

where ν is any closed individual canonical type with the property that $\alpha{\leq}\nu \not\vdash \nu \leq \alpha$. (For example, take $\nu \equiv \forall\gamma{\leq}\bigwedge[].\ \gamma.$) Then it is easy to check that the following subtype relations hold in the empty context:



Note, however, that $\not\vdash \mu_1 \leq \mu_2$.

Now, assume that κ and ι have some least upper bound; call it λ. Then by the syntax-directedness of canonical subtyping (4.2.6.3) and the fact that λ is a supertype of κ, λ must have the form

$$\lambda \equiv \forall\alpha{\leq}L_1.\ \forall\beta{\leq}L_2.\ \lambda_3.$$

By syntax-directedness again and the fact that $\vdash \lambda \leq \mu_1$ (since μ_1 is a common upper bound of κ and ι),

$\vdash \bigwedge[] \leq L_1$
i.e. $L_1 \equiv \bigwedge[]$

$\alpha{\leq}\bigwedge[] \vdash \bigwedge[\alpha] \leq L_2$
i.e. $\lambda_2 \in L_2$ implies $\alpha{\leq}\bigwedge[] \vdash \alpha \leq \lambda_2$
i.e. $\lambda_2 \in L_2$ implies $\lambda_2 \equiv \alpha$
i.e. $L_2 \equiv \bigwedge[]$ or $L_2 \equiv \bigwedge[\alpha]$ (up to equivalence) ,

and if $L_2 \equiv \bigwedge[\alpha]$ then

$\alpha{\leq}\bigwedge[], \beta{\leq}\bigwedge[\alpha] \vdash \lambda_3 \leq \alpha$
i.e. $\lambda_3 \equiv \alpha$ or $\lambda_3 \equiv \beta$,

while if $L_2 \equiv \bigwedge[]$, then $\lambda_3 \equiv \alpha$.

Using the assumption that $\vdash \kappa \leq \lambda$, we may eliminate the case $\lambda_3 \equiv \beta$. Then, using $\vdash \iota \leq \lambda$, we may eliminate the case $L_2 \equiv \bigwedge[]$. In short, if κ and ι have any lub then it is equivalent to μ_1, which must therefore also be a lub. But μ_1 is not a subtype of μ_2, which is a the common upper bound of κ and ι; so μ_1 is *not* a lub of κ and ι. This contradicts our assumption.

To show that composite canonical types lack lubs, we actually need to show something stronger about individual canonical types: that they do not even possess complete finite sets of upper bounds.

5.2.8. Definition: Let σ and τ be types, both closed under Γ. Then a complete finite set of upper bounds for σ and τ under Γ is a finite set $T \equiv \{\theta_1..\theta_n\}$ such that:

1. $\Gamma \vdash \sigma \leq \theta_i$ and $\Gamma \vdash \tau \leq \theta_i$ for each θ_i;
2. if ϕ is a type such that $\Gamma \vdash \sigma \leq \phi$ and $\Gamma \vdash \tau \leq \phi$, then there is some θ_i such that $\Gamma \vdash \theta_i \leq \phi$.

5.2.9. Definition: Define the following infinite series of types:

$$\begin{array}{lcl} \nu_0 & \equiv & \forall\alpha{\leq}\bigwedge[].\ \alpha \\ \nu_{n+1} & \equiv & \forall\alpha{\leq}\bigwedge[].\ \nu_n. \end{array}$$

5.2.10. Lemma: If $\Delta \vdash \nu_i \sim \iota$ for some Δ and ι, then $\iota \equiv \nu_i$.

Proof: By induction on i.

Case: $i = 0$

Since $\Delta \vdash \nu_0 \leq \iota$, syntax-directedness (4.2.6.3) gives $\iota \equiv \forall\alpha{\leq}I_1.\ \iota_2$ and $\Delta, \alpha{\leq}I_1 \vdash \alpha \leq \iota_2$. From $\Delta \vdash \iota \leq \nu_0$, syntax-directedness gives $\Delta \vdash \bigwedge[] \leq I_1$, hence $I_1 \equiv \bigwedge[]$. Performing this substitution, we have $\Delta, \alpha{\leq}\bigwedge[] \vdash \alpha \leq \iota_2$, hence (by syntax-directedness again) $\iota_2 \equiv \alpha$.

Case: $i = n + 1$

By syntax-directedness, $\Delta \vdash \nu_{n+1} \leq \iota$ gives

$\iota \equiv \forall\alpha{\leq}I_1.\ \iota_2$
$\Delta, \alpha{\leq}I_1 \vdash \nu_n \leq \iota_2.$

Using syntax-directedness on $\Delta \vdash \iota \leq \nu_{n+1}$, we also have

$\Delta \vdash \bigwedge[] \leq I$
i.e. $I \equiv \bigwedge[]$
$\Delta, \alpha{\leq}\bigwedge[] \vdash \iota_2 \leq \nu_n.$

By the induction hypothesis, $\iota_2 \equiv \nu_n$, so $\iota \equiv \forall\alpha{\leq}\bigwedge[].\ \nu_n$, which is just ν_{n+1}. □

5.2.11. Lemma: There exists a pair of individual canonical types in $F_\wedge$ with no complete finite set of upper bounds.

Proof: Assume, for a contradiction, that $B \equiv \{\lambda_1..\lambda_n\}$ is a complete finite set of upper bounds for the types

$$\kappa \equiv \forall\alpha{\leq}\bigwedge[].\ \forall\beta{\leq}\bigwedge[].\ \alpha$$
$$\iota \equiv \forall\alpha{\leq}\bigwedge[].\ \forall\beta{\leq}\bigwedge[].\ \beta$$

and let

$$\mu_i \equiv \forall\alpha{\leq}\bigwedge[\nu_i].\ \forall\beta{\leq}\bigwedge[\nu_i].\ \nu_i$$

for every i. Note that each μ_i is a common supertype of κ and ι. Also, since there are more μ's than λ's, we can choose some $\lambda \in B$ and some μ_i and μ_j (with $i \neq j$) such that $\vdash \lambda \leq \mu_i$ and $\vdash \lambda \leq \mu_j$.

From $\vdash \kappa \leq \lambda$ and $\vdash \iota \leq \lambda$, syntax-directedness gives

$\lambda \equiv \forall\alpha{\leq}L_1.\ \forall\beta{\leq}L_2.\ \lambda_3$
$\alpha{\leq}L_1, \beta{\leq}L_2 \vdash \alpha \leq \lambda_3$
$\alpha{\leq}L_1, \beta{\leq}L_2 \vdash \beta \leq \lambda_3.$

Since $\vdash \lambda \leq \mu_i$, syntax-directedness again yields

$\vdash \bigwedge[\nu_i] \leq L_1$
$\alpha{\leq}\bigwedge[\nu_i] \vdash \bigwedge[\nu_i] \leq L_2$
$\alpha{\leq}\bigwedge[\nu_i], \beta{\leq}\bigwedge[\nu_i] \vdash \lambda_3 \leq \nu_i.$

By canonical narrowing (4.2.3.6),

$\alpha{\leq}\bigwedge[\nu_i], \beta{\leq}L_2 \vdash \alpha \leq \lambda_3$
$\alpha{\leq}\bigwedge[\nu_i], \beta{\leq}L_2 \vdash \beta \leq \lambda_3,$

and again

$\alpha{\leq}\bigwedge[\nu_i], \beta{\leq}\bigwedge[\nu_i] \vdash \alpha \leq \lambda_3$
$\alpha{\leq}\bigwedge[\nu_i], \beta{\leq}\bigwedge[\nu_i] \vdash \beta \leq \lambda_3.$

Now by syntax-directedness,

$\lambda_3 \equiv \alpha$ or $\alpha{\leq}\bigwedge[\nu_i], \beta{\leq}\bigwedge[\nu_i] \vdash \bigwedge[\nu_i] \leq \bigwedge[\lambda_3]$
$\lambda_3 \equiv \beta$ or $\alpha{\leq}\bigwedge[\nu_i], \beta{\leq}\bigwedge[\nu_i] \vdash \bigwedge[\nu_i] \leq \bigwedge[\lambda_3].$

Since $\lambda \equiv \alpha$ and $\lambda \equiv \beta$ cannot both be true, we have $\alpha \leq \bigwedge[\nu_i]$, $\beta \leq \bigwedge[\nu_i] \vdash \bigwedge[\nu_i] \leq \bigwedge[\lambda_3]$, i.e. (by syntax-directedness),

$$\alpha \leq \bigwedge[\nu_i], \beta \leq \bigwedge[\nu_i] \vdash \nu_i \leq \lambda_3.$$

Combining this with the type inclusion in the opposite direction (which we derived above), we get

$$\alpha \leq \bigwedge[\nu_i], \beta \leq \bigwedge[\nu_i] \vdash \nu_i \sim \lambda_3.$$

So, by Lemma 5.2.10, $\lambda_3 \equiv \nu_i$.

But starting from $\vdash \lambda \leq \mu_j$ and reasoning analogously, we can also obtain $\lambda_3 \equiv \nu_j$. Since $\nu_i \not\equiv \nu_j$, this is a contradiction. Our assumption that B is a complete finite set of upper bounds for κ and ι must therefore be false. □

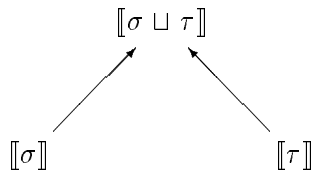
Clearly, if κ and ι have no complete finite set of upper bounds, then the composite canonical types $\bigwedge[\kappa]$ and $\bigwedge[\iota]$ have no lub. As in the first-order case, by Theorem 4.2.7.6, $F_\wedge$ has lubs iff its canonical formulation does, so this counterexample for the canonical system amounts to a proof of the nonexistence of lubs for the original formulation of $F_\wedge$. (Also, in ordinary $F_\wedge$ a complete finite set of upper bounds can always be conjoined to form a single least upper bound, so the nonexistence of lubs is equivalent to the nonexistence of complete finite sets of upper bounds for ordinary types.)

The most immediate implication of the nonexistence of least upper bounds is that standard techniques developed by Reynolds [123] for constructing and analyzing models of first-order intersection types will not generalize straightforwardly to $F_\wedge$.

Reynolds' model construction proceeds as follows. First, the set of canonical type expressions is defined as the limit of a series formed by beginning with the primitives and, at each stage, first closing under the $\rightarrow$ constructor and then forming all finite meets of the resulting set. The semantics of types is defined by induction on the same series of sets of types: the interpretation of a type τ at stage $n+1$ is defined in terms of the interpretations of the components of τ at stage n.

The intended interpretation of an intersection $\sigma \wedge \tau$ is the limit of a diagram containing the interpretations of σ and τ and all their common supertypes (c.f. Section 2.4.2). But even if σ and τ both exist at level n, there might be many common supertypes that will not appear until some later stage, so the limit with respect to only those supertypes that exist at level n might be too large. At each level, then, it appears that we would need to recalculate the interpretations of all the intersection types from previous levels. It is not obvious that this process would converge.

Fortunately, in $\lambda_\wedge$, every σ and τ possess a least upper bound $\sigma \sqcup \tau$, which, furthermore, always appears at the first stage containing both σ and τ. So $\sigma \wedge \tau$ may be interpreted as the limit of a very tidy diagram



with no fear that this interpretation will ever need to be revised.

The nonexistence of least upper bounds in $F_\wedge$ renders this important simplification useless. It is not clear whether a model could be constructed by "incrementally revising" the interpretations of intersections at each level, as described above. This kind of construction, if it worked at all, would almost certainly be much more complex than the known models of $\lambda_\wedge$.

5.3 Translation Semantics

Our translation semantics for $F_\wedge$ follows the style of Breazu-Tannen, Coquand, Gunter, and Scedrov's translation semantics for $F_\leq$ [10], appropriately extended to deal with intersection types (c.f. Section 2.4). Intuitively, we read $F_\wedge$ typing derivations as terms of the ordinary second-order λ-calculus extended with surjective tupling (system $F_\times$) by taking explicit account of the coercions introduced by the subtyping rules:

- Each $F_\wedge$ type τ is translated to an $F_\times$ type $[\![\tau]\!]$. In particular, a quantified type $\forall\alpha{\leq}\sigma.\ \tau$ is translated as $\forall\alpha.\ (\alpha{\to}[\![\sigma]\!]){\to}[\![\tau]\!]$, which makes explicit the required coercion function into σ from each appropriate value for α.
- Each subtyping derivation $c :: \Gamma \vdash \sigma \leq \tau$ is translated as an $F_\times$ term $[\![c]\!]$ such that $[\![\Gamma]\!] \vdash [\![c]\!] \in [\![\sigma]\!]{\to}[\![\tau]\!]$.
- Each typing derivation $s :: \Gamma \vdash e \in \tau$ is translated as an $F_\times$ term $[\![s]\!]$ such that $[\![\Gamma]\!] \vdash [\![s]\!] \in [\![\tau]\!]$. In particular, the translation of a type application supplies both a type and an appropriate coercion function as arguments.
- Intersection types are translated as Cartesian products (leaving their coherence properties implicit in the translation). This means, in particular, that a phrase of type $\top$ will be interpreted as an empty tuple, effectively throwing away any ill-typed subphrases.

5.3.1 Target Calculus

This section gives the syntax, typing rules, and equational theory of the polymorphic λ-calculus, system F, extended with surjective tuples, which we call $F_\times$ (c.f. [49]). Rather than choose a particular denotational or operational semantics for $F_\times$, we state an equational theory *constraining* a later choice of semantics; this gives us enough information to study both the properties of the translation in the following section and the equational theory of $F_\wedge$ given later on.

5.3.1.1. Definition: The set of $F_\times$ types is defined by the following abstract grammar:

$$\begin{array}{lll} \tau & ::= & \alpha \\ & | & \tau_1 \to \tau_2 \\ & | & \forall\alpha.\ \tau \\ & | & \Pi[\tau_1..\tau_n] \end{array}$$

5.3.1.2. Definition: The set of $F_\times$ terms is defined by the following abstract grammar:

$$\begin{array}{lll} e & ::= & x \\ & | & \lambda x{:}\tau.\ e \\ & | & e_1\ e_2 \\ & | & \Lambda\alpha.\ e \\ & | & e[\tau] \\ & | & \langle e_1..e_n\rangle \\ & | & \mathit{proj}_i\ e \end{array}$$

5.3.1.3. Convention: For the the following translations, we assume that the sets of term and type variables of $F_\times$ include at least the following: a term variable x for each $F_\wedge$ term variable x; a type variable α and a term variable c_α for each $F_\wedge$ type variable α; and the term variables y, z, f, v, c, and p.

5.3.1.4. Definition: An $F_\times$ *context* is a finite sequence of distinct type variables (with no bounds) and term variables with associated types:

$$\Gamma \quad ::= \quad \{\} \quad | \quad \Gamma, \alpha \quad | \quad \Gamma, x{:}\tau$$

5.3.1.5. Definition: The three-place typing relation $\Gamma \vdash e \in \tau$ of $F_\times$ is the least relation closed under the following rules:

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(F-Var)}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ e \in \tau_1 \to \tau_2} \qquad \text{(F-Arrow-I)}$$

$$\frac{\Gamma \vdash e_1 \in \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash e_1\ e_2 \in \tau_2} \qquad \text{(F-Arrow-E)}$$

$$\frac{\Gamma, \alpha \vdash e \in \tau}{\Gamma \vdash \Lambda\alpha.\ e \in \forall\alpha.\ \tau} \qquad \text{(F-All-I)}$$

$$\frac{\Gamma \vdash e \in \forall\alpha.\ \tau}{\Gamma \vdash e[\sigma] \in \{\sigma/\alpha\}\tau} \qquad \text{(F-All-E)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash e_i \in \tau_i}{\Gamma \vdash \langle e_1..e_n\rangle \in \Pi[\tau_1..\tau_n]} \qquad \text{(F-Prod)}$$

$$\frac{\Gamma \vdash e \in \Pi[\tau_1..\tau_n]}{\Gamma \vdash proj_i\ e \in \tau_i} \qquad \text{(F-Proj)}$$

5.3.1.6. Convention: When necessary to prevent confusion with other calculi, turnstiles in $F_\times$ derivations are written $\vdash^F$.

5.3.1.7. Definition: The equality relation on $F_\times$ terms is the least four-place relation closed under the following rules:

Conversion rules:

$$\frac{\Gamma \vdash (\lambda x{:}\sigma.\ f)\ v \in \tau}{\Gamma \vdash (\lambda x{:}\sigma.\ f)\ v = \{v/x\}f \in \tau} \qquad \text{(FEq-Beta)}$$

$$\frac{\Gamma \vdash (\Lambda\alpha.\ f)\ [\phi] \in \tau}{\Gamma \vdash (\Lambda\alpha.\ f)\ [\phi] = \{\phi/\alpha\}f \in \tau} \qquad \text{(FEq-Beta2)}$$

$$\frac{\Gamma \vdash \lambda x{:}\sigma.\ f\ x \in \tau \qquad x \notin FV(f)}{\Gamma \vdash \lambda x{:}\sigma.\ f\ x = f \in \tau} \qquad \text{(FEq-Eta)}$$

$$\frac{\Gamma \vdash \Lambda\alpha.\ f\ [\alpha] \in \tau \qquad \alpha \notin FTV(f)}{\Gamma \vdash \Lambda\alpha.\ f\ [\alpha] = f \in \tau} \qquad \text{(FEq-Eta2)}$$

$$\frac{\Gamma \vdash proj_i\ \langle e_1..e_n\rangle \in \tau}{\Gamma \vdash proj_i\ \langle e_1..e_n\rangle = e_i \in \tau} \qquad \text{(FEq-Pi)}$$

$$\frac{\Gamma \vdash e \in \langle \tau_1..\tau_n\rangle}{\Gamma \vdash e = \langle (proj_1\ e)\,..\,(proj_n\ e)\rangle \in \Pi[\tau_1..\tau_n]} \qquad \text{(FEq-Surj)}$$

Congruence rules:

$$\frac{\Gamma \vdash e \in \tau}{\Gamma \vdash e = e \in \tau} \qquad \text{(FEq-Refl)}$$

$$\frac{\Gamma \vdash e = e' \in \tau}{\Gamma \vdash e' = e \in \tau} \quad \text{(FEq-Symm)}$$

$$\frac{\Gamma \vdash e_1 = e_2 \in \tau \qquad \Gamma \vdash e_2 = e_3 \in \tau}{\Gamma \vdash e_1 = e_3 \in \tau} \quad \text{(FEq-Trans)}$$

$$\frac{\Gamma, x{:}\sigma \vdash e = e' \in \tau}{\Gamma \vdash \lambda x{:}\sigma.\ e = \lambda x{:}\sigma.\ e' \in \tau} \quad \text{(FEq-Abs)}$$

$$\frac{\Gamma \vdash e_1 = e_1' \in \sigma{\to}\tau \qquad \Gamma \vdash e_2 = e_2' \in \sigma}{\Gamma \vdash e_1\ e_2 = e_1'\ e_2' \in \tau} \quad \text{(FEq-App)}$$

$$\frac{\Gamma, \alpha \vdash e = e' \in \tau}{\Gamma \vdash \Lambda\alpha.\ e = \Lambda\alpha.\ e' \in \tau} \quad \text{(FEq-TAbs)}$$

$$\frac{\Gamma \vdash e = e' \in \forall\alpha.\ \tau}{\Gamma \vdash e\ [\phi] = e'\ [\phi] \in \{\phi/\alpha\}\tau} \quad \text{(FEq-TApp)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash e_i = e_i' \in \tau_i}{\Gamma \vdash \langle e_1..e_n\rangle = \langle e_1'..e_n'\rangle \in \Pi[\tau_1..\tau_n]} \quad \text{(FEq-Tuple)}$$

$$\frac{\Gamma \vdash e = e' \in \Pi[\tau_1..\tau_n]}{\Gamma \vdash \mathit{proj}_i\ e = \mathit{proj}_i\ e' \in \tau_i} \quad \text{(FEq-Proj)}$$

5.3.2 Ordinary Derivations

It is technically convenient to give translations for both ordinary subtyping and typing derivations and the algorithmic forms discussed in Sections 4.2.8 and 4.3.2. We begin by translating ordinary derivations.

5.3.2.1. Definition:

$$\begin{array}{lcl}
[\![\alpha]\!] & = & \alpha \\
[\![\tau_1{\to}\tau_2]\!] & = & [\![\tau_1]\!]{\to}[\![\tau_2]\!] \\
[\![\forall\alpha{\le}\tau_1.\ \tau_2]\!] & = & \forall\alpha.\ (\alpha{\to}[\![\tau_1]\!]){\to}[\![\tau_2]\!] \\
[\![\bigwedge[\tau_1..\tau_n]]\!] & = & \Pi[[\![\tau_1]\!] .. [\![\tau_n]\!]].
\end{array}$$

5.3.2.2. Lemma: $\{[\![\sigma]\!]/\alpha\}[\![\tau]\!] = [\![\{\sigma/\alpha\}\tau]\!]$.

Proof: Straightforward. □

5.3.2.3. Definition: The following abbreviations for $F_\times$ terms are used in the translation:

$$\begin{array}{lcl}
f_1\ ;\ f_2 & \stackrel{\text{def}}{=} & \lambda v{:}\tau_1.\ f_2\ (f_1\ v) \\
 & & \text{where } \Gamma \stackrel{F}{\vdash} f_1 \in \tau_1{\to}\tau_2 \text{ and } \Gamma \stackrel{F}{\vdash} f_2 \in \tau_2{\to}\tau_3 \\
\mathit{dist}_{\bigwedge[\sigma\to\tau_1\ ..\ \sigma\to\tau_n]} & \stackrel{\text{def}}{=} & \lambda p{:}[\![\bigwedge[\sigma{\to}\tau_1\ ..\ \sigma{\to}\tau_n]]\!]. \\
 & & \quad \lambda v{:}[\![\sigma]\!].\ \langle(\mathit{proj}_1\ p)\ v\ ..\ (\mathit{proj}_n\ p)\ v\rangle \\
\mathit{dist}_{\bigwedge[\forall\alpha\le\sigma.\ \tau_1\ ..\ \forall\alpha\le\sigma.\ \tau_n]} & \stackrel{\text{def}}{=} & \lambda p{:}[\![\bigwedge[\forall\alpha{\le}\sigma.\ \tau_1\ ..\ \forall\alpha{\le}\sigma.\ \tau_n]]\!]. \\
 & & \quad \Lambda\alpha.\ \lambda c_\alpha{:}\alpha{\to}[\![\sigma]\!].\ \langle(\mathit{proj}_1\ p)\ [\alpha]\ c_\alpha\ ..\ (\mathit{proj}_n\ p)\ [\alpha]\ c_\alpha\rangle.
\end{array}$$

5.3.2.4. Definition:

$$\begin{array}{lcl}
[\![\{\}]\!] & = & \{\} \\
[\![\Gamma, x{:}\tau]\!] & = & [\![\Gamma]\!], x{:}[\![\tau]\!] \\
[\![\Gamma, \alpha{\le}\tau]\!] & = & [\![\Gamma]\!], \alpha, c_\alpha{:}\alpha{\to}[\![\tau]\!]
\end{array}$$

5.3.2.5. Definition:

$[\![id :: \Gamma \vdash \tau \leq \tau]\!]$ (T-Sub-Refl)
$= \quad \lambda v{:}[\![\tau]\!].\ v$

$[\![c\ ;\ d :: \Gamma \vdash \tau_1 \leq \tau_3]\!]$ (T-Sub-Trans)
$= \quad [\![c]\!]\ ;\ [\![d]\!]$

$[\![V_\alpha :: \Gamma \vdash \alpha \leq \Gamma(\alpha)]\!]$ (T-Sub-TVar)
$= \quad c_\alpha$

$[\![c{\rightarrow}d :: \Gamma \vdash \sigma_1{\rightarrow}\sigma_2 \leq \tau_1{\rightarrow}\tau_2]\!]$ (T-Sub-Arrow)
$= \quad \lambda f{:}[\![\tau_1{\rightarrow}\tau_2]\!].\ [\![c]\!]\ ;\ f\ ;\ [\![d]\!]$

$[\![\forall\alpha{\leq}c.\ d :: \Gamma \vdash \forall\alpha{\leq}\sigma_1.\ \sigma_2 \leq \forall\alpha{\leq}\tau_1.\ \tau_2]\!]$ (T-Sub-All)
$= \quad \lambda f{:}[\![\forall\alpha{\leq}\sigma_1.\ \sigma_2]\!].\ \Lambda\alpha.\ \lambda c_\alpha{:}\alpha{\rightarrow}[\![\tau_1]\!].$
$\qquad [\![d]\!] \cdot (f \cdot [\alpha] \cdot (c_\alpha\ ;\ [\![c]\!]))$

$[\![\langle c_1..c_n \rangle :: \Gamma \vdash \sigma \leq \bigwedge[\tau_1..\tau_n]]\!]$ (T-Sub-Inter-G)
$= \quad \lambda x{:}[\![\sigma]\!].\ \langle [\![c_1]\!] \cdot x\ ..\ [\![c_n]\!] \cdot x \rangle$

$[\![proj_i :: \Gamma \vdash \bigwedge[\tau_1..\tau_n] \leq \tau_i]\!]$ (T-Sub-Inter-LB)
$= \quad proj_i$

$[\![dist\text{-}ia :: \Gamma \vdash \bigwedge[\sigma{\rightarrow}\tau_1..\sigma{\rightarrow}\tau_n] \leq \sigma \rightarrow \bigwedge[\tau_1..\tau_n]]\!]$ (T-Sub-Dist-IA)
$= \quad dist_{\bigwedge[\sigma\rightarrow\tau_1\ ..\ \sigma\rightarrow\tau_n]}$

$[\![dist\text{-}iq :: \Gamma \vdash \bigwedge[\forall\alpha{\leq}\sigma.\ \tau_1..\forall\alpha{\leq}\sigma.\ \tau_n] \leq \forall\alpha{\leq}\sigma.\ \bigwedge[\tau_1..\tau_n]]\!]$ (T-Sub-Dist-IQ)
$= \quad dist_{\bigwedge[\forall\alpha\leq\sigma.\ \tau_1\ ..\ \forall\alpha\leq\sigma.\ \tau_n]}$

5.3.2.6. Lemma: If $\Gamma \vdash^{\wedge} \sigma \leq \tau$, then $[\![\Gamma]\!] \vdash^{F} [\![\Gamma \vdash \sigma \leq \tau]\!] \in [\![\sigma]\!]{\rightarrow}[\![\tau]\!]$.

Proof: By induction on the structure of the given derivation. □

5.3.2.7. Definition:

$[\![V_x :: \Gamma \vdash x \in \Gamma(x)]\!]$ (T-Var)
$= \quad x$

$[\![\lambda x{:}\tau_1.\ s :: \Gamma \vdash \lambda x{:}\tau_1.\ e \in \tau_1{\rightarrow}\tau_2]\!]$ (T-Arrow-I)
$= \quad \lambda x{:}[\![\tau_1]\!].\ [\![s]\!]$

$[\![s_1\ s_2 :: \Gamma \vdash (e_1\ e_2) \in \tau_2]\!]$ (T-Arrow-E)
$= \quad [\![s_1]\!] \cdot [\![s_2]\!]$

$[\![\Lambda\alpha{\leq}\tau_1.\ s :: \Gamma \vdash \Lambda\alpha{\leq}\tau_1.\ e \in \forall\alpha{\leq}\tau_1.\ \tau_2]\!]$ (T-All-I)
$= \quad \Lambda\alpha.\ \lambda c_\alpha{:}\alpha{\rightarrow}[\![\tau_1]\!].\ [\![s]\!]$

$[\![s\ [c] :: \Gamma \vdash e[\tau] \in \{\tau/\alpha\}\tau_2]\!]$ (T-All-E)
$= \quad [\![s]\!] \cdot [[\![\tau]\!]] \cdot [\![c]\!]$

$[\![for\ \alpha\ in\ \sigma_1..\sigma_n.\ s_i :: \Gamma \vdash for\ \alpha\ in\ \sigma_1..\sigma_n.\ e \in \tau_i]\!]$ (T-For)
$= \quad [\![s_i]\!]$

$[\![\langle s_1..s_n \rangle :: \Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]]\!]$ (T-Inter-I)
$= \quad \langle [\![s_1]\!]\ ..\ [\![s_n]\!] \rangle$

$[\![s \uparrow c :: \Gamma \vdash e \in \tau]\!]$ (T-Sub)
$= \quad [\![c]\!] \cdot [\![s]\!]$

5.3.2.8. Theorem: If $\Gamma \vdash^{\wedge} e \in \tau$, then $[\![\Gamma]\!] \vdash^{F} [\![\Gamma \vdash e \in \tau]\!] \in [\![\tau]\!]$.

Proof: By induction on the structure of the given derivation, using Lemma 5.3.2.6 for the cases involving subtyping derivations. □

5.3.2.9. Remark: This amounts to a kind of type-soundness property for the pure calculus: well-formed $F_\wedge$ typing derivations translate to well-typed — hence well-behaved — $F_\times$ terms.

5.3.3 Algorithmic Derivations

We can give an analogous translation for the forms of derivations used by the subtyping algorithm of Section 4.2.8 and the type synthesis algorithm of Section 4.3.2. This is essentially just the composition of the translation functions $(\text{—})^\wedge$ of Definitions 4.2.8.7 and 4.3.2.5 with the translation given in the previous section, but it is worth writing out in its own right because it suggests a possible architecture for the back end of a compiler for $F_\wedge$.

5.3.3.1. Definition:

$$dist^*_{\Gamma,\wedge[[\,]\Rightarrow\tau_1..[\,]\Rightarrow\tau_n]} \stackrel{\text{def}}{=} \lambda v{:}[\![\textstyle\bigwedge[[\,]\Rightarrow\tau_1\,..\,[\,]\Rightarrow\tau_n]]\!].\ v$$

$$dist^*_{\Gamma,\wedge[\sigma\to(X\Rightarrow\tau_1)..\sigma\to(X\Rightarrow\tau_n)]} \stackrel{\text{def}}{=} dist_{\wedge[\sigma\to(X\Rightarrow\tau_1)..\sigma\to(X\Rightarrow\tau_n)]}\ ;\ dist^*_{\Gamma,\wedge[X\Rightarrow\tau_1..X\Rightarrow\tau_n]}$$

$$dist^*_{\Gamma,\wedge[\forall\alpha\le\sigma.\ X\Rightarrow\tau_1..\forall\alpha\le\sigma.\ X\Rightarrow\tau_n]} \stackrel{\text{def}}{=} dist_{\wedge[\forall\alpha\le\sigma.\ X\Rightarrow\tau_1..\forall\alpha\le\sigma.\ X\Rightarrow\tau_n]}\ ;\ dist^*_{(\Gamma,\alpha\le\sigma),\wedge[X\Rightarrow\tau_1..X\Rightarrow\tau_n]}.$$

5.3.3.2. Definition: We also need to introduce a *tuple comprehension* notation analogous to the finite sequence comprehensions used earlier (c.f. 2.1.1). For example, the expression $\langle\langle e_i,\, e_i\rangle \mid e_i \in [f_1\,..\,f_n]\rangle$ stands for the tuple of tuples $\langle\langle f_1,\, f_1\rangle\,..\,\langle f_n,\, f_n\rangle\rangle$.

5.3.3.3. Theorem: The composition of the translation $(\text{—})^\wedge$ and the translation $[\![\text{—}]\!]$ from ordinary $F_\wedge$ subtyping and typing derivations into $F_\times$ terms can be characterized by the following equations:

$[\![\Gamma \vdash \sigma \le X\Rightarrow\textstyle\bigwedge[\tau_1..\tau_n]]\!]$ (T-ASubR-Inter)
$\quad = \quad dist^*_{\Gamma,X\Rightarrow\wedge[\tau_1..\tau_n]} \cdot \langle [\![\Gamma \vdash \sigma \le X\Rightarrow\tau_1]\!]\,..\,[\![\Gamma \vdash \sigma \le X\Rightarrow\tau_n]\!]\rangle$

$[\![\Gamma \vdash \bigwedge[\sigma_1..\sigma_n] \le X\Rightarrow\alpha]\!]$ (T-ASubL-Inter)
$\quad = \quad proj_i\ ;\ [\![\Gamma \vdash \sigma_i \le X\Rightarrow\alpha]\!]$

$[\![\Gamma \vdash \sigma_1\to\sigma_2 \le [\tau_1, X_2]\Rightarrow\alpha]\!]$ (T-ASubL-Arrow)
$\quad = \quad \lambda f{:}[\![\sigma_1]\!]\to[\![\sigma_2]\!].\ [\![\Gamma \vdash \tau_1 \le [\,]\Rightarrow\sigma_1]\!]\ ;\ f\ ;\ [\![\Gamma \vdash \sigma_2 \le X_2\Rightarrow\alpha]\!]$

$[\![\Gamma \vdash \forall\beta\le\sigma_1.\ \sigma_2 \le [\beta\le\tau_1, X_2]\Rightarrow\alpha]\!]$ (T-ASubL-All)
$\quad = \quad \lambda f{:}(\forall\alpha.\ (\alpha\to[\![\sigma_1]\!])\to[\![\sigma_2]\!]).\ \Lambda\alpha.\ \lambda c_\alpha{:}(\alpha\to[\![\tau_1]\!]).$
$\qquad\quad [\![\Gamma, \beta\le\tau_1 \vdash \sigma_2 \le X_2\Rightarrow\alpha]\!] \cdot (f \cdot [\alpha] \cdot (c_\alpha\ ;\ [\![\Gamma \vdash \tau_1 \le [\,]\Rightarrow\sigma_1]\!]))$

$[\![\Gamma \vdash \alpha \le [\,]\Rightarrow\alpha]\!]$ (T-ASubL-Refl)
$\quad = \quad \lambda v{:}\alpha.\ v$

$[\![\Gamma \vdash \beta \le X\Rightarrow\alpha]\!]$ (T-ASubL-TVar)
$\quad = \quad c_\beta\ ;\ [\![\Gamma \vdash \Gamma(\beta) \le X\Rightarrow\alpha]\!]$

$[\![\Gamma \vdash x \in \Gamma(x)]\!]$ (TA-Var)
$\quad = \quad x$

$[\![\Gamma \vdash \lambda x{:}\tau_1.\ e \in \tau_1\to\tau_2]\!]$ (TA-Arrow-I)
$\quad = \quad \lambda x{:}[\![\tau_1]\!].\ [\![\Gamma, x{:}\tau_1 \vdash e \in \tau_2]\!]$

$[\![\Gamma \vdash e_1\ e_2 \in \bigwedge[\psi_i \mid \phi_i\to\psi_i \in arrowbasis_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \sigma_2 \le \phi_i]]\!]$ (TA-Arrow-E)
$\quad = \quad \langle\, ([\![\Gamma \vdash \sigma_1 \le \phi_i\to\psi_i]\!] \cdot [\![\Gamma \vdash e_1 \in \sigma_1]\!])$
$\qquad\qquad \cdot ([\![\Gamma \vdash \sigma_2 \le \phi_i]\!] \cdot [\![\Gamma \vdash e_2 \in \sigma_2]\!])$
$\qquad\quad \mid \phi_i\to\psi_i \in arrowbasis_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \sigma_2 \le \phi_i\,\rangle$

$[\![\Gamma \vdash \Lambda\alpha\le\tau_1.\ e \in \forall\alpha\le\tau_1.\ \tau_2]\!]$ (TA-All-I)
$\quad = \quad \Lambda\alpha.\ \lambda c_\alpha{:}(\alpha\to[\![\tau_1]\!]).\ [\![\Gamma, \alpha\le\tau_1 \vdash e \in \tau_2]\!]$

$$[\![\Gamma \stackrel{\perp}{\vdash} e[\tau] \in \bigwedge[\{\tau/\alpha\}\psi_i \mid (\forall\alpha{\leq}\phi_i.\ \psi_i) \in \mathit{allbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \stackrel{\perp}{\vdash} \tau \leq \phi_i]]\!] \qquad \text{(TA-All-E)}$$
$$= \quad \langle\, ([\![\Gamma \stackrel{\perp}{\vdash} \sigma_1 \leq \forall\alpha{\leq}\phi_i.\ \psi_i]\!] \cdot [\![\Gamma \stackrel{\perp}{\vdash} e_1 \in \sigma_1]\!]) \cdot [[\![\tau]\!]] \cdot [\![\Gamma \stackrel{\perp}{\vdash} \tau \leq \phi_i]\!]$$
$$\mid \forall\alpha{\leq}\phi_i.\ \psi_i \in \mathit{allbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \stackrel{\perp}{\vdash} \tau \leq \phi_i \,\rangle$$
$$[\![\Gamma \stackrel{\perp}{\vdash} \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e \in \bigwedge[\tau_1..\tau_n]]\!] \qquad \text{(TA-For)}$$
$$= \quad \langle [\![\Gamma \stackrel{\perp}{\vdash} \{\sigma_1/\alpha\}e \in \tau_1]\!] \,..\, [\![\Gamma \stackrel{\perp}{\vdash} \{\sigma_n/\alpha\}e \in \tau_n]\!] \rangle$$

Proof: By induction on algorithmic derivations. □

5.4 Coherence (Preliminary Results)

This section states an appropriate coherence property (c.f. Section 2.4) for the translation functions on ordinary subtyping and typing derivations. Unfortunately, because $F_\wedge$ does not have least upper bounds, a proof of this property lies beyond the scope of this thesis. Section 8.2.2 reviews the difficulties with extending standard methods of proving coherence and suggests some possible approaches.

5.4.1. Conjecture: [Coherence of subtyping] If $c :: \Gamma \stackrel{\wedge}{\vdash} \sigma \leq \tau$ and $d :: \Gamma \stackrel{\wedge}{\vdash} \sigma \leq \tau$, then $[\![\Gamma]\!] \stackrel{F}{\vdash} [\![c]\!] = [\![d]\!] \in [\![\sigma]\!]{\to}[\![\tau]\!]$.

5.4.2. Conjecture: [Coherence of typing] If $s :: \Gamma \stackrel{\wedge}{\vdash} e \in \tau$ and $t :: \Gamma \stackrel{\wedge}{\vdash} e \in \tau$, then $[\![\Gamma]\!] \stackrel{F}{\vdash} [\![s]\!] = [\![t]\!] \in [\![\tau]\!]$.

5.4.3. Remark: For the remainder of the chapter, we *assume* that the translation semantics is coherent.

5.4.4. Lemma: If $c :: \Gamma \stackrel{\wedge}{\vdash} \sigma \leq \tau$, then $[\![\Gamma]\!] \stackrel{F}{\vdash} [\![c]\!] = [\![c^!]\!] \in [\![\sigma]\!]{\to}[\![\tau]\!]$, where $c^! :: \Gamma \stackrel{\perp}{\vdash} \sigma \leq \tau$ is the algorithmic derivation whose existence is guaranteed by Theorem 4.2.8.12.

Proof: By Theorem 5.3.3.3 and the coherence of subtyping (5.4.1). □

5.4.5. Lemma: If $s :: \Gamma \stackrel{\wedge}{\vdash} e \in \tau$, then $[\![\Gamma]\!] \stackrel{F}{\vdash} [\![s]\!] = [\![c]\!] \cdot [\![s^!]\!] \in [\![\tau]\!]$, where $s^! :: \Gamma \stackrel{\perp}{\vdash} e \in \tau$ and $c^! :: \Gamma \stackrel{\perp}{\vdash} \sigma \leq \tau$ are the algorithmic derivations of $\Gamma \vdash e \in \sigma$ and $\Gamma \vdash \sigma \leq \tau$ whose existence is guaranteed by Theorem 4.3.2.7 and Lemma 5.4.4.

Proof: By Theorem 5.3.3.3 and Lemma 5.4.4. □

5.5 Equational Theory

As an alternative perspective on the meaning of $F_\wedge$ programs, we offer a theory of provable equality for $F_\wedge$ terms. Like the equational theory of $F_\leq$ studied by Cardelli, Martini, Mitchell, and Scedrov [30], this equational theory is based on a notion of "equality at a type": $\Gamma \vdash e = e' \in \tau$. It includes typed analogues of the familiar β and η conversion rules for both values and types, plus the usual collection of rules to ensure that the equality relation forms a congruence. The two novel elements are:

- A rule of *intersection equality*, Eq-Inter, which states that whenever e and e' are known to be equal at all of the types $\tau_1..\tau_n$ separately, they may be judged equal at $\bigwedge[\tau_1..\tau_n]$. In particular, every pair of terms is equal at type $\top$ (c.f. Curien and Ghelli's *Top*-equality rule [50]).

- A collection of rules for reorganizing *for* expressions. The main goal of these rules is to ensure that the *for* marker can never block a β- or η-conversion step. For example, the "potential β-redex"

  $$(\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ \lambda x{:}\sigma.\ f)\ v$$

  is equal to the expression

  $$\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ (\lambda x{:}\sigma.\ f)\ v$$

  with an actual β-redex.

We begin by presenting the equality rules, establishing some basic properties, and checking that equality is well-defined with respect to the typing relation, in the sense that $\Gamma \vdash e = e' \in \tau$ implies $\Gamma \vdash e \in \tau$ and $\Gamma \vdash e' \in \tau$. We then establish a connection between the equational theory and both the untyped semantics of Section 5.1 and the translation semantics given in Section 5.3 by showing that the equational theory correctly (though incompletely) describes the behavior of the interpretations of terms. (A more informative equational description of $F_\wedge$'s semantics might try to characterize *exactly* the valid equivalences between $F_\wedge$ derivations induced by the translation semantics.)

The theory described in this section owes a great deal to conversations with QingMing Ma, who has studied a related equational theory for an extension of $F_\wedge$ [89].

5.5.1 Definitions

5.5.1.1. Definition: The *pure equational theory of* $F_\wedge$ is the least four-place relation $\Gamma \vdash e = e' \in \tau$ closed under the following rules:

Conversion rules:

$$\frac{\Gamma \vdash (\lambda x{:}\sigma.\ f)\ v \in \tau}{\Gamma \vdash (\lambda x{:}\sigma.\ f)\ v = \{v/x\}f \in \tau} \quad \text{(Eq-Beta)}$$

$$\frac{\Gamma \vdash (\Lambda\alpha{\leq}\sigma.\ f)\ [\phi] \in \tau}{\Gamma \vdash (\Lambda\alpha{\leq}\sigma.\ f)\ [\phi] = \{\phi/\alpha\}f \in \tau} \quad \text{(Eq-Beta2)}$$

$$\frac{\Gamma \vdash \lambda x{:}\sigma.\ f\ x \in \tau \qquad \Gamma \vdash f \in \tau}{\Gamma \vdash \lambda x{:}\sigma.\ f\ x = f \in \tau} \quad \text{(Eq-Eta)}$$

$$\frac{\Gamma \vdash \Lambda\alpha{\leq}\sigma.\ f\ [\alpha] \in \tau \qquad \Gamma \vdash f \in \tau}{\Gamma \vdash \Lambda\alpha{\leq}\sigma.\ f\ [\alpha] = f \in \tau} \quad \text{(Eq-Eta2)}$$

Intersection rule:

$$\frac{\text{for all } i,\ \Gamma \vdash e = e' \in \tau_i}{\Gamma \vdash e = e' \in \bigwedge[\tau_1..\tau_n]} \quad \text{(Eq-Inter)}$$

Reorganization rules:

$$\frac{\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ x \in \tau}{\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ x = x \in \tau} \quad \text{(Eq-For/Var)}$$

$$\frac{\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ \lambda x{:}\sigma.\ e \in \tau \qquad \alpha \notin FTV(\sigma)}{\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ \lambda x{:}\sigma.\ e = \lambda x{:}\sigma.\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e \in \tau} \quad \text{(Eq-For/Abs)}$$

$$\frac{\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e_1\ e_2 \in \tau}{\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e_1\ e_2 = (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e_1)\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e_2) \in \tau} \quad \text{(Eq-For/App)}$$

$$\frac{\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ \Lambda\beta{\leq}\sigma.\ e \in \tau \qquad \alpha \notin FTV(\sigma)}{\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ \Lambda\beta{\leq}\sigma.\ e = \Lambda\beta{\leq}\sigma.\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e \in \tau} \quad \text{(Eq-For/TAbs)}$$

$$\frac{\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e\,[\sigma] \in \tau \qquad \alpha \notin FTV(\sigma)}{\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e\,[\sigma] = (\mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e)\,[\sigma] \in \tau} \quad \text{(Eq-For/TApp)}$$

$$\frac{\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ \mathit{for}\,\beta\ \mathit{in}\,\tau_1..\tau_n.\ e \in \tau \qquad \alpha \notin \bigcup_i FTV(\tau_i)}{\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ \mathit{for}\,\beta\ \mathit{in}\,\tau_1..\tau_n.\ e = \mathit{for}\,\beta\ \mathit{in}\,\tau_1..\tau_n.\ \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e \in \tau} \quad \text{(Eq-For/For)}$$

Congruence rules:

$$\frac{\Gamma \vdash e \in \tau}{\Gamma \vdash e = e \in \tau} \quad \text{(Eq-Refl)}$$

$$\frac{\Gamma \vdash e = e' \in \tau}{\Gamma \vdash e' = e \in \tau} \quad \text{(Eq-Symm)}$$

$$\frac{\Gamma \vdash e_1 = e_2 \in \tau \qquad \Gamma \vdash e_2 = e_3 \in \tau}{\Gamma \vdash e_1 = e_3 \in \tau} \quad \text{(Eq-Trans)}$$

$$\frac{\Gamma, x{:}\sigma \vdash e = e' \in \tau}{\Gamma \vdash \lambda x{:}\sigma.\ e = \lambda x{:}\sigma.\ e' \in \sigma{\to}\tau} \quad \text{(Eq-Abs)}$$

$$\frac{\Gamma \vdash e_1 = e_1' \in \sigma{\to}\tau \qquad \Gamma \vdash e_2 = e_2' \in \sigma}{\Gamma \vdash e_1\,e_2 = e_1'\,e_2' \in \tau} \quad \text{(Eq-App)}$$

$$\frac{\Gamma, \alpha{\le}\sigma \vdash e = e' \in \tau}{\Gamma \vdash \Lambda\alpha{\le}\sigma.\ e = \Lambda\alpha{\le}\sigma.\ e' \in \forall\alpha{\le}\sigma.\ \tau} \quad \text{(Eq-TAbs)}$$

$$\frac{\Gamma \vdash e = e' \in \forall\alpha{\le}\sigma.\ \tau \qquad \Gamma \vdash \phi \le \sigma}{\Gamma \vdash e\,[\phi] = e'\,[\phi] \in \{\phi/\alpha\}\tau} \quad \text{(Eq-TApp)}$$

$$\frac{\Gamma \vdash \{\sigma_i/\alpha\}e = \{\sigma_i/\alpha\}e' \in \tau_i}{\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e = \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e' \in \tau_i} \quad \text{(Eq-For)}$$

5.5.1.2. Remark: In general, the conversion, intersection, and reorganization rules are formulated so that it is obvious that the left-hand side of each equality has the appropriate type, while the type of the right-hand side is not explicitly mentioned. We could give both types as premises, of course, but this extra clutter is unnecessary, since it will be easy to show that the right-hand side also has the appropriate type (c.f. 5.5.2.10). The one exception is the rules Eq-Eta and Eq-Eta2, where the proof that the right-hand side has the same type as the left-hand side requires a strengthening lemma thas has not been proved for this system. We give typing premises for both sides of these rules.

5.5.1.3. Remark: Note that the second premise in Eq-Eta implies the more familiar side condition "$x \notin FV(f)$." A similar remark applies to Eq-Eta2.

5.5.2 Basic Properties

5.5.2.1. Convention: By Lemmas 5.4.4 and 5.4.5, the interpretation of each algorithmic derivation is equal to the interpretation of some ordinary derivation with the same conclusion. Since, by the assumption of coherence, the interpretations of all ordinary derivations are equal, and since arbitrary subphrases of $F_\times$ equality statements may be replaced by equal subphrases without affecting derivability, we often simplify arguments below by dropping the decorations $\vdash$ and $\vdash^{\wedge}$ and regarding any two derivations of the same statement in *either* typing system as identical.

5.5.2.2. Lemma: [Equality context permutation] If Γ is a permutation of Γ' and both are closed, then $\Gamma \vdash e = e' \in \tau$ iff $\Gamma' \vdash e = e' \in \tau$.

Proof: By induction on derivations. □

5.5.2.3. Convention: [c.f. Convention 4.1.4] Two equality statements or derivations differing only in the ordering of contexts are considered identical.

5.5.2.4. Lemma: [Equality weakening] Let $(\Gamma, x{:}\phi)$ and $(\Gamma, \alpha{\leq}\phi)$ be closed contexts. Then

1. $\Gamma \vdash e = e' \in \tau$ implies $\Gamma, x{:}\phi \vdash e = e' \in \tau$.
2. $\Gamma \vdash e = e' \in \tau$ implies $\Gamma, \alpha{\leq}\phi \vdash e = e' \in \tau$.

Proof: Straightforward. □

5.5.2.5. Lemma: [Congruence] The following rule is derivable:

$$\frac{\Gamma \vdash v = v' \in \sigma \qquad \Gamma, x{:}\sigma \vdash e \in \tau}{\Gamma \vdash \{v/x\}e = \{v'/x\}e \in \tau} \qquad \text{(Eq-Cong)}$$

Proof: By induction on a derivation of $\Gamma, x{:}\sigma \vdash e \in \tau$, using equality strengthening for the base case $e \equiv y \not\equiv x$ and equality weakening for the Arrow-I and All-I cases. □

5.5.2.6. Lemma: [Equality subsumption] The following rule is derivable:

$$\frac{\Gamma \vdash e = e' \in \sigma \qquad \Gamma \vdash \sigma \leq \tau}{\Gamma \vdash e = e' \in \tau} \qquad \text{(D-Eq-Sub)}$$

Proof: Choose $x \notin dom(\Gamma)$. Then $\Gamma, x{:}\sigma \vdash x \in \sigma$ by rule Var. By Lemma 4.1.5, $\Gamma, x{:}\sigma \vdash \sigma \leq \tau$. By Sub, $\Gamma, x{:}\sigma \vdash x \in \tau$. By Eq-Refl, $\Gamma, x{:}\sigma \vdash x = x \in \tau$. By Eq-Abs, $\Gamma \vdash (\lambda x{:}\sigma.\ x) = (\lambda x{:}\sigma.\ x) \in \sigma{\to}\tau$. By Eq-App and the left-hand assumption, $\Gamma \vdash (\lambda x{:}\sigma.\ x)\ e = (\lambda x{:}\sigma.\ x)\ e' \in \tau$. By Eq-Beta (twice), Eq-Symm, and Eq-Trans, $\Gamma \vdash e = e' \in \tau$. □

5.5.2.7. Lemma: [*for* introduction] If $\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e \in \tau$ is a closed statement and $\Gamma \vdash e \in \tau$, then $\Gamma \vdash e = \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ e \in \tau$.

Proof: By induction on a derivation of $\Gamma \vdash e \in \tau$, using, in turn, rules Eq-For/Var ... Eq-For/For, Eq-Inter, and Eq-Sub. □

The next lemma verifies that the *for* construct never blocks potential β- or η reductions.

5.5.2.8. Lemma:

1. If $\Gamma \vdash (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ v \in \tau$, then
$$\Gamma \vdash (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ v = \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ (f\ v) \in \tau.$$
2. If $\Gamma \vdash (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ [\phi] \in \tau$, then
$$\begin{array}{lll} \Gamma & \vdash & (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ [\phi] \\ & = & \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ (f\ [\phi]) \\ & \in & \tau. \end{array}$$
3. If $\Gamma \vdash \lambda x{:}\sigma.\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ x \in \tau$, then
$$\begin{array}{lll} \Gamma & \vdash & \lambda x{:}\sigma.\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ x \\ & = & \lambda x{:}\sigma.\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ x \\ & \in & \tau. \end{array}$$
4. If $\Gamma \vdash \Lambda\beta{\leq}\sigma.\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ [\beta] \in \tau$, then
$$\begin{array}{lll} \Gamma & \vdash & \Lambda\beta{\leq}\sigma.\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ [\beta] \\ & = & \Lambda\beta{\leq}\sigma.\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ [\beta] \\ & \in & \tau. \end{array}$$

Proof:

1. By minimal typing (4.3.2.7), there is a type θ such that

   $\Gamma \stackrel{!}{\vdash} (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ v\ \in\ \theta$
   $\Gamma\ \vdash\ \theta\ \leq\ \tau.$

   By the syntax-directedness of type synthesis (4.3.2.3),

   $\Gamma \stackrel{!}{\vdash} \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ \in\ \theta_1$
   $\Gamma \stackrel{!}{\vdash} v\ \in\ \theta_2$
   $\theta \equiv \bigwedge[\psi_i \mid (\phi_i{\rightarrow}\psi_i)\ \in\ \mathit{arrowbasis}_\Gamma(\theta_1)$ and $\Gamma\ \vdash\ \theta_2\ \leq\ \phi_i]$,

   and again,

   $\Gamma \stackrel{!}{\vdash} \{\sigma_j/\alpha\}f\ \in\ \theta_{1j}$ for each j
   $\theta_1 \equiv \bigwedge[\theta_{11}\ ..\ \theta_{1n}]$.

   Choose $\psi_i \in \theta$. By Sub, $\Gamma \vdash v \in \phi_i$. By the definition of $\mathit{arrowbasis}_\Gamma$ (4.3.1.1), there is some θ_{1j} such that $\phi_i{\rightarrow}\psi_i \in \mathit{arrowbasis}_\Gamma(\theta_{1j})$, i.e. (by Lemma 4.3.1.3) such that $\Gamma \vdash \theta_{1j} \leq \phi_i{\rightarrow}\psi_i$. By Sub, $\Gamma \vdash \{\sigma_j/\alpha\}f\ \in\ \phi_i{\rightarrow}\psi_i$. By For, $\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ \in\ \phi_i{\rightarrow}\psi_i$. Using *for* introduction (5.5.2.7) to get $\Gamma \vdash v = \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ v \in \phi_i$, we then have, by Eq-Refl and Eq-App,

   $\Gamma\ \vdash\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ v\ =\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ v)\ \in\ \psi_i.$

   On the other hand, by Arrow-E, $\Gamma \vdash (\{\sigma_j/\alpha\}f)\ v \in \psi_i$, i.e., $\Gamma \vdash (\{\sigma_j/\alpha\}f\ v) \in \psi_i$. By For, $\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ v \in \psi_i$. So by Eq-For/App,

   $\Gamma\ \vdash\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ v)\ =\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ (f\ v)\ \in\ \psi_i.$

   By Eq-Trans,

   $\Gamma\ \vdash\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ v\ =\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ (f\ v)\ \in\ \psi_i.$

   By Eq-Inter,

   $\Gamma\ \vdash\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ v\ =\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ (f\ v)\ \in\ \theta,$

   and by Eq-Sub,

   $\Gamma\ \vdash\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ v\ =\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ (f\ v)\ \in\ \tau.$

2. By minimal typing (4.3.2.7), there is some θ such that $\Gamma \stackrel{!}{\vdash} (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ [\phi] \in \theta$ and $\Gamma \vdash \theta \leq \tau$. By the syntax-directedness of type synthesis (4.3.2.3),

   $\Gamma \stackrel{!}{\vdash} \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ \in\ \theta_1$
   $\theta \equiv \bigwedge[\{\phi/\beta\}\psi_i \mid (\forall\beta{\leq}\phi_i.\ \psi_i)\ \in\ \mathit{allbasis}_\Gamma(\theta_1)$ and $\Gamma\ \vdash\ \phi\ \leq\ \phi_i]$
   $\Gamma \stackrel{!}{\vdash} \{\sigma_j/\alpha\}f\ \in\ \theta_{1j}$ for each j
   $\theta_1 \equiv \bigwedge[\theta_{11}\ ..\ \theta_{1n}]$.

   Choose $\psi_i' \in \theta$, i.e., $\psi_i' \equiv \{\psi/\beta\}\psi_i$ for some ϕ_i such that $(\forall\beta{\leq}\phi_i.\ \psi_i) \in \mathit{allbasis}_\Gamma(\theta_1)$ and $\Gamma \vdash \phi \leq \phi_i$. Then by the definition of *allbasis* (4.3.1.1), there is some $\theta_{1j} \in \theta_1$ such that $\forall\beta{\leq}\phi_i.\ \psi_i \in \mathit{allbasis}_\Gamma(\theta_{1j})$. By Lemma 4.3.1.3 and Sub, $\Gamma \vdash \{\sigma_j/\alpha\}f \in \forall\beta{\leq}\phi_i.\ \psi_i$. By All-E, $\Gamma \vdash (\{\sigma_j/\alpha\}f)\ [\phi] \in \{\phi/\beta\}\psi_i$, i.e. $\Gamma \vdash (\{\sigma_j/\alpha\}f\ [\phi]) \in \{\phi/\beta\}\psi_i$. By For, $\Gamma \vdash \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ [\phi] \in \{\phi/\beta\}\psi_i$. By Eq-For/TApp,

   $\Gamma\ \vdash\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ [\phi]\ =\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ [\phi]\ \in\ \{\phi/\beta\}\psi_i.$

   By Eq-Inter and Eq-Sub,

   $\Gamma\ \vdash\ \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f\ [\phi]\ =\ (\mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ f)\ [\phi]\ \in\ \tau.$

3. Straightforward.

4. Straightforward. □

5.5.2.9. Remark: The equational theory of $F_{\leq}$ studied by Cardelli, Martini, Mitchell, and Scedrov includes a more general version of the Eq-Tapp rule, intended to capture the notion of *parametricity* [119] in the model:

$$\frac{\Gamma \vdash e = e' \in \forall\alpha{\leq}\sigma.\ \tau \qquad \Gamma \vdash \phi \leq \sigma \qquad \Gamma \vdash \phi' \leq \sigma \qquad \Gamma \vdash \{\phi/\alpha\}\tau \leq \psi \qquad \Gamma \vdash \{\phi'/\alpha\}\tau \leq \psi}{\Gamma \vdash e\ [\phi] = e'\ [\phi'] \in \psi} \qquad \text{(Eq-TApp')}$$

By analogy, it might be interesting to consider an extended Eq-For rule:

$$\frac{\Gamma \vdash \{\sigma/\alpha\}e = \{\sigma'/\alpha\}e' \in \tau}{\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma..\sigma_m.\ e = \mathit{for}\,\alpha\ \mathit{in}\,\sigma'_1..\sigma'..\sigma'_n.\ e' \in \tau} \qquad \text{(Eq-For')}$$

For our present purposes, however, it is enough to study the simpler system with the Eq-Tapp and Eq-For rules presented above.

We may verify that the equational theory of $F_\wedge$ is "well typed" in the sense that equality at a type τ implies membership in τ.

5.5.2.10. Lemma: If $\Gamma \vdash e = e' \in \tau$, then $\Gamma \vdash e \in \tau$ and $\Gamma \vdash e' \in \tau$.

Proof: By induction on the given derivation. In each case, the first conclusion, $\Gamma \vdash e \in \tau$, follows either immediately or by straightforward application of the induction hypothesis. The second conclusion is established as follows:

Case Eq-Beta: $e \equiv (\lambda x{:}\sigma.\ f)\ v \qquad e' \equiv \{v/x\}f \qquad \Gamma \vdash e \in \tau$

By minimal typing (4.3.2.7), there is some type θ such that $\Gamma \stackrel{!}{\vdash} e \in \theta$ and $\Gamma \vdash \theta \leq \tau$. By the syntax-directedness of type synthesis,

$\Gamma, x{:}\sigma \stackrel{!}{\vdash} f \in \theta_1$
$\Gamma \stackrel{!}{\vdash} \lambda x{:}\sigma.\ f \in \sigma{\to}\theta_1$
$\Gamma \stackrel{!}{\vdash} v \in \theta_2$
$\theta \equiv \bigwedge[\psi_i \mid (\phi_i{\to}\psi_i) \in \mathit{arrowbasis}_\Gamma(\sigma{\to}\theta_1) \text{ and } \Gamma \vdash \theta_2 \leq \phi_i]$
$\equiv \begin{cases} \bigwedge[\theta_1] & \text{if } \Gamma \vdash \theta_2 \leq \sigma \\ \top & \text{if } \Gamma \not\vdash \theta_2 \leq \sigma. \end{cases}$

If $\Gamma \vdash \theta_2 \leq \sigma$, then $\Gamma \vdash v \in \sigma$ by rule Sub, and, by Lemma 4.1.10, Sub-Inter-LB, and Sub, $\Gamma \vdash e' \in \theta$. If $\Gamma \not\vdash \theta_2 \leq \sigma$, then $\theta \equiv \top$ and $\Gamma \vdash e' \in \theta$ directly by Sub. In either case, $\Gamma \vdash e' \in \tau$ by Sub.

Case Eq-Beta2: $e \equiv (\Lambda\alpha{\leq}\sigma.\ f)\ [\phi] \qquad e' \equiv \{\phi/\alpha\}f \qquad \Gamma \vdash e \in \tau$

Similar, using Lemma 4.1.11 instead of 4.1.10.

Case Eq-Eta: $e \equiv \lambda x{:}\sigma.\ f\ x \qquad x \notin FV(f) \qquad e' \equiv f \qquad \Gamma \vdash e \in \tau \qquad \Gamma \vdash e' \in \tau$

pImmediate.

Case Eq-Eta2: $e \equiv \Lambda\alpha{\leq}\sigma.\ f\ [\alpha] \qquad \alpha \notin FTV(f) \qquad e' \equiv f \qquad \Gamma \vdash e \in \tau \qquad \Gamma \vdash e' \in \tau$

Immediate.

Case EQ-FOR/VAR: $e \equiv \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ x \qquad e' \equiv x \qquad \Gamma \vdash e \in \tau$

By minimal typing 4.3.2.7, there is some θ such that $\Gamma \stackrel{!}{\vdash} e \in \theta$ and $\Gamma \vdash \theta \leq \tau$. By the syntax-directedness of type synthesis (4.3.2.3),

$\Gamma \stackrel{!}{\vdash} \{\sigma_i/\alpha\}x \in \theta_i$ for each i
$\theta \equiv \bigwedge[\theta_1 .. \theta_n]$.

If $n = 0$, then $\theta \equiv \top$ and $\Gamma \vdash e' \in \theta$ by SUB. Otherwise, by VAR, each $\theta_i \equiv \Gamma(x)$, so $\Gamma \vdash \Gamma(x) \leq \theta$ by SUB-INTER-LB and SUB-TRANS, and $\Gamma \vdash e' \in \theta$ by VAR and SUB. In either case, $\Gamma \vdash e' \in \tau$ by one more application of SUB.

Case EQ-FOR/ABS: $e \equiv \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ \lambda x{:}\sigma.\ b \qquad \Gamma \vdash e \in \tau \qquad \alpha \notin \mathit{FTV}(\sigma)$
$e' \equiv \lambda x{:}\sigma.\ \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ b$

By minimal typing (4.3.2.7), there is some θ such that $\Gamma \stackrel{!}{\vdash} e \in \theta$ and $\Gamma \vdash \theta \leq \tau$. By the syntax-directedness of type synthesis (4.3.2.3),

$\Gamma \stackrel{!}{\vdash} \{\sigma_j/\alpha\}(\lambda x{:}\sigma.\ b) \in \theta_i$
$\theta \equiv \bigwedge[\theta_1 .. \theta_n]$
$\Gamma, x{:}\sigma \stackrel{!}{\vdash} \{\sigma_j/\alpha\}b \in \theta_{i2}$
$\theta_i \equiv \sigma \rightarrow \theta_{i2}$.

By FOR, for each i,

$\Gamma, x{:}\sigma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ b \in \theta_{i2}$.

By ARROW-I,

$\Gamma \vdash \lambda x{:}\sigma.\ \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ b \in \sigma \rightarrow \theta_{i2}$,

i.e., $\Gamma \vdash e' \in \theta_i$. By INTER-I, $\Gamma \vdash e' \in \theta$. By SUB, $\Gamma \vdash e' \in \tau$.

Cases EQ-FOR/APP, ..., EQ-FOR/FOR:

Similar.

Cases EQ-INTER, EQ-REFL, ..., EQ-FOR:

Straightforward. □

5.5.3 Soundness for the Untyped Semantics

It is a simple matter to show that the equational theory is validated by the untyped semantics of $F_\wedge$ in Section 5.1.

5.5.3.1. Theorem: If

$\eta_1 \models \Gamma$
$\eta_2 \models \Gamma$
$\forall \alpha \in \mathit{dom}(\Gamma).\ \eta_1(\alpha) = \eta_2(\alpha) = \eta(\alpha)$
$\forall x \in \mathit{dom}(\Gamma).\ \eta_1(x)\ \{[\![\tau]\!]_\eta\}\ \eta_2(x)$
$\Gamma \vdash e_1 = e_2 \in \tau$,

then

$[\![e_1]\!]_{\eta_1}\ \{[\![\tau]\!]_\eta\}\ [\![e_2]\!]_{\eta_2}$.

Proof: By induction on a derivation of $\Gamma \vdash e_1 = e_2 \in \tau$.

Case EQ-BETA: $e_1 \equiv (\lambda x{:}\sigma.\ f)\ v \qquad e_2 \equiv \{v/x\}f \qquad \Gamma \vdash e_1 \in \tau$

$$\begin{array}{lll}
[\![e_1]\!]_{\eta_1} \ \{[\![\tau]\!]_\eta\} & [\![e_1]\!]_{\eta_2} & \text{by Lemma 5.1.3.8} \\
= & [\![(\lambda x{:}\sigma.\ f)\ v]\!]_{\eta_2} & \\
= & [\![\lambda^\star x.\ |erase(f)|\ |erase(v)|]\!]_{\eta_2} & \\
= & [\![\lambda^\star x.\ |erase(f)|]\!]_{\eta_2} \cdot [\![|erase(v)|]\!]_{\eta_2} & \\
= & [\![|erase(f)|]\!]_{\eta_2[x \leftarrow [\![|erase(v)|]\!]_{\eta_2}]} & \text{by Lemma 5.1.1.9} \\
= & [\![\{|erase(v)|/x\}(|erase(f)|)]\!]_{\eta_2} & \text{by Lemma 5.1.3.9(4)} \\
= & [\![|erase(\{v/x\}f)|]\!]_{\eta_2} & \text{by Lemma 5.1.3.9(5)} \\
= & [\![\{v/x\}f]\!]_{\eta_2}. &
\end{array}$$

Case EQ-BETA2: $e_1 \equiv (\Lambda\alpha{\le}\sigma.\ f)\ [\phi] \qquad e_2 \equiv \{\phi/\alpha\}f \qquad \Gamma \vdash e_1 \in \tau$

$$\begin{array}{lll}
[\![e_1]\!]_{\eta_1} \ \{[\![\tau]\!]_\eta\} & [\![e_1]\!]_{\eta_2} & \text{by Lemma 5.1.3.8} \\
= & [\![f]\!]_{\eta_2} & \text{by definition} \\
= & [\![e_2]\!]_{\eta_2} & \text{by Lemma 5.1.3.9(2).}
\end{array}$$

Case EQ-ETA: $e_1 \equiv \lambda x{:}\sigma.\ f\ x \qquad e_2 \equiv f \qquad \Gamma \vdash e_1 \in \tau \qquad \Gamma \vdash e_2 \in \tau$

$$\begin{array}{lll}
[\![e_1]\!]_{\eta_1} \ \{[\![\tau]\!]_\eta\} & [\![e_1]\!]_{\eta_2} & \text{by Lemma 5.1.3.8} \\
= & [\![\lambda^\star x.\ |erase(f)|\ |erase(x)|]\!]_{\eta_2} & \\
= & [\![K\ |erase(f)|\ (\lambda^\star x.\ x)]\!]_{\eta_2} & \text{since } x \notin FV(f) = FV(|erase(f)|) \\
= & k \cdot [\![|erase(f)|]\!]_{\eta_2} \cdot [\![\lambda^\star x.\ x]\!]_{\eta_2} & \\
= & [\![|erase(f)|]\!]_{\eta_2} & \\
= & [\![f]\!]_{\eta_2}. &
\end{array}$$

Case EQ-ETA2: $e_1 \equiv \Lambda\alpha{\le}\sigma.\ f\ [\alpha] \qquad e_2 \equiv f \qquad \Gamma \vdash e_1 \in \tau \qquad \Gamma \vdash e_2 \in \tau$

As for EQ-BETA2.

Case EQ-INTER: for all i, $\Gamma \vdash e_1 = e_2 \in \tau_i \qquad \tau \equiv \bigwedge[\tau_1..\tau_n]$

By the induction hypothesis, $[\![e_1]\!]_{\eta_1}\ \{[\![\tau_i]\!]_\eta\}\ [\![e_2]\!]_{\eta_2}$ for each i; hence, by the definition of $\bigcap$, $[\![e_1]\!]_{\eta_1}\ \{\bigcap_{1\le i\le n} [\![\tau_i]\!]_\eta\}\ [\![e_2]\!]_{\eta_2}$, i.e., $[\![e_1]\!]_{\eta_1}\ \{[\![\bigwedge[\tau_1..\tau_n]]\!]_\eta\}\ [\![e_2]\!]_{\eta_2}$.

Cases EQ-FOR/VAR.. EQ-FOR/FOR:

Immediate from the definition of *erase* and Lemma 5.1.3.10.

Case EQ-REFL: $\Gamma \vdash e_1 \in \tau$

By Lemma 5.1.3.10.

Cases EQ-SYMM, EQ-TRANS:

By the induction hypothesis and the syummetry and transitivity of PERs.

Case EQ-ABS: $e_1 \equiv \lambda x{:}\sigma.\ e_1' \qquad e_2 \equiv \lambda x{:}\sigma.\ e_2' \qquad \Gamma, x{:}\sigma \vdash e_1' = e_2' \in \sigma'$

Choose m and n such that $m\ \{[\![\sigma]\!]_\eta\}\ n$. By the induction hypothesis,

$$[\![e_1']\!]_{\eta_1[x\leftarrow m]}\ \{[\![\sigma']\!]_\eta\}\ [\![e_2']\!]_{\eta_2[x\leftarrow n]}.$$

By Lemmas 5.1.3.4 and 5.1.1.10

$$[\![\lambda x{:}\sigma.\ e_1']\!]_{\eta_1} \cdot m\ \{[\![\sigma']\!]_\eta\}\ [\![\lambda x{:}\sigma.\ e_2']\!]_{\eta_2} \cdot n,$$

hence, by the definition of $\to$,

$$[\![\lambda x{:}\sigma.\ e_1']\!]_{\eta_1}\ \{[\![\sigma{\to}\sigma']\!]_\eta\}\ [\![\lambda x{:}\sigma.\ e_2']\!]_{\eta_2}.$$

Case EQ-APP: $e_1 \equiv f_1\, v_1 \qquad e_2 \equiv f_2\, v_2 \qquad \Gamma \vdash f_1 = f_2 \in \sigma{\to}\tau \qquad \Gamma \vdash v_1 = v_2 \in \sigma$

By the induction hypothesis,

$$[\![f_1]\!]_{\eta_1}\ \{[\![\sigma{\to}\tau]\!]_\eta\}\ [\![f_2]\!]_{\eta_2},$$

i.e.,

$$[\![f_1]\!]_{\eta_1}\ \{[\![\sigma]\!]_\eta{\to}[\![\tau]\!]_\eta\}\ [\![f_2]\!]_{\eta_2},$$

and

$$[\![v_1]\!]_{\eta_1}\ \{[\![\sigma]\!]_\eta\}\ [\![v_2]\!]_{\eta_2},$$

so by the definition of $\to$,

$$[\![f_1]\!]_{\eta_1} \cdot [\![v_1]\!]_{\eta_1}\ \{[\![\tau]\!]_\eta\}\ [\![f_2]\!]_{\eta_2} \cdot [\![v_2]\!]_{\eta_2},$$

i.e.,

$$[\![f_1\, v_1]\!]_{\eta_1}\ \{[\![\tau]\!]_\eta\}\ [\![f_2\, v_2]\!]_{\eta_2}.$$

Case EQ-TABS: $e_1 \equiv \Lambda\alpha{\le}\sigma.\ e_1' \qquad e_2 \equiv \Lambda\alpha{\le}\sigma.\ e_2' \qquad \Gamma, \alpha{\le}\sigma \vdash e_1' = e_2' \in \sigma' \qquad \tau \equiv \forall\alpha{\le}\sigma.\ \sigma'$

Choose some $A \subseteq [\![\sigma]\!]_\eta$. By the induction hypothesis,

$$[\![e_1']\!]_{\eta_1[\alpha \leftarrow A]}\ \{[\![\sigma']\!]_{\eta[\alpha \leftarrow A]}\}\ [\![e_2']\!]_{\eta_2[\alpha \leftarrow A]}.$$

By Lemma 5.1.1.9,

$$[\![e_1']\!]_{\eta_1}\ \{[\![\sigma']\!]_{\eta[\alpha \leftarrow A]}\}\ [\![e_2']\!]_{\eta_2}.$$

By the definition of $\bigcap$,

$$[\![e_1']\!]_{\eta_1}\ \{\textstyle\bigcap_{A \subseteq [\![\sigma]\!]_\eta} [\![\sigma']\!]_{\eta[\alpha \leftarrow A]}\}\ [\![e_2']\!]_{\eta_2},$$

i.e.

$$[\![e_1']\!]_{\eta_1}\ \{[\![\forall\alpha{\le}\sigma.\ \sigma']\!]_\eta\}\ [\![e_2']\!]_{\eta_2}.$$

Case EQ-TAPP: $e_1 \equiv e_1'\,[\phi] \qquad e_2 \equiv e_2'\,[\phi]$
$\Gamma \vdash e_1' = e_2' \in \forall\alpha{\le}\sigma.\ \sigma' \qquad \Gamma \vdash \phi \le \sigma \qquad \tau \equiv \{\phi/\alpha\}\sigma'$

By the induction hypothesis,

$$[\![e_1']\!]_{\eta_1}\ \{[\![\forall\alpha{\le}\sigma.\ \sigma']\!]_\eta\}\ [\![e_2']\!]_{\eta_2},$$

i.e.,

$$[\![e_1']\!]_{\eta_1}\ \{\textstyle\bigcap_{A \subseteq [\![\sigma]\!]_\eta} [\![\sigma']\!]_{\eta[\alpha \leftarrow A]}\}\ [\![e_2']\!]_{\eta_2}.$$

By Lemma 5.1.3.8 and the fact that $(\bigcap_{i \in I} A_i) \subseteq A_i$ for each i,

$$[\![e_1']\!]_{\eta_1}\ \{[\![\sigma']\!]_{\eta[\alpha \leftarrow [\![\phi]\!]_\eta]}\}\ [\![e_2']\!]_{\eta_2},$$

hence (by Lemma 5.1.3.9(3)),

$$[\![e_1']\!]_{\eta_1}\ \{[\![\{\phi/\alpha\}\sigma']\!]_\eta\}\ [\![e_2']\!]_{\eta_2},$$

i.e.,

$$[\![e_1]\!]_{\eta_1}\ \{[\![\{\phi/\alpha\}\sigma']\!]_\eta\}\ [\![e_2]\!]_{\eta_2}.$$

Case EQ-FOR: $e_1 \equiv \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e_1' \qquad e_2 \equiv \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e_2'$
$\Gamma \vdash \{\sigma_i/\alpha\}e_1' = \{\sigma_i/\alpha\}e_2' \in \tau_i \qquad \tau \equiv \tau_i$

By the induction hypothesis,

$$[\![\{\sigma_i/\alpha\}e_1']\!]_{\eta_1}\ \{[\![\tau_i]\!]_\eta\}\ [\![\{\sigma_i/\alpha\}e_2']\!]_{\eta_2},$$

i.e. (by Lemma 5.1.3.9(3)),

$$[\![e_1']\!]_{\eta_1}\ \{[\![\tau_i]\!]_\eta\}\ [\![e_2']\!]_{\eta_2},$$

i.e. (by the definition of *erase*),

$$[\![e_1]\!]_{\eta_1}\ \{[\![\tau_i]\!]_\eta\}\ [\![e_2]\!]_{\eta_2}.$$

□

5.5.4 Soundness for the Translation Semantics

5.5.4.1. Remark: The soundness of our equational theory for the translation-style semantics given in Section 5.3 depends in many places on the coherence of the translation. Although we lack a proof of this property, it is instructive to write out the proof of soundness modulo coherence.

5.5.4.2. Lemma:

1. If $\Gamma \vDash^{\wedge} v \in \phi$ and $\Gamma, x{:}\phi \vDash^{\wedge} f \in \psi$, then
$$[\![\Gamma \vDash^{\wedge} \{v/x\}f \in \psi]\!] = \{[\![\Gamma \vDash^{\wedge} v \in \phi]\!]/x\}\,[\![\Gamma, x{:}\phi \vDash^{\wedge} f \in \psi]\!].$$
2. If $\Gamma, \alpha{\leq}\sigma \vdash f \in \psi$ and $\Gamma \vdash \phi \leq \sigma$, then
$$[\![\Gamma \vdash \{\phi/\alpha\}f \in \{\phi/\alpha\}\psi]\!] = \{[\![\Gamma \vdash \phi \leq \sigma]\!]/c_\alpha\}\{[\![\phi]\!]/\alpha\}[\![\Gamma, \alpha{\leq}\sigma \vdash f \in \psi]\!].$$

Proof:

1. By induction on the structure of a derivation of $\Gamma, x{:}\phi \vDash^{\wedge} f \in \psi$.
2. Similar. □

5.5.4.3. Theorem: If $\Gamma \vDash^{\wedge} e = e' \in \tau$, then $[\![\Gamma]\!] \vDash^{F} [\![\Gamma \vDash^{\wedge} e \in \tau]\!] = [\![\Gamma \vDash^{\wedge} e' \in \tau]\!] \in [\![\tau]\!]$.

Proof: By induction on the given derivation.

Case Eq-Beta: $e \equiv (\lambda x{:}\sigma.\ f)\ v \qquad e' \equiv \{v/x\}f \qquad \Gamma \vDash^{\wedge} e \in \tau$

By minimal typing (4.3.2.7), there is a type θ such that $\Gamma \vdash^{\downarrow} e \in \theta$ and $\Gamma \vdash \theta \leq \tau$. By the syntax-directedness of type synthesis (4.3.2.3),

$\Gamma \vdash^{\downarrow} \lambda x{:}\sigma.\ f \in \sigma{\to}\theta_2$
$\Gamma, x{:}\sigma \vdash^{\downarrow} f \in \theta_2$
$\Gamma \vdash v \in \theta_1$
$\theta \equiv \bigwedge[\psi_i \mid (\phi_i{\to}\psi_i) \in \mathit{arrowbasis}_\Gamma(\sigma{\to}\theta_2) \text{ and } \Gamma \vdash \theta_1 \leq \phi_i]$.

Subcase: $\Gamma \vdash \theta_1 \leq \sigma$

Then $\theta \equiv \bigwedge[\theta_2]$ and
$$[\![\Gamma \vdash^{\downarrow} e \in \theta]\!] = \langle([\![\Gamma \vdash \sigma{\to}\theta_2 \leq \sigma{\to}\theta_2]\!] \cdot [\![\Gamma \vdash^{\downarrow} (\lambda x{:}\sigma.\ f) \in \sigma{\to}\theta_2]\!]) \cdot ([\![\Gamma \vdash \theta_1 \leq \sigma]\!] \cdot [\![\Gamma \vdash v \in \theta_1]\!])\rangle.$$

By Sub-Refl, T-Sub-Refl, and FEq-Beta,
$$[\![\Gamma]\!] \vDash^{F} [\![\Gamma \vdash^{\downarrow} e \in \theta]\!] = \langle([\![\Gamma \vdash^{\downarrow} (\lambda x{:}\sigma.\ f) \in \sigma{\to}\theta_2]\!]) \cdot ([\![\Gamma \vdash \theta_1 \leq \sigma]\!] \cdot [\![\Gamma \vdash v \in \theta_1]\!])\rangle \in [\![\textstyle\bigwedge[\theta_2]]\!],$$

i.e. (by T-Sub and TA-Arrow-I),
$$[\![\Gamma]\!] \vDash^{F} [\![\Gamma \vdash^{\downarrow} e \in \theta]\!] = \langle(\lambda x{:}[\![\sigma]\!].\ [\![\Gamma, x{:}\sigma \vdash^{\downarrow} f \in \theta_2]\!]) \cdot ([\![\Gamma \vdash v \in \sigma]\!])\rangle \in [\![\textstyle\bigwedge[\theta_2]]\!].$$

By FEq-Beta, Lemma 5.5.4.2, and FEq-Tuple,
$$[\![\Gamma]\!] \vDash^{F} \langle(\lambda x{:}[\![\sigma]\!].\ [\![\Gamma, x{:}\sigma \vdash f \in \theta_2]\!]) \cdot [\![\Gamma \vdash v \in \sigma]\!]\rangle = \langle\{[\![\Gamma \vdash v \in \sigma]\!]/x\}\,[\![\Gamma, x{:}\sigma \vDash^{\wedge} f \in \theta_2]\!]\rangle \in [\![\textstyle\bigwedge[\theta_2]]\!].$$

By FEq-Refl and FEq-App,
$$[\![\Gamma]\!] \vDash^{F} [\![\Gamma \vdash \textstyle\bigwedge[\theta_2] \leq \tau]\!] \cdot \langle(\lambda x{:}[\![\sigma]\!].\ [\![\Gamma, x{:}\sigma \vdash f \in \theta_2]\!]) \cdot [\![\Gamma \vdash v \in \sigma]\!]\rangle = [\![\Gamma \vdash \textstyle\bigwedge[\theta_2] \leq \tau]\!] \cdot \langle\{[\![\Gamma \vdash v \in \sigma]\!]/x\}\,[\![\Gamma, x{:}\sigma \vDash^{\wedge} f \in \theta_2]\!]\rangle \in [\![\tau]\!],$$

i.e.,
$$[\![\Gamma]\!] \vDash^{F} [\![\Gamma \vdash e \in \tau]\!] = [\![\Gamma \vdash e' \in \tau]\!] \in [\![\tau]\!].$$

Subcase: $\Gamma \not\vdash \theta_1 \leq \sigma$

Then $\theta \equiv \top$ and $[\![\Gamma \vdash e \in \theta]\!] = \langle\rangle$. By INTER-I, $\Gamma \vdash \{v/x\}f \in \top$. By T-INTER-I, $[\![\Gamma \vdash \{v/x\}f \in \top]\!] = \langle\rangle$. By FEQ-TUPLE and FEQ-TRANS,

$$[\![\Gamma]\!] \models^F [\![\Gamma \vdash e \in \top]\!] = [\![\Gamma \vdash e' \in \top]\!] \in [\![\top]\!].$$

By FEQ-REFL and FEQ-APP,

$$[\![\Gamma]\!] \models^F [\![\Gamma \vdash \top \leq \tau]\!] \cdot [\![\Gamma \vdash e \in \top]\!] = [\![\Gamma \vdash \top \leq \tau]\!] \cdot [\![\Gamma \vdash e' \in \top]\!] \in [\![\tau]\!].$$

Case EQ-BETA2: $e \equiv (\Lambda\alpha{\leq}\sigma.\ f)\ [\phi]$ $\quad e' \equiv \{\phi/\alpha\}f$ $\quad \Gamma \vdash e \in \tau$

Similar.

Case EQ-ETA: $e \equiv \lambda x{:}\sigma.\ f\ x$ $\quad \Gamma \vdash e \in \tau$ $\quad x \notin FV(f)$ $\quad e' \equiv f$

By minimal typing (4.3.2.7), there is some θ such that $\Gamma \stackrel{\perp}{\vdash} e \in \theta$ and $\Gamma \vdash \theta \leq \tau$. By the syntax-directedness of type synthesis,

$\Gamma, x{:}\sigma \vdash f\ x \in \theta_2$
$\theta \equiv \sigma{\rightarrow}\theta_2$,

and again,

$\Gamma, x{:}\sigma \stackrel{\perp}{\vdash} f \in \zeta$
$\Gamma, x{:}\sigma \stackrel{\perp}{\vdash} x \in \sigma$
$\theta_2 \equiv \bigwedge[\psi_i \mid (\phi_i{\rightarrow}\psi_i) \in \mathit{arrowbasis}_{(\Gamma,\, x:\sigma)}(\zeta)$ and $\Gamma, x{:}\sigma \vdash \sigma \leq \phi_i]$.

By Lemma 4.3.1.3, D-ALL-SOME, SUB-ARROW, and SUB-DIST-IA,

$$\begin{aligned}\Gamma \vdash \zeta &\leq \bigwedge[\mathit{arrowbasis}_{(\Gamma,\, x:\sigma)}(\zeta)] \\ &\leq \bigwedge[\phi_i{\rightarrow}\psi_i \mid (\phi_i{\rightarrow}\psi_i) \in \mathit{arrowbasis}_{(\Gamma,\, x:\sigma)}(\zeta) \text{ and } \Gamma, x{:}\sigma \vdash \sigma \leq \phi_i] \\ &\equiv \bigwedge[\phi_i{\rightarrow}\psi_i \mid (\phi_i{\rightarrow}\psi_i) \in \mathit{arrowbasis}_{(\Gamma,\, x:\sigma)}(\zeta)] \\ &\leq \bigwedge[\sigma{\rightarrow}\psi_i \mid (\phi_i{\rightarrow}\psi_i) \in \mathit{arrowbasis}_{(\Gamma,\, x:\sigma)}(\zeta) \text{ and } \Gamma, x{:}\sigma \vdash \sigma \leq \phi_i] \\ &\leq \sigma{\rightarrow}\theta_2.\end{aligned}$$

By SUB, $\Gamma, x{:}\sigma \vdash f \in \sigma{\rightarrow}\theta_2$. By strengthening (4.1.9), $\Gamma \vdash f \in \sigma{\rightarrow}\theta_2$. By FEQ-ETA,

$$[\![\Gamma]\!] \models^F \lambda x{:}\sigma.\ [\![\Gamma, x{:}\sigma \vdash f \in \sigma{\rightarrow}\theta_2]\!]\ x = [\![\Gamma \vdash f \in \sigma{\rightarrow}\theta_2]\!] \in [\![\sigma]\!]{\rightarrow}[\![\theta_2]\!],$$

i.e.,

$$\begin{aligned}[\![\Gamma]\!] \models^F \lambda x{:}\sigma.\ &[\![\Gamma, x{:}\sigma \vdash f \in \sigma{\rightarrow}\theta_2]\!] \cdot [\![\Gamma, x{:}\sigma \vdash x \in \sigma]\!] \\ &= [\![\Gamma \vdash f \in \sigma{\rightarrow}\theta_2]\!] \\ &\in [\![\sigma]\!]{\rightarrow}[\![\theta_2]\!],\end{aligned}$$

i.e.,

$$[\![\Gamma]\!] \models^F [\![\Gamma \vdash \lambda x{:}\sigma.\ f\ x \in \sigma{\rightarrow}\theta_2]\!] = [\![\Gamma \vdash f \in \sigma{\rightarrow}\theta_2]\!] \in [\![\sigma{\rightarrow}\theta_2]\!].$$

By FEQ-REFL and FEQ-APP,

$$\begin{aligned}[\![\Gamma]\!] &\models^F [\![\Gamma \vdash \sigma{\rightarrow}\theta_2 \leq \tau]\!] \cdot [\![\Gamma \vdash \lambda x{:}\sigma.\ f\ x \in \sigma{\rightarrow}\theta_2]\!] \\ &= [\![\Gamma \vdash \sigma{\rightarrow}\theta_2 \leq \tau]\!] \cdot [\![\Gamma \vdash f \in \sigma{\rightarrow}\theta_2]\!] \\ &\in [\![\tau]\!],\end{aligned}$$

which, by T-SUB, is the desired result.

Case EQ-ETA2: $e \equiv \Lambda\alpha{\leq}\sigma.\ f\ [\alpha]$ $\quad \Gamma \vdash e \in \tau$ $\quad \alpha \notin FTV(f)$ $\quad e' \equiv f$

Similar.

Case EQ-FOR/VAR: $e \equiv \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ x$ $\quad e' \equiv x$
$\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ x \in \tau$

By minimal typing (4.3.2.7), there is some θ such that $\Gamma \stackrel{\perp}{\vdash} e \in \theta$ and $\Gamma \vdash \theta \leq \tau$. Since $\{\sigma_i/\alpha\}x \equiv x$, we have $\theta \equiv \bigwedge[\Gamma(x)\ ..\ \Gamma(x)]$ by the syntax-directedness of type synthesis (4.3.2.3).

Subcase: $n = 0$

Then $\theta \equiv \top$. By FEQ-TUPLE,

$$\begin{aligned}[\![\Gamma]\!] \vDash^F & [\![\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ x \in \top]\!] \\ &= [\![\Gamma \vdash x \in \top]\!] \\ &\in [\![\top]\!].\end{aligned}$$

By FEQ-APP,

$$\begin{aligned}[\![\Gamma]\!] \vDash^F & [\![\Gamma \vdash \top \leq \tau]\!] \cdot [\![\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ x \in \top]\!] \\ &= [\![\Gamma \vdash \top \leq \tau]\!] \cdot [\![\Gamma \vdash x \in \top]\!] \\ &\in [\![\tau]\!],\end{aligned}$$

which, by T-SUB, is the desired result.

Subcase: $n > 0$

By TA-FOR and TA-VAR,

$$\begin{aligned} & [\![\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ x \in \textstyle\bigwedge[\Gamma(x)\,..\,\Gamma(x)]]\!] \\ &= \langle [\![\Gamma \vdash \{\sigma_1/\alpha\}x \in \Gamma(x)]\!]\,..\,[\![\Gamma \vdash \{\sigma_n/\alpha\}x \in \Gamma(x)]\!]\rangle \\ &= \langle [\![\Gamma \vdash x \in \Gamma(x)]\!]\,..\,[\![\Gamma \vdash x \in \Gamma(x)]\!]\rangle \\ &= \langle x\,..\,x\rangle.\end{aligned}$$

By FEQ-REFL and coherence,

$$\begin{aligned}[\![\Gamma]\!] \vDash^F & [\![\Gamma \vdash \theta \leq \tau]\!] \cdot \langle x\,..\,x\rangle \\ &= (\mathit{proj}_1\,; [\![\Gamma \vdash \Gamma(x) \leq \tau]\!]) \cdot \langle x\,..\,x\rangle \\ &\in [\![\tau]\!].\end{aligned}$$

By FEQ-PI,

$$\begin{aligned}[\![\Gamma]\!] \vDash^F & (\mathit{proj}_1\,; [\![\Gamma \vdash \Gamma(x) \leq \tau]\!]) \cdot \langle x\,..\,x\rangle \\ &= [\![\Gamma \vdash \Gamma(x) \leq \tau]\!] \cdot x \\ &\in [\![\tau]\!],\end{aligned}$$

from which the desired result follows by transitivity.

Case EQ-FOR/ABS: $e \equiv \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ \lambda x{:}\sigma.\ b \qquad e' \equiv \lambda x{:}\sigma.\ \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ b$
$\alpha \notin FTV(\sigma) \qquad \Gamma \vdash e \in \tau$

By minimal typing (4.3.2.7), there is some θ such that $\Gamma \vdash^{\cdot} e \in \theta$ and $\Gamma \vdash \theta \leq \tau$. By the syntax-directedness of type synthesis (4.3.2.3),

$$\Gamma \vdash^{\cdot} e \in \textstyle\bigwedge[\sigma{\to}\tau_1\,..\,\sigma{\to}\tau_n],$$

where, for each i,

$$\Gamma, x{:}\{\sigma_i/\alpha\}\sigma \vdash \{\sigma_i/\alpha\}b \in \tau_i,$$

i.e.,

$$\Gamma, x{:}\sigma \vdash \{\sigma_i/\alpha\}b \in \tau_i.$$

By A-ARROW-I and A-FOR,

$$\Gamma \vdash^{\cdot} e' \in \sigma{\to}\textstyle\bigwedge[\tau_1\,..\,\tau_n].$$

Let

$$\begin{aligned} t &\stackrel{\text{def}}{=} \langle \lambda x{:}[\![\sigma]\!].\ [\![\Gamma, x{:}\sigma \vdash \{\sigma_1/\alpha\}b \in \tau_1]\!]\,..\,\lambda x{:}[\![\sigma]\!].\ [\![\Gamma, x{:}\sigma \vdash \{\sigma_n/\alpha\}b \in \tau_n]\!]\rangle \\ t' &\stackrel{\text{def}}{=} \lambda x{:}[\![\sigma]\!].\ \langle [\![\Gamma, x{:}\sigma \vdash \{\sigma_1/\alpha\}b \in \tau_1]\!]\,..\,[\![\Gamma, x{:}\sigma \vdash \{\sigma_n/\alpha\}b \in \tau_n]\!]\rangle.\end{aligned}$$

Then by TA-FOR and TA-ARROW-I,

$$\begin{aligned} t &= [\![\Gamma \vdash^{\cdot} e \in \textstyle\bigwedge[\sigma{\to}\tau_1\,..\,\sigma{\to}\tau_n]]\!] \\ t' &= [\![\Gamma \vdash^{\cdot} e' \in \sigma \to \textstyle\bigwedge[\tau_1..\tau_n]]\!].\end{aligned}$$

By FEq-Beta and FEq-Pi (n times),

$$\begin{aligned}
&[\![\Gamma]\!] \models^F dist_{\bigwedge[\sigma\to\tau_1\,..\,\sigma\to\tau_n]} \cdot t \\
&\quad = \lambda x{:}[\![\sigma]\!].\ \langle (proj_1\ t)\ x\ ..\ (proj_n\ t)\ x\rangle \\
&\quad = t' \\
&\quad \in [\![\sigma\to\textstyle\bigwedge[\tau_1..\tau_n]]\!].
\end{aligned}$$

By FEq-App and FEq-Refl,

$$\begin{aligned}
&[\![\Gamma]\!] \models^F [\![\Gamma \vdash \sigma\to\textstyle\bigwedge[\tau_1..\tau_n] \le \tau]\!] \cdot (dist_{\bigwedge[\sigma\to\tau_1\,..\,\sigma\to\tau_n]}\ t) \\
&\quad = [\![\Gamma \vdash \sigma\to\textstyle\bigwedge[\tau_1..\tau_n] \le \tau]\!] \cdot t' \\
&\quad \in [\![\tau]\!].
\end{aligned}$$

By T-Sub,

$$\begin{aligned}
&[\![\Gamma]\!] \models^F [\![\Gamma \vdash \textstyle\bigwedge[\sigma\to\tau_1\,..\,\sigma\to\tau_n] \le \tau]\!] \cdot t \\
&\quad = [\![\Gamma \vdash \sigma\to\textstyle\bigwedge[\tau_1..\tau_n] \le \tau]\!] \cdot t' \\
&\quad \in [\![\tau]\!],
\end{aligned}$$

as required.

Case Eq-For/App: $e \equiv for\ \alpha\ in\ \sigma_1..\sigma_n.\ b_1\ b_2$
$e' \equiv (for\ \alpha\ in\ \sigma_1..\sigma_n.\ b_1)\ (for\ \alpha\ in\ \sigma_1..\sigma_n.\ b_2)$
$\Gamma \vdash e \in \tau$

By minimal typing (4.3.2.7), there is some θ such that $\Gamma \stackrel{\perp}{\vdash} e \in \theta$ and $\Gamma \vdash \theta \le \tau$. For each i, we have (by the syntax-directedness of type synthesis (4.3.2.3)

$$\begin{aligned}
&\Gamma \stackrel{\perp}{\vdash} \{\sigma_i/\alpha\}b_1 \in \theta_{i1} \\
&\Gamma \stackrel{\perp}{\vdash} \{\sigma_i/\alpha\}b_2 \in \theta_{i2} \\
&\Gamma \stackrel{\perp}{\vdash} (\{\sigma_i/\alpha\}b_1)\,(\{\sigma_i/\alpha\}b_2) \\
&\quad \in \textstyle\bigwedge[\psi_{ij} \mid (\phi_{ij}\to\psi_{ij}) \in arrowbasis_\Gamma(\theta_{i1}) \text{ and } \Gamma \vdash \theta_{i2} \le \psi_{ij}].
\end{aligned}$$

Let

$$\theta_i \stackrel{\text{def}}{=} \textstyle\bigwedge[\psi_{ij} \mid (\phi_{ij}\to\psi_{ij}) \in arrowbasis_\Gamma(\theta_{i1}) \text{ and } \Gamma \vdash \theta_{i2} \le \psi_{ij}].$$

Then $\theta \equiv \bigwedge[\theta_1\ ..\ \theta_n]$. By T-For (twice), FEq-App, and T-For

$$\begin{aligned}
&[\![\Gamma]\!] \models^F [\![\Gamma \vdash e' \in \theta_i]\!] \\
&\quad = [\![\Gamma \vdash (\{\sigma_i/\alpha\}b_1)\,(\{\sigma_i/\alpha\}b_2) \in \theta_i]\!] \\
&\quad = [\![\Gamma \vdash e \in \theta_i]\!] \\
&\quad \in [\![\theta_i]\!].
\end{aligned}$$

By FEq-Tuple,

$$\begin{aligned}
&[\![\Gamma]\!] \models^F \langle [\![\Gamma \vdash e' \in \theta_1]\!]\ ..\ [\![\Gamma \vdash e' \in \theta_n]\!]\rangle \\
&\quad = \langle [\![\Gamma \vdash e \in \theta_1]\!]\ ..\ [\![\Gamma \vdash e \in \theta_n]\!]\rangle \\
&\quad \in \Pi[[\![\theta_1]\!]\ ..\ [\![\theta_n]\!]],
\end{aligned}$$

i.e.,

$$[\![\Gamma]\!] \models^F [\![\Gamma \vdash e' \in \theta]\!] = [\![\Gamma \vdash e \in \theta]\!] \in [\![\theta]\!].$$

By FEq-App and T-Sub,

$$[\![\Gamma]\!] \models^F [\![\Gamma \vdash e' \in \tau]\!] = [\![\Gamma \vdash e \in \tau]\!] \in [\![\tau]\!].$$

Cases Eq-For/TAbs, EQ-For/TApp, EQ-For/For:

Similar.

Case EQ-INTER: $\tau \equiv \bigwedge[\tau_1..\tau_n]$ for all i, $\Gamma \vdash e = e' \in \tau_i$

By the induction hypothesis,

$$[\![\Gamma]\!] \models^F [\![\Gamma \vdash e \in \tau_i]\!] = [\![\Gamma \vdash e' \in \tau_i]\!] \in [\![\tau_i]\!]$$

for each i. By coherence,

$$[\![\Gamma]\!] \models^F proj_i \cdot [\![\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]]\!] = proj_i \cdot [\![\Gamma \vdash e' \in \bigwedge[\tau_1..\tau_n]]\!] \in [\![\tau_i]\!]$$

By FEQ-TUPLE,

$$\begin{aligned}[\![\Gamma]\!] &\models^F \langle proj_1 \cdot [\![\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]]\!] .. proj_n \cdot [\![\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]]\!]\rangle \\ &= \langle proj_1 \cdot [\![\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]]\!] .. proj_n \cdot [\![\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]]\!]\rangle \in \Pi[[\![\tau_1]\!] .. [\![\tau_n]\!]].\end{aligned}$$

By FEQ-SURJ (twice) and FEQ-TRANS,

$$[\![\Gamma]\!] \models^F [\![\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]]\!] = [\![\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]]\!] \in [\![\bigwedge[\tau_1..\tau_n]]\!].$$

Cases EQ-REFL, EQ-SYMM, EQ-TRANS, EQ-ABS, EQ-APP:

By straightforward application of the induction hypothesis.

Case EQ-TABS: $e \equiv \Lambda\alpha{\le}\phi.\ b$ $e \equiv \Lambda\alpha{\le}\phi.\ b$ $\tau \equiv \forall\alpha{\le}\phi.\ \psi$
$\Gamma, \alpha{\le}\phi \models^{\wedge} b = b' \in \psi$

By the induction hypothesis,

$$[\![\Gamma, \alpha{\le}\phi]\!] \models^F [\![\Gamma, \alpha{\le}\phi \vdash b \in \psi]\!] = [\![\Gamma, \alpha{\le}\phi \vdash b' \in \psi]\!] \in [\![\psi]\!],$$

i.e.,

$$[\![\Gamma]\!], \alpha, c_\alpha{:}(\alpha{\to}[\![\phi]\!]) \models^F [\![\Gamma, \alpha{\le}\phi \vdash b \in \psi]\!] = [\![\Gamma, \alpha{\le}\phi \vdash b' \in \psi]\!] \in [\![\psi]\!].$$

By FEQ-ABS and FEQ-TABS,

$$\begin{aligned}[\![\Gamma]\!] &\models^F \Lambda\alpha.\ \lambda c_\alpha{:}(\alpha{\to}[\![\phi]\!]).\ [\![\Gamma, \alpha{\le}\phi \vdash b \in \psi]\!] \\ &= \Lambda\alpha.\ \lambda c_\alpha{:}(\alpha{\to}[\![\phi]\!]).\ [\![\Gamma, \alpha{\le}\phi \vdash b \in \psi]\!] \\ &\in \forall\alpha.\ (\alpha{\to}[\![\phi]\!]){\to}\psi,\end{aligned}$$

i.e.,

$$[\![\Gamma]\!] \models^F [\![\Gamma \vdash \Lambda\alpha{\le}\phi.\ b \in \forall\alpha{\le}\phi.\ \psi]\!] = [\![\Gamma \vdash \Lambda\alpha{\le}\phi.\ b \in \forall\alpha{\le}\phi.\ \psi]\!] \in [\![\forall\alpha{\le}\phi.\ \psi]\!].$$

Case EQ-TAPP: $e \equiv b\,[\phi]$ $e' \equiv b'\,[\phi]$ $\tau \equiv \{\phi/\alpha\}\psi$
$\Gamma \models^{\wedge} b = b' \in \forall\alpha{\le}\theta.\ \psi$ $\Gamma \vdash \phi \le \theta$

By the induction hypothesis,

$$\begin{aligned}[\![\Gamma]\!] &\models^F [\![\Gamma \vdash b \in \forall\alpha{\le}\theta.\ \psi]\!] \\ &= [\![\Gamma \vdash b' \in \forall\alpha{\le}\theta.\ \psi]\!] \\ &\in [\![\forall\alpha{\le}\theta.\ \psi]\!],\end{aligned}$$

i.e.,

$$\begin{aligned}[\![\Gamma]\!] &\models^F [\![\Gamma \vdash b \in \forall\alpha{\le}\theta.\ \psi]\!] \\ &= [\![\Gamma \vdash b' \in \forall\alpha{\le}\theta.\ \psi]\!] \\ &\in \forall\alpha.\ (\alpha{\to}[\![\theta]\!]){\to}[\![\psi]\!].\end{aligned}$$

By Lemma 5.3.2.6, $[\![\Gamma]\!] \models^F [\![\Gamma \vdash \phi \le \theta]\!] \in [\![\phi]\!]{\to}[\![\theta]\!]$. So by FEQ-TAPP, Lemma 5.3.2.2, and FEQ-APP,

$$\begin{aligned}[\![\Gamma]\!] &\models^F [\![\Gamma \vdash b \in \forall\alpha{\le}\theta.\ \psi]\!] \cdot [[\![\phi]\!]] \cdot [\![\Gamma \vdash \phi \le \theta]\!] \\ &= [\![\Gamma \vdash b' \in \forall\alpha{\le}\theta.\ \psi]\!] \cdot [[\![\phi]\!]] \cdot [\![\Gamma \vdash \phi \le \theta]\!] \\ &\in [\![\{\phi/\alpha\}\psi]\!],\end{aligned}$$

i.e.,

$$\begin{aligned}[\![\Gamma]\!] &\models^F [\![\Gamma \vdash b\,[\phi] \in \{\psi/\alpha\}\psi]\!] \\ &= [\![\Gamma \vdash b'\,[\phi] \in \{\psi/\alpha\}\psi]\!] \\ &\in [\![\{\phi/\alpha\}\psi]\!].\end{aligned}$$

Case Eq-For: $e \equiv \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ b \qquad e' \equiv \mathit{for}\ \alpha\ \mathit{in}\ \sigma_1..\sigma_n.\ b' \qquad \tau \equiv \tau_i$
$\Gamma \vdash \{\sigma_i/\alpha\}b = \{\sigma_i/\alpha\}b' \in \tau_i$

By the induction hypothesis and T-For. □

Chapter 6

Undecidability of Subtyping

In this chapter, we show that the typing relation of $F_{\leq}$ (and, as an easy corollary, that of $F_{\wedge}$) is undecidable. The crux of the difficulty lies in the subtype relation, specifically in the subtyping rule for quantified types:

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma, \alpha{\leq}\tau_1 \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \forall\alpha{\leq}\sigma_1.\ \sigma_2 \leq \forall\alpha{\leq}\tau_1.\ \tau_2} \qquad \text{(Sub-All)}$$

Though semantically appealing, this rule creates serious problems for reasoning about the subtype relation. In a quantified type $\forall\alpha{\leq}\sigma_1.\ \sigma_2$, instances of α in σ_2 are naturally thought of as being bounded by their lexically declared bound σ_1. But this connection is destroyed by the second premise: when $\forall\alpha{\leq}\sigma_1.\ \sigma_2$ is compared to $\forall\alpha{\leq}\tau_1.\ \tau_2$, instances of α in *both* σ_2 and τ_2 are bounded by τ_1 in the premise $\Gamma, \alpha{\leq}\tau_1 \vdash \sigma_2 \leq \tau_2$.

Cardelli and Wegner's original definition of Fun [33] used a weaker quantifier rule in which $\forall\alpha{\leq}\sigma_1.\ \sigma_2$ is a subtype of $\forall\alpha{\leq}\tau_1.\ \tau_2$ only if σ_1 and τ_1 are identical; this variant can easily be shown to be decidable. Later authors, including Cardelli, have chosen to work with the more powerful formulation considered in this thesis.

Curien and Ghelli used a proof-normalization argument to show that $F_{\leq}$ typechecking is coherent (that is, that all derivations of a statement $\Gamma \vdash e \in \tau$ have the same meaning under certain assumptions about the semantics). One corollary of their proof is the soundness and completeness of a natural syntax-directed procedure for computing minimal typings of $F_{\leq}$ terms, with a subroutine for checking the subtype relation (algorithm $F^N_{\leq}$ of Section 2.6); the same procedure had been developed by the group at Penn and by Cardelli for use in his Quest typechecker [Gunter, personal communication, 1990]. The termination of Curien and Ghelli's typechecking procedure is equivalent to the termination of the subtyping algorithm. Ghelli, in his Ph.D. thesis [63], gave a proof of termination; unfortunately, this proof was later found to contain a subtle mistake. In fact, Ghelli soon realized that there are inputs for which the subtyping algorithm does *not* terminate [personal communication, 1991]. Worse yet, these cases are not amenable to any simple form of cycle detection: when presented with one of them, the algorithm generates an infinite sequence of recursive calls with larger and larger contexts. This discovery reopened the question of the decidability of $F_{\leq}$.

The undecidability result presented here began as an attempt to formulate a more refined algorithm capable of detecting the kinds of divergence that could be induced in the simpler one. A series of partial results about decidable subsystems eventually led to the discovery of a class of input problems for which increasing the size the input by a constant factor would increase the search depth of a *succeeding* execution of the algorithm by an exponential factor. Besides dispelling

previous intuitions about why the problem ought to be decidable, this construction suggested a trick for encoding natural numbers, from which it was a short step to an encoding of two-counter Turing machines.

After reviewing the flaw in Ghelli's earlier proof of termination for the subtyping algorithm $F^N_{\leq}$ (Section 6.1) and presenting an example where the algorithm fails to terminate (Section 6.2), we identify a fragment of $F_{\leq}$ that forms a convenient target for the reductions to follow (Sections 6.3 and 6.4). The main result is then presented in two steps.

1. We first define an intermediate abstraction, called *rowing machines* (Section 6.5); these bridge the gap between $F_{\leq}$ subtyping problems and two-counter machines by retaining the notions of bound variables and substitution from $F_{\leq}$ while introducing a computational abstraction with a finite collection of registers and an evaluation regime based on state transformation.

   An encoding of rowing machines as $F_{\leq}$ subtyping statements is given and proven correct, in the sense that a rowing machine R halts iff its translation $\mathcal{F}(R)$ is a derivable statement in $F_{\leq}$ (Section 6.6).
2. We then review the definition of two-counter machines (Section 6.7) and show how a two-counter machine T may be encoded as a rowing machine $\mathcal{R}(T)$ such that T halts iff $\mathcal{R}(T)$ does (Section 6.8).

Section 6.9 shows that the undecidability of subtyping implies the undecidability of typechecking for $F_{\leq}$; Sections 6.10 and 6.11 extend the result to $F_{\wedge}$ and some related systems. Section 6.12 discusses its pragmatic import.

6.1 A Flawed Decidability Argument for $F_{\leq}$

Ghelli's Ph.D. thesis [63, pp. 80–83] argues that the algorithm $F^N_{\leq}$ always terminates and is therefore a decision procedure for $F_{\leq}$ typechecking. This section briefly sketches Ghelli's argument and shows where it goes wrong. The problem is quite subtle: the incorrect proof was read by a number of people (including the present author) before the flaw was detected, independently, by Curien and Reynolds.

The idea, as usual, is to define a well-founded complexity metric and show that if J' is a subproblem of J, then *complexity*(J') is strictly less than *complexity*(J).

6.1.1. Definition: The function $index_\Gamma(\alpha)$ gives the index in Γ (counting from right to left) of the binding of α.

6.1.2. Definition: The *left depth* of a type variable α in a type τ and a context Γ is the number of bound type variables in both τ and Γ at α's point of definition. To formalize this concept, it is convenient to assume that all binding occurrences of type variables in τ and Γ have been renamed so as to be distinct from each other (or better yet, that deBruijn indices are used instead of variable names). Now define:

$$ld(\alpha, \tau, \Gamma) = 1 + \begin{cases} len(\Gamma) + ld(\alpha, \tau) & \text{if } \alpha \text{ is bound in } \tau \\ index_\Gamma(\alpha) & \text{otherwise} \end{cases}$$

$ld(\alpha, \tau) =$ the number of instances of $\forall$ in whose scope the binding occurrence of α (in τ) falls.

(As Ghelli observes, "The definition is simpler in terms of DeBruijn indices, as the left-depth of any variable is simply the difference among the indexes of that variable and the 'outermost variable',

taken in any environment [i.e. context] where they are both defined, plus one; the "outermost variable" is the first variable bound in the environment, or the first one bound in the term if the environment is empty.")
Define the left depth of a type σ in a context Γ to be the maximum left depth of any type variable in σ:

$$ld(\sigma, \Gamma) = max(\{0\} \cup \{ld(\alpha, \sigma, \Gamma) \mid \alpha \in TV(\sigma)\}).$$

Define the complexity of a subtyping statement $(\Gamma \vdash \sigma \leq \tau)$ to be the following pair:

$$complexity(\Gamma \vdash \sigma \leq \tau) = (ld(\sigma, \Gamma) + ld(\tau, \Gamma), size(\sigma) + size(\tau)).$$

Order the range of $complexity(\Gamma \vdash \sigma \leq \tau)$ lexicographically. Note that this ordering is well founded (contains no infinite descending chains).

This ordering operates as desired for all the rules of $F^N_{\leq}$ with the exception of NVAR. For an example of its misbehavior in this case, let

$$\Gamma \equiv \alpha{\leq}(\forall\beta{\leq}\mathit{Top}.\ \mathit{Top}), \alpha'{\leq}\mathit{Top}.$$

Then

$$\begin{aligned} ld(\alpha, \Gamma) &= ld(\alpha, \alpha, \Gamma) \\ &= 1 + index_{\Gamma}(\alpha) \\ &= 2, \end{aligned}$$

whereas

$$\begin{aligned} ld((\forall\gamma{\leq}\mathit{Top}.\ \mathit{Top}), \Gamma) &= ld(\gamma, (\forall\gamma{\leq}\mathit{Top}.\ \mathit{Top}), \Gamma) \\ &= 1 + len(\Gamma) + ld(\gamma, (\forall\gamma{\leq}\mathit{Top}.\ \mathit{Top})) \\ &= 3 + ld(\gamma, (\forall\gamma{\leq}\mathit{Top}.\ \mathit{Top})) \\ &= 3. \end{aligned}$$

So the instance

$$\frac{\Gamma \vdash \forall\gamma{\leq}\mathit{Top}.\ \mathit{Top} \leq \mathit{Top}}{\Gamma \vdash \alpha \leq \mathit{Top}}$$

of NVAR has a premise of greater complexity than its conclusion.

6.2 Nontermination of the $F_{\leq}$ Subtyping Algorithm

Ghelli recently dispelled the widely held belief that the algorithm $F^N_{\leq}$ terminates on all inputs, by discovering the following example.

6.2.1. Definition: In this example (and below), an additional abbreviation is used:

$$\neg\tau \stackrel{\text{def}}{=} \forall\alpha{\leq}\tau.\ \alpha$$

The salient property of this notation is that it allows the right- and left-hand sides of subtyping statements to be swapped:

6.2.2. Fact: $\Gamma \vdash \neg\sigma \leq \neg\tau$ is derivable iff $\Gamma \vdash \tau \leq \sigma$ is.

6.2.3. Example: Let $\theta \equiv \forall\alpha.\ \neg(\forall\beta{\leq}\alpha.\ \neg\beta)$. Then executing the algorithm $F^N_{\leq}$ on the input problem $\alpha_0{\leq}\theta \vdash \alpha_0 \leq (\forall\alpha_1{\leq}\alpha_0.\ \neg\alpha_1)$ leads to the following infinite sequence of recursive calls:

$$
\begin{array}{llll}
\alpha_0{\leq}\theta & \vdash\ \alpha_0 & \leq & \forall\alpha_1{\leq}\alpha_0.\ \neg\alpha_1 \\
\alpha_0{\leq}\theta & \vdash\ \forall\alpha_1.\ \neg(\forall\alpha_2{\leq}\alpha_1.\ \neg\alpha_2) & \leq & \forall\alpha_1{\leq}\alpha_0.\ \neg\alpha_1 \\
\alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0 & \vdash\ \neg(\forall\alpha_2{\leq}\alpha_1.\ \neg\alpha_2) & \leq & \neg\alpha_1 \\
\alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0 & \vdash\ \alpha_1 & \leq & \forall\alpha_2{\leq}\alpha_1.\ \neg\alpha_2 \\
\alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0 & \vdash\ \alpha_0 & \leq & \forall\alpha_2{\leq}\alpha_1.\ \neg\alpha_2 \\
\alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0 & \vdash\ \forall\alpha_2.\ \neg(\forall\alpha_3{\leq}\alpha_2.\ \neg\alpha_3) & \leq & \forall\alpha_2{\leq}\alpha_1.\ \neg\alpha_2 \\
\alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0,\ \alpha_2{\leq}\alpha_1 & \vdash\ \neg(\forall\alpha_3{\leq}\alpha_2.\ \neg\alpha_3) & \leq & \neg\alpha_2 \\
\alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0,\ \alpha_2{\leq}\alpha_1 & \vdash\ \alpha_2 & \leq & \forall\alpha_3{\leq}\alpha_2.\ \neg\alpha_3 \\
\alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0,\ \alpha_2{\leq}\alpha_1 & \vdash\ \alpha_1 & \leq & \forall\alpha_3{\leq}\alpha_2.\ \neg\alpha_3 \\
\alpha_0{\leq}\theta,\ \alpha_1{\leq}\alpha_0,\ \alpha_2{\leq}\alpha_1 & \vdash\ \alpha_0 & \leq & \forall\alpha_3{\leq}\alpha_2.\ \neg\alpha_3 \\
\text{etc.} & & &
\end{array}
$$

(The α-conversion steps necessary to maintain the well-formedness of the context when new variables are added are performed tacitly here, choosing new names so as to clarify the pattern of regress.)

6.2.4. Remark: This example is apparently the smallest subtyping statement that causes the $F^N_{\leq}$ algorithm to diverge [Ghelli, personal communication, 1991].

6.3 A Deterministic Fragment of $F_{\leq}$

The pattern of recursion in Ghelli's example is an instance of a more general scheme — one so general, in fact, that it can be used to encode termination problems for two-counter Turing machines. We now turn to demonstrating this fact.

6.3.1. Definition: The *positive* and *negative occurrences* in a statement $\Gamma \vdash \sigma \leq \tau$ are defined as follows:

- The type σ and the bounds in Γ are negative occurrences; τ is a positive occurrence.
- If $\tau_1{\to}\tau_2$ is a positive (respectively, negative) occurrence, then τ_1 is a negative (positive) occurrence and τ_2 is a positive (negative) occurrence.
- If $\forall\alpha{\leq}\tau_1.\ \tau_2$ is a positive (negative) occurrence, then τ_1 is a negative (positive) occurrence and τ_2 is a positive (negative) occurrence.

6.3.2. Fact: The rules defining $F^N_{\leq}$ (2.6.10) preserve the signs of occurrences: wherever a metavariable τ appears in a premise of one of the rules, it has the same sign as the corresponding occurrence of τ in the conclusion.

In what follows, it will be convenient to work with a fragment of $F^N_{\leq}$ with somewhat simpler behavior:

- we drop the $\to$ type constructor and its subtyping rule;
- we introduce a negation operator explicitly into the syntax and include a rule for comparing negated expressions;
- we drop the left-hand premise from the rule for comparing quantifiers, requiring instead that when two quantified types are compared, the bound of the one on the left must be *Top*;
- we consider only statements where no variable occurs positively, allowing us to drop the NREFL rule; and

- we disallow *Top* in negative positions.

Since the $F^N_{\leq}$ rules preserve positive and negative occurrences, we may redefine the set of types so that positive and negative types are separate syntactic categories. At the same time, we simplify each category appropriately.

6.3.3. Definition: The sets of *positive types* τ^+ and *negative types* τ^- are defined by the following abstract grammar:

$$\begin{array}{lcl} \tau^+ & ::= & \mathit{Top} \mid \neg\tau^- \mid \forall\alpha{\leq}\tau^-.\ \tau^+ \\ \tau^- & ::= & \alpha \mid \neg\tau^+ \mid \forall\alpha.\ \tau^- \end{array}$$

A *negative context* Γ^- is one whose bounds are all negative types.

6.3.4. Definition: $F^P_{\leq}$ (P for polarized) is the least relation closed under the following rules:

$$\Gamma^- \vdash \tau^- \leq \mathit{Top} \qquad (\textsc{PTop})$$

$$\frac{\Gamma^- \vdash \Gamma^-(\alpha) \leq \tau^+}{\Gamma^- \vdash \alpha \leq \tau^+} \qquad (\textsc{PVar})$$

$$\frac{\Gamma^-, \alpha{\leq}\phi^- \vdash \sigma^- \leq \tau^+}{\Gamma^- \vdash \forall\alpha.\ \sigma^- \leq \forall\alpha{\leq}\phi^-.\ \tau^+} \qquad (\textsc{PAll})$$

$$\frac{\Gamma^- \vdash \tau^- \leq \sigma^+}{\Gamma^- \vdash \neg\sigma^+ \leq \neg\tau^-} \qquad (\textsc{PNeg})$$

$F^P_{\leq}$ is almost the system we need, but it still lacks one important property: $F_{\leq}$ is not a conservative extension of $F^P_{\leq}$. For example, the non-derivable $F^P_{\leq}$ statement

$$\vdash \neg\,\mathit{Top} \leq \forall\alpha.\ \alpha$$

corresponds, under the abbreviations for $\neg$ and $\forall\alpha.\ \alpha$, to the derivable $F_{\leq}$ statement

$$\vdash \forall\alpha{\leq}\mathit{Top}.\ \alpha \leq \forall\alpha{\leq}\mathit{Top}.\ \alpha.$$

To achieve conservativity, we restrict the form of $F^P_{\leq}$ statements even further so that negated types can never be compared with quantified types.

6.3.5. Definition: Let n be a fixed nonnegative number. The sets of *n-positive* and *n-negative types* are defined by the following abstract grammar:

$$\begin{array}{lcl} \tau^+ & ::= & \mathit{Top} \mid \forall\alpha_0{\leq}\tau_0^-\ ..\ \alpha_n{\leq}\tau_n^-.\ \neg\tau^- \\ \tau^- & ::= & \alpha \mid \forall\alpha_0\ ..\ \alpha_n.\ \neg\tau^+ \end{array}$$

We stipulate, moreover, that an n-positive type $\forall\alpha_0{\leq}\tau_0^-\ ..\ \alpha_n{\leq}\tau_n^-.\ \neg\tau^-$ is closed only if no α_i appears free in any τ_i.

An *n-negative context* Γ_n^- is one whose bounds are all n-negative types.

6.3.6. Definition: An $F^D_{\leq}$ *statement* has the form $\Gamma_n^- \vdash \sigma_n^- \leq \tau_n^+$, where Γ_n^- is an n-negative context, σ_n^- is an n-negative type, and τ_n^+ is an n-positive type.

6.3.7. Convention: To reduce clutter, we drop the superscripts $+$ and $-$ and usually leave n implicit in what follows.

6.3.8. Definition: $F^D_{\leq}$ (D for deterministic) is the least relation closed under the following rules:

$$\Gamma \vdash \tau \leq \mathit{Top} \qquad (\textsc{DTop})$$

$$\frac{\Gamma \vdash \Gamma(\alpha) \leq \forall \alpha_0 \leq \phi_0 .. \alpha_n \leq \phi_n .\ \neg\tau}{\Gamma \vdash \alpha \leq \forall \alpha_0 \leq \phi_0 .. \alpha_n \leq \phi_n .\ \neg\tau} \quad \text{(DVar)}$$

$$\frac{\Gamma, \alpha_0 \leq \phi_0 .. \alpha_n \leq \phi_n \vdash \tau \leq \sigma}{\Gamma \vdash \forall \alpha_0 .. \alpha_n .\ \neg\sigma \leq \forall \alpha_0 \leq \phi_0 .. \alpha_n \leq \phi_n .\ \neg\tau} \quad \text{(DAllNeg)}$$

Using the earlier abbreviations for negation, multiple quantification, and unbounded quantification, we may read every $F^D_{\leq}$ statement as an $F^N_{\leq}$ statement. Under this interpretation, the two subtype relations coincide for statements in their common domain:

6.3.9. Lemma: $F^N_{\leq}$ is a conservative extension of $F^D_{\leq}$: if J is an $F^D_{\leq}$ statement, then J is derivable in $F^D_{\leq}$ iff it is derivable in $F^N_{\leq}$.

Proof: The implication $F^D_{\leq} \Rightarrow F^N_{\leq}$ is straightforward. The other direction, $F^N_{\leq} \Rightarrow F^D_{\leq}$ proceeds by induction on $F^N_{\leq}$ derivations, using the form of $F^D_{\leq}$ statements to constrain the possible forms of $F^N_{\leq}$ derivations whose conclusions are de-abbreviated $F^D_{\leq}$ statements. □

These simplifications justify a useful change of perspective. Since the only rule in $F^N_{\leq}$ with two premises has been replaced by a rule with one premise, derivations in this fragment are linear (each node has at most one subderivation). Moreover, every metavariable in the premise of each rule also appears in the conclusion, which makes the step from conclusion to premise deterministic. The syntax-directed construction of such a derivation may be thus be viewed as a deterministic state transformation process, where the subtyping statement being verified is the current state and the single premise that must be recursively verified, if any, is the next state. In other words, a subtyping statement is thought of as an instantaneous description of a kind of automaton.

From now on we use terminology that makes the intuition of "subtyping as state transformation" more explicit. Analogous terminology and notation will be used to describe the other calculi introduced below.

6.3.10. Definition: The *one-step elaboration* function $\mathcal{E}$ for $F^D_{\leq}$-statements is the partial mapping defined by:

$$\mathcal{E}(J) = \begin{cases} J' & \text{if } J \text{ is the conclusion of an instance of DVar or DAllNeg} \\ & \text{and } J' \text{ is the corresponding premise} \\ \text{undefined} & \text{if } J \text{ is an instance of DTop.} \end{cases}$$

6.3.11. Definition: J' is an *immediate subproblem* of J in $F^D_{\leq}$, written $J \longrightarrow_D J'$, if $J' = \mathcal{E}(J)$.

6.3.12. Definition: J' is a *subproblem* of J in $F^D_{\leq}$, written $J \xrightarrow{\star}_D J'$, if either $J \equiv J'$ or $J \longrightarrow_D J_1$ and $J_1 \xrightarrow{\star}_D J'$.

6.3.13. Definition: The *elaboration* of a statement J is the sequence of subproblems encountered by the subtyping algorithm given J as input.

6.4 Eager Substitution

To make a smooth transition between the subtyping statements of $F_{\leq}$ and the rowing machine abstraction to be introduced in Section 6.5, we need one more variation in the definition of subtyping, where, instead of maintaining a context with the bounds of free variables, the quantifier rule immediately substitutes the bounds into the body of the statement.

6.4.1. Definition: The simultaneous, capture-avoiding substitution of ϕ_0 through ϕ_n, respectively, for α_0 through α_n in τ, is written $\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\tau$.

6.4.2. Definition: An $F^F_{\leq}$ *statment* is an $F^D_{\leq}$ statement with empty context.

6.4.3. Definition: $F^F_{\leq}$ (F for flattened) is the least relation closed under the following rules:

$$\vdash \tau \leq \mathit{Top} \qquad \text{(FTop)}$$

$$\frac{\vdash \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\tau \leq \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\sigma}{\vdash \forall\alpha_0 .. \alpha_n.\ \neg\sigma \leq \forall\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n.\ \neg\tau} \qquad \text{(FAllNeg)}$$

6.4.4. Remark: Of course, an analogous reformulation of full $F_{\leq}$ would not be correct. For example, in the non-derivable statement

$$\vdash (\forall\alpha{\leq}\mathit{Top}.\ \mathit{Top}) \leq (\forall\alpha{\leq}\mathit{Top}.\ \alpha)$$

substituting *Top* for α in the bodies of the quantifiers yields the derivable statement $\vdash \mathit{Top} \leq \mathit{Top}$. But having restricted our attention to statements where variables appear only negatively, we are guaranteed that the only position where the elaboration of a statement can cause a variable to appear by itself in the body of a subproblem is on the left-hand side, where it will immediately be replaced by its bound. We are therefore safe in making the substitution eagerly.

In the remainder of this section, we show that $F^D_{\leq}$ is a conservative extension of $F^F_{\leq}$.

6.4.5. Lemma: Let $\phi_0 .. \phi_n$ be n-negative types and assume that the $F^D_{\leq}$ statement $\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma \vdash \tau \leq \sigma$ is closed. Then if $\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\Gamma \vdash \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\tau \leq \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\sigma$ is derivable in $F^D_{\leq}$, so is $\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma \vdash \tau \leq \sigma$.

Proof: By induction on the size of the given derivation. (Observe that by the stipulation in 6.3.5 that no α_i appears in any ϕ_i, the ϕ_i must all be closed.)

Case DTop: $\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\sigma \equiv \mathit{Top}$

Since variables can only occur negatively, σ cannot be a variable, so $\sigma \equiv \mathit{Top}$ and the result is immediate.

Case DVar: $\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\tau \equiv \beta$

We may assume that $\tau \not\equiv \alpha_i$ for any of the distinguished α_i's, since otherwise we would have $\phi_i \equiv \beta$ and the statement $\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma \vdash \tau \leq \sigma$ would not be closed. So τ must itself be β. By assumption, we have a subderivation

$$\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\Gamma \vdash (\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\Gamma)(\beta) \leq \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\sigma,$$

that is,

$$\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\Gamma \vdash \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,(\Gamma(\beta)) \leq \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\sigma.$$

By the induction hypothesis,

$$\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma \vdash \Gamma(\beta) \leq \sigma.$$

By DVar,

$$\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma \vdash \beta \leq \sigma.$$

Case DAllNeg: $\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\tau \equiv \forall\beta_0 .. \beta_n.\ \neg\tau'_2$
$\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\sigma \equiv \forall\beta_0{\leq}\psi'_0 .. \beta_n{\leq}\psi'_n.\ \neg\sigma'_2$

Since σ cannot be a variable (else $\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma \vdash \tau \leq \sigma$ would not be an $F^D_{\leq}$ statement), we have

$$\begin{aligned} \sigma &\equiv \forall\beta_0{\leq}\psi_0 .. \beta_n{\leq}\psi_n.\ \neg\sigma_2 \\ \psi'_i &\equiv \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\psi_i \\ \sigma'_2 &\equiv \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\sigma_2. \end{aligned}$$

For τ, there are two cases to consider:

Subcase: $\tau \equiv \alpha_i$

Then

$$\phi_i \equiv \forall\beta_0 .. \beta_n.\ \neg\tau_2'.$$

By assumption, there is a subderivation

$$\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\Gamma,\ \beta_0{\leq}\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\psi_0 .. \beta_n{\leq}\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\psi_n \ \vdash\ \sigma_2' \ \leq\ \tau_2',$$

i.e. (since we stipulated $\alpha_j \notin FTV(\phi_i)$, so $\alpha_j \notin FTV(\tau_i')$ for any j),

$$\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\Gamma,\ \beta_0{\leq}\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\psi_0 .. \beta_n{\leq}\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\psi_n \ \vdash\ \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\sigma_2 \ \leq\ \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\tau_2',$$

By the induction hypothesis,

$$\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma, \beta_0{\leq}\psi_0 .. \beta_n{\leq}\psi_n \ \vdash\ \sigma_2 \ \leq\ \tau_2'.$$

By DAllNeg,

$$\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma \ \vdash\ \forall\beta_0 .. \beta_n.\ \neg\tau_2' \ \leq\ \forall\beta_0{\leq}\psi_0 .. \beta_n{\leq}\psi_n.\ \neg\sigma_2.$$

By DVar,

$$\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma \ \vdash\ \alpha_i \ \leq\ \forall\beta_0{\leq}\psi_0 .. \beta_n{\leq}\psi_n.\ \neg\sigma_2$$

Subcase: $\tau \not\equiv \alpha_i$

Then

$$\tau \ \equiv \forall\beta_0 .. \beta_n.\ \neg\tau_2$$
$$\tau_2' \equiv \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\tau_2.$$

By assumption, we again have a subderivation

$$\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\Gamma,\ \beta_0{\leq}\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\psi_0 .. \beta_n{\leq}\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\psi_n \ \vdash\ \sigma_2' \ \leq\ \tau_2',$$

that is,

$$\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\Gamma,\ \beta_0{\leq}\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\psi_0 .. \beta_n{\leq}\{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\psi_n \ \vdash\ \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\sigma_2 \ \leq\ \{\phi_0/\alpha_0 .. \phi_n/\alpha_n\}\,\tau_2.$$

By the induction hypothesis,

$$\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma, \beta_0{\leq}\psi_0 .. \beta_n{\leq}\psi_n \ \vdash\ \sigma_2 \ \leq\ \tau_2.$$

By DAllNeg,

$$\alpha_0{\leq}\phi_0 .. \alpha_n{\leq}\phi_n, \Gamma \vdash \forall\beta_0 .. \beta_n.\ \neg\tau_2 \leq \forall\beta_0 .. \beta_n.\ \neg\sigma_2.$$ □

6.4.6. Lemma: If $\vdash \sigma \leq \tau$ is derivable in $F^F_{\leq}$, then it is derivable in $F^D_{\leq}$.

Proof: By induction on the original derivation, using Lemma 6.4.5 for the FAllNeg case. □

6.4.7. Lemma: If $\alpha{\leq}\phi, \Gamma \vdash \sigma \leq \tau$ is derivable in $F^D_{\leq}$, then $\{\phi/\alpha\}\Gamma \vdash \{\phi/\alpha\}\sigma \leq \{\phi/\alpha\}\tau$ has an $F^D_{\leq}$-derivation of equal or lesser size.

Proof: By induction on the given derivation. □

6.4.8. Lemma: If $\vdash \sigma \leq \tau$ is an $F^F_{\leq}$-statement and is derivable in $F^D_{\leq}$, then it is derivable in $F^F_{\leq}$.

Proof: By induction on the size of the original derivation, using Lemma 6.4.7 for the DAllNeg case. □

6.4.9. Lemma: $F^D_{\leq}$ is a conservative extension of $F^F_{\leq}$.

Proof: By Lemmas 6.4.6 and 6.4.8. □

6.5 Rowing Machines

The reduction from two-counter Turing machines to $F_{\leq}$ subtyping statements is easiest to understand in terms of an intermediate abstraction called a rowing machine, which makes more stylized use of bound variables.

A rowing machine is a tuple of *registers*

$$\langle \rho_1 \,..\, \rho_n \rangle,$$

where the contents of each register is a *row*. By convention, the first register is the machine's *program counter* (PC). To move to the next state, the PC is used as a template to construct the new contents of each of the registers from the current contents of all of the registers (including the PC).

6.5.1. Definition: The set of *rows* (of width n) is defined by the following abstract grammar:

$$\begin{array}{lcll} \rho & ::= & \alpha_m & 1 \leq m \leq n \\ & | & [\alpha_1..\alpha_n]\langle \rho_1..\rho_n \rangle & \\ & | & \text{HALT} & \end{array}$$

The variables $\alpha_1..\alpha_n$ on the left of $[\alpha_1..\alpha_n]\langle \rho_1..\rho_n \rangle$ are binding occurrences whose scope is the rows ρ_1 through ρ_n. We regard rows that differ only in the names of bound variables as identical.

6.5.2. Definition: A *rowing machine* (of width n) is a tuple $\langle\!\langle \rho_1..\rho_n \rangle\!\rangle$, where each ρ_i is a row of width n with no free variables.

6.5.3. Definition: The *one-step elaboration* function $\mathcal{E}$ for rowing machines of width n is the partial mapping

$$\mathcal{E}(\langle\!\langle \rho_1..\rho_n \rangle\!\rangle) = \begin{cases} \langle \{\rho_1/\alpha_1 \,..\, \rho_n/\alpha_n\}\, \rho_{11} \,..\, \{\rho_1/\alpha_1 \,..\, \rho_n/\alpha_n\}\, \rho_{1n} \rangle & \text{if } \rho_1 = [\alpha_1..\alpha_n]\langle \rho_{11}..\rho_{1n} \rangle \\ \text{undefined} & \text{if } \rho_1 = \text{HALT}. \end{cases}$$

(Since rowing machines consist only of closed rows, we need not define the evaluation function for the case where the PC is a variable. Also, since all the ρ_n are closed, the substitution is trivially capture-avoiding.)

6.5.4. Notational conventions:

1. When the symbol "—" appears as the ith component of a compound row $[\alpha_1..\alpha_n]\langle \rho_1..\rho_n \rangle$, it stands for the variable α_i.
2. To avoid a proliferation of variable names in the examples and definitions below, we sometimes use numerical indices (like deBruijn indices [56]) rather than names for variables: the "variable" $\#n$ refers to the n^{th} bound variable of the row in which it appears; $\#\#n$ refers to the n^{th} bound variable of the row enclosing the one in which it appears; and so on.
3. When these abbreviations are used, the binding lists $[\alpha_1..\alpha_n]$ are omitted.

For example, the nested row $[\alpha_1..\alpha_3]\langle \alpha_1, [\beta_1..\beta_3]\langle \alpha_1, \beta_1, \beta_3 \rangle, \alpha_1 \rangle$ would be abbreviated as $\langle —, \langle \#\#1, \#1, — \rangle, \#1 \rangle$.

4. It is convenient to introduce names for closed rows and use these to build up descriptions of other rows. For example, the compound row

   $$\langle \langle \langle \#1, \#1, \#1 \rangle, \#3, \#2 \rangle, \langle —, —, — \rangle, \langle \#1, \#1, \#1 \rangle \rangle$$

   might be written as

   $$\langle \text{Z}, \text{Y}, \text{X} \rangle,$$

where

$$\begin{array}{lcl} \mathrm{X} & \equiv & \langle \#1, \#1, \#1 \rangle \\ \mathrm{Y} & \equiv & \langle —, —, — \rangle \\ \mathrm{Z} & \equiv & \langle \mathrm{X}, \#3, \#2 \rangle. \end{array}$$

6.5.5. Definition: A rowing machine R *halts* if there is a machine R' such that $R \longrightarrow_R^\star R'$ and the PC of R' is the instruction HALT.

6.5.6. Example: The simplest rowing machine, $\langle\!\langle \textsc{halt} \rangle\!\rangle$, halts immediately. The next simplest, $\langle\!\langle \langle \textsc{halt} \rangle \rangle\!\rangle$, takes one step and then halts. Another simple one, $\langle\!\langle \langle — \rangle \rangle\!\rangle$, leads to an infinite elaboration with every state identical to the first.

6.5.7. Example: The machine

$$\langle\!\langle \textsc{loop}, \textsc{a}, \textsc{b} \rangle\!\rangle,$$

where

$$\begin{array}{lcl} \textsc{loop} & \equiv & \langle —, \#3, \#2 \rangle \\ \textsc{a} & \equiv & \text{an arbitrary row} \\ \textsc{b} & \equiv & \text{an arbitrary row} \end{array}$$

executes an infinite loop where the contents of the second and third register are exchanged at successive steps:

$$\begin{array}{ll} & \langle\!\langle \textsc{loop}, \textsc{a}, \textsc{b} \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \textsc{loop}, \textsc{b}, \textsc{a} \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \textsc{loop}, \textsc{a}, \textsc{b} \rangle\!\rangle \\ \longrightarrow_R & \ldots \end{array}$$

6.5.8. Example: The row

$$\textsc{bri} \equiv \langle \#2, — \rangle$$

encodes an *indirect branch* to the contents of register 2 at the moment when BRI is executed. The machine

$$\langle\!\langle \textsc{bri}, \langle \textsc{bri}, \langle \textsc{bri}, \textsc{halt} \rangle \rangle \rangle\!\rangle$$

elaborates as follows:

$$\begin{array}{ll} & \langle\!\langle \textsc{bri}, \langle \textsc{bri}, \langle \textsc{bri}, \textsc{halt} \rangle \rangle \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \langle \textsc{bri}, \langle \textsc{bri}, \textsc{halt} \rangle \rangle, \langle \textsc{bri}, \langle \textsc{bri}, \textsc{halt} \rangle \rangle \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \textsc{bri}, \langle \textsc{bri}, \textsc{halt} \rangle \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \langle \textsc{bri}, \textsc{halt} \rangle, \langle \textsc{bri}, \textsc{halt} \rangle \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \textsc{bri}, \textsc{halt} \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \textsc{halt}, \textsc{halt} \rangle\!\rangle. \end{array}$$

6.6 Encoding Rowing Machines as Subtyping Problems

We now show how a rowing machine R can be encoded as a subtyping problem $\mathcal{F}(R)$ such that R halts iff $\mathcal{F}(R)$ is derivable in $F^F_{\leq}$.

The idea of the translation is that a rowing machine $R = \langle\!\langle \rho_1..\rho_n \rangle\!\rangle$ becomes a subtyping statement $\mathcal{F}(\rho)$ of the form

$$\vdash \ldots \leq (\ldots \mathcal{F}(\rho_1) \ldots),$$

(where we use $\mathcal{F}$ to denote the translation of both rowing machines and rows), constructed so that

- if $\rho_1 = \text{HALT}$, then the elaboration of $\mathcal{F}(R)$ halts (by reaching a subproblem where *Top* appears on the right-hand side);
- if $\rho_1 = [\alpha_1..\alpha_n]\langle \rho_{11}..\rho_{1n} \rangle$, then the elaboration of $\mathcal{F}(R)$ reaches a subproblem that encodes the rowing machine $\mathcal{E}(\langle\!\langle \rho_1..\rho_n \rangle\!\rangle) = \langle\!\langle \{\rho_1/\alpha_1 \,..\, \rho_n/\alpha_n\}\, \rho_{11} \,..\, \{\rho_1/\alpha_1 \,..\, \rho_n/\alpha_n\}\, \rho_{1n} \rangle\!\rangle$.

In more detail, if $R = \langle\!\langle [\alpha_1..\alpha_n]\langle \rho_{11}..\rho_{1n} \rangle..\rho_n \rangle\!\rangle$, then $\mathcal{F}(R)$ is essentially the following:

$$\begin{array}{lll} \vdash & \forall\gamma_1..\gamma_n. & \neg(\forall\gamma'_1{\leq}\gamma_1 \,..\, \gamma'_n{\leq}\gamma_n.\ \neg \ldots) \\ \leq & \forall\gamma_1{\leq}\mathcal{F}(\rho_1) \,..\, \gamma_n{\leq}\mathcal{F}(\rho_n).\ \neg(\forall\alpha_1..\alpha_n. & \neg(\forall\alpha'_1{\leq}\mathcal{F}(\rho_{11}) \,..\, \alpha'_n{\leq}\mathcal{F}(\rho_{1n}).\ \neg\mathcal{F}(\rho_{11}))). \end{array}$$

The elaboration of this statement proceeds as follows:

1. The current contents of the registers $\rho_1..\rho_n$ are temporarily saved by matching the quantifiers on the right with the ones on the left; this has the effect of substituting the bounds $\mathcal{F}(\rho_1) \,..\, \mathcal{F}(\rho_n)$ for free occurrences of the variables $\gamma_1 \,..\, \gamma_n$ on the left-hand side.

   The right- and left-hand sides are also swapped (by the $\neg$ constructor on both sides), so that what now appears on the left is a sequence of variable bindings for the free variables $\alpha_1..\alpha_n$ of ρ_1:

   $$\begin{array}{ll} \vdash & \forall\alpha_1..\alpha_n.\ \neg(\forall\alpha'_1{\leq}\mathcal{F}(\rho_{11}) \,..\, \alpha'_n{\leq}\mathcal{F}(\rho_{1n}).\ \neg\mathcal{F}(\rho_{11})) \\ \leq & \forall\gamma'_1{\leq}\mathcal{F}(\rho_1) \,..\, \gamma'_n{\leq}\mathcal{F}(\rho_n).\ \neg \ldots \end{array}$$

2. The saved contents of the original registers now appear on the right-hand side. When these are matched with the quantifiers on the left, the result is that the old values of the registers are substituted for the variables $\alpha_1..\alpha_n$ in the body

   $$\neg(\forall\alpha'_1{\leq}\mathcal{F}(\rho_{11}) \,..\, \alpha'_n{\leq}\mathcal{F}(\rho_{1n}).\ \neg\mathcal{F}(\rho_{11}))$$

   of the left-hand side.

   Swapping right- and left-hand sides again yields a statement of the same form as the original, where the appropriate instances of $\mathcal{F}(\rho_{11}) \,..\, \mathcal{F}(\rho_{1n})$ appear as the bounds of the outer quantifiers on the right:

   $$\begin{array}{l} \vdash \ldots \leq (\forall\alpha'_1{\leq}\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\, \mathcal{F}(\rho_{11}) \,.. \\ \qquad\qquad \alpha'_n{\leq}\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\, \mathcal{F}(\rho_{1n}). \\ \qquad\qquad\qquad \neg\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\, \mathcal{F}(\rho_{11}) \end{array}$$

   i.e.,

   $$\begin{array}{l} \vdash \ldots \leq (\forall\gamma_1{\leq}\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\, \mathcal{F}(\rho_{11}) \,.. \\ \qquad\qquad \gamma_n{\leq}\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\, \mathcal{F}(\rho_{1n}). \\ \qquad\qquad\qquad \neg\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\, \mathcal{F}(\rho_{11}). \end{array}$$

To be able to get back to a statement of the same form as the original, one piece of additional mechanism is required: besides the n variables used to store the old state of the registers, a variable γ_0 is used to hold the original value of the entire left-hand side of $\mathcal{F}(R)$. This variable is used at the

end of a cycle to set up the left hand side of the statement encoding the next state of the rowing machine.

The formal definition of the translation is as follows.

6.6.1. Definition: Let ρ be a row of width n. The $F^F_{\leq}$-*translation* of ρ, written $\mathcal{F}(\rho)$, is the n-negative type

$$\mathcal{F}(\rho) = \begin{cases} \alpha_i & \text{if } \rho = \alpha_i \\ \forall\gamma_0, \alpha_1 .. \alpha_n.\ \neg\Big(\forall\gamma_0'{\leq}\gamma_0, \alpha_1'{\leq}\mathcal{F}(\rho_1) .. \alpha_n'{\leq}\mathcal{F}(\rho_n).\ \neg\mathcal{F}(\rho_1)\Big) & \text{if } \rho = [\alpha_1..\alpha_n]\langle\rho_1..\rho_n\rangle \\ \forall\gamma_0, \alpha_1 .. \alpha_n.\ \neg \mathit{Top} & \text{if } \rho = \textsc{halt}, \end{cases}$$

where γ_0, γ_0', and α_1' through α_n' are fresh variables.

The proofs below rely on two simple observations:

6.6.2. Fact:

1. The free variables of ρ coindide with the free type variables of $\mathcal{F}(\rho)$.
2. $\mathcal{F}(\{\rho_1/\alpha_1 .. \rho_n/\alpha_n\}\,\rho) = \{\mathcal{F}(\rho_1)/\alpha_1 .. \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho)$.

6.6.3. Definition: Let $R = \langle\!\langle\rho_1..\rho_n\rangle\!\rangle$ be a rowing machine. The $F^F_{\leq}$-*translation* of R, written $\mathcal{F}(R)$, is the $F^F_{\leq}$ statement

$$\vdash\ \sigma\ \leq\ \forall\gamma_0{\leq}\sigma, \gamma_1{\leq}\mathcal{F}(\rho_1) .. \gamma_n{\leq}\mathcal{F}(\rho_n).\ \neg\mathcal{F}(\rho_1),$$

where $\sigma \equiv \forall\gamma_0, \gamma_1..\gamma_n.\ \neg(\forall\gamma_0'{\leq}\gamma_0, \gamma_1'{\leq}\gamma_1 .. \gamma_n'{\leq}\gamma_n.\ \neg\gamma_0)$ and $\gamma_0, \gamma_1, \ldots, \gamma_n$ are fresh type variables. (Note that σ occurs on both sides.)

6.6.4. Fact: This definition is proper — i.e., $\mathcal{F}(R)$ is a well-formed $F^F_{\leq}$-statement for every rowing machine R.

6.6.5. Lemma: If $R \longrightarrow_R R'$, then $\mathcal{F}(R) \xrightarrow{\star}_F \mathcal{F}(R')$.

Proof: By the definition of the elaboration function for rowing machines, $R \equiv \langle\!\langle\rho_1..\rho_n\rangle\!\rangle$, where $\rho_1 \equiv [\alpha_1..\alpha_n]\langle\rho_{11}..\rho_{1n}\rangle$, and $R' \equiv \langle\!\langle\{\rho_1/\alpha_1 .. \rho_n/\alpha_n\}\,\rho_{11} .. \{\rho_1/\alpha_1 .. \rho_n/\alpha_n\}\,\rho_{1n}\rangle\!\rangle$. Let

$$\sigma \equiv \forall\gamma_0, \gamma_1..\gamma_n.\ \neg(\forall\gamma_0'{\leq}\gamma_0, \gamma_1'{\leq}\gamma_1 .. \gamma_n'{\leq}\gamma_n.\ \neg\gamma_0).$$

Now calculate as follows:

$$\begin{array}{rll} & \mathcal{F}(R) & \\ \equiv & \vdash & \sigma \\ & \leq & \forall\gamma_0{\leq}\sigma, \gamma_1{\leq}\mathcal{F}(\rho_1) .. \gamma_n{\leq}\mathcal{F}(\rho_n).\ \neg\mathcal{F}(\rho_1) \\ \equiv & \vdash & \forall\gamma_0, \gamma_1..\gamma_n.\ \neg(\forall\gamma_0'{\leq}\gamma_0, \gamma_1'{\leq}\gamma_1 .. \gamma_n'{\leq}\gamma_n.\ \neg\gamma_0) \\ & \leq & \forall\gamma_0{\leq}\sigma, \gamma_1{\leq}\mathcal{F}(\rho_1) .. \gamma_n{\leq}\mathcal{F}(\rho_n).\ \neg\mathcal{F}(\rho_1) \\ \longrightarrow_F & \vdash & \{\sigma/\gamma_0, \mathcal{F}(\rho_1)/\gamma_1 .. \mathcal{F}(\rho_n)/\gamma_n\}\,\mathcal{F}(\rho_1) \\ & \leq & \{\sigma/\gamma_0, \mathcal{F}(\rho_1)/\gamma_1 .. \mathcal{F}(\rho_n)/\gamma_n\}\,(\forall\gamma_0'{\leq}\gamma_0, \gamma_1'{\leq}\gamma_1 .. \gamma_n'{\leq}\gamma_n.\ \neg\gamma_0) \\ \equiv & \vdash & \mathcal{F}(\rho_1) \\ & \leq & \forall\gamma_0'{\leq}\sigma, \gamma_1'{\leq}\mathcal{F}(\rho_1) .. \gamma_n'{\leq}\mathcal{F}(\rho_n).\ \neg\sigma \\ \equiv & \vdash & \forall\gamma_0, \alpha_1..\alpha_n.\ \neg(\forall\gamma_0'{\leq}\gamma_0, \alpha_1'{\leq}\mathcal{F}(\rho_{11}) .. \alpha_n'{\leq}\mathcal{F}(\rho_{1n}).\ \neg\mathcal{F}(\rho_{11})) \\ & \leq & \forall\gamma_0'{\leq}\sigma, \gamma_1'{\leq}\mathcal{F}(\rho_1) .. \gamma_n'{\leq}\mathcal{F}(\rho_n).\ \neg\sigma \\ \equiv & \vdash & \forall\gamma_0, \alpha_1..\alpha_n.\ \neg(\forall\gamma_0'{\leq}\gamma_0, \alpha_1'{\leq}\mathcal{F}(\rho_{11}) .. \alpha_n'{\leq}\mathcal{F}(\rho_{1n}).\ \neg\mathcal{F}(\rho_{11})) \\ & \leq & \forall\gamma_0{\leq}\sigma, \alpha_1{\leq}\mathcal{F}(\rho_1) .. \alpha_n{\leq}\mathcal{F}(\rho_n).\ \neg\sigma \end{array}$$

$$
\begin{array}{lll}
\longrightarrow_F & \vdash & \{\sigma/\gamma_0,\, \mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\,\sigma \\
& \leq & \{\sigma/\gamma_0,\, \mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\} \\
& & \quad (\forall \gamma_0' \leq \gamma_0,\, \alpha_1' \leq \mathcal{F}(\rho_{11}) \,..\, \alpha_n' \leq \mathcal{F}(\rho_{1n}).\ \neg \mathcal{F}(\rho_{11})) \\
\equiv & \vdash & \sigma \\
& \leq & \forall\ \gamma_0' \leq \sigma, \\
& & \quad \alpha_1' \leq (\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{11})) \,.. \\
& & \quad \alpha_n' \leq (\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{1n})). \\
& & \quad\quad \neg(\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{11})) \\
\equiv & \vdash & \sigma \\
& \leq & \forall\ \gamma_0 \leq \sigma, \\
& & \quad \gamma_1 \leq (\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{11})) \,.. \\
& & \quad \gamma_n \leq (\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{1n})). \\
& & \quad\quad \neg(\{\mathcal{F}(\rho_1)/\alpha_1 \,..\, \mathcal{F}(\rho_n)/\alpha_n\}\,\mathcal{F}(\rho_{11})) \\
\equiv & \mathcal{F}(R'). & \qquad \square
\end{array}
$$

6.6.6. Lemma: If $R \equiv \langle \textsc{halt}, \rho_2..\rho_n \rangle$, then $\mathcal{F}(R)$ is derivable in $F^F_{\leq}$.

Proof: Let

$$\sigma \equiv \forall \gamma_0, \gamma_1..\gamma_n.\ \neg(\forall \gamma_0' \leq \gamma_0,\, \gamma_1' \leq \gamma_1 \,..\, \gamma_n' \leq \gamma_n.\ \neg \gamma_0).$$

Then

$$
\begin{array}{lll}
& \mathcal{F}(R) & \\
\equiv & \vdash & \sigma \\
& \leq & \forall \gamma_0 \leq \sigma,\, \gamma_1 \leq \mathcal{F}(\textsc{halt}) \,..\, \gamma_n \leq \mathcal{F}(\rho_n).\ \neg \mathcal{F}(\textsc{halt}) \\
\equiv & \vdash & \forall \gamma_0, \gamma_1..\gamma_n.\ \neg(\forall \gamma_0' \leq \gamma_0,\, \gamma_1' \leq \gamma_1 \,..\, \gamma_n' \leq \gamma_n.\ \neg \gamma_0) \\
& \leq & \forall \gamma_0 \leq \sigma,\, \gamma_1 \leq \mathcal{F}(\textsc{halt}) \,..\, \gamma_n \leq \mathcal{F}(\rho_n).\ \neg \mathcal{F}(\textsc{halt}) \\
\longrightarrow_F & \vdash & \{\sigma/\gamma_0,\, \mathcal{F}(\textsc{halt})/\gamma_1 \,..\, \mathcal{F}(\rho_n)/\gamma_n\}\,\mathcal{F}(\textsc{halt}) \\
& \leq & \{\sigma/\gamma_0,\, \mathcal{F}(\textsc{halt})/\gamma_1 \,..\, \mathcal{F}(\rho_n)/\gamma_n\}\,(\forall \gamma_0' \leq \gamma_0,\, \gamma_1' \leq \gamma_1 \,..\, \gamma_n' \leq \gamma_n.\ \neg \gamma_0) \\
\equiv & \vdash & \mathcal{F}(\textsc{halt}) \\
& \leq & \forall \gamma_0' \leq \sigma,\, \gamma_1' \leq \mathcal{F}(\textsc{halt}) \,..\, \gamma_n' \leq \mathcal{F}(\rho_n).\ \neg \sigma \\
\equiv & \vdash & \forall \gamma_0, \alpha_1 \,..\, \alpha_n.\ \neg \mathit{Top} \\
& \leq & \forall \gamma_0' \leq \sigma,\, \gamma_1' \leq \mathcal{F}(\textsc{halt}) \,..\, \gamma_n' \leq \mathcal{F}(\rho_n).\ \neg \sigma \\
\longrightarrow_F & \vdash & \sigma \\
& \leq & \mathit{Top},
\end{array}
$$

which is an instance of FTop. $\square$

6.6.7. Corollary: The rowing machine R halts iff $\mathcal{F}(R)$ is derivable in $F^F_{\leq}$.

6.6.8. Remark: It is natural to ask whether Ghelli's nonterminating example (6.2.3) is the image of some rowing machine under this translation. The answer is "almost." Although the style of divergence in Ghelli's example is suggestive of the stepping behavior of translated rowing machines, every rowing machine translation involves a type σ of appropriate width, which is not present in Ghelli's example.

6.7 Two-counter Machines

This section reviews the definition of two-counter Turing machines; see, e.g., Hopcroft and Ullman [78] for more details.

6.7.1. Definition: A *two-counter machine* is a tuple $\langle PC, A, B, I_1..I_w \rangle$, where A and B are nonnegative numbers and PC and I_1 through I_w are instructions of the following forms:

INCA⇒m
INCB⇒m
TSTA⇒m/n
TSTB⇒m/n
HALT.

with m and n in the range 1 to w.

6.7.2. Definition: The *elaboration function* $\mathcal{E}$ for two-counter machines is the partial function mapping $T = \langle PC, A, B, I_1..I_w \rangle$ to

$$\mathcal{E}(T) = \begin{cases} \langle I_m, A{+}1, B, I_1..I_w \rangle & \text{if } PC \equiv \text{INCA}{\Rightarrow}m \\ \langle I_m, A, B{+}1, I_1..I_w \rangle & \text{if } PC \equiv \text{INCB}{\Rightarrow}m \\ \langle I_m, A, B, I_1..I_w \rangle & \text{if } PC \equiv \text{TSTA}{\Rightarrow}m/n \text{ and } A = 0 \\ \langle I_n, A{-}1, B, I_1..I_w \rangle & \text{if } PC \equiv \text{TSTA}{\Rightarrow}m/n \text{ and } A > 0 \\ \langle I_m, A, B, I_1..I_w \rangle & \text{if } PC \equiv \text{TSTB}{\Rightarrow}m/n \text{ and } B = 0 \\ \langle I_n, A, B{-}1, I_1..I_w \rangle & \text{if } PC \equiv \text{TSTB}{\Rightarrow}m/n \text{ and } B > 0 \\ \text{undefined} & \text{if } PC \equiv \text{HALT.} \end{cases}$$

6.7.3. Convention: For the following examples, it is convenient to assign alphabetic labels to the instructions of a program. By convention, the instruction with label START is used as the initial PC, and the initial value in both registers is 0.

6.7.4. Example: This program loads register A with the value 5, then tests the parity of register A, halting if it is even and looping forever if it is odd:

```
START  INCA⇒I1

I1     INCA⇒I2
I2     INCA⇒I3
I3     INCA⇒I4
I4     INCA⇒E

E      TSTA⇒OK/O
O      TSTA⇒LOOP/E

LOOP   INCA⇒LOOP
OK     HALT.
```

6.7.5. Example: This program loads 5 into register A and 3 into register B, then compares A and B for equality by repeatedly decrementing them until one or both become zero; if both do so

on the same iteration, the program halts; otherwise it goes into an infinite loop.

```
START   INCA⇒I1

I1      INCA⇒I2
I2      INCA⇒I3
I3      INCA⇒I4
I4      INCA⇒J0

J0      INCB⇒J1
J1      INCB⇒J2
J2      INCB⇒LL

LL      TSTA⇒AZ/AS
AZ      TSTB⇒AZBZ/AZBS
AS      TSTB⇒ASBZ/LL
AZBZ    HALT
AZBS    INCA⇒AZBS
ASBZ    INCA⇒ASBZ.
```

6.7.6. Definition: A two-counter machine T *halts* if $T \xrightarrow{\star}_T T'$ for some machine $T' \equiv \langle \text{HALT}, A', B', I_1..I_w \rangle$.

6.7.7. Fact: The halting problem for two-counter machines is undecidable.

Proof sketch: Hopcroft and Ullman [78, pp. 171–173] show that a similar formulation of two-counter machines is Turing-equivalent. (Their two-counter machines have test instructions that do not change the contents of the register being tested and separate decrement instructions. It is easy to check that this formulation and the one used here are inter-encodable.) □

6.8 Encoding Two-counter Machines as Rowing Machines

We can now finish the proof of the undecidability of $F_{\leq}$ subtyping by showing that any two-counter machine T can be encoded as a rowing machine $\mathcal{R}(T)$ such that T halts iff $\mathcal{R}(T)$ does.

The main trick of the encoding lies in the representation of natural numbers as rows. Each number n is encoded as a *program* (i.e., a row) that, when executed, branches indirectly through one of two registers whose contents have been set beforehand to appropriate destinations for the zero and nonzero cases of a test; in other words, n itself encapsulates the behavior of the test instruction on a register containing n. The increment operation simply builds a new program of this sort from an existing one. The new program saves a pointer to the present contents of the register in a local variable so that it can restore the old value (i.e., one less than its own value) before executing the branch.

The encoding $\mathcal{R}(T)$ of a two-counter machine $T \equiv \langle PC, A, B, I_1..I_w \rangle$ comprises the following registers:

#1	$\mathcal{R}^w(PC)$
#2	$\mathcal{R}^w_A(A)$
#3	$\mathcal{R}^w_B(B)$
#4	address register for zero branches
#5	address register for nonzero branches
#6	$\mathcal{R}^w(I_1)$
...	
#6+w−1	$\mathcal{R}^w(I_w)$.

We use four translation functions for the various components:

1. $\mathcal{R}(T)$ is the encoding of a the two-counter machine T as a rowing machine of width w+5;
2. $\mathcal{R}^w(I)$ is the encoding of a two-counter instruction I as a row of width w+5;
3. $\mathcal{R}^w_A(n)$ is the encoding of the natural number n, when it appears as the contents of register A, as a row of width w+5;
4. $\mathcal{R}^w_B(n)$ is the encoding of the natural number n, when it appears as the contents of register B, as a row of width w+5.

6.8.1. Definition: The *row-encoding* (for w instructions) of a natural number n in register A, written $\mathcal{R}^w_A(n)$, is defined as follows:

$$\begin{aligned}
\mathcal{R}^w_A(0) &= \langle \#4, —, —, \textsc{halt}, \textsc{halt}, \underbrace{— \,..\, —}_{w \text{ times}} \rangle \\
\mathcal{R}^w_A(n{+}1) &= \langle \#5, \mathcal{R}^w_A(n), —, \textsc{halt}, \textsc{halt}, \underbrace{— \,..\, —}_{w \text{ times}} \rangle.
\end{aligned}$$

The row-encoding (for w instructions) of a natural number n in register B, written $\mathcal{R}^w_B(n)$, is defined as follows:

$$\begin{aligned}
\mathcal{R}^w_B(0) &= \langle \#4, —, —, \textsc{halt}, \textsc{halt}, \underbrace{— \,..\, —}_{w \text{ times}} \rangle \\
\mathcal{R}^w_B(n{+}1) &= \langle \#5, —, \mathcal{R}^w_B(n), \textsc{halt}, \textsc{halt}, \underbrace{— \,..\, —}_{w \text{ times}} \rangle.
\end{aligned}$$

6.8.2. Definition: The *row-encoding* (for w instructions) of an instruction I, written $\mathcal{R}^w(I)$, is defined as follows:

$$\begin{aligned}
\mathcal{R}^w(\textsc{inca}{\Rightarrow}m) &= \langle \#m{+}5, \langle \#5, \#\#2, —, \textsc{halt}, \textsc{halt}, — \,..\, — \rangle, —, \textsc{halt}, \textsc{halt}, — \,..\, — \rangle \\
\mathcal{R}^w(\textsc{incb}{\Rightarrow}m) &= \langle \#m{+}5, —, \langle \#5, —, \#\#3, \textsc{halt}, \textsc{halt}, — \,..\, — \rangle, \textsc{halt}, \textsc{halt}, — \,..\, — \rangle \\
\mathcal{R}^w(\textsc{tsta}{\Rightarrow}m/n) &= \langle \#2, —, —, \#m{+}5, \#n{+}5, — \,..\, — \rangle \\
\mathcal{R}^w(\textsc{tstb}{\Rightarrow}m/n) &= \langle \#3, —, —, \#m{+}5, \#n{+}5, — \,..\, — \rangle \\
\mathcal{R}^w(\textsc{halt}) &= \langle \textsc{halt}, —, —, \textsc{halt}, \textsc{halt}, — \,..\, — \rangle.
\end{aligned}$$

6.8.3. Definition: Let $T \equiv \langle PC, A, B, I_1..I_w \rangle$ be a two-counter machine. The *row-encoding* of T, written $\mathcal{R}(T)$, is the rowing machine of width w+5 defined as follows:

$$\mathcal{R}(T) = \langle\!\langle \mathcal{R}^w(PC), \mathcal{R}^w_A(A), \mathcal{R}^w_B(B), \textsc{halt}, \textsc{halt}, \mathcal{R}^w(I_1) \,..\, \mathcal{R}^w(I_w) \rangle\!\rangle.$$

6.8.4. Lemma: If $T \longrightarrow_T T'$, then $\mathcal{R}(T) \xrightarrow{\star}_R \mathcal{R}(T')$.

Proof: Let $T = \langle\!\langle PC, A, B, I_1..I_w \rangle\!\rangle$. Proceed by cases on the form of PC.

Case: $PC = \text{INCA} \Rightarrow m$

Then $T' = \langle\!\langle I_m, A{+}1, B, I_1..I_w \rangle\!\rangle$. Calculate as follows:

$$\begin{array}{rl} & \mathcal{R}(T) \\ \equiv & \langle\!\langle \langle \#m{+}5, \langle \#5, \#\#2, —, \text{HALT}, \text{HALT}, — \,..\, — \rangle, —, \text{HALT}, \text{HALT}, — \,..\, — \rangle, \\ & \mathcal{R}^w_A(A), \mathcal{R}^w_B(B), \\ & \text{HALT}, \text{HALT}, \\ & \mathcal{R}^w(I_1) \,..\, \mathcal{R}^w(I_w) \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \mathcal{R}^w(I_m), \\ & \langle \#5, \mathcal{R}^w_A(A), —, \text{HALT}, \text{HALT}, — \,..\, — \rangle, \mathcal{R}^w_B(B), \\ & \text{HALT}, \text{HALT}, \\ & \mathcal{R}^w(I_1) \,..\, \mathcal{R}^w(I_w) \rangle\!\rangle \\ \equiv & \mathcal{R}(T'). \end{array}$$

Case: $PC = \text{INCB} \Rightarrow m$

Similar.

Case: $PC = \text{TSTA} \Rightarrow m/n$

Calculate as follows:

$$\begin{array}{rl} & \mathcal{R}(T) \\ \equiv & \langle\!\langle \langle \#2, —, —, \#m{+}5, \#n{+}5, — \,..\, — \rangle, \\ & \mathcal{R}^w_A(A), \mathcal{R}^w_B(B), \\ & \text{HALT}, \text{HALT}, \\ & \mathcal{R}^w(I_1) \,..\, \mathcal{R}^w(I_w) \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \mathcal{R}^w_A(A), \\ & \mathcal{R}^w_A(A), \mathcal{R}^w_B(B), \\ & \mathcal{R}^w(I_m), \mathcal{R}^w(I_n), \\ & \mathcal{R}^w(I_1) \,..\, \mathcal{R}^w(I_w) \rangle\!\rangle \end{array}$$

There are two subcases to consider:

Subcase: $A = 0$

Then

$$\begin{array}{rcl} \mathcal{R}^w_A(A) & = & \langle \#4, —, —, \text{HALT}, \text{HALT}, — \,..\, — \rangle \\ T' & = & \langle\!\langle I_m, A, B, I_1..I_w \rangle\!\rangle. \end{array}$$

Continue calculating as follows:

$$\begin{array}{rl} & \langle\!\langle \langle \#4, —, —, \text{HALT}, \text{HALT}, — \,..\, — \rangle, \\ & \mathcal{R}^w_A(A), \mathcal{R}^w_B(B), \\ & \mathcal{R}^w(I_m), \mathcal{R}^w(I_n), \\ & \mathcal{R}^w(I_1) \,..\, \mathcal{R}^w(I_w) \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \mathcal{R}^w(I_m), \\ & \mathcal{R}^w_A(A), \mathcal{R}^w_B(B), \\ & \text{HALT}, \text{HALT}, \\ & \mathcal{R}^w(I_1) \,..\, \mathcal{R}^w(I_w) \rangle\!\rangle \end{array}$$

$\equiv \quad \mathcal{R}(T')$.

Subcase: $A > 0$

Then

$$\begin{array}{lcl} \mathcal{R}^w_A(A) & = & \langle \#5, \mathcal{R}^w_A(A{-}1), —, \text{HALT}, \text{HALT}, — \,..\, — \rangle \\ T' & = & \langle I_n, A{-}1, B, I_1..I_w \rangle. \end{array}$$

Continue calculating as follows:

$$\begin{array}{ll} & \langle\!\langle \langle \#5, \mathcal{R}^w_A(A{-}1), —, \text{HALT}, \text{HALT}, — \,..\, — \rangle, \\ & \mathcal{R}^w_A(A), \mathcal{R}^w_B(B), \\ & \mathcal{R}^w(I_m), \mathcal{R}^w(I_n), \\ & \mathcal{R}^w(I_1) \,..\, \mathcal{R}^w(I_w) \rangle\!\rangle \\ \longrightarrow_R & \langle\!\langle \mathcal{R}^w(I_n), \\ & \mathcal{R}^w_A(A{-}1), \mathcal{R}^w_B(B), \\ & \text{HALT}, \text{HALT}, \\ & \mathcal{R}^w(I_1) \,..\, \mathcal{R}^w(I_w) \rangle\!\rangle \\ \equiv & \mathcal{R}(T'). \end{array}$$

Case: $PC = \text{TSTB} \Rightarrow m/n$

Similar.

Case: $PC = \text{HALT}$

Can't happen. □

6.8.5. Lemma: If $T = \langle \text{HALT}, A, B, I_1..I_w \rangle$, then $\mathcal{R}(T)$ halts.

Proof: Immediate. □

6.8.6. Corollary: T halts iff $\mathcal{R}(T)$ does.

6.8.7. Theorem: The $F_{\leq}$ subtyping relation is undecidable.

Proof: Assume, for a contradiction, that we had a total-recursive procedure for testing the derivability of subtyping statements in $F_{\leq}$. Then to decide whether a two-counter machine T halts, we could use this procedure to test whether $\mathcal{F}(\mathcal{R}(T))$ is derivable, since

	T halts	
iff	$\mathcal{R}(T)$ halts	by Corollary 6.8.6
iff	$\mathcal{F}(\mathcal{R}(T))$ is derivable in $F^F_{\leq}$	by Corollary 6.6.7
iff	$\mathcal{F}(\mathcal{R}(T))$ is derivable in $F^D_{\leq}$	by Lemma 6.4.9
iff	$\mathcal{F}(\mathcal{R}(T))$ is derivable in $F^N_{\leq}$	by Lemma 6.3.9
iff	$\mathcal{F}(\mathcal{R}(T))$ is derivable in $F_{\leq}$	by Lemma 2.6.11.

□

6.9 Undecidability of $F_{\leq}$ Typechecking

From the undecidability of $F_{\leq}$ subtyping, the undecidability of typechecking follows immediately: we need only show how to write down a term that is well typed iff a given subtyping statement $\vdash \sigma \leq \tau$ is derivable. One such term is: $\lambda f{:}\tau \to \mathit{Top}.\ \lambda a{:}\sigma.\ f\ a$.

6.10 Undecidability of $\mathbf{F}_\wedge$

6.10.1. Definition: Let τ be an $F^D_{\leq}$ type. Then $\tau_\top$ is the $F_\wedge$ type formed by replacing instances of *Top* in τ by $\top$. This translation is extended to $F^D_{\leq}$ terms, contexts, and statements in the obvious way.

6.10.2. Lemma: $(\text{—})_\top$ is an embedding of $F^D_{\leq}$ into $F_\wedge$: if $J \equiv \Gamma \vdash \sigma \leq \tau$ is an $F^D_{\leq}$ subtyping statement, then $J_\top$ is derivable in $F_\wedge$ iff J is derivable in $F^D_{\leq}$.

Proof: ($\Longleftarrow$) Straightforward.

($\Longrightarrow$) By Theorem 4.2.8.12, $\Gamma_\top \stackrel{!}{\vdash} \sigma_\top \leq \tau_\top$ — that is, $\Gamma_\top \stackrel{!}{\vdash} \sigma_\top \leq X \Rightarrow \psi$, where $\tau_\top \equiv X \Rightarrow \psi$ and either $\psi \equiv \bigwedge[\psi_1..\psi_n]$ or $\psi \equiv \alpha$. Proceed by induction on this derivation.

Case ASubR-Inter: $\psi \equiv \bigwedge[\psi_1..\psi_m]$

By the form of $F^D_{\leq}$ statements (6.3.6), τ must have either the form $\forall \alpha_0 \leq \tau_0 \,..\, \alpha_n \leq \tau_n.\ \neg\tau'$ or the form *Top*. The first case does not apply, since then $\tau_\top$ could not have the form $X \Rightarrow \bigwedge[\psi_1..\psi_m]$. In the second case, the result is immediate by rule DTop.

Case ASubL-Inter: $\psi \equiv \alpha \qquad \sigma_\top \equiv \bigwedge[\phi_1..\phi_m]$

Can't happen: there is no n-negative type σ such that $\sigma_\top \equiv \bigwedge[\phi_1..\phi_m]$.

Case ASubL-Arrow: $\psi \equiv \alpha \qquad \sigma_\top \equiv \phi_1 \rightarrow \phi_2$

Can't happen: there is no n-negative type σ such that $\sigma_\top \equiv \phi_1 \rightarrow \phi_2$.

Case ASubL-All: $\psi \equiv \alpha \qquad \sigma_\top \equiv \forall \alpha_0 \leq \phi_0.\ \phi_b$

By the form of $\sigma_\top$,

$$\sigma \equiv \forall \alpha_0..\alpha_n.\ \neg\sigma',$$

so

$$\sigma_\top \equiv \forall \alpha_0 \leq \top..\alpha_n \leq \top.\ \forall \alpha' \leq \sigma'.\ \alpha'.$$

Now by the form of algorithmic derivations (4.2.8.4), a derivation of

$$\Gamma_\top \stackrel{!}{\vdash} \forall \alpha_0 \leq \top..\alpha_n \leq \top.\ \forall \alpha' \leq \sigma'.\ \alpha' \leq X \Rightarrow \alpha$$

must end with a chain of n instances of ASubL-All with trivial instances of ASubR-Inter as their left-hand subderivations, preceded by an instance of ASubL-All whose left-hand subderivation has the conclusion

$$\Gamma_\top, \alpha_0 \leq \psi_0 \,..\, \alpha_n \leq \psi_n \stackrel{!}{\vdash} \psi' \leq [\,] \Rightarrow \sigma'_\top,$$

where

$$\tau \equiv \forall \alpha_0 \leq \tau_0 \,..\, \alpha_n \leq \tau_n.\ \neg\tau'$$
$$\tau_{i\top} \equiv \psi_i$$
$$\tau'_\top \equiv \psi'.$$

By the induction hypothesis,

$$\Gamma, \alpha_0 \leq \tau_0 \,..\, \alpha_n \leq \tau_n \vdash \tau' \leq \sigma'.$$

By DAllNeg,

$$\Gamma \vdash \forall \alpha_0..\alpha_n.\ \neg\sigma' \leq \forall \alpha_0 \leq \tau_0 \,..\, \alpha_n \leq \tau_n.\ \neg\tau'.$$

Case ASubL-Refl: $\psi \equiv \alpha \qquad \sigma_\top \equiv \alpha \qquad X \equiv [\,]$

Can't happen: there is no n-positive type τ such that $\tau_\top \equiv \alpha$.

Case ASUBL-TVAR: $\psi \equiv \alpha \qquad \sigma_\top \equiv \beta \qquad \Gamma_\top \vdash \Gamma_\top(\beta) \leq \tau_\top$

By the induction hypothesis, $\Gamma \vdash \Gamma(\beta) \leq \tau$. By DVAR, $\Gamma \vdash \beta \leq \tau$. □

6.10.3. Corollary: The $F_\wedge$ subtyping relation is undecidable.

6.10.4. Theorem: The $F_\wedge$ typing relation is undecidable.

Proof: The $F_\wedge$ term $\Lambda\alpha.\ \lambda f{:}\tau{\to}\alpha.\ \lambda a{:}\sigma.\ f\ a$ has type $\forall\alpha.\ (\tau{\to}\alpha){\to}\sigma{\to}\alpha$ iff $\Gamma \vdash^\wedge \sigma \leq \tau$. □

6.11 Related Systems

The proof of undecidability presented here extends straightforwardly to the notion of F-bounded quantification proposed by Canning, Cook, Hill, Olthoff, and Mitchell [18].

It appears likely that a similar argument can be used to show that Pavel Curtis's more general system of constrained quantification [53] is undecidable.

6.12 Discussion

The undecidability of $F_\leq$ will perhaps surprise many of those who have studied, extended, and applied it since its introduction in 1985. But it may turn out that language designs and implementations based on $F_\leq$ will not be greatly affected by this discovery. Here are some reasons for optimism:

1. The algorithm has been used for several years now without any sign of misbehavior in any situation arising in practice. Indeed, constructing even the simplest nonterminating example requires a contortion that is difficult to imagine anyone performing by accident.
2. A number of useful fragments of $F_\leq$ are easily shown to be decidable. For example:
   - The prenex fragment, where all quantifiers appear at the outside and quantifiers are instantiated only at monotypes (types containing no quantifiers).
   - A predicative fragment where types are stratified into universes and the bound of a quantified type lives in a lower universe than the quantified type itself.
   - Cardelli and Wegner's original formulation where the bounds of two quantified types must be identical in order for one to be a subtype of the other.

     Though semantically unappealing, this formulation of $F_\leq$ appears strong enough to include essentially all useful programming examples. The only known examples that require the more general quantifier subtyping rule are those involving bounded existential types, which correspond to "partially abstract types" (c.f. [33]) under Mitchell and Plotkin's correspondence between abstract types and existential types [97]. Partially abstract types are a generalization of abstract types where some of the structure of the representation type is known but its exact identity remains hidden. Interesting subtype relations between partially abstract types can only arise from the full $F_\leq$ quantifier subtyping rule.
3. The best known subtyping algorithms for these fragments are essentially identical to the algorithm $F^N_\leq$.
4. On well-typed expressions, a type synthesis algorithm based on $F^N_\leq$ is guaranteed to terminate, since it will only ask questions to which the answer is "yes." (Note that this is not true of the type synthesis algorithm for $F_\wedge$, however: the algorithm given in Section 4.3.2 may need to ask both "yes" and "no" subtyping questions to synthesize a minimal type for a given term.)

Chapter 7

Examples

This chapter develops a broad collection of examples illustrating the expressiveness and exploring the potential practicality of type systems based on $F_\wedge$. We first describe the notational conventions used by the prototype implementation (Section 7.1), then proceed to the first set of examples (Section 7.2). These are largely based on sample programs given in Reynolds' report on the Forsythe language [121] — Forsythe being the closest extant relative of $F_\wedge$ — and serve both to give the feel of programming in a Forsythe-like language and to show some specific points where Forsythe itself could be simplified and generalized using mechanisms studied in this thesis. Section 7.3 makes a short digression to show how intersection types can be used to define procedures with optional arguments; Section 7.4 generalizes this idea to suggest a mechanism for user-defined coherent overloading. The next examples (Sections 7.5 and 7.6) illustrate a novel style of programming using intersection types, where basic datatypes can be refined into small abstract lattices (distinguishing, for example, the type of empty lists from that of nonempty lists) and functions over them given correspondingly refined types, encoding the kind of information that might be obtained by conventional abstract interpretation or strictness analysis. In Section 7.7, we verify that the ability to perform these refined static analyses during typechecking is inherent in the core calculus itself (rather than arising from some special choice of primitive types and constants) by showing how to express some of the earlier examples in pure $F_\wedge$ using extensions of the well-known encodings of inductive datatypes in the polymorphic λ-calculus. In Section 7.8, we discuss the process of programming in $F_\wedge$ and illustrate some useful debugging techniques. Section 7.9 applies the "encoding" of bounded quantification as unbounded quantification and intersections (c.f. Section 3.5) to some of the earlier examples.

7.1 Conventions

The examples in the remainder of the chapter rely on a prototype typechecker for $F_\wedge$, implemented in about 5000 lines of Standard ML [93]. Sample sessions with the typechecker have been typeset directly from the output of the running system: text files containing both the input portions of the examples and raw TEX sources for the running commentary are passed through the typechecker, which inserts its responses at the appropriate points.

The system maintains a notion of the "current pervasive context," to which new definitions are cumulatively added. For example, a new type variable `Real` may be introduced by specifying its bound:

```
> Real < T;
```

Later definitions and expressions are understood relative to a pervasive context in which `Real` is defined:

```
> idReal = \x:Real. x;
idReal : Real -> Real

> Int < Real;

> check Real->Int < Int->Real;
Yes

> check Int->Int < Real->Real;
No

> polyIdInt = \\A<Int. \a:A. a;
polyIdInt : All A<Int. A -> A
```

Similarly, new term variables, corresponding to primitive constants, may be added to the pervasive context:

```
> zero : Int,
> plus : Int -> Int -> Int;
```

Definitions are terminated by either a semicolon or a comma. The system keeps reading comma-separated definitions until it reaches a terminating semicolon, at which time the whole collection is processed and any responses printed:

```
> one : Int,
> two = plus one one,
> four = plus two two,
> double = \x:Int. plus x x;
two : Int
four : Int
double : Int -> Int
```

If a term is entered without a name, it is named "`it`" by default:

```
> double (plus four two);
it : Int

> plus it it;
it : Int
```

The typechecker provides a simple facility for transparent type abbreviation. An identifier may be associated with a type expression introduced by the "`==`" symbol:

```
> BinFun == Int->Int->Int;
```

The abbreviation `BinFun` is completely equivalent to the longer expression. Instances of `BinFun` are expanded to `Int->Int->Int`, as necessary, during typechecking:

```
> \f:BinFun. \x:Int. f x x;
it : BinFun -> Int -> Int
```

Conversely, when types are printed, instances of `Int->Int->Int` are collapsed to `BinFun`:

```
> \x:Int. \y:Int. plus x y;
it : BinFun
```

7.2 Examples from the Forsythe Report

The Forsythe language [121] is in many respects the closest relative of the $F_\wedge$ calculus. Briefly, Forsythe is a normal-order language combining functional and imperative features, whose execution model is based on Algol 60 [98] and whose core type system is the first-order calculus of intersection types described in Section 2.3.

Since we have not proposed a specific set of basic datatypes for a language based on $F_\wedge$, we are not in a position to make a detailed comparison of the two systems as programming languages. Instead, the examples in this section illustrate some of the possible properties of a Forsythe-like programming language based on $F_\wedge$, and underscore some points where the additional expressive power of second-order polymorphism might be used to generalize constructs already present in Forsythe.

It is also worth noting that we make only minimal use of bounded quantification (as opposed to ordinary unbounded quantification) here. Practical examples motivating bounded quantification tend to involve language features (e.g., records) that we have not considered.

The primitive datatypes of Forsythe include the numeric types of integers and reals, the type of booleans, and the type of characters, plus a primitive type `Value` that forms a common supertype of all of the rest. These are modeled in $F_\wedge$ by the following declarations:

```
> Value < T,
> Real < Value,
> Int < Real,
> Bool < Value,
> Char < Value;
```

Forsythe includes a variety of primitive operators on these types. Here we give only a few that will be needed later in this section:

```
> 0 : Int,
> 1 : Int,
> plus: Int->Int->Int /\ Real->Real->Real,
> minus: Int->Int->Int /\ Real->Real->Real,
> times: Int->Int->Int /\ Real->Real->Real;

> true : Bool,
> false : Bool,
> if : All A. Bool -> A -> A -> A,
> not : Bool -> Bool;

> eq :    Int->Int->Bool /\ Real->Real->Bool /\ Bool->Bool->Bool
>      /\ Char->Char->Bool,
> neq :    Int->Int->Bool /\ Real->Real->Bool /\ Bool->Bool->Bool
>       /\ Char->Char->Bool,
> leq : Int->Int->Bool /\ Real->Real->Bool;
```

We depart from Forsythe in the typing of the `if` primitive. Forsythe includes a "generalized `if`" construct as a built-in syntactic form. The `if` used here drops the convenience of Forsythe's `if` in favor of the broader generality of a polymorphic constant.

The primitive type `Comm` is the type of *commands* — simple imperative programs whose execution may affect the store:

```
> Comm < T;
```

The related type `Compl` is used for "completions": imperative programs that are guaranteed never to return (e.g., because they escape to a previously captured continuation). By neglecting the fact that a `Compl` will never return, we can regard it as a simple `Comm`; this observation is formalized as a primitive coercion from `Compl` to `Comm`:

```
> Compl < Comm;
```

The primitive constructor for commands is the sequencing operator `before` (we also allow `before` to apply to a command and a completion, producing a completion in this case):

```
> before : Comm->Comm->Comm /\ Comm->Compl->Compl;
```

To make programs involving commands easier to read, we introduce some special syntax reminiscent of the single-semicolon syntax for sequencing in many familiar languages: the expression `begin e1 ;; e2 ;; ... ;; en end` is translated by the parser into `(before e1 (before e2 (... (before en-1 en))))`.

```
> repeat5 = \c:Comm. begin c ;; c ;; c ;; c ;; c end;
repeat5 : Comm -> Comm
```

The `while` operator provides basic iteration:

```
> while : Bool -> Comm -> Comm;
```

The `skip` command provides a convenient way of doing nothing:

```
> skip : Comm;
```

The primitive means of creating completions is the `escape` operator. The argument to `escape` is a function that accepts a completion and computes a command. Operationally, the completion passed to this function aborts execution of the function and continues immediately from the point where `escape` was called:

```
> escape : (Compl->Comm) -> Comm;
```

Forsythe includes a built-in syntactic form `rec` for defining recursive values. In $F_\wedge$, we may avoid dealing with recursive definition in the core language by introducing it as a polymorphic constant `fix`. This is slightly more verbose that Forsythe's `rec`, but avoids the problem (still open for Forsythe) of synthesizing minimal types for programs involving `rec`.

```
> fix : All A. (A->A) -> A;
```

The `while` operator may also be defined in terms of `fix`:

```
> while =
>   \b:Bool. \c:Comm.
>      fix [Comm] \next:Comm.
>        if [Comm] b
>            begin c ;; next end
>            skip;
while : Bool -> Comm -> Comm
```

Notice that the correct behavior of `while` depends crucially on the normal-order reduction strategy of Forsythe and our hypothetical language based on $F_\wedge$, since it requires that the guard `b` be executed repeatedly.

One of the principal innovations of Forsythe was the separation of the normally atomic concept of "variable" into two yet smaller units: expressions and acceptors, or sources and sinks. Intuitively, a variable should be thought of not as a cell capable of either receiving or producing a value, but as two separate (but normally connected) entities, one capable of producing values when evaluated

and one capable of accepting values. We retain the names `Int`, `Real`, `Bool`, and `Char` for the primitive expression types and introduce the abbreviation `XAcc = X -> Comm` for the four acceptor types. Then an `XVar` is just the intersection (thought of as a product, since the relevant coherence condition is vacuous) of an `X` and an `XAcc`:

```
> IntAcc == Int -> Comm,
> RealAcc == Real -> Comm,
> BoolAcc == Bool -> Comm,
> CharAcc == Char -> Comm,
> IntVar == Int /\ IntAcc,
> RealVar == Real /\ RealAcc,
> BoolVar == Bool /\ BoolAcc,
> CharVar == Char /\ CharAcc;
```

Again, the readability of programs is enhanced by a pinch of syntactic sugar; we let `v := e` abbreviate `v e`, so that assignment statements may be written in the familiar way:

```
> \v:IntVar. begin v := plus v 1 ;; v := plus v v end;
it : IntVar -> Comm
```

Variables are created by a primitive constructor `newcell`. Here we follow the style of Forsythe by letting the body of a `newcell` expression be a function that expects to be passed the newly created variable as its argument.

```
> newcell : All A<Value. All R. A -> ((A/\A->Comm)->R) -> R;
```

The first argument to `newcell` is the type of the cell to be created. The second argument is the final result type of the body that uses the new variable. The third argument is an expression whose result will be the initial value of the new variable, and the fourth is the body itself. (N.b.: we generalize Forsythe's variable declarators by allowing the newly created cell to contain a value of any subtype of `Value`, rather than explicitly mentioning the primitive types `Int`, `Real`, `Bool`, and `Char`. We also allow the body of the `newcell` construct to have any type, where Forsythe restricts it to one of six possibilities: `Comm`, `Compl`, `Int`, `Real`, `Bool`, and `Char`.)

The variable constructors of Forsythe may now be defined using `newcell`:

```
> newIntCell = newcell [Int],
> newRealCell = newcell [Real],
> newBoolCell = newcell [Bool],
> newCharCell = newcell [Char];
newIntCell : All R. Int -> (IntVar->R) -> R
newRealCell : All R. Real -> (RealVar->R) -> R
newBoolCell : All R. Bool -> (BoolVar->R) -> R
newCharCell : All R. Char -> (CharVar->R) -> R

> newIntVar = newIntCell [Int,Real,Bool,Char,Comm,Compl],
> newRealVar = newRealCell [Int,Real,Bool,Char,Comm,Compl],
> newBoolVar = newBoolCell [Int,Real,Bool,Char,Comm,Compl],
> newCharVar = newCharCell [Int,Real,Bool,Char,Comm,Compl];
newIntVar : Int->(IntVar->Int)->Int
         /\ Int->(IntVar->Real)->Real
         /\ Int->(IntVar->Bool)->Bool
         /\ Int->(IntVar->Char)->Char
         /\ Int->(IntVar->Comm)->Comm
         /\ Int->(IntVar->Compl)->Compl
newRealVar : Real->(RealVar->Int)->Int
          /\ Real->(RealVar->Real)->Real
```

```
          /\ Real->(RealVar->Bool)->Bool
          /\ Real->(RealVar->Char)->Char
          /\ Real->(RealVar->Comm)->Comm
          /\ Real->(RealVar->Compl)->Compl
newBoolVar : Bool->(BoolVar->Int)->Int
          /\ Bool->(BoolVar->Real)->Real
          /\ Bool->(BoolVar->Bool)->Bool
          /\ Bool->(BoolVar->Char)->Char
          /\ Bool->(BoolVar->Comm)->Comm
          /\ Bool->(BoolVar->Compl)->Compl
newCharVar : Char->(CharVar->Int)->Int
          /\ Char->(CharVar->Real)->Real
          /\ Char->(CharVar->Bool)->Bool
          /\ Char->(CharVar->Char)->Char
          /\ Char->(CharVar->Comm)->Comm
          /\ Char->(CharVar->Compl)->Compl
```

The Forsythe report develops a number of programs using these primitives. For example, here is a simple iterative definition of the factorial function (c.f. [121, p. 29]):

```
> fact = \n:Int. \f:IntVar.
>   newIntVar 0 \k:IntVar.
>      begin
>        f := 1 ;;
>        while (neq k n)
>          begin
>             k := plus k 1 ;;
>             f := times k f
>          end
>      end;
fact : Int -> IntVar -> Comm
```

Using an extra `newIntVar`, this version of factorial may be improved so that it evaluates its initial argument only once. This is a common idiom in Forsythe, since it amounts to a call-by-value parameter passing regime:

```
> fact2 = \n:Int. \f:IntVar.
>   newIntVar n \n:IntVar.
>      newIntVar 0 \k:IntVar.
>          begin
>            f := 1 ;;
>            while (neq k n)
>               begin k := plus k 1 ;; f := times k f end
>          end;
fact2 : Int -> IntVar -> Comm
```

Similarly, we can use `newIntVar` to build a version that makes only one assignment of the final result to the second parameter; this is essentially a call-by-result parameter:

```
> fact3 = \n:Int. \f:IntAcc.
>    newIntVar n \n:IntVar. newIntVar 1 \localf:IntVar.
>      begin
>         (newIntVar 0 \k:IntVar.
>             while (neq k n)
>                begin k := plus k 1 ;; localf := times k localf end) ;;
>         f := localf
```

```
>      end;
fact3 : Int -> IntAcc -> Comm
```

We may also, of course, give a traditional recursive formulation of factorial:

```
> fact = fix [Int->Int] \fact:Int->Int.
>   \n:Int. newIntVar n \n:IntVar.
>     if [Int] (eq n 0) 1 (times n (fact (minus n 1)));
fact : Int -> Int
```

The default call-by-name procedure call semantics allows a variety of syntactic extensions to be defined as user-level operations. Here is another kind of iteration construct:

```
> forup = \from:Int. \to:Int. \b:IntAcc.
>   newIntVar (minus from 1) \k:IntVar.
>   newIntVar to \to:IntVar.
>     while (leq k to) begin k := plus k 1 ;; b k end;
forup : Int -> Int -> IntAcc -> Comm
```

The `forup` iterator provides a good opportunity for an illustration of the `escape` procedure. The following function accepts an integer function of one argument and a range of integers in which to search (linearly) for a particular value `y` of the function. If this value is found within the specified range, the function immediately returns with no further search, using `escape` to jump out of the body of the `forup`:

```
> linsearch = \X:Int->Int. \from:Int. \to:Int. \y:Int.
>             \present:BoolAcc. \j:IntAcc.
>   escape \out:Compl.
>      begin
>         forup from to \k:Int.
>           if [Comm] (eq (X k) y)
>              begin present := true ;; j := k ;; out end
>              skip ;;
>         present := false
>      end;
linsearch : (Int->Int) -> Int -> Int -> Int -> BoolAcc -> IntAcc -> Comm
```

7.3 Procedures With Optional Arguments

A completely different example of the practical utility of intersection types (in this case, even first-order intersection types) comes from their ability to express procedures with default parameters. For example, we can give the type `String->Int->(String/\Char->String)` to a built-in function that takes a string `s` and an integer `i` and returns *both* the string `s` padded with enough blanks to make its length `i` *and* a function that, given a character `c`, returns `s` padded with enough `c`'s to make its length `i`. The result of applying `pad` to `s` and `i` can either be used directly as a string (by applying a `print` function to it, for example) or further applied to a character `c`.

To implement this scheme, we assume the following primitives:

```
> pad : String -> Int -> (String/\Char->String);
> print : String -> Unit;
> blank : Char;
> dot : Char;
> mesg : String;
```

Now we can use `pad` and `print` as described above:

```
> print (pad mesg 10);
it : Unit
> print (pad mesg 10 dot);
it : Unit
```

In fact, this notion could be supported as a general language extension by introducing a built-in polymorphic constant that is used to build functions with default parameters:

```
> default : All A. All B. (A -> B) -> A -> (B/\A->B);
```

```
> myprimpad : String -> Int -> Char -> String;
> mypad = \s:String. \l:Int. default [Char] [String] (myprimpad s l) blank;
mypad : String->Int->String /\ String->Int->Char->String
```

A particularly interesting case occurs when `B` $\equiv$ `A->C` for some `C`. Then the value built by `default` cannot "tell," from the context in which it is used (an application to something of type `A`), whether its `B` component or its `A->B` component is desired. In must essentially produce both components, so that the application results in a new overloaded value of type `C/\B`, i.e., `C/\A->C`, to which the same considerations apply.

7.4 User-defined Coherent Overloading

A further language extension along the lines of the previous section is to provide a primitive `glue` that allows user-defined coherent overloading. Given two values `a` and `b`, of types `A` and `B`, the expression `glue a b` yields a value that, in a context where a value of type `A` is expected, behaves like `a`, and, in a context expecting a `B`, behaves like `b`.

```
> glue : All A. All B. A -> B -> (A/\B);
```

Then, for example, we can define our usual `plus` function operating on both `Int` and `Real` from two more specialized versions:

```
> plusInt : Int->Int->Int;
> plusReal : Real->Real->Real;
```

```
> plus = glue [Int->Int->Int] [Real->Real->Real] plusInt plusReal;
plus : Int->Int->Int /\ Real->Real->Real
```

Of course, if `glue a b` is used in a context where either an `A` or a `B` is appropriate, the compiler is free to choose either `a` or `b` as its value. If these do not behave coherently, then any coherence guarantees provided by the language designer for the built-in types and type constructors are nullified. Such constructs blur the distinction between the language designer and the expert systems programmer, a facility that can be invaluable in rare circumstances but that should be used sparingly. It may be advisable to explicitly mark sections of code where `glue` may be used as "unsafe," in the terminology of Modula-3 [28, 99].

7.5 Modeling Abstract Interpretation

Perhaps the most useful property of programming languages with intersection types is that they allow extremely *refined* types to be given for expressions — much more refined than is possible in conventional polymorphic languages. Rather than a single description, each expression may be assigned any finite collection of descriptions, each capturing some aspect of its behavior. This means that, in the limit, the behavior of a program can be *exactly* described by the types assignable to it. In more practical contexts, it offers both language designer and programmer a great deal of flexibility in choosing behavioral primitives that capture salient properties of programs and obtaining good descriptions of programs in terms of these primitives. Since we are working with explicitly typed calculi, this requires effort in the form of type assumptions or annotations; in general, as more effort is expended, better typings are obtained.

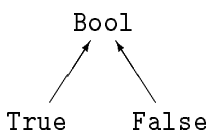
This section and the next illustrate these general observations by showing how various forms of program analysis can be mimicked in the type system of languages with intersection types. We concentrate first on performing some simple kinds of abstract interpretation, using the typechecker, under explicit programmer control, to derive types encoding the same sorts of information as might be obtained by a static analyzer during the code generation phase of a modern compiler.

Some of the following examples were suggested by conversations with Tim Freeman and Frank Pfenning, whose independent work on *refinement types* for languages in the ML class [60] shares many motivations and technical intuitions with what appears here. Hayashi has described related work using refinement types in extracting programs from proofs [72].

It is important to note that we are switching semantic intuitions at this point. Previous sections relied on the intuition that values of intersection types were represented at run time as tuples of different (though coherent) values, as described in Sections 2.4.2 and 5.3. Here, and for the remainder of the chapter, it makes more sense to think of expressions as denoting single run time values, which an intersection type simply describes in several different ways. This point of view corresponds to the untyped semantic models presented in Sections 2.4.1 and 5.1.

7.5.1 Booleans

The examples in previous sections used a primitive type `Bool` with two elements, `true` and `false`. This picture can be refined by introducing two subtypes of `Bool`, called `True` and `False`,



and giving more exact types for the constants `true` and `false` in terms of these refinements:

```
> Bool < T,
> True < Bool,
> False < Bool;

> true : True,
> false : False;
```

The primitive `if` can also be given a more refined type: if we know whether the value of the test lies in the type `True` or the type `False`, we can tell in advance which of the branches will be chosen:

```
> if : All A.    (True -> A -> T -> A)
>             /\ (False -> T -> A -> A)
>             /\ (Bool -> A -> A -> A);
```

(The third typing is needed here because $F_\wedge$'s types cannot express the idea that every element of `Bool` is an element of either `True` or `False`. This shortcoming, while not serious in practice, motivates the investigation of a dual notion of *union types*, which we discuss briefly in Section 8.3.3.)

The refinement in the types of `true`, `false`, and `if` can now be inherited by new functions defined from these:

```
> or =
>    \x:True,False,Bool. \y:True,False,Bool.
>       for R in True,False,Bool.
>          if [R] x true y;
or : True->Bool->True
  /\ False->False->False
  /\ Bool->True->True
  /\ Bool->Bool->Bool
```

This example illustrates several novel aspects of the style of programming being explored here. For one thing, note the typechecker does not *automatically* discover the more refined type for `or`: it must explicitly be instructed to consider all the necessary sets of assumptions. Annotating the two abstractions with only the type `Bool` results in a strictly less refined typing:

```
> or = \x:Bool. \y:Bool. if [Bool] x true y;
or : Bool -> Bool -> Bool
```

A stranger element of the example is the type variable `R`. The *for* expression introducing `R` provides a kind of "guessing" behavior that is often necessary to achieve the desired type for functions like `or`. Without `R`, we would get stuck trying to decide what type to provide as the argument to `if`. None of `Bool`, `True`, or `False` will do the trick, since providing any one of these would amount to asserting that this will be the result type of the `if` in every case:

```
> or =
>    \x:True,False,Bool. \y:True,False,Bool.
>          if [Bool] x true y;
or : Bool -> Bool -> Bool

> or =
>    \x:True,False,Bool. \y:True,False,Bool.
>          if [True] x true y;
or : True->Bool->True /\ Bool->True->True

> or =
>    \x:True,False,Bool. \y:True,False,Bool.
>          if [False] x true y;
or : False -> False -> False
```

The actual type that should be provided as the first argument to `if` depends on the current set of assumptions for the variables `x` and `y`: if both have type `True`, then the argument to `if` should be `True`, if both `False`, then `False`, etc. But this sort of calculation clearly cannot be expressed in the type system of $F_\wedge$. What turns out to work is simply *guessing* all three possibilities in turn. Two out of three times, the result will be too large — it will come out as `Bool` when `True` or `False` would have been achievable, or it will simply be $\top$ — but one of the three will produce the desired type. This type is guaranteed to be a subtype of the two "wrong" results, and so the intersection

of the three is equivalent to the desired one. Here we show the internal form of the type actually derived by the type synthesis algorithm, before it is simplified for printing:

```
> or =
>    \x:True,False,Bool. \y:True,False,Bool.
>       for R in True,False,Bool.
>          if [R] x true y;
or : True
   ->(True->((True/\True)/\T/\(Bool/\Bool))
    /\False->(/\[True]/\T/\(Bool/\Bool))
    /\Bool->(/\[True]/\T/\(Bool/\Bool)))
  /\ False
   ->(True->((True/\True)/\T/\(Bool/\Bool))
    /\False->(T/\/\[False]/\(Bool/\Bool))
    /\Bool->(T/\T/\(Bool/\Bool)))
  /\ Bool
   ->(True->(/\[True]/\T/\/\[Bool])
    /\False->(T/\T/\/\[Bool])
    /\Bool->(T/\T/\/\[Bool]))
i.e. True->Bool->True
  /\ False->False->False
  /\ Bool->True->True
  /\ Bool->Bool->Bool
```

This idiom — a *for* whose body is a type application where the only use of the type variable introduced by the *for* is as the argument to the application — is so common that we introduce a new abbreviatory form for it:

$$e\,[\tau_1..\tau_n] \stackrel{\text{def}}{=} \mathit{for}\,\alpha\ \mathit{in}\,\tau_1..\tau_n.\ e\,[\alpha] \qquad \text{(where } \alpha \text{ is fresh).}$$

Then,

```
> or =
>    \x:True,False,Bool. \y:True,False,Bool.
>          if [True,False,Bool] x true y;
or : True->Bool->True
  /\ False->False->False
  /\ Bool->True->True
  /\ Bool->Bool->Bool
```

7.5.2 Lists

More interesting kinds of abstract interpretation can be performed on programs involving structured data like lists and trees.

For this example, we assume that the property of lists we are concerned with is whether they are of even or odd length. As we did with the booleans, we assume a type `List` of finite lists of natural numbers

```
> List < T;
```

with two immediate subtypes, `Even` and `Odd`, and one further refinement of `Even`, a special type containing only `Nil`:

```
> Even < List,
> Odd < List,
> Nil < Even;
```

```
> nil : Nil;
```

We then state types for the primitive list operations in terms of this partial order:

```
> car : List->Nat;
> cdr : Even->Odd /\ Odd->Even /\ List->List,
> cons : Nat->Even->Odd /\ Nat->Odd->Even /\ Nat->List->List,
> null : Odd->False /\ Nil->True /\ List->Bool;
```

(Of course, even more refined types might be given for these. For example, we could distinguish another type `EvenCons` of even-length, nonempty lists and give `car` the type `EvenCons->Nat/\Odd->Nat`. However, this typing for `car` would prevent us from obtaining the desired type for `append` below. This is a reminder that we are still in the business of typechecking; the full power of arbitrary abstract interpretation methods should not be expected.)

As usual, we may define higher-level functions so that they inherit similar typings from the primitives:

```
> cddr = \l:Even,Odd,Nil,List. cdr (cdr l);
cddr : Even->Even /\ Odd->Odd /\ List->List
```

Finally, we can give a type to the `append` function showing that it maps, for example, pairs of even-length inputs into even-length results.

Since `append` is defined using the explicitly typed fixpoint operator, we must begin by stating the type we hope to obtain:

```
> AppType == Even->Even->Even
>         /\ Even->Odd ->Odd
>         /\ Odd ->Even->Odd
>         /\ Odd ->Odd ->Even
>         /\ List->List->List;
```

Now `append` is expressed as follows:

```
> append =
>   fix [AppType] \app:AppType.
>      \l1:Even,Odd,Nil,List. \l2:Even,Odd,Nil,List.
>         if [Even,Odd,Nil,List] (null l1)
>            l2
>            (cons (car l1) (app (cdr l1) l2));
append : AppType
```

By providing more refined type information to the fixed point operator, we can obtain an even more refined typing for `append`:

```
> AppType2 == Even->Even->Even
>          /\ Even->Odd ->Odd
>          /\ Odd ->Even->Odd
>          /\ Odd ->Odd ->Even
>          /\ Nil ->Nil ->Nil
>          /\ List->List->List;

> append2 =
>   fix [AppType2] \app:AppType2.
>      \l1:Even,Odd,Nil,List. \l2:Even,Odd,Nil,List.
>         if [Even,Odd,Nil,List] (null l1)
>            l2
>            (cons (car l1) (app (cdr l1) l2));
append2 : AppType2
```

7.5.3 Natural Numbers

Similar tricks also apply to programs involving natural numbers (which, after all, can be viewed as lists of marks). Here we distinguish zero from the rest of the natural numbers:

```
> Nat < T,
> Zero < Nat,
> Pos < Nat;

> succ : Nat -> Pos,
> pred : Nat -> Nat,
> iszero : Zero->True /\ Pos->False /\ Nat -> Bool;
```

Like `append`, the `plus` function can be defined via an explicit fixed point. We state the type that we hope to obtain,

```
> PlusType == Zero->Zero-> Zero
>          /\ Nat ->Pos -> Pos
>          /\ Pos ->Nat -> Pos
>          /\ Nat ->Nat -> Nat;
```

and provide sufficient guidance for the typechecker to infer this type for the body of the `plus`:

```
> plus =
>   fix [PlusType] \plus:PlusType.
>     \m:Zero,Pos,Nat. \n:Zero,Pos,Nat.
>         if [Zero,Pos,Nat] (iszero m) n (succ (plus (pred m) n));
plus : PlusType
```

Our treatment of both `append` and `plus` suffers from the necessity of deciding, beforehand, what type we want to obtain for them. If we program using a more restricted set of primitives — for which, naturally, more exact types can be given — we can make the typechecker do more of the work of discovering types for functions like `plus`.

The `plus` function may be defined in terms of a primitive *iteration* construct that takes a number, together with a single-argument function on some *result type* `N` and a starting value of type `N`, and computes the result of applying the function an appropriate number of times to the given starting value:

```
> oiternat : Nat -> All N. (N->N) -> N -> N;

> oplus =
>   \m:Nat. \n:Nat.
>      oiternat m [Nat] succ n;
oplus : Nat -> Nat -> Nat
```

A more refined typing for the natural number iterator can be expressed in terms of `Zero` and `Pos`:

```
> iternat :
>      (Zero-> All N, P<=N, Z<=N. (N->P) -> Z -> Z)
>   /\ (Pos -> All N, P<=N, Z<=N. (N->P) -> Z -> P)
>   /\ (Nat -> All N, P<=N, Z<=N. (N->P) -> Z -> N);
```

Using this iterator, the same type as above may be obtained for `plus` without declaring it in advance:

```
> plus =
>   for M in Zero,Pos,Nat. for N in Zero,Pos,Nat.
>     \m:M. \n:N.
>         iternat m [Nat,Pos] [Pos] [N] succ n;
plus : Zero->Zero->Zero /\ Pos->Nat->Pos /\ Nat->Pos->Pos /\ Nat->Nat->Nat
```

We allow the types of both `m` and `n` to be any of `Nat`, `Zero`, or `Pos`, checking the body separately in each case. We need to apply `m` alternatively to both the types `Nat` and `Pos`, to be sure of having a name for the type of `n`; when `m` has type `Pos`, the result type of the iteration is always `Pos`. (Note that applying `m` to `Zero` does not make sense, since this would amount to asserting that the iteration of `m` over `succ` and `n` can return a zero result even when `m` is positive.)

7.6 Modelling Strictness Analysis

Strictness analysis [16] discovers situations in which the argument to a function will always be evaluated in the course of evaluating the function's body. This sort of information is useful, for example, in the optimization phases of compilers for lazy functional languages. When a given function is known to be strict in its argument, then whenever this function is applied, its argument may immediately be evaluated (rather than being encapsulated in a closure) without fear of introducing spurious nontermination in the transformed program. Since closures are generally expensive to build, good strictness analysis can greatly enhance the quality of code generated by compilers that use it. Like abstract interpretation, a simple form of strictness analysis may be encoded in $F_\wedge$ as a typechecking task. (For related treatments of strictness analysis as a typechecking problem, see [84, 86].)

The technical approach here is slightly different than in the previous section: rather than introducing subtypes encoding strictness information for *every* type used in a program, we deal with strictness separately from ordinary typing, carrying along the partial results of strictness analysis "beside" the normal typing information derived for expressions.

We begin with a new type constant `Btm`, representing divergent computations:

```
> Btm < T;
```

The types `Btm` and `T` together form a two-point abstract lattice of strictness assertions: `Btm` encodes the information that a particular expression (typically an argument to a function) necessarily diverges; `T` encodes the absence of such information.

We now add to the types of our primitive functions appropriate annotations in terms of `Btm` and `T`, indicating which arguments each primitive can be counted upon to evaluate. For example, the constant `true` always terminates

```
> true: Bool;
```

and the `if` operation maps a `Bool` and a pair of `A`'s into an `A` (for any type `A`) and, furthermore, that it always evaluates its first argument and also always evaluates either its second or its third argument.

```
> if : All A. (Bool -> A -> A -> A)
>          /\ (Btm -> T -> T -> Btm)
>          /\ (T -> Btm -> Btm -> Btm);
```

This type can be read as asserting that this information can now be inherited by functions built up from `if`:

```
> or = \a:Bool,Btm. \b:Bool,Btm. if [Bool] a true b;
or : Bool->Bool->Bool /\ Btm->Bool->Btm /\ Btm->Btm->Btm
```

Some other useful primitives can be given similar typings:

```
> nil : List,
> cons : Nat -> List -> List
>     /\ Btm -> T -> Btm
>     /\ T -> Btm -> Btm,
> car : List -> Nat
>    /\ Btm -> Btm,
> cdr : List -> List
>    /\ Btm -> Btm,
> null : List -> Bool
>    /\ Btm -> Btm;
```

7.6.1. Remark: Like the examples involving type refinement and abstract interpretation, this application of intersection types strongly suggests an untyped semantic model. We do not think of the typing information associated with strictness analysis as giving rise to any behavior at run time. For example, the second and third elements of the type of `cons` are not thought of as actual functions, but as predicates describing the behavior of a single function whose "real type" is `Nat->List->List`. Of course, there is no harm in keeping the typed perspective and requiring that some kind of functions with types `Btm->T->Btm` and `T->Btm->Btm` be present at run time, but it is less natural to do so.

Using this information, we build higher-level functions, as before, for which the typechecker can infer appropriate strictness information:

```
> AppType == List->List->List
>         /\ Btm->T->Btm /\ T->Btm->Btm;

> append =
>   fix [AppType] (\app:AppType.
>       \l1:List,Btm,T.
>         \l2:List,Btm,T.
>            (if [List] (null l1) l2 (cons (car l1) (app (cdr l1) l2))));
append : AppType
```

Finally, we can make use of the strictness information obtained by this method to introduce a behavior-preserving "strictifier" that can only be applied to functions guaranteed to be strict in their first argument. (This version of `strictify` works on functions of between one and four arguments that are strict:)

```
> strictify :  All A.
>                   (A /\ Btm->Btm) -> A
>               /\  (A /\ Btm->T->Btm) -> A
>               /\  (A /\ Btm->T->T->Btm) -> A
>               /\  (A /\ Btm->T->T->T->Btm) -> A;
```

Using `strictify`, we may define a version of `append` that evaluates its left argument before its own body:

```
> leftstrictappend = strictify [AppType] append;
leftstrictappend : AppType
```

7.7 Refining Pure Encodings of Inductive Types

One of the main advantages of working with impredicative polymorphism in foundational investigations of static type systems is that a great variety of datatypes that ordinarily have to be given as explicit extensions can be encoded directly in the pure calculus. These encodings allow a broad range of issues to be investigated using very economical formal means. Later, when the time arrives to design a full-scale language based on the results of these preliminary investigations, the behavior of the encodings provides a strong guide for the proper behavior of the corresponding primitive datatypes.

In this section, we use variants of the familiar encodings of algebraic datatypes such as natural numbers and booleans to verify that the "abstract interpretation" behavior investigated in the previous section arises naturally in the system and is not "rigged" by our assumptions about primitive values like `cdr` and `succ`.

7.7.1 Church Arithmetic

We begin by reviewing the well-known encoding of the Church numerals in the polymorphic λ-calculus. (Readers unfamiliar with this encoding may find the more expository presentations in [108, 120] helpful. Also see [38, 15, 103, 63, 29, 27].) We then show how to enrich this encoding to model the "abstract lattice" used in Section 7.5.3, where zero is distinguished from the rest of the numbers. Analogous extensions of the usual encodings of the arithmetic operators may now be given types identical to those we *assumed* for them in Section 7.5.3.

Recall that Church's numerals [38] are encoded in the ordinary polymorphic λ-calculus as elements of the following type:

```
> OrigNat == All N. (N->N) -> N -> N;
```

Operationally, the type argument `N` to an element `e` of type `OrigNat` specifies the type of the result of the n-fold iteration of the argument `s` over the argument `z`, where n is the number coded by `e`. In other words, a number is its own iterator:

```
> origIterNat =
>   \m:OrigNat.
>      \\N. \s:N->N. \z:N.
>         m [N] s z;
origIterNat : OrigNat -> OrigNat
```

The first few natural numbers are encoded as follows:

```
> origzero = \\N. \s:N->N. \z:N. z,
> origone = \\N. \s:N->N. \z:N. s z,
> origtwo = \\N. \s:N->N. \z:N. s (s z);
origzero : OrigNat
origone : OrigNat
origtwo : OrigNat
```

Since we intend to distinguish zero from all other natural numbers, our refined encoding will take *three* type arguments — one for the result of the n-fold iteration of `s` over `z` where n may be either zero or positive, one for the result type of a 0-fold iteration (that is, the type of `z` itself) and one for the result type of an n-fold iteration for some $n \geq 1$. Also, the function `s` must map arbitrary elements of `N` to elements of `P`, and the starting point for the iteration, `z`, must have type `Z`:

```
> Nat  == All N. All P<N. All Z<N. (N->P) -> Z -> N,
> Zero == All N. All P<N. All Z<N. (N->P) -> Z -> Z,
> Pos  == All N. All P<N. All Z<N. (N->P) -> Z -> P;

> check Zero < Nat;
Yes
> check Pos < Nat;
Yes
```

Aside from their types, elements of `Nat` are precisely the same as the corresponding elements of `OrigNat`:

```
> zero = \\N. \\P<N. \\Z<N. \s:(N->P). \z:Z. z,
> one  = \\N. \\P<N. \\Z<N. \s:(N->P). \z:Z. s z,
> two  = \\N. \\P<N. \\Z<N. \s:(N->P). \z:Z. s (s z);
zero : Zero
one : Pos
two : Pos
```

The successor function for ordinary church numerals takes a numeral `n` as argument and returns a new numeral that iterates `s` over `z` `n` times and then once more.

```
> origsucc = \n:OrigNat. \\N. \s:N->N. \z:N. s (n [N] s z);
origsucc : OrigNat -> OrigNat
```

Successor for our new encoding is exactly the same, except that we explicitly allow for the argument `n` to be of type `Zero` or `Pos`, in addition to `Nat`, and check the body separately for each case.

```
> succ = \n:Zero,Pos,Nat.
>           \\N. \\P<N. \\Z<N. \s:(N->P). \z:Z.
>              s (n [N] [P] [Z] s z);
succ : Nat -> Pos
```

The sum of original-style Church numerals `m` and `n` is obtained by iterating the successor function `m` times over `n`:

```
> origplus = \m:OrigNat. \n:OrigNat. m [OrigNat] origsucc n;
origplus : OrigNat -> OrigNat -> OrigNat
```

Again, addition of our numerals is exactly the same, except that we need to be more careful about the types.

```
> plus = for M in Zero,Pos,Nat. for N in Zero,Pos,Nat.
>          \m:M. \n:N.
>              m [Nat,Pos] [Pos] [N] succ n;
plus : Zero->Zero->Zero /\ Pos->Nat->Pos /\ Nat->Pos->Pos /\ Nat->Nat->Nat

> two = plus one one;
two : Pos
```

Multiplication and exponentiation of our numerals can be defined in the same way.

```
> times = for M in Zero,Pos,Nat. for N in Zero,Pos,Nat.
>          \m:M. \n:N.
>             m [Nat,Zero] [N] [Zero] (plus n) zero;
times : Zero->Nat->Zero /\ Pos->Pos->Pos /\ Nat->Zero->Zero /\ Nat->Nat->Nat
```

```
> exp = for M in Zero,Pos,Nat. for N in Zero,Pos,Nat.
>          \m:M. \n:N.
>             n [Nat,Pos] [M] [Pos] (times m) one;
exp : Zero->Pos->Zero /\ Pos->Nat->Pos /\ Nat->Zero->Pos /\ Nat->Nat->Nat
```

Defining the predecessor function on Church's original encoding was a significant feat in the early days of λ-calculus. To mimic it here, we first need pairing functions for numerals:

```
> ZeroZeroPr == All R. (Zero->Zero->R)->R,
> ZeroPosPr == All R. (Zero->Pos->R)->R,
> ZeroNatPr == All R. (Zero->Nat->R)->R,
> PosZeroPr == All R. (Pos->Zero->R)->R,
> PosPosPr == All R. (Pos->Pos->R)->R,
> PosNatPr == All R. (Pos->Nat->R)->R,
> NatZeroPr == All R. (Nat->Zero->R)->R,
> NatPosPr == All R. (Nat->Pos->R)->R,
> NatNatPr == All R. (Nat->Nat->R)->R;
```

```
> pair = for P1 in Zero,Pos,Nat.
>          for P2 in Zero,Pos,Nat.
>            \p1:P1. \p2:P2.
>               \\R. \f:P1->P2->R.
>                 f p1 p2;
pair : Zero->Zero->ZeroZeroPr
    /\ Zero->Pos->ZeroPosPr
    /\ Zero->Nat->ZeroNatPr
    /\ Pos->Zero->PosZeroPr
    /\ Pos->Pos->PosPosPr
    /\ Pos->Nat->PosNatPr
    /\ Nat->Zero->NatZeroPr
    /\ Nat->Pos->NatPosPr
    /\ Nat->Nat->NatNatPr
```

```
> fst = for P1 in Zero,Pos,Nat.
>          \p: (All R. (P1->T->R)->R).
>              p [P1] (\p1:P1. \p2:T. p1),
> snd = for P2 in Zero,Pos,Nat.
>          \p: (All R. (T->P2->R)->R).
>              p [P2] (\p1:T. \p2:P2. p2);
fst : (All R. (Zero->T->R)->R)->Zero
   /\ (All R. (Pos->T->R)->R)->Pos
   /\ (All R. (Nat->T->R)->R)->Nat
snd : (All R. (T->Zero->R)->R)->Zero
   /\ (All R. (T->Pos->R)->R)->Pos
   /\ (All R. (T->Nat->R)->R)->Nat
```

```
> pred = \n:Pos.
>          snd (n [NatNatPr] [PosNatPr] [ZeroZeroPr]
>               (\p:NatNatPr.
>                pair (succ (fst p)) (fst p))
>               (pair zero zero));
pred : Pos -> Nat
```

(For the sake of the example, this version of `pred` only takes positive numbers as arguments. Of course, by giving more possible types for the parameter `n`, we could allow `pred` to accept arbitrary natural numbers.)

There is another way — somewhat less well known — of encoding the basic arithmetic functions on Church numerals (see [120]):

```
> altorigplus =
>    \m:OrigNat. \n:OrigNat.
>       \\N. \s:N->N. \z:N.
>          m [N] s (n [N] s z);
altorigplus : OrigNat -> OrigNat -> OrigNat

> altorigmult =
>    \m:OrigNat. \n:OrigNat.
>        \\N. \s:N->N.
>            m [N] (n [N] s);
altorigmult : OrigNat -> OrigNat -> OrigNat

> altorigexp =
>    \m:OrigNat. \n:OrigNat.
>         \\N.
>            n [N->N] (m [N]);
altorigexp : OrigNat -> OrigNat -> OrigNat
```

This version of the arithmetic functions is interesting to try to emulate on our new encoding; the solution involves some fairly tricky use of the *for* construct. Also, the exponential function in this encoding requires iteration at higher types, which provides another good test of the limits of this encoding. (It may provide an even better test of the limits of the encoder. It took several hours to find a set of type annotations that would produce the desired typing for this version of the exponential function and it seems likely that this set of annotations is not the simplest available.)

```
> altplus =
>    \m:Nat,Zero,Pos. \n:Nat,Zero,Pos.
>        \\N. \\P<N. \\Z<N.
>            \s: N->P. \z:Z.
>                m [N,P] [P] [N,P,Z] s (n [N] [P] [Z] s z);
altplus : Nat->Nat->Nat
       /\ Nat->Pos->Pos
       /\ Zero->Zero->Zero
       /\ Pos->Nat->Pos
```

The cases for multiplication and exponentiation are similar, but slightly more complicated.

```
> altmult =
>    \m:Nat,Zero,Pos. \n:Nat,Zero,Pos.
>       \\N. \\P<N. \\Z<N.
>           \s: N->P.
>                m [N,Z] [N,P,Z] [Z]
>                    (n [N,Z] [N,P,Z] [N,P,Z] s);
altmult : Nat->Nat->Nat
       /\ Nat->Zero->Zero
       /\ Zero->Nat->Zero
       /\ Pos->Pos->Pos

> altexp =
>  \m:Nat,Zero,Pos. \n:Nat,Zero,Pos.
>    \\N. \\P<N. \\Z<N.
>       n [N->N,N->P,N->N/\Z->Z/\P->P]
>         [N->N,N->P,N->N/\Z->Z/\P->P]
```

```
>         [N->P,N->N/\Z->Z/\P->P]
>         (m [N,P,Z] [N,P,Z] [N,P,Z]);
altexp : Nat->Nat->(All N. All P<N. All Z<N. (N->P)->N->N)
      /\ Nat->Nat->(All N. All P<N. All Z<N. (N->N/\Z->Z/\P->P)->N->N)
      /\ Nat->Nat->(All N. All P<N. All Z<N. (N->N/\Z->Z/\P->P)->Z->Z)
      /\ Nat->Nat->(All N. All P<N. All Z<N. (N->N/\Z->Z/\P->P)->P->P)
      /\ Nat->Zero->(All N. All P<N. All Z<N. (N->P)->N->P)
      /\ Zero->Pos->Zero
      /\ Zero->Pos->(All N. All P<N. All Z<N. (N->P)->P->P)
      /\ Pos->Nat->(All N. All P<N. All Z<N. (N->P)->N->P)
```

The type for this version of `altexp` looks strange because it is actually smaller than we wanted. By η-expanding its body, we can force it to have only the familiar typing:

```
> altexp =
>  \m:Nat,Zero,Pos. \n:Nat,Zero,Pos.
>    \\N. \\P<N. \\Z<N.
>       \s:N->P. \z:Z.
>          n [N->N,N->P,N->N/\Z->Z/\P->P]
>            [N->N,N->P,N->N/\Z->Z/\P->P]
>            [N->P,N->N/\Z->Z/\P->P]
>            (m [N,P,Z] [N,P,Z] [N,P,Z])
>            s z;
altexp : Nat->Nat->Nat /\ Nat->Zero->Pos /\ Zero->Pos->Zero /\ Pos->Nat->Pos
```

The diagonalization of this exponential function is particularly interesting, since it involves a polymorphic self-application.

```
> diag = \n:Nat,Zero,Pos. altexp n n;
diag : Nat->Nat /\ Zero->Pos /\ Pos->Pos
```

7.7.2 Booleans

Similarly, the usual Church-encoding of the booleans can be refined to distinguish between `true` and `false`, obtaining typings similar to those of Section 7.5.

```
> True  == All B. All TT<B. All FF<B. TT  -> T -> TT,
> False == All B. All TT<B. All FF<B. T -> FF  -> FF,
> Bool  == All B. All TT<B. All FF<B. TT  -> FF  -> B;
```

```
> true = \\B. \\TT<B. \\FF<B. \x:TT. \y:T. x,
> false = \\B. \\TT<B. \\FF<B. \x:T. \y:FF. y;
true : True
false : False
```

```
> not = \m:True,False,Bool.
>          m [Bool] [False] [True] false true;
not : True->False /\ False->True /\ Bool->Bool
```

```
> or = for M in True,False,Bool.
>        for N in True,False,Bool.
>          \m:M. \n:N.
>             m [Bool] [True] [N] true n;
or : True->Bool->True
  /\ False->True->True
  /\ False->False->False
  /\ Bool->Bool->Bool
```

7.8 Observations on Programming with $F_\wedge$

There are two novel aspects of the style of programming explored in this chapter that, together, require new ways of thinking about the task of programming:

- The typechecker *never* fails outright. Since every parseable term can validly be given the type $\top$, the notion of a term being ill-typed does not make sense in this framework. Instead, we are forced to think in terms of a term having a minimal type that is larger than the one we expect or prefer.
- We often specify multiple typing assumptions for several bound variables or type applications. We do not usually expect that every combination of assumptions is going to lead to an interesting (non-$\top$) typing for the term, so it would be irritating to have the typechecker generate a warning when the best type for some subphrase is $\top$. (This would, however, be a good idea in the case where the subphrase is in the scope of only one set of possible assumptions.)

Consider the following, slightly incorrect, version of the `altplus` operator from Section 7.7.1:

```
> altplus =
>    for MM in Nat,Zero,Pos. for NN in Nat,Zero,Pos.
>      \m:MM. \n:NN.
>          \\N. \\P<N. \\Z<N.
>              \s: N->P. \z:Z.
>                  m [N,P] [P] [N,Z] s (n [N] [P] [Z] s z);
altplus : Nat->Nat->Nat /\ Zero->Zero->Zero /\ Pos->Nat->Pos
```

This expression has some of the types we expect, but it is missing `Nat->Pos->Pos`. To understand what is wrong, we need some way of gaining insight into what is happening under the specific set of assumptions where `MM` = `Nat` and `NN` = `Pos`.

One helpful tool that the present prototype implementation provides for this purpose is a *query operator* that has no effect on the typing or execution of terms, but that has the side effect during typechecking of printing out the minimal type that has been synthesized for its body under the current prevailing assumptions. We write query expressions as `?i:e`, where `e` is the body and `i` is an identifying tag used to distinguish output from this query from that generated by other queries. For example, here is the broken version of `altplus` with three queries added at salient points in its body:

```
> altplus =
>    for MM in Nat,Zero,Pos. for NN in Nat,Zero,Pos.
>      \m:MM. \n:NN.
>          \\N. \\P<N. \\Z<N.
>              \s: N->P. \z:Z.
>                   ?body:
>                     (?m: (m [N,P] [P] [N,Z]))
>                        s
>                        (?n: (n [N] [P] [Z] s z));
  MM=Nat, NN=Nat  => m: (N->P)->N->N
  MM=Nat, NN=Nat  => n: N
  MM=Nat, NN=Nat  => body: N
  MM=Nat, NN=Zero  => m: (N->P)->N->N
  MM=Nat, NN=Zero  => n: Z
  MM=Nat, NN=Zero  => body: N
  MM=Nat, NN=Pos  => m: (N->P)->N->N
  MM=Nat, NN=Pos  => n: P
```

```
  MM=Nat, NN=Pos  => body: N
  MM=Zero, NN=Nat  => m: (N->P)->N->N/\(N->P)->Z->Z
  MM=Zero, NN=Nat  => n: N
  MM=Zero, NN=Nat  => body: N
  MM=Zero, NN=Zero  => m: (N->P)->N->N/\(N->P)->Z->Z
  MM=Zero, NN=Zero  => n: Z
  MM=Zero, NN=Zero  => body: Z
  MM=Zero, NN=Pos  => m: (N->P)->N->N/\(N->P)->Z->Z
  MM=Zero, NN=Pos  => n: P
  MM=Zero, NN=Pos  => body: N
  MM=Pos, NN=Nat  => m: (N->P)->N->P
  MM=Pos, NN=Nat  => n: N
  MM=Pos, NN=Nat  => body: P
  MM=Pos, NN=Zero  => m: (N->P)->N->P
  MM=Pos, NN=Zero  => n: Z
  MM=Pos, NN=Zero  => body: P
  MM=Pos, NN=Pos  => m: (N->P)->N->P
  MM=Pos, NN=Pos  => n: P
  MM=Pos, NN=Pos  => body: P
altplus : Nat->Nat->Nat /\ Zero->Zero->Zero /\ Pos->Nat->Pos
```

From this trace, it is clear what the problem is: when `MM` is `Nat` and `NN` is `Pos`, the type of `m`'s application to the three following types yields a function mapping a successor function and an element of `P` (the type of `n`'s application to its type arguments) to `N`, not to `P` as we expected. Since the relation between the type of the phrase marked `m` and the type of its final argument depends on the third type application, we are led to try adding an annotation that will cause the case where this argument is `P` to be considered separately. As we saw in Section 7.7.1, this is enough to get the desired typing for `altplus`:

```
> altplus =
>    for MM in Nat,Zero,Pos. for NN in Nat,Zero,Pos.
>      \m:MM. \n:NN.
>          \\N. \\P<N. \\Z<N.
>              \s: N->P. \z:Z.
>                   ?body:
>                     (?m: (m [N,P] [P] [N,Z,P]))
>                        s
>                        (?n: (n [N] [P] [Z] s z));
  MM=Nat, NN=Nat  => m: (N->P)->N->N/\(P->P)->P->P
  MM=Nat, NN=Nat  => n: N
  MM=Nat, NN=Nat  => body: N
  MM=Nat, NN=Zero  => m: (N->P)->N->N/\(P->P)->P->P
  MM=Nat, NN=Zero  => n: Z
  MM=Nat, NN=Zero  => body: N
  MM=Nat, NN=Pos  => m: (N->P)->N->N/\(P->P)->P->P
  MM=Nat, NN=Pos  => n: P
  MM=Nat, NN=Pos  => body: P
  MM=Zero, NN=Nat  => m: (N->P)->N->N/\(N->P)->Z->Z/\(P->P)->P->P
  MM=Zero, NN=Nat  => n: N
  MM=Zero, NN=Nat  => body: N
  MM=Zero, NN=Zero  => m: (N->P)->N->N/\(N->P)->Z->Z/\(P->P)->P->P
  MM=Zero, NN=Zero  => n: Z
  MM=Zero, NN=Zero  => body: Z
  MM=Zero, NN=Pos  => m: (N->P)->N->N/\(N->P)->Z->Z/\(P->P)->P->P
```

```
  MM=Zero, NN=Pos  => n: P
  MM=Zero, NN=Pos  => body: P
  MM=Pos, NN=Nat  => m: (N->P)->N->P/\(P->P)->P->P
  MM=Pos, NN=Nat  => n: N
  MM=Pos, NN=Nat  => body: P
  MM=Pos, NN=Zero  => m: (N->P)->N->P/\(P->P)->P->P
  MM=Pos, NN=Zero  => n: Z
  MM=Pos, NN=Zero  => body: P
  MM=Pos, NN=Pos  => m: (N->P)->N->P/\(P->P)->P->P
  MM=Pos, NN=Pos  => n: P
  MM=Pos, NN=Pos  => body: P
altplus : Nat->Nat->Nat
       /\ Nat->Pos->Pos
       /\ Zero->Zero->Zero
       /\ Pos->Nat->Pos
```

7.9 An Experiment with a Simpler Formulation of $F_\wedge$

In Section 3.5 we mentioned a trick for "encoding" bounded quantification in a system with intersections and pure (unbounded) second-order polymorphism:

$$\forall\alpha{\leq}\sigma.\ \tau \quad \stackrel{\text{def}}{=} \quad \forall\alpha.\ \{\sigma{\wedge}\alpha/\alpha\}\tau$$

This is not an encoding in the true sense; for example, it does not validate the Sub-All rule. Still, since it mimics something like bounded quantification in a significantly simpler system, it is worth exploring the limits of this technique. Here are some of the more complex examples from Section 7.7.1, with the translation applied by hand:

```
> Nat  == All N. All P. All Z. (N->(P/\N)) -> (Z/\N) -> N,
> Zero == All N. All P. All Z. (N->(P/\N)) -> (Z/\N) -> (Z/\N),
> Pos  == All N. All P. All Z. (N->(P/\N)) -> (Z/\N) -> (P/\N);

> check Zero < Nat;
Yes
> check Pos < Nat;
Yes

> zero = \\N. \\P. \\Z. \s:(N->(P/\N)). \z:(Z/\N). z,
> one  = \\N. \\P. \\Z. \s:(N->(P/\N)). \z:(Z/\N). s z,
> two  = \\N. \\P. \\Z. \s:(N->(P/\N)). \z:(Z/\N). s (s z);
zero : All N. All P. All Z. (N->P/\N->N)->(Z/\N)->Z /\ Nat
one : All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P /\ Nat
two : All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P /\ Nat
```

Note that these types are equivalent to `zero : Zero`, `one : Pos`, etc:

```
> check (All N. All P. All Z. (N->P/\N->N)->(Z/\N)->Z) /\ Nat < Zero;
Yes
> check Zero < (All N. All P. All Z. (N->P/\N->N)->(Z/\N)->Z) /\ Nat;
Yes
> check (All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P) /\ Nat < Pos;
Yes
> check Pos < (All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P) /\ Nat;
Yes
```

```
> succ = \n:Zero,Pos,Nat.
>           \\N. \\P. \\Z. \s:(N->(P/\N)). \z:(Z/\N).
>              s (n [N] [(P/\N)] [(Z/\N)] s z);
succ : Nat->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P) /\ Nat->Nat

> plus = for M in Zero,Pos,Nat. for N in Zero,Pos,Nat.
>          \m:M. \n:N.
>              m [Nat,Pos] [Pos] [N] succ n;
plus : Zero->Zero->Zero /\ Pos->Nat->Pos /\ Nat->Pos->Pos /\ Nat->Nat->Nat

> times = for M in Zero,Pos,Nat. for N in Zero,Pos,Nat.
>          \m:M. \n:N.
>             m [Nat,Zero] [N] [Zero] (plus n) zero;
times : Zero->Nat->Zero /\ Pos->Pos->Pos /\ Nat->Zero->Zero /\ Nat->Nat->Nat

> exp = for M in Zero,Pos,Nat. for N in Zero,Pos,Nat.
>          \m:M. \n:N.
>             n [Nat,Pos] [M] [Pos] (times m) one;
exp : Zero->Pos->Zero /\ Pos->Nat->Pos /\ Nat->Zero->Pos /\ Nat->Nat->Nat

> altplus =
>    \m:Nat,Zero,Pos. \n:Nat,Zero,Pos.
>        \\N. \\P. \\Z.
>            \s: N->(P/\N). \z:(Z/\N).
>                m [N,(P/\N)] [(P/\N)] [N,(P/\N),(Z/\N)]
>                  s (n [N] [(P/\N)] [(Z/\N)] s z);
altplus : Nat->Nat->Nat
       /\ Nat->Pos->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P)
       /\ Zero->Zero->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->Z)
       /\ Pos->Nat->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P)
```

Again, note that this type is equivalent to the expected type for `altplus`:

```
> check
>     Nat->Nat->Nat
>      /\ Nat->Pos->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P)
>      /\ Zero->Zero->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->Z)
>      /\ Pos->Nat->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P)
>  <  Nat->Nat->Nat
>      /\ Nat->Pos->Pos
>      /\ Pos->Nat->Pos
>      /\ Zero->Zero->Zero;
Yes
> check
>     Nat->Nat->Nat
>      /\ Nat->Pos->Pos
>      /\ Pos->Nat->Pos
>      /\ Zero->Zero->Zero
>  <  Nat->Nat->Nat
>      /\ Nat->Pos->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P)
>      /\ Zero->Zero->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->Z)
>      /\ Pos->Nat->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P);
Yes

> altmult =
>    \m:Nat,Zero,Pos. \n:Nat,Zero,Pos.
```

```
>       \\N. \\P. \\Z.
>           \s: N->(P/\N).
>                m [N,(Z/\N)] [N,(P/\N),(Z/\N)] [(Z/\N)]
>                    (n [N,(Z/\N)] [N,(P/\N),(Z/\N)] [N,(P/\N),(Z/\N)] s);
altmult : Nat->Nat->Nat
       /\ Nat->Zero->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->Z)
       /\ Zero->Nat->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->Z)
       /\ Pos->Pos->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P)

> altexp =
>  \m:Nat,Zero,Pos. \n:Nat,Zero,Pos.
>    \\N. \\P. \\Z.
>       \s:N->(P/\N). \z:(Z/\N).
>          n [N->N,N->(P/\N),N->N/\(Z/\N)->(Z/\N)/\(P/\N)->(P/\N)]
>            [N->N,N->(P/\N),N->N/\(Z/\N)->(Z/\N)/\(P/\N)->(P/\N)]
>            [N->(P/\N),N->N/\(Z/\N)->(Z/\N)/\(P/\N)->(P/\N)]
>            (m [N,(P/\N),(Z/\N)] [N,(P/\N),(Z/\N)] [N,(P/\N),(Z/\N)])
>            s z;
altexp : Nat->Nat->Nat
      /\ Nat->Zero->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P)
      /\ Zero->Pos->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->Z)
      /\ Pos->Nat->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P)

> diag = \n:Nat,Zero,Pos. altexp n n;
diag : Nat->Nat
    /\ Zero->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P)
    /\ Pos->(All N. All P. All Z. (N->P/\N->N)->(Z/\N)->P)
```

Our preliminary conclusion from these examples is that the encoding trick works better than might be expected. Its main apparent defect is that it disrupts our notion of typed semantics, since it replaces the type $\forall\alpha{\leq}\sigma.\ \tau$, which, roughly speaking, describes a function expecting a type and a coercion from this type into σ, by the type $\forall\alpha.\ \{\sigma{\wedge}\alpha/\alpha\}\tau$, which describes a function expecting any type whatsoever, but requiring additional proof that whatever elements of this type are actually used can also be coerced to type σ.

Chapter 8

Evaluation and Future Work

Having presented our results in detail, we conclude by evaluating them in terms of the goals articulated in the introductory chapter and suggesting some likely paths for future research.

The naturalness and formal power of the $F_\wedge$ calculus seem well established. Is is based on three elegant and appealing notions of typing — subtyping, finitary polymorphism, and bounded parametric polymorphism — and combines them nearly orthogonally so that the programming idioms of all the components are fully supported. When the two styles of polymorphism are used together, a fascinating new class of encodings of algebraic datatypes arises (Section 7.7).

The thesis presents both positive and negative results about the system's tractability. On the positive side, we have proofs of the partial correctness and of simple algorithms for checking the subtype relation and for computing minimal types for programs (Chapter 4). We also have a simple untyped semantic model (Section 5.1), a natural framework for typed models based on a translation into system F (Section 5.3), and a preliminary equational theory (Section 5.5). On the negative side is the discovery that the subtype relation of $F_\wedge$ lacks least upper bounds (Section 5.2), which blocks some of the known methods of semantic analysis and entails significant complication for future efforts along these lines. The observation that the subtype relation is, in fact, undecidable (Chapter 6), though not overly worrysome in practical terms, is further evidence for the underlying complexity that $F_\wedge$ inherits from $F_{\leq}$.

The difficulty of analyzing $F_\wedge$, together with the possibility that some of the programming idioms arising from pure bounded quantification may have useful analogues in languages with intersection types and only unbounded quantification (Section 7.9), suggests, in fact, that $F_\wedge$ may be *too* powerful, and that future investigations might profitably focus on simpler fragments instead of treating the whole calculus.

The suitability of $F_\wedge$ as a basis for language designs is partially, but incompletely, demonstrated by the work described here.

The prototype implementation used to typeset the examples throughout the thesis establishes the viability of the naive algorithms described in Chapter 4 for small examples and suggests numerous convenient syntactic sugarings, programming techniques, and debugging tools (Section 7.8). Sections 7.2, 7.3, and 7.4 present some experiments with language features using combinations of finitary and parametric polymorphism. Sections 7.5 and 7.7 explore a novel style of programming where intersections are used to obtain typings similar to the results of conventional abstract interpretation, and Section 7.6 hints at a similar treatment of strictness analysis. Among these examples, however, only those in the section discussing extensions of Forsythe (7.2) could be called "practical." To fully justify $F_\wedge$ as a core type system for programming languages, a much larger suite of examples illustrating its application to real programming problems would be required. In

particular, the set of examples given here lacks convincing evidence that bounded polymorphism is more useful than ordinary unbounded polymorphism. (This is not surprising, since the standard examples using bounded polymorphism rely on the presence of recursive types or updateable record types, or both.)

Another concern raised by the prototype implementation is the practical efficiency of typechecking for larger examples. Naive implementations of the simple algorithms in Chapter 4 exhibit exponential behavior — in practice — in both type synthesis (because of the *for* construct) and subtyping (because of rules ASubR-Inter and ASubL-Inter in Definition 4.2.8.4). Fortunately, this behavior normally occurs as a result of explicit programmer directives — requests, in effect, for an exponential amount of analysis of the program during typechecking. Still, a serious typechecker implementation would need to find ways to save some of this cost by caching the partial results of previous analysis. The exact form of the typing derivations constructed by the type analyzer can also have significant effects on the code generation phase of an implementation based on something like the translation semantics given in Section 5.3.3, giving rise to a whole collection of practical issues not considered here.

A third practical consideration for any language based on second-order polymorphism is the problem of verbosity. Without some means of abbreviation (partial type reconstruction), even modest programs quickly become overburdened with type abstractions and applications and long type annotations on λ-abstractions. We have chosen to ignore this set of issues here, since it is not yet well understood even for pure polymorphic λ-calculi without subtyping, but an eventual full-scale language design based on $F_\wedge$ would need to face it somehow.

With these remarks in mind, we now discuss some areas where future research might fruitfully extend or complement the work described in this thesis.

8.1 Alternative Formulations

As we mentioned in the Introduction and in Section 3.5, the $F_\wedge$ calculus is just one representative of a whole space of calculi combining some form of polymorphism with some presentation of intersection types.

Intersection types allow for relatively few degrees of freedom. Besides the version given here, which slightly generalizes the core type system of Forsythe, there are three variants that may be worth considering in more detail:

- The calculus identical to $\lambda_\wedge$ but without the type $\top$ — i.e., where n is required to be at least 1 in every rule involving an intersection — seems less elegant than $\lambda_\wedge$ itself. However, it has the possible practical advantage that it retains a conventional notion of typechecking failure, since it provides no type that can be assigned to every phrase. Combined with ordinary $F_\leq$ (with $F_\leq$'s rules for the *Top* type), this version of intersection types might provide much of the expressiveness that we have demonstrated here, without requiring such a radical change in the notion of well-typedness.
- The distributivity laws of $\lambda_\wedge$ could perhaps also be dropped without greatly affecting expressiveness. Our guess, however, is that this restriction would make types much more clumsy to manipulate. For example, many of the simplifications performed by the prototype implementation before printing the type inferred for an expression would be blocked by this restriction.

- A much more severe restriction on the use of intersection types would be to prevent them from appearing on the left-hand sides of arrows. This would greatly reduce their expressive power, perhaps bringing them within the reach of conventional type reconstruction techniques.

Polymorphism has been studied in many different forms. The one used in this thesis is among the most powerful, combining full second-order quantification over types with a notion of quantification over a collection of types determined by the subtype ordering. Many of the others, though, are possible candidates for integration with intersection types.

- Generalizing our results about $F_\wedge$ to systems based on F-bounded polymorphism [18, 39] or ω-order polymorphism [66, 102, 62, 108] would seem to be a straightforward process.
- Versions of $F_\leq$ with stronger subtyping rules (c.f. 3.5) can also be combined with intersection types. These combinations are of dubious value as bases for practical programming languages, since they have typechecking problems that seem to be of similar difficulty to the full type reconstruction problem for the polymorphic λ-calculus, but they may be suitable foundations for more theoretical investigations. (See [89], for example.)
- Replacing $F_\wedge$'s quantifier subtyping rule with the weaker "equal-bounds" rule of Cardelli and Wegner's original Fun [33]

$$\frac{\Gamma, \alpha{\leq}\theta \vdash \sigma \leq \tau}{\Gamma \vdash \forall\alpha{\leq}\theta.\ \sigma \leq \forall\alpha{\leq}\theta.\ \tau} \qquad \text{(Sub-All-Eq)}$$

  yields a decidable system. This rule is hard to justify semantically, however.
- Languages with prenex (ML-style) polymorphism [92, 55] have been investigated quite thoroughly, but we are not aware of a formulation of prenex bounded quantification and, in general, the work of adding subtyping to the ML type system in such a way as to retain its crucial properties — especially decidable inference of principal types — is less developed than the study of second-order type systems with subtyping.
- The ordinary polymorphic λ-calculus (system F) can also be extended with a subtype relation. When this calculus is combined with intersection types, some of the behavior of the bounded quantifier can be recovered using intersections (c.f. Section 7.9). Although more investigation is needed to determine the limitations of this trick, the proof theory and semantics of this combination are likely to be so much simpler than those of $F_\wedge$ that it seems an excellent avenue to pursue.

8.2 Foundations

The most significant unfinished aspect of our theoretical study of $F_\wedge$ is the investigation of its typed semantics. We gave, in Chapter 5, two partial approaches to this question: a translation from $F_\wedge$ typing derivations into the pure polymorphic λ-calculus, and an equational characterization of equalities between $F_\wedge$ terms. However, we were unable to show that the translation semantics was coherent. Here we sketch some other approaches to the semantics of the calculus and some possible methods by which the coherence of the translation semantics might be established.

8.2.1 Semantics

The partial equivalence relation model of $F_\wedge$ given in Section 5.1 is a simple extension of Bruce and Longo's PER model of $F_\leq$ [12]. Our presentation, however, was much more elementary than theirs, which began by giving a general definition of an *environment model* of $F_\leq$ (extending Bruce, Meyer, and Mitchell's familiar notion of a *second-order environment model* [13] for the polymorphic λ-calculus) and then showed how an instance of this framework could be constructed in the category of ω-sets (c.f. [3]). This general construction could presumably also be extended to an environment model for $F_\wedge$. PER models for $F_\wedge$ may also arise from Bruce and Mitchell's work on models of $F_\leq$ extended with recursive types [14].

A more general *categorical semantics* for $F_\wedge$ along the lines of Seely's semantics for system F [129] would have to rest on a categorical semantics for $F_\leq$ — currently an open problem.

A different view of $F_\wedge$'s semantics might come from a complete equational theory — an extension of the rules in Section 5.5 with the additional property that they characterize *all* the valid equivalences between terms with respect to some class of models.

8.2.2 Coherence

In Section 5.4 we stated the following conjecture for the translation semantics of $F_\wedge$:

(5.4.2). Conjecture: [Coherence of typing] If $s :: \Gamma \vdash^\wedge e \in \tau$ and $t :: \Gamma \vdash^\wedge e \in \tau$, then $[\![\Gamma]\!] \vdash^F [\![s]\!] = [\![t]\!] \in [\![\tau]\!]$.

Two general methods are known for establishing conjectures of this sort. One, formalized most cleanly by Curien and Ghelli [50] (also see [63, 10]), has been applied successfully to second-order bounded quantification. The other, due to Reynolds [123], works for first-order intersection types. The extension of either to $F_\wedge$ is problematic.

Before applying either method, the translation semantics should be slightly refined. Rather than interpreting subtyping derivations directly as terms of $F_\times$, they should be interpreted as combinations of a set of *coercion combinators*, which capture the notion of a semantic subcategory of coercions. (See [10].) This refinement of $F_\wedge$ is straightforward.

Curien and Ghelli's method is based on a derivation normalization argument for typing derivations similar to the one given for canonical subtyping in Section 4.2. In outline, the argument proceeds as follows:

- A set of rules is given for rewriting derivations into a standard normal form.
- A terminating rewriting strategy for these rules is exhibited.
- The set of normal forms is shown to be sufficiently restricted that there is at most one normal-form derivation with any given conclusion.
- Each of the rewriting rules is shown to be "locally coherent" with respect to the given semantic interpretation of derivations: if $s \longrightarrow_1 t$ then $[\![s]\!] = [\![t]\!]$.
- Given two derivations with the same conclusion, the termination of the rewrite rules and the unicity of normal forms guarantee that both can be rewritten to the same normal form. The local confluence of the rewriting rules then establishes the equality of the interpretations of the original derivations.

The main difficulty with extending this approach to $F_\wedge$ is that it is not clear how to write normalization rules for typing derivations that rewrite any derivation into a unique normal form. For example, if $f \in (Int \to Char) \wedge (Bool \to Char)$ and $v \in Int \wedge Bool$, then the term

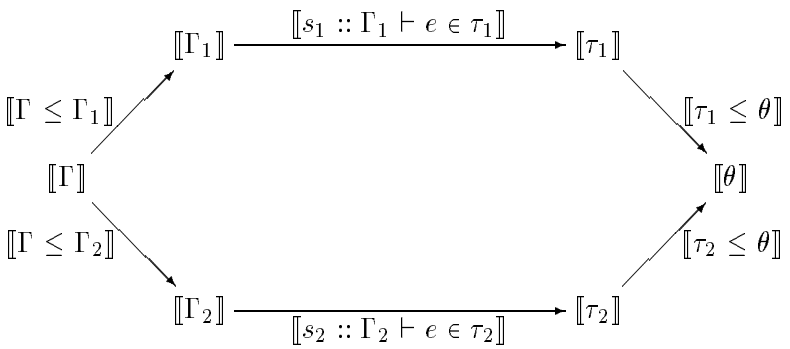
$$(for\ \alpha\ in\ Int, Bool.\ \lambda x{:}\alpha.\ f\ x)\ v$$

can be given the type *Char* in at least two different ways — one using the substitution $\{Int/\alpha\}$ and another using $\{Bool/\alpha\}$. The first derivation contains no subderivation for the term $\lambda x{:}Bool.\ f\ x$, and the second contains no subderivation for $\lambda x{:}Int.\ f\ x$. So in order to rewrite both of them into a common derivation, a whole new subderivation would need to be created "on the fly" by the rewriting rules. This does not seem impossible, but it is certainly more difficult than the task accomplished by Curien and Ghelli's rules, which can simply rearrange the existing structure of derivations.

Reynolds' method for proving coherence is based on a category-theoretic presentation of the semantics of $\lambda_\wedge$ in which intersections are interpreted as limits. The interpretation of a derivation $\Gamma \vdash e \in \tau$ is a morphism $[\![\Gamma \vdash e \in \tau]\!] \in [\![\Gamma]\!] \to [\![\tau]\!]$, where $[\![x_1{:}\tau_1,\ \ldots x_n{:}\tau_n]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$. Proving coherence in this presentation amounts to establishing the commutativity of all diagrams of the following form:

$$[\![\Gamma]\!] \underset{[\![s_2 :: \Gamma \vdash e \in \tau]\!]}{\overset{[\![s_1 :: \Gamma \vdash e \in \tau]\!]}{\rightrightarrows}} [\![\tau]\!]$$

The proof actually requires a stronger induction hypothesis, the commutativity of every diagram of the following form,



which can be established by simultaneous induction on s_1 and s_2.

When the final steps of both derivations are applications of syntax-directed rules such as Arrow-I, the induction hypothesis is used together with properties of the the model (such as cartesian closure) to obtain the desired result.

For the non-syntax-directed rules Sub and Inter-I, the proof depends on two crucial properties. When the last rule of one of the derivations is Sub, the result follows from the coherence of subtyping (which in category-theoretic terms, can be stated more simply as "the function $[\![—]\!]$ from types to objects of the semantic category and from subtyping derivations to coercion morphisms is a functor"). A proof of this property for $F_\wedge$ using an extension of Curien and Ghelli's method appears to be messy but fairly straightforward. The crux of the proof is an analog of the subtyping-derivation-normalization argument in Section 4.2, where types are left in their ordinary form instead of being flattened to canonical types.

The second property needed for Reynolds' proof, unfortunately, is the existence of least upper bounds in the subtype relation, which we showed fails for $F_\wedge$. This is used in the case where one of the two typing derivations ends with rule Inter-I, to "glue together" the diagrams obtained by applying the induction hypothesis to the subderivations.

It is conceivable that this proof technique could be extended to $F_\wedge$ by strengthening the induction hypothesis again to consider *all* of the supertypes of τ_1 and τ_2 simultaneously, rather than just a single given τ.

8.3 Extensions

In addition to more tractable fragments of $F_\wedge$, there are several important extensions that should be considered.

8.3.1 Records

To model more of the features of full-scale object-oriented programming languages — an even to allow useful programming in more conventional idioms — it is critical that we consider extending $F_\wedge$ with a flexible facility for record manipulation.

In Forsythe, Reynolds introduced the following elegant treatment of records. Let $I = \{\iota_1, \iota_2, \ldots\}$ be a set of *labels*. For each $\iota \in I$, introduce a type constructor "ι:" describing the set of single-field records with label ι. Next, introduce a field selection operator "$.\iota$" for each i. If $\Gamma \vdash v \in \iota{:}\tau$ then $\Gamma \vdash v.\iota \in \tau$. Multi-field records can now be built from these primitives: instead of $\{\iota_1{:}\tau_1, \iota_2, \tau_2\}$, we write $(\iota_1{:}\tau_1) \wedge (\iota_2{:}\tau_2)$. Lastly, we need a way of building new records. Forsythe uses the construct "v *with* $\iota \equiv w$" to denote a value with all the same fields as v, but with v's ι field, if any, replaced with $\iota = w$. The type of *with* can be stated in terms of an operator "$\backslash\iota$," which removes any existing ι fields from a given type:

$$\frac{\Gamma \vdash e \in \tau \qquad \Gamma \vdash e' \in \tau'}{\Gamma \vdash e \textit{ with } \iota \equiv e' \in (\tau \backslash \iota) \wedge (\iota, \tau')}$$

where

$$\begin{array}{lcl}
\rho\backslash\iota & = & \rho \\
(\bigwedge[\tau_1 .. \tau_n])\backslash\iota & = & \bigwedge[(\tau_1\backslash\iota) .. (\tau_n\backslash\iota)] \\
(\tau_1 \to \tau_2)\backslash\iota & = & \tau_1 \to \tau_2 \\
(\iota'{:}\tau)\backslash\iota & = & \iota'{:}\tau \quad \text{when } \iota \not\equiv \iota' \\
(\iota{:}\tau)\backslash\iota & = & \top.
\end{array}$$

The principal difficulty with adding this treatment of records to $F_\wedge$ is that there is no way to define the behavior of the operation $\backslash\iota$ applied to a type variable α: we cannot tell from the shape of α itself whether it will later be instantiated with a type containing an ι field, and even if we could, the current language of types gives us no way of "remembering" to remove the ι field when this instantiation actually occurs. The $\backslash\iota$ operation must be introduced as a new *constructor* in the language of types:

$$\begin{array}{lcl}
\tau & ::= & \alpha \\
& | & \tau_1 \to \tau_2 \\
& | & \forall\alpha{\leq}\tau_1.\ \tau_2 \\
& | & \bigwedge[\tau_1 .. \tau_n] \\
& | & \iota{:}\tau \\
& | & \tau\backslash\iota
\end{array}$$

New rules must be given for the behavior of $\backslash\iota$ and its interaction with the other constructors, and a new typechecking algorithm must be given and proved correct.

Luckily, the $\backslash\iota$ constructor has been extensively studied in recent years — albeit for languages without intersection types [112, 135, 82, 83, 113, 114, 32, 31, 71, 70, 115]. We hope that existing intuitions and techniques can be extended to $F_\wedge$ straightforwardly.

8.3.2 Recursive Types

Another extension of particular importance for $F_\wedge$'s role in modeling object-oriented languages is *recursive types*. Again, a great deal has been learned recently about calculi with recursive types and subtyping [2]. But previous work has focused on systems with substantially simpler subtype relations; there is little reason to believe that extending existing techniques will be straightforward.

8.3.3 Union Types

Having studied the properties of a type system whose subtype relation is closed under finite meets, it is natural to consider introducing finite joins as well. In practical terms, the main effect of this extension is that we gain the ability to express, say, that the type *Bool* is completely partitioned by *True* and *False* (as opposed to knowing only that *True* and *False* are both contained in *Bool*).

Calculi incorporating various formulations of this notion have been proposed by the present author [107] and a number of other researchers [4, 60, 122, 36, 59, 73, 72], but their practicality and tractability remain unclear.

A related extension of $F_\wedge$ arises from "dualizing" the upper bound of the bounded quantifier so that each variable is introduced with both an upper and a lower bound: $\forall\sigma_1{\leq}\alpha{\leq}\sigma_2.\ \tau$. A fragment of this calculus with double-bounded variables but no quantification (the bounds on variables are given in advance and no mechanism is provided for extending the context) is shown to be decidable in [104].

8.3.4 Type Reconstruction

In order for languages based on second-order polymorphism to be usable on a large scale, some form of partial type reconstruction is a critical requirement. Though satisfactory algorithms exist for the pure system F and its higher-order variants [102, 9, 109], there has been little progress to date on extending these ideas to calculi with subtyping (see, however, [27]).

Less critical in practical terms, but intriguing, is the possibility of integrating polymorphic type reconstruction with a known semi-algorithm for intersection type inference [125].

8.4 Implementation

Our prototype implementation of $F_\wedge$ uses some slight extensions of the algorithms we analyzed in Chapter 4. We perform the type substitutions introduced by the FOR rule lazily by storing them in the context rather than inserting them in the term. This mechanism also provides for transparent type abbreviations that are somewhat more efficient than their most naive implementation (simple replacement by the parser) would suggest. Our data structure for types, which is based on DeBruijn indexing [56], is also implemented lazily; instead of renumbering the indices of the free variables in a type when it is extracted from a context, the extracted type is reindexed, incrementally, as needed. (Related schemes for lazy implementations of the data structures used in typechecking have been studied by Abadi, Cardelli, Curien, and Lévy [1].)

These refinements substantially improve the speed of the implementation, compared to a naive transcription of the typechecking algorithms. But some much more serious efficiency issues remain to be addressed. These have to do primarily with the exponential behavior of the typechecker in situations where the programmer has requested that some part of a program be checked under many different sets of assumptions. To some degree, this exponential behavior is justified, since the programmer has asked for it and since it can be shown [Reynolds, personal communication,

1990] that there are $F_\wedge$ programs for which an exponential amount of work must be expended to discover their minimal types. Still, the compiler implementor must try to make common cases as inexpensive as possible.

The most promising technique for accomplishing this is some form of *memoization* or *caching* of previous partial results of subtyping and typechecking. For example, if α does not appear free in e_1, then the type of e_1 should only be analyzed once during the analysis of *for* α *in* $\sigma_1..\sigma_n$. $e_1\ e_2$. Of course, determining that α is not free in e_1 may itself require some work; if we are not careful, we will spend more time discovering that we've already computed and cached a type for e_1 than we would spend computing it over again from scratch. Both careful tuning of the data structures used for caching and careful performance measurements will be crucial to the success of this sort of improvement.

Another class of issues that we have dealt with only superficially concerns the structure of efficient code generators based on our typed semantics of $F_\wedge$. (Compilation based on the untyped semantics is less problematic.) One of the largest of these is the sensitivity of the generated code to the specific shapes of typing derivations. It will be important to consider alternative formulations of the typing and (especially) the subtyping rules that give rise to efficient translations. Some compile-time proof normalization to eliminate useless coercions also seems necessary.

It may also become important, in practical terms, to try to distinguish the "real" overloading of values like + from the "typechecking only" overloading associated with our examples of abstract interpretation, strictness analysis, and so on. This would amount to taking a hybrid view of semantics, allowing some intersections to be interpreted as intersections in the semantics while others were interpreted as coherent tuples. It might be interesting to try to reflect this distinction in the syntax of the language by introducing two different kinds of intersection types (with a coercion from the untyped to the typed variety).

Appendix A

Summary of Major Definitions

A.1 $F_\wedge$

A.1.1 Subtyping

$$\Gamma \vdash \tau \leq \tau \qquad \text{(Sub-Refl)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \tau_2 \qquad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3} \qquad \text{(Sub-Trans)}$$

$$\Gamma \vdash \alpha \leq \Gamma(\alpha) \qquad \text{(Sub-TVar)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2} \qquad \text{(Sub-Arrow)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma, \alpha \leq \tau_1 \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \forall \alpha \leq \sigma_1.\ \sigma_2 \leq \forall \alpha \leq \tau_1.\ \tau_2} \qquad \text{(Sub-All)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash \sigma \leq \tau_i}{\Gamma \vdash \sigma \leq \bigwedge[\tau_1..\tau_n]} \qquad \text{(Sub-Inter-G)}$$

$$\Gamma \vdash \bigwedge[\tau_1..\tau_n] \leq \tau_i \qquad \text{(Sub-Inter-LB)}$$

$$\Gamma \vdash \bigwedge[\sigma \to \tau_1\ ..\ \sigma \to \tau_n] \leq \sigma \to \bigwedge[\tau_1..\tau_n] \qquad \text{(Sub-Dist-IA)}$$

$$\Gamma \vdash \bigwedge[\forall \alpha \leq \sigma.\ \tau_1\ ..\ \forall \alpha \leq \sigma.\ \tau_n] \leq \forall \alpha \leq \sigma.\ \bigwedge[\tau_1..\tau_n] \qquad \text{(Sub-Dist-IQ)}$$

A.1.2 Typing

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(Var)}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ e \in \tau_1 \to \tau_2} \qquad \text{(Arrow-I)}$$

$$\frac{\Gamma \vdash e_1 \in \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash e_1\ e_2 \in \tau_2} \qquad \text{(Arrow-E)}$$

$$\frac{\Gamma, \alpha \leq \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \Lambda \alpha \leq \tau_1.\ e \in \forall \alpha \leq \tau_1.\ \tau_2} \qquad \text{(All-I)}$$

$$\frac{\Gamma \vdash e \in \forall \alpha \leq \tau_1.\ \tau_2 \qquad \Gamma \vdash \tau \leq \tau_1}{\Gamma \vdash e[\tau] \in \{\tau/\alpha\}\tau_2} \qquad \text{(All-E)}$$

$$\frac{\Gamma \vdash \{\sigma_i/\alpha\}e \in \tau_i}{\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e \in \tau_i} \quad (\textsc{For})$$

$$\frac{\text{for all } i,\ \Gamma \vdash e \in \tau_i}{\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]} \quad (\textsc{Inter-I})$$

$$\frac{\Gamma \vdash e \in \tau_1 \qquad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e \in \tau_2} \quad (\textsc{Sub})$$

A.1.3 Syntax-Directed Subtyping

$$\frac{\text{for all } i,\ \Gamma \vdash \sigma \leq X \Rightarrow \tau_i}{\Gamma \vdash \sigma \leq X \Rightarrow \bigwedge[\tau_1..\tau_n]} \quad (\textsc{ASubR-Inter})$$

$$\frac{\text{for some } i,\ \Gamma \vdash \sigma_i \leq X \Rightarrow \alpha}{\Gamma \vdash \bigwedge[\sigma_1..\sigma_n] \leq X \Rightarrow \alpha} \quad (\textsc{ASubL-Inter})$$

$$\frac{\Gamma \vdash \tau_1 \leq [\,] \Rightarrow \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq X_2 \Rightarrow \alpha}{\Gamma \vdash \sigma_1 \to \sigma_2 \leq [\tau_1, X_2] \Rightarrow \alpha} \quad (\textsc{ASubL-Arrow})$$

$$\frac{\Gamma \vdash \tau_1 \leq [\,] \Rightarrow \sigma_1 \qquad \Gamma, \beta \leq \tau_1 \vdash \sigma_2 \leq X_2 \Rightarrow \alpha}{\Gamma \vdash \forall \beta \leq \sigma_1.\ \sigma_2 \leq [\beta \leq \tau_1, X_2] \Rightarrow \alpha} \quad (\textsc{ASubL-All})$$

$$\Gamma \vdash \alpha \leq [\,] \Rightarrow \alpha \quad (\textsc{ASubL-Refl})$$

$$\frac{\Gamma \vdash \Gamma(\beta) \leq X \Rightarrow \alpha}{\Gamma \vdash \beta \leq X \Rightarrow \alpha} \quad (\textsc{ASubL-TVar})$$

A.1.4 Type Synthesis

$$\Gamma \vdash x \in \Gamma(x) \quad (\textsc{A-Var})$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ e \in \tau_1 \to \tau_2} \quad (\textsc{A-Arrow-I})$$

$$\frac{\Gamma \vdash e_1 \in \sigma_1 \qquad \Gamma \vdash e_2 \in \sigma_2}{\Gamma \vdash e_1\ e_2 \in \bigwedge[\psi_i \mid (\phi_i \to \psi_i) \in \mathit{arrowbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \sigma_2 \leq \phi_i]} \quad (\textsc{A-Arrow-E})$$

$$\frac{\Gamma, \alpha \leq \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \Lambda\alpha \leq \tau_1.\ e \in \forall \alpha \leq \tau_1.\ \tau_2} \quad (\textsc{A-All-I})$$

$$\frac{\Gamma \vdash e \in \sigma_1}{\Gamma \vdash e[\tau] \in \bigwedge[\{\tau/\alpha\}\psi_i \mid (\forall \alpha \leq \phi_i.\ \psi_i) \in \mathit{allbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \tau \leq \phi_i]} \quad (\textsc{A-All-E})$$

$$\frac{\text{for all } i,\ \Gamma \vdash \{\sigma_i/\alpha\}e \in \tau_i}{\Gamma \vdash \mathit{for}\,\alpha\ \mathit{in}\,\sigma_1..\sigma_n.\ e \in \bigwedge[\tau_1..\tau_n]} \quad (\textsc{A-For})$$

A.2 $\mathbf{F}_{\leq}$

A.2.1 Subtyping

$$\Gamma \vdash \sigma \leq \mathit{Top} \qquad \text{(Sub-Top)}$$

$$\Gamma \vdash \tau \leq \tau \qquad \text{(Sub-Refl)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \tau_2 \qquad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3} \qquad \text{(Sub-Trans)}$$

$$\Gamma \vdash \alpha \leq \Gamma(\alpha) \qquad \text{(Sub-TVar)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2} \qquad \text{(Sub-Arrow)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma, \alpha \leq \tau_1 \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \forall \alpha \leq \sigma_1.\ \sigma_2 \leq \forall \alpha \leq \tau_1.\ \tau_2} \qquad \text{(Sub-All)}$$

A.2.2 Typing

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(Var)}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ e \in \tau_1 \to \tau_2} \qquad \text{(Arrow-I)}$$

$$\frac{\Gamma \vdash e_1 \in \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash e_1\ e_2 \in \tau_2} \qquad \text{(Arrow-E)}$$

$$\frac{\Gamma, \alpha \leq \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \Lambda \alpha \leq \tau_1.\ e \in \forall \alpha \leq \tau_1.\ \tau_2} \qquad \text{(All-I)}$$

$$\frac{\Gamma \vdash e \in \forall \alpha \leq \tau_1.\ \tau_2 \qquad \Gamma \vdash \tau \leq \tau_1}{\Gamma \vdash e[\tau] \in \{\tau/\alpha\}\tau_2} \qquad \text{(All-E)}$$

$$\frac{\Gamma \vdash e \in \tau_1 \qquad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e \in \tau_2} \qquad \text{(Sub)}$$

A.3 $\lambda_{\wedge}$

A.3.1 Subtyping

$$\frac{\Gamma \vdash \rho_1 \leq_P \rho_2}{\Gamma \vdash \rho_1 \leq \rho_2} \qquad \text{(Sub-Prim)}$$

$$\Gamma \vdash \tau \leq \tau \qquad \text{(Sub-Refl)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \tau_2 \qquad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3} \qquad \text{(Sub-Trans)}$$

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \qquad \Gamma \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2} \qquad \text{(Sub-Arrow)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash \sigma \leq \tau_i}{\Gamma \vdash \sigma \leq \bigwedge[\tau_1..\tau_n]} \qquad \text{(Sub-Inter-G)}$$

$$\Gamma \vdash \bigwedge[\tau_1..\tau_n] \leq \tau_i \qquad \text{(Sub-Inter-LB)}$$

$$\Gamma \vdash \bigwedge[\sigma \to \tau_1\ ..\ \sigma \to \tau_n] \leq \sigma \to \bigwedge[\tau_1..\tau_n] \qquad \text{(Sub-Dist-IA)}$$

A.3.2 Typing

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(Var)}$$

$$\frac{\Gamma, x{:}\sigma_i \vdash e \in \tau_i}{\Gamma \vdash \lambda x{:}\sigma_1..\sigma_n.\ e \in \sigma_i \to \tau_i} \qquad \text{(Arrow-I')}$$

$$\frac{\Gamma \vdash e_1 \in \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash e_1\ e_2 \in \tau_2} \qquad \text{(Arrow-E)}$$

$$\frac{\text{for all } i,\ \Gamma \vdash e \in \tau_i}{\Gamma \vdash e \in \bigwedge[\tau_1..\tau_n]} \qquad \text{(Inter-I)}$$

$$\frac{\Gamma \vdash e \in \tau_1 \qquad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e \in \tau_2} \qquad \text{(Sub)}$$

Appendix B

Glossary of Notation

α, β, γ	type variables	p. 14
$\sigma, \tau, \theta, \phi, \psi, \zeta$	types	p. 14
S, T, U, V, M, P	finite sequences of types	p. 14
x, y	term variables	p. 14
e, f, v, b	terms	p. 14
Γ	contexts	p. 14
J	subtyping and typing statements	p. 15
c, d	subtyping derivations	p. 15
s, t	typing derivations	p. 15
ρ	primitive types	p. 16
Γ_P	the pervasive context	p. 18
K, I	composite canonical types	p. 50
κ, ι	individual canonical types	p. 50
k, i	all canonical types	p. 50
$\stackrel{\flat}{\vdash}$	canonical derivations	p. 51
C, D	canonical subtyping derivations	p. 51
ξ, δ	individual canonical subtyping derivations	p. 51
c, d	both canonical and individual canonical subtyping derivations	p. 51

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, CA, January 1990.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 104–118, Orlando, FL, January 1991.

[3] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist*. The MIT Press, 1991.

[4] Franco Barbanera and Mariangiola Dezani-Ciancaglini. Intersection and union types. In Ito and Meyer [80], pages 651–674.

[5] H. P. Barendregt. *The Lambda Calculus*. North Holland, revised edition, 1984.

[6] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[7] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[8] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Studentlitteratur (Lund, Sweden), Bratt Institute Fuer Neues Lerned (Goch, FRG), Chartwell-Bratt Ltd (Kent, England), 1979.

[9] Hans-J. Boehm. Type inference in the presence of type abstraction. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 192–206, Portland, OR, June 1989.

[10] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[11] Kim B. Bruce. The equivalence of two semantic definitions for inheritance in object-oriented languages. In *Proceedings of Mathematical Foundations of Programming Semantics*, Pittsburgh, PA, March 1991. To appear.

[12] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.

[13] Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. The semantics of second-order lambda calculus. In Huet [79], pages 213–272. Also appeared in Information and Computation 84, 1 (January 1990).

[14] Kim Bruce and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque, NM, January 1992. To appear.

[15] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[16] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory and practice of strictness analysis. *Science of Programming*, 7:249–278, 1986.

[17] Peter Canning, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *Object Oriented Programing: Systems, Languages, and Applications (Conference Proceedings)*, pages 457–467, 1989.

[18] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[19] Peter Canning, Walt Hill, and Walter Olthoff. A kernel language for object-oriented programming. Technical Report STL-88-21, Hewlett-Packard Labs, 1988.

[20] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[21] Luca Cardelli. Amber. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*. Springer-Verlag, 1986. Lecture Notes in Computer Science No. 242.

[22] Luca Cardelli. Typechecking dependent types and subtypes. In *Proc. of the Workshop on Foundations of Logic and Functional Programming*, Trento, Italy, December 1987.

[23] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[24] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988.

[25] Luca Cardelli. Typeful programming. Research Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1989.

[26] Luca Cardelli. Extensible records in a pure calculus of subtyping. To appear, 1991.

[27] Luca Cardelli. F-sub, the system. Unpublished manuscript, July 1991.

[28] Luca Cardelli, James Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, January 1989.

[29] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest: (Extended abstract). In *ACM Conference on Lisp and Functional Programming*, pages 30–43, Nice, France, June 1990. Extended version available as DEC SRC Research Report 55, Feb. 1990.

[30] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In Ito and Meyer [80], pages 750–770.

[31] Luca Cardelli and John C. Mitchell. Operations on records. Research Report 48, Digital Equipment Corporation, Systems Research Center, August 1989.

[32] Luca Cardelli and John Mitchell. Operations on records (summary). In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of Fifth International Conference on Mathematical Foundations of Programming Language Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 22–52, Tulane University, New Orleans, March 1989. Springer Verlag. To appear in Mathematical Structures in Computer Science; also available as DEC Systems Research Center Research Report #48, August, 1989.

[33] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[34] Felice Cardone. Relational semantics for recursive types and bounded quantification. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 164–178, Stresa, Italy, July 1989. Springer-Verlag.

[35] Felice Cardone and Mario Coppo. Two extensions of Curry's type inference system. In Odifreddi [100], pages 19–76.

[36] Robert Cartwright and Mike Fagan. Soft typing. Submitted to PLDI '91, November 1990.

[37] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[38] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[39] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990.

[40] M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv. Math. Logik*, 19:139–156, 1978.

[41] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre-Dame Journal of Formal Logic*, 21(4):685–693, October 1980.

[42] M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo. Extended type structures and filter lambda models. In G. Lolli, G. Longo, and A. Marja, editors, *Logic Colloquium 82*, pages 241–262, Amsterdam, 1983. North-Holland.

[43] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and lambda calculus semantics. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 535–560, New York, 1980. Academic Press.

[44] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[45] M. Coppo, M. Dezani-Ciancaglini, and M. Zacchi. Type theories, normal forms and D_∞-lambda-models. *Information and Computation*, 72:85–116, 1987.

[46] M. Coppo, M. Dezani, and P. Sallé. Functional characterization of some semantic equalities inside λ-calculus. Number 81 in Lecture Notes in Computer Science, pages 133–146. Springer-Verlag, 1979.

[47] Thierry Coquand. *Une Théorie des Constructions*. PhD thesis, University Paris VII, January 1985.

[48] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[49] Pierre-Louis Curien and Roberto Di Cosmo. A confluent reduction for the λ-calculus with surjective pairing and terminal object. In *ICALP '91*, 1991.

[50] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption. *Mathematical Structures in Computer Science*, 1991. To appear.

[51] Pierre-Louis Curien and Giorgio Ghelli. Subtyping + extensionality: Confluence of $\beta\eta$-reductions in $F_{\leq}$. In Ito and Meyer [80], pages 731–749.

[52] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

[53] Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox Palo Alto Research Center, February 1990.

[54] O. J. Dahl and K. Nygaard. SIMULA–An ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

[55] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[56] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[57] M. Dezani-Ciancaglini and I. Margaria. F-semantics for intersection type discipline. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 279–300. Springer-Verlag, 1984.

[58] Mariangiola Dezani-Ciancaglini and Ines Margaria. A characterisation of F-complete type assignments. *Theoretical Computer Science*, 45:121–157, 1986.

[59] Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages.* PhD thesis, Rice University, December 1990.

[60] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario.* ACM Press, June 1991.

[61] P. Freyd, P. Mulry, G. Rosolini, and D. Scott. Extensional PERs. In *Fifth Annual Symposium on Logic in Computer Science (Philadelphia, PA)*, pages 346–354. IEEE Computer Society Press, June 1990.

[62] Jean H. Gallier. On Girard's "candidats de reductibilité". In Odifreddi [100], pages 123–203.

[63] Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Università di Pisa, March 1990. Technical report TD–6/90, Dipartimento di Informatica, Università di Pisa.

[64] Giorgio Ghelli. A static type system for message passing. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 129–143, Phoenix, Arizona, October 1991. Distributed as Sigplan Notices, Volume 26, Number 11, November 1991.

[65] Paola Giannini and Simona Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *IEEE Symposium on Logic in Computer Science*, pages 61–70, 1988.

[66] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[67] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[68] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[69] Robert Harper and John Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 1992. To appear. An earlier version titled "The Essence of ML" (Mitchell and Harper), appeared in the Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages, San Diego, CA, January 1988.

[70] Robert W. Harper and Benjamin C. Pierce. Extensible records without subsumption. Technical Report CMU-CS-90-102, School of Computer Science, Carnegie Mellon University, Feburary 1990.

[71] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando FL*, pages 131–142. ACM, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

[72] Susumu Hayashi. Singleton, union and intersection types for program extraction. In Ito and Meyer [80], pages 701–730.

[73] Susumu Hayashi and Yukihide Takayama. Extended projection method with Kreisel-Troelstra realizability. Submitted to Information and Computation, 1990.

[74] Fritz Henglein and Harry G. Mairson. The complexity of type inference for higher-order typed lambda-calculi. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 119–130, Orlando, FL, January 1991.

[75] J. R. Hindley. The simple semantics for Coppo-Dezani-Sallé types. In Dezani-Ciancaglini and Montanari, editors, *Proceedings of the International Symposium on Programming*, pages 212–226. Springer-Verlag, 1982. Lecture Notes in Computer Science No. 137.

[76] J. Roger Hindley. Coppo-Dezani-Sallé types in lambda-calculus, an introduction. Draft manuscript, February 1989.

[77] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

[78] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[79] Gérard Huet, editor. *Logical Foundations of Functional Programming*. University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

[80] T. Ito and A. R. Meyer, editors. *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science. Springer-Verlag, September 1991.

[81] Bart Jacobs, Ines Margaria, and Maddalena Zacchi. Expansion and conversion models in the lambda calculus from filters with polymorphic types. Manuscript, March 1989.

[82] Lalita A. Jategaonkar. ML with extended pattern matching and subtypes. Master's thesis, MIT, August 1989.

[83] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[84] Thomas P. Jensen. Strictness analysis in logical form. Unpublished manuscript, 1991.

[85] A.J. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the polymorphic λ-calculus. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 2–11, Philadelphia, PA, June 1990.

[86] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Proceedings of the fourth International Conference on Functional Programming and Computer Architecture*, pages 260–272, September 1989.

[87] Daniel Leivant. Typing and computational properties of lambda expressions. *Theoretical Computer Science*, 44:51–68, 1986.

[88] Daniel Leivant. Discrete polymorphism (summary). In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 288–297, 1990.

[89] QingMing Ma. Parametricity as subtyping. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque, NM, January 1992. To appear.

[90] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[91] Simone Martini. Bounded quantifiers have interval models. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988. ACM.

[92] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[93] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[94] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

[95] John C. Mitchell. A type-inference approach to reduction properties and semantics of polymorphic expressions. In Huet [79], pages 195–212.

[96] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 270–278, Orlando, FL, January 1991.

[97] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.

[98] P. Naur et al. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6:1–17, January 1963.

[99] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[100] Piergiorgio Odifreddi, editor. *Logic and Computer Science.* Number 31 in APIC Studies in Data Processing. Academic Press, 1990.

[101] Atsushi Ohori and Peter Buneman. Static type inference for parametric classes. In *OOPSLA '89: Object-Oriented Programming Systems, Languages, and Applications, Conference Proceedings*, pages 445–456, October 1989.

[102] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah*, pages 153–163. ACM Press, July 1988. Also available as Ergo Report 88–048, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[103] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics, Tulane University, New Orleans, Louisiana*, pages 209–228. Springer-Verlag LNCS 442, March 1989. Also available as Ergo Report 88–069, School of Computer Science, Carnegie Mellon University.

[104] Benjamin Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical Report CMU-CS-89-169, School of Computer Science, Carnegie Mellon University, September 1989.

[105] Benjamin C. Pierce. Preliminary investigation of a calculus with intersection and union types. Unpublished manuscript, June 1990.

[106] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[107] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[108] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, March 1989.

[109] Randy Pollack. Implicit syntax. In G. Huet and G. Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 421–434. Preliminary Version, May 1990.

[110] Garrell Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 561–577. Academic Press, New York, 1980.

[111] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 289–297, Snowbird, Utah, July 1988.

[112] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*, pages 242–249. ACM, January 1989.

[113] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels*. PhD thesis, Université Paris VII, 1990.

[114] Didier Rémy. Typechecking records in a natural extension of ML. Submitted to TOPLAS, June 1990.

[115] Didier Rèmy. Typing record concatenation for free. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque, NM, January 1992. To appear.

[116] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[117] John Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science. Springer-Verlag, January 1980.

[118] J. C. Reynolds. The essence of algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland.

[119] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[120] John Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development*. Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

[121] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[122] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[123] John C. Reynolds. The coherence of languages with intersection types. In Ito and Meyer [80], pages 675–700.

[124] Edmund Robinson and Robert Tennent. Bounded quantification and record-update problems. Message to `Types` electronic mail list, October 1988.

[125] Simona Ronchi della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.

[126] S. Ronchi della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.

[127] P. Sallé. Une extension de la theorie des types en λ-calcul. pages 398–410. Springer-Verlag, 1982. Lecture Notes in Computer Science No. 62.

[128] Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

[129] R. A. G. Seely. Categorical semantics for higher order polymorphic lambda calculus. *Journal of Symbolic Logic*, 52(4):969–988, December 1987.

[130] Ryan Stansifer. Type inference with subtypes. In *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 88–97, San Diego, CA, January 1988.

[131] Steffen van Bakel. Principal type schemes for the strict type assignment system. Technical report 91-6, University of Nijmegen, 1991.

[132] Steffen van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 99, 1992. To appear.

[133] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[134] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

[135] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989.

[136] Hirofumi Yokouchi. Relationship between polymorphic types and intersection types (extended abstract). Unpublished manuscript, December 1990.