Fundamentals of type inference systems

Fritz Henglein
DIKU, University of Copenhagen
henglein@diku.dk

February 1, 1991; revised January 31, 1994; updated August 9, 2009

1 Introduction

These notes give a compact overview of established core type systems and of their fundamental properties. We emphasize the use and application of type systems in programming languages, but also mention their role in logic. Proofs are omitted, but references to relevant sources in the literature are usually given.

1.1 What is a "type"?

There are many examples of *types* in programming languages:

- Primitive types: int, float, bool
- Compound types: products (records), sums (disjoint unions), lists, arrays
- Recursively definable types, such as tree data types
- Function types, e.g. int →int
- Parametric polymorphic types
- Abstract types, e.g. Java interfaces

Loosely speaking a type is a *description of a collection of related values.*

- A type has *syntax* ("description"): It *denotes* a collection.
- A type has *elements* ("collection"): It makes sense to talk about elements of a type; in particular, it may have zero, one or many elements

- A type's elements have common properties ("related"): Users of a type can rely on each element having some common properties—an interface— without having to know the identity of particular elements.

Types incorporate multiple aspects:

- Type as a set of values: This view focuses on how values are constructed to be elements of a type; e.g. constructing the natural numbers from 0 and the successor function;
- Type as a an interface: This view focuses on how values can be *used* ("deconstructed") by a client; e.g. testing whether a number is 0 or a successor and choosing different branches based on that.

These aspects are *complementary*, with none superseding the other.

- If there is no test or other operation for making computations distinguish between 0 from nonzero values we can "implement" the successor function by just returning 0 (or any other value for that matter).
- If there is no successor function we can statically simplify each test for 0

We can think of a type as a *contract* between an implementor and a user of a software component that enforces information hiding with the well-known trade-off between choosing very concrete or very abstract (highly encapsulated) interfaces:

- The fewer elements a type has the more common properties they share that can be exploited by clients, but the fewer alternatives there are for implementing the type (low degree of information hiding);
- The more elements a type has the fewer common properties they share that can be exploited by clients, but the more alterantives there are for implementing the type (high degree of information hiding).

2 What is a type system?

A *type system* is logical system of rules for inferring valid judgements The ingredients of a type system for programming languages are:

- expressions e: syntactically well-formed program fragments
- types τ: a language of interfaces (properties) we are interested in

- typings (judgments) $e : \tau$ and more generally $A \vdash e : \tau$ expressing that e "expression e has type τ if assumptions A hold".
- inference rules for deriving valid typings for compound expressions from their constituent expressions.

3 Soundness

Derivable typing judgements are intended to guarantee some *safety property* about program execution, namely that certain errors cannot occur. This is popularized in the slogan:

> *Well-typed programs don't go wrong.*

A type system is *sound* if its rules are such that only typings for safe programs are derivable. Soundness relates the derivable typings to what happens during computation and can be considered a core characteristic of a type system.

3.1 Applications of type systems

- Language-based security: No spoofing or injection attacks possible. Typing derivations provide efficiently checkable *logical certificates* of a program's safety ("does not go wrong") property.
- Component-oriented software development: Types express explicit contracts between implementation and client of a software component. Either one can change without having to notify the other as long as the type is preserved. This supports software evolution and software reuse. Parametricity expresses powerful program properties and gives encapsulation guarantees.
- Type-based program analysis: Types as properties of programs. The whole derivation, not just the property of the whole program, can be used for program transformation/novel implementations (e.g. for pointer analysis, binding-time analysis, dynamic type inference, region inference, information flow, communication pattern inference). Control of expressiveness, supported by efficient constraint-solving methods.

4 The descriptive and prescriptive views of typings

There are different philosophical views (with technical ramifications, mostly in the model theories of the resulting type theories) we can take of the interaction of programs and their properties. They have been termed *descriptive* and *prescriptive* or, respectively, *Curry* and *Church* for historical reasons. In the descriptive view programs and properties are defined independently (typically by some simple syntactic mechanism such as context-free grammars), and the typings *describe* (properties of) programs; in particular, some programs may not have any properties whatsoever, but are programs nonetheless. In the prescriptive view the very notion of program (or expression, term, *etc.*) is defined in such a way where their properties (types) *prescribe* how they may be put together to form new programs. In particular, programs (in the descriptive sense) with no type are not even considered programs in the first place, but called something like "pre-programs" (or pre-terms, pre-expressions, ...).

4.1 Types in logic

Type systems are used in logic, especially in the proof theory of constructive logics because of the *propositions-as-types* (and *deductions-as-programs*) principle, also called the *Curry-Howard isomorphism*, since Curry and Howard were the first ones to notice these correspondences for intuitionistic propositional logic.

In logic it is most natural to adopt the prescriptive viewpoint as programs describe deductions and their type corresponds to a theorem. Theorems are defined in terms of deductions and, in turn, deductions are defined in terms of what they prove. There is only little interest in considering "pre-deductions".

A derivable typing $e : p$ gives not only that p is a theorem, but also represents an explicit deduction e of p. The type inference system provides rules for how deductions of theorems can be composed to give deductions of other theorems. There is usually a reduction relation on deductions corresponding to cut-elimination, which one might also call lemma elimination — replacing uses of a lemma in a deduction by (copies of) the deduction of the lemma itself. A central property of such type systems is that all deductions strongly normalize: any reduction sequence is finite (see strong normalization theorem below). Viewing deductions as programs, and normalization as evaluation this implies that the "programming language" of

deductions only contains uniformly terminating programs; i.e., it is *not* a *general (Turing-complete)* programming language.

The correspondences between proof theory and λ-calculus are:

$$\begin{array}{rcl} \text{proposition} & = & \text{type} \\ \text{deduction} & = & \text{expression} \\ \text{normalization} & = & \text{reduction} \\ \text{cut elimination} & = & \text{strong normalization} \end{array}$$

4.2 Types in programming languages

The presence of a general recursion mechanism in programming languages destroys the propositions-as-types principle: the type $(\alpha \to \alpha) \to \alpha$ of a fixed point operator is *not* a (valid) proposition.

In type inference applied to programming language processing and analysis the descriptive viewpoint is often natural since programs and properties are typically defined independently of each other:

Type checking/inference. In type checking the question is whether programs satisfy the typing rules or not; in particular the notion of "program" — the input to this process — is defined independently of the typing rules.

Program analysis. In program analysis we use types to express properties of programs (and program parts). Again, what constitutes programs is defined completely independently of types. Programs for which no properties can be inferred are not in any sense rejected, as is implicit in the prescriptive view. Furthermore, we might even apply a program analysis formulated as a type inference system to a typed language, where the object language type system and the analysis type system serve different purposes.

More often than not these viewpoints are interchangeable, however, and have actually led to more difference in terminology than in anything of substance — with the exception of models and semantics.

5 Why study core type systems?

Type systems for realistic programming languages can be quite complex. The *principles* they follow are fewer, and it is often possible to study them in a tiny core language.

We shall restrict ourselves to the type systems for λ-calculus, the core of functional languages. This is partially because type systems have historically evolved for λ-calculus. Type systems that start with a other primitive notions than functions, such as objects (Theory of Objects) or processes (behavioral type systems) or imperative (Hoare logic) exist, but are arguably less mature.

6 Untyped λ-calculus

6.1 Syntax of expressions

In the *(pure) λ-calculus* the *λ-expressions* Λ are:

$$e ::= x \mid e'e'' \mid \lambda x.e'$$

where x ranges over an infinite set V of *variables* and λ binds its variable.

6.2 λ-reduction

Reduction can be viewed both as an operational semantics for the λ-calculus and as a form of *program simplification*. It is defined by the following inference system.

$$(\beta) \qquad (\lambda x.e)e' \longrightarrow_\beta e[e'/x]$$

$$\text{(COMP)} \qquad \frac{e \longrightarrow e'}{C[e] \longrightarrow C[e']} \quad \text{for any context } C$$

Here, $C[\ldots]$ denotes a *context*; that is, a λ-expression with one hole.[1]

A *normal form* is an expression e such that $e \longrightarrow e'$ for no e'. An expression e is *(weakly) normalizing* (has a normal form) if $e \longrightarrow^* e'$ for some normal form e'; i.e., if there is a finite reduction sequence $e \longrightarrow e_1 \longrightarrow \ldots \longrightarrow e'$ starting in e and ending in e'. Expression e is *strongly normalizing* if there is no infinite reduction sequence starting at e.

Note: There is also η-reduction with the rule

$$(\eta) \qquad \lambda x.ex \longrightarrow_\eta e \quad \text{if } x \notin FV(e)$$

[1] In fact, the context could also have an arbitrary number of contexts.

In general, a β-rule shows what happens when a value constructor (λ) meets a deconstructor ($C[] = []e$); which can be thought of as a *computation step*. In contrast, an η-rule, read from right to left, expresses intuitively that that a type has no other value constructors than given ones; in particular, it cannot be extended with new ones.

6.3 λ-theory

A λ-*theory* is a theory of equalities between λ-expressions. These equalities can be viewed as specifying the (extensional) behavior of λ-expressions. For any reduction relation $\longrightarrow$ we define the equality theory induced by it as follows.

$$(\alpha) \qquad \lambda x.e = \lambda y.e[y/x]$$

$$(\text{INTRO}) \qquad \frac{e \longrightarrow e'}{e = e'}$$

$$(\text{REFL}) \qquad e = e$$

$$(\text{SYMM}) \qquad \frac{e = e'}{e' = e}$$

$$(\text{TRANS}) \qquad \frac{e = e' \quad e' = e''}{e = e''}$$

$$(\text{COMP}) \qquad \frac{e = e'}{C[e] = C[e']}$$

So, β-*equality* $e =_\beta e'$ is the equality theory induced by β-reduction.

6.4 λ-models

A (syntactic) λ-model is an *applicative structure* $(D, \bullet)$ together with a mapping $[\![\ldots]\!] : \Lambda \to (V \to D) \to D$ such that the following properties hold:

$$\begin{aligned} [\![x]\!]\rho &= \rho(x) \\ [\![ee']\!]\rho &= ([\![e]\!]\rho) \bullet ([\![e']\!]\rho) \\ [\![\lambda x.e]\!]\rho \bullet d &= [\![e]\!](\rho\{x : d\}) \\ \rho|_{FV(e)} = \rho'|_{FV(e)} &\Rightarrow [\![e]\!]\rho = [\![e]\!]\rho' \end{aligned}$$

Note: $e =_\beta e'$ implies $[\![e]\!]\rho = [\![e']\!]\rho$ for all ρ in any λ-model. The map $[\![\ldots]\!]$ interprets every λ-expression inside an applicative structure; the properties this applicative structure has to satisfy guarantee that λ-expressions that are β-equal are interpreted as the same element.

7 Simply typed λ-calculus

7.1 Simple types

The *(simple) types* T are

$$\tau ::= \alpha \mid \tau' \to \tau''$$

where α ranges over an infinite set TV of *type variables*. In type systems for actual programming languages there are usually also primitive types such as *bool* and *integer*, and other type constructors such as pair and sum type constructors and *list*.

A *simple model* of types is an applicative structure $(D, \bullet)$ with a map $[\![\ldots]\!] : T \to (TV \to 2^D) \to 2^D$ that satisfies the following properties:

$$\begin{aligned} [\![\alpha]\!]\upsilon &= \upsilon(\alpha) \\ [\![\tau \to \tau']\!]\upsilon &= \{d \in D : (\forall d' \in D) d' \in [\![\tau]\!]\upsilon \Rightarrow \\ &\quad d \bullet d' \in [\![\tau']\!]\upsilon\} \end{aligned}$$

7.2 Typings

A *type environment* A is a mapping from variables to types. A *typing formula* is a formal statement of the form $A \vdash e : \tau$ where A is a type environment, e an expression and τ a type. The *(derivable) typings* are all those (and only those) typing formulae derivable by the following inference system.

$$\text{(VAR)} \qquad A\{x : \tau\} \vdash x : \tau$$

$$\text{(ABSTR)} \qquad \frac{A\{x : \tau\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \to \tau'}$$

$$\text{(APPL)} \qquad \frac{A \vdash e : \tau \to \tau' \quad A \vdash e' : \tau}{A \vdash ee' : \tau'}$$

($A\{x : \tau\}$ is the type environment A updated at x to map x to τ.)

A typing $\vdash e : \tau$ is interpreted in a λ-model as a containment statement: the value denoted by e is contained in the set denoted by τ. More precisely, for λ-model $\mathcal{M} = (D, \bullet)$ we write

$$A \models_{\mathcal{M}} e : \tau$$

if for every ρ, υ such that $[\![x]\!]\rho \in [\![A(x)]\!]\upsilon$ for every x in the domain of A we have $[\![e]\!]\rho \in [\![\tau]\!]\upsilon$. If this holds for *every* λ-model $\mathcal{M}$ we write

$$A \models e : \tau.$$

7.3 Syntactic properties of typings

The following lemmas express that

1. type variables really represent "arbitrary types", and
2. a typing for e depends only on type assumptions for the free variables of e.

Lemma 7.1 *(Substitution Lemma)*
If $A \vdash e : \tau$ then $S(A) \vdash e : S(\tau)$ for any substitution S (of types for type variables).

PROOF By induction on typing derivations. □

Lemma 7.2 *(Irrelevance of type assumptions for nonfree variables)*
If $A \vdash e : \tau$ then $A|_{FV(e)} \vdash e : \tau$.

PROOF By induction on typing derivations. □

A very important general property of type systems is the *principal type property*. Its definition varies from type system to type system but always embodies the following principle: there is a single typing such that all other typings for the same expression can be derived from this one, the principal one, by application of "logical" (nonstructural) rules; i.e., by reasoning that applies to *all* expressions not just to those with a specific syntactic structure.[2] For the 1st order typed λ-calculus the typings can be derived from the principal one by substitution.

Theorem 7.3 *(Principal types)*
For every typable expression e there is a typing $A \vdash e : \tau$ such that for every typing $A' \vdash e : \tau'$ there is a substitution S such that

- $S(A) = A'|_{FV(e)}$,
- $S(\tau) = \tau'$.

PROOF (Cur69; Hin69) □

Remark 7.4 Principal type properties have also been shown for ML typing (DM82; Dam84), ML typing with polymorphic recursion (Myc84), intersection type discipline (RDR88), and higher order modules (Tof92). □

[2] "Reasoning" in this sense includes both nonstructural type inference rules such as (GEN) and (INST) further below, and logical rules of inference such as substitution.

7.4 Soundness

We say a type system is invariant under (β-)equality (or any other binary relation) if $A \vdash e : \tau \Leftrightarrow A \vdash e' : \tau$ whenever $e =_\beta e'$.

Typings capture aspects both of the syntax and the (reduction) semantics of an expression, but they are generally not invariant under equality. They do, however, respect β-reduction, in the sense that typings are preserved under reduction. Historically this is called the *subject reduction property*. A modern expression is *(type) preservation* under reduction (evaluation).

Theorem 7.5 *(Subject reduction property)*
If $A \vdash e : \tau$ and $e \longrightarrow_\beta e'$ then $A \vdash e' : \tau$.

PROOF (CF58; CHS72) □

The *subject expansion property*, preservation of typings under β-expansion, does not hold in general.

Exercise 1 Show that the subject expansion property does not hold in general; i.e., there exist e, e', τ such that $\vdash e' : \tau$ and $e \longrightarrow_\beta e'$, but *not* $e : \tau$. □

Exercise 2 (•) Define the *linear* λ-expressions $L\Lambda$ as those expressions where every bound and free variable has exactly one occurrence. Show that simple typings are invariant under β-equality for $L\Lambda$ (see below for definition of type invariance). □

Note: Viewed as a program analysis the subject reduction property expresses that the program analysis does not "lose" properties under program "simplification". (This is a stronger requirement on a program analysis than just stipulating soundness.)

Remark: The subject reduction property holds for all established type systems and can be considered a fundamental characteristic and necessary property of type inference systems.

7.5 Completeness

The typing rules are logically sound, but not complete w.r.t. the interpretation of typings. This is to say that whenever $A \vdash e : \tau$ then we have $A \models e : \tau$, but the converse does not generally hold. This follows (how?) immediately from the fact that typings are not invariant under β-equality.

The "smallest" possible extension of the type system to make typings invariant, however, is complete.

$$\text{(EQUAL)} \qquad \frac{A \vdash e : \tau \quad e =_\beta e'}{A \vdash e' : \tau}$$

Theorem 7.6 *(Soundness and completeness)*
$A \vdash e : \tau$ *(with rule EQUAL) if and only if* $A \models e : \tau$.

PROOF (Hin83) □

Remark 7.7 This soundness and completeness result has also been extended to subtypings (Mit91). □

8 Predicative (ML-style) parametric polymorphism

8.1 Type schemes

Type schemes are generated by

$$\sigma ::= \tau \mid \forall\alpha.\sigma,$$

where τ ranges over simple types.

They are interpreted as intersections in an applicative structure $(D, \bullet)$:

$$[\![\forall\alpha.\sigma]\!]\upsilon = \bigcap_{D' \subseteq D} [\![\sigma]\!](\upsilon\{\alpha : D'\}).$$

They give rise to two (nonstructural) rules:

$$\text{(GEN)} \qquad \frac{A \vdash e : \sigma}{A \vdash e : \forall\alpha.\sigma} \quad \text{if } \alpha \text{ not free in } A$$

$$\text{(INST)} \qquad \frac{A \vdash e : \forall\alpha.\sigma}{A \vdash e : \sigma[\tau/\alpha]}$$

The generalization rule, (GEN), shows how type schemes can be inferred: by abstracting over type variables that do not occur in the assumptions. The instantiation rule, (INST), shows how type schemes can be used: by instantiating the quantified type variable with an arbitrary type. Note, however, that the type substituted must be a *simple* type, not a type scheme. Since different (simple) types can be given to different occurrences of a variable this is called a *polymorphic* typing discipline.

8.2 Nonrecursive definitions

We extend expressions with (nonrecursive) *let-expressions*,

$$e ::= \ldots \mid \mathbf{let}\, x = e' \,\mathbf{in}\, e''.$$

Their reduction semantics is given by

$$\mathbf{let}\, x = e' \,\mathbf{in}\, e'' \longrightarrow_{\mathbf{let}} e''[e'/x] \quad .$$

These are nonrecursive definitions together with a scope (the expression e'') where they may be used.

We extend the simply typed λ-calculus with the rule

$$\text{(LET)} \qquad \frac{A \vdash e' : \sigma' \quad A\{x : \sigma'\} \vdash e : \sigma}{A \vdash \mathbf{let}\, x = e' \,\mathbf{in}\, e : \sigma}$$

where σ, σ' range over type schemes. Note that the let-bound variable x may have a type scheme associated with it, whereas λ-bound variables are always bound to simple types.

The polymorphic typing rule (LET) is very powerful in that it is invariant under $\longrightarrow_{\mathbf{let}}$.

Theorem 8.1 *(Type invariance under* **let** *-reduction)*
$A \vdash \mathbf{let}\, x = e' \,\mathbf{in}\, e : \sigma$ *if and only if* $A \vdash e[e'/x] : \sigma$ *and* $A \vdash e' : \tau'$ *for some* τ'.

PROOF (Dam84) □

Exercise 3 Is this theorem still correct if we drop the phrase "and $A \vdash e' : \tau'$ for some τ'"? □

Expressions are strongly normalizing under **let** -reduction (with no β-reduction steps!). This theorem suggests a simple algorithm for type checking an expression e: **let** -reduce e until it contains no more let-expressions. The resulting expression is simply typable if and only if e is typable with the (LET) rule.

Exercise 4 The principal type property for core ML (the simply typed λ-calculus with polymorphic let-expressions) can be stated as follows: For every e typable under type environment A there exists a *principal* type $A \vdash e : \sigma$ such that every other typing $A \vdash e : \sigma'$ can be derived from the principal one by application of the typing rules (GEN) and (INST) alone. Use this property to prove the right-to-left implication of Theorem 8.1. □

8.3 Recursive definitions

We extend expressions with recursive definitions (fixed points),

$$e ::= \ldots \mid \mathbf{fix}\ f.e'.$$

Their reduction semantics is given by

$$\mathbf{fix}\ f.e' \longrightarrow_{\mathbf{fix}} e'[\mathbf{fix}\ f.e'/f]$$

An expression $\mathbf{fix}\ f.e'$ represents the value defined by the (recursive) definition $f = e'$ (note that e' can contain occurrences of f). The use of f is suggestive of functions being mostly recursively defined; e.g. $f = \lambda x.\mathbf{if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ f(x-1)$.

We extend the typing rules of the simply typed λ-calculus (and let-expressions) with the rule

$$\text{(FIX-P)} \qquad \frac{A\{x:\sigma\} \vdash e' : \sigma}{A \vdash \mathbf{fix}\ x.e' : \sigma}$$

This polymorphic rule, (FIX-P), is invariant under **fix** -reduction.

Theorem 8.2 *(Type invariance under* **fix** *-reduction)*
$A \vdash \mathbf{fix}\ x.e' : \sigma$ *if and only if* $A \vdash e'[\mathbf{fix}\ x.e'/x] : \sigma$.

Exercise 5 (••) Prove this theorem. □

Note: ML (as well as other languages, such as Miranda and Haskell) employs the polymorphic typing rule (LET) for nonrecursive definitions, but uses a *monomorphic* typing rule for recursive definitions:

$$\text{(FIX-M)} \qquad \frac{A\{x:\tau\} \vdash e' : \tau}{A \vdash \mathbf{fix}\ x.e' : \tau}$$

where τ must be a simple type.

Exercise 6 Show there is an expression e that is typable with polymorphic recursion (FIX-P), but untypable with monomorphic recursion (FIX-M). □

9 System F: Impredicative parametric polymorphism

The polymorphic typing systems above are called impredicative because there is a hierarchy of type expressions (simple types, type schemes) where type expressions higher up in the hierarchy may be quantified only over type expressions strictly below them. For instance, rule (INST) requires that the quantified type variables in a type scheme are instantiated to simple types, but not to type schemes. In System F (also called Girard/Reynolds Calculus, 2nd order λ-calculus, polymorphic λ-calculus) this restriction is dropped. The types are defined by

$$\tau ::= \alpha \mid \tau' \to \tau'' \mid \forall\alpha.\tau'$$

and the type inference system consists of the following rules:

$$\text{(VAR)} \qquad A\{x:\tau\} \vdash x : \tau$$

$$\text{(GEN)} \qquad \frac{A \vdash e : \tau}{A \vdash e : \forall\alpha.\tau} \quad \text{if } \alpha \text{ not free in } A$$

$$\text{(INST)} \qquad \frac{A \vdash e : \forall\alpha.\tau}{A \vdash e : \tau[\tau'/\alpha]}$$

$$\text{(ABSTR)} \qquad \frac{A\{x:\tau\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \to \tau'}$$

$$\text{(APPL)} \qquad \frac{A \vdash e : \tau \to \tau' \quad A \vdash e' : \tau}{A \vdash ee' : \tau'}$$

Exercise 7 Consider the expression $I' = (\lambda y.yy)(\lambda x.x)$. Show that I' is not typable in the simply typed λ-calculus, but that it is typable in F_2. ($\bullet$) Show that if we translate every let-expression $\mathbf{let}\, x = e' \,\mathbf{in}\, e$ by $(\lambda x.e)e'$ then the translation of every ML typable expression is F_2-typable. □

9.1 Parametricity

We can think of a type as a subset of the universe of values. A type thus acts as a unary predicate: $int(v)$ holds if v satisfies being an integer. A function f of type $int \to real$ then guarantees to map each value that satisfies the *int*-predicate to a value that satisfies the *real*-predicate if it terminates at all. We can express this as

$$\forall v.int(v) \Rightarrow real(f(v)).$$

(Note how the function arrow is interpreted as an implication.)

Polymorphic types can be interpreted in the same fashion. Assume f has type $\forall\alpha.\alpha \to \alpha$. As above, from the (INST) rule we can conclude that

$$\forall v.\alpha(v) \Rightarrow \alpha(f(v))$$

holds for each instantion of α as a type (*int*, *bool*, etc.). But an even stronger interpretation is possible: We can instantiate α with any subset (predicate) of the value universe, whether or not denotable by a type:

$$\forall^2 P.\forall v.P_\alpha(v) \Rightarrow P_\alpha(f(v)).$$

Instantiating with singleton set $\{v\}$ we obtain that f maps v to v. Since v can be chosen arbitrarily we can conclude that f must be the identity function from its type alone!

A more powerful interpretation yet of a typing is where we allow binary predicates (or n-ary) predicates (relations), not just unary ones.

If a language admits an interpretation of typings as "free theorems" (Wad89) similar to the one above we call it *parametric*. In a parametric language the presence of a universally quantified type variable intuitively conveys a very strong guarantee: A (part of a) value passed to a parametric polymorphic function in the position of a type variable is guaranteed only to be copied and moved around, but never inspected in the function.

Theorem 9.1 *System F is parametric.*

PROOF (Rey83) □

9.2 Strong normalization

A property of central importance in logic and also of great relevance in programming language theory is that only uniformly terminating λ-expressions are typable.

Theorem 9.2 *(Strong Normalization)*
If $A \vdash e : \tau$ then e is strongly normalizing.

PROOF (Gir71; GLT89) □

Note: Call-by-value reduction (innermost redex first) or any other reduction strategy may be applied to typable expressions *without* fixed point operators, even when (sub)expressions are nonstrict!

Exercise 8 A *fixed point operator* is a λ-expression F that satisfies $fe =_\beta e(fe)$. Use the strong normalization theorem to show that no fixed point operator that actually satisfies the stronger property $fe \longrightarrow_\beta e(fe)$ is simply typable. ($\bullet$) Strengthen your proof to show that no (arbitrary) fixed point operator is simply typable. $\square$

10 Subtyping

A *type containment* is a formula of the form $\tau \leq \tau'$. Type containments are reflexive, transitive and compatible. To express this we add the following inference rules to the simply typed λ-calculus.

$$\text{(REFL)} \qquad \tau \leq \tau$$

$$\text{(TRANS)} \qquad \frac{\tau \leq \tau' \quad \tau' \leq \tau''}{\tau \leq \tau''}$$

$$\text{(ARROW-COMP)} \qquad \frac{\tau_d \leq \tau'_d \quad \tau_r \leq \tau'_r}{\tau'_d \to \tau_r \leq \tau_d \to \tau'_r}$$

$$\text{(COERCE)} \qquad \frac{A \vdash e : \tau \quad \tau \leq \tau'}{A \vdash e : \tau'}$$

We write the *subtyping* $C, A \vdash e : \tau$ if $A \vdash e : \tau$ is derivable from the type containments C.

Exercise 9 ($\bullet$) Replace the rule (ARROW-COMP) by the rule

$$\frac{A \vdash \lambda x.ex : \tau}{A \vdash e : \tau} \quad x \notin FV(e) \qquad .$$

Is every subtyping $C, A \vdash e : \tau$ derivable in the original subtyping system (simply typed λ-calculus together with the above four subtyping rules) also derivable in the changed type system? Is the converse also true; i.e., is every subtyping derivable in the changed type system derivable in the original type system? $\square$

There are two basic interpretations of a type containment $\tau \leq \tau'$:

1. (subset interpretation) the set denoted by τ is contained in the set denoted by τ';

2. (coercion interpretation) there is a canonical (unique) mapping (*coercion*) from τ to τ'.

10.1 Atomic subtyping

In *atomic* subtyping the types in all given type containments are atomic; that is, primitive types such as integer, real, bool, etc. For example, $\mathit{integer} \leq \mathit{real}$ is an atomic type containment.

10.2 Record subtyping

In *record* (sub)typing we have record types

$$\tau ::= \ldots \mid \{x_1 : \tau_1, \ldots, x_k : \tau_k\}$$

and the following typing and subtyping rules.

$$\text{(REC-DEF)} \qquad \frac{e_1 : \tau_1, \ldots, e_k : \tau_k}{\{x_1 = e_1, \ldots, x_k = e_k\} : \{x_1 : \tau_1, \ldots, x_k : \tau_k\}}$$

$$\text{(REC-SEL)} \qquad \frac{e : \{x_1 : \tau_1, \ldots, x_k : \tau_k\}}{e.x_i : \tau_i}$$

$$\text{(REC-SUB)} \qquad \{x_1 : \tau_1, \ldots, x_k : \tau_k\} \leq \{x_{i_1} : \tau_{i_1}, \ldots, x_{i_l} : \tau_{i_l}\}$$

$$\text{(REC-COMP)} \qquad \frac{\tau_1 \leq \tau'_1 \ldots \tau_k \leq \tau'_k}{\{x_1 : \tau_1, \ldots, x_k : \tau_k\} \leq \{x_1 : \tau'_1 \ldots x_k : \tau'_k\}}$$

In the rule (REC-SUB) the indices $i_1, \ldots, i_l$ are pairwise distinct and form a subset of $1 \ldots k$. It expresses that every record can be considered a record with fewer fields, simply by ignoring some fields.

10.3 Subtyping with bottom type

Type $\bot$ ("bottom") models the empty type that no terminating expressions belong to. It has the rule

$$\text{(BOT)} \qquad \bot \leq \tau$$

but no introduction rule; that is, there is no rule that "introduces" expressions of type $\bot$.

Note: This type can be used in *strictness* analysis: if $\vdash e : \bot \to \bot$ then e is an expression that when given a nonterminating expression returns a nonterminating expression (KM89).

10.4 Subtyping with top type

Type $\top$ ("top") models the universal type: all expressions have this type. Its only rule is

$$\text{(TOP)} \qquad\qquad \tau \leq \top$$

Together with rule (COERCE) this rule implies that every typable expression e has type $\top$.

Note: This type can be used in *binding-time* analysis and *partial type* discipline: in the former $\top$ is the (compile-time) type of expressions that are evaluated at run-time (not already at compile-time) (Gom91), and in the latter $\top$ is the type of all tagged objects (Tha88).

11 Complexity of type inference

How difficult is it to decide whether an expression e is typable in these type systems? Here are a few results.

Theorem 11.1 *(Complexity of simple type inference)*
There is a linear-time algorithm for deciding whether an expression is simply typable*. Furthermore, every polynomial-time decidable problem can be reduced to simple typability in logarithmic space. In other words: simple typability is P-complete.*

PROOF Linear-time reduction of type inference to unification (Wan87), linear-time algorithm for unification (PW78); *log*-space reduction of type inference to unification (folk theorem), P-completeness of unification (DKM84) □

Theorem 11.2 *(Complexity of ML type inference)*
ML typability (typability with polymorphic definitions) is DEXPTIME-complete.

PROOF (Mai90; KMM91; KTU90a) □

Theorem 11.3 *(Complexity of type inference with polymorphic recursion)*
Typability with polymorphic recursion is (recursively) undecidable.

PROOF Reduction of semi-unification to type inference with polymorphic recursion (Hen89; KTU89), undecidability of semi-unification (KTU90b) □

Theorem 11.4 *(Complexity of type inference with atomic subtypings)*
Typability with (certain, fixed) atomic type containments is PSPACE-complete.

PROOF (LM92; Fre02) □

Theorem 11.5 *(Complexity of type inference for System F) Typability in System F (Girard/Reynolds Calculus) is undecidable.*

PROOF (Wel93) □

Most of the lower bounds can be strengthened using the technique of type-invariant simulation (Hen90a; Hen90b; HM94; Urz97). It consists of identifying reduction steps that not only have a subject reduction property, but also the converse; that is where the reduct has a type if *and only if* the reductum has that type and then showing how many steps of an arbitrary Turing Machine can be encoded using those reduction steps.

In this fashion *inseparability* results can be obtained, which roughly say not only that a particular type system has a particular complexity, but no type system that *contains it* can have a *lower* complexity.

- Any sound type system that contains simple typability is P-hard.
- Any sound type system that contains ML-typability or rank-2 bounded System F typability is DEXPTIME-hard (Hen90a; Hen90b; HM94)
- Any sound type system that contains rank-1 of System F_ω is undecidable (Urz97).

Interestingly, it is not known whether or not there exists a decidable sound type system that contains all of System F.

12 Constraint-based polymorphic type inference

Historically, Milner introduced Algorithms W and J (Mil78) for ML-type inference, which execute by structurally building a principal type for an expression from the the principal types of their subexpressions. Nowadays, constraint-solving techniques are increasinglhy used. They consist of the following steps:

1. Find a *properly syntax-oriented* version of the type system where there is precisely one typing rule for each construct that derives the type for the whole construct from the types of its (immediate) subexpressions.

2. For each such rule characterize the relation of the types of the subexpressions and the whole expression that must be satisfied by constraints between them.

3. For a given program, parse it, annotate the subexpressions with unique type variables (the "unknowns") and collect all constraints according to the inference rules.

4. Solve the constraints; that is find a *substitution* for the type variables that satisfies all the constraints.

Note that the separation of constraint generation and constraint solving reflects a separation of concerns: The constraints and their generation depend on the input program's syntax, but the constraint solution is independent of where the constraints come from.

Let us execute these steps for ML-style type inference. The type system is as follows:

$$\text{(VAR)} \qquad A\{x:\tau\} \vdash x:\tau$$

$$\text{(ABSTR)} \qquad \frac{A\{x:\tau\} \vdash e:\tau'}{A \vdash \lambda x.e : \tau \to \tau'}$$

$$\text{(APPL)} \qquad \frac{A \vdash e:\tau \to \tau' \quad A \vdash e':\tau}{A \vdash ee' : \tau'}$$

$$\text{(FIX-M)} \qquad \frac{A\{x:\tau\} \vdash e':\tau}{A \vdash \mathbf{fix}\, x.e' : \tau}$$

$$\text{(LET)} \qquad \frac{A \vdash e':\sigma' \quad A\{x:\sigma'\} \vdash e:\sigma}{A \vdash \mathbf{let}\, x = e' \,\mathbf{in}\, e : \sigma}$$

$$\text{(GEN)} \qquad \frac{A \vdash e:\sigma}{A \vdash e : \forall\alpha.\sigma} \quad \text{if } \alpha \text{ not free in } A$$

$$\text{(INST)} \qquad \frac{A \vdash e : \forall\alpha.\sigma}{A \vdash e : \sigma[\tau/\alpha]}$$

Note that it is syntax-oriented (the premises contain only expressions that occur in the corresponding consequents), but not properly syntax-oriented: The (GEN) and (INST) rules do not change the subject expression e.

12.1 Properly syntax-oriented type system

An equivalent properly syntax-oriented version is as follows. For convenience we retain which construct gives rise to which variable binding and we writing type assumptions on variables as a sequence reflecting the scoping in the program (the rightmost binding represents the innermost scope).

$$\text{(VAR-M)} \qquad A, \lambda x : \tau, A' \vdash x : \tau$$

$$\text{(VAR-P)} \qquad A, \mathbf{let}\ x : \sigma, A' \vdash x : \tau \qquad (\sigma \preceq \tau)$$

$$\text{(VAR-F)} \qquad A, \mathbf{fix}\ x : \tau, A' \vdash x : \tau$$

$$\text{(ABSTR)} \qquad \frac{A, \lambda x : \tau\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \to \tau'}$$

$$\text{(APPL)} \qquad \frac{A \vdash e : \tau \to \tau' \quad A \vdash e' : \tau}{A \vdash ee' : \tau'}$$

$$\text{(FIX-M)} \qquad \frac{A\{x : \tau\} \vdash e' : \tau}{A \vdash \mathbf{fix}\ x.e' : \tau}$$

$$\text{(LET)} \qquad \frac{A \vdash e' : \tau' \quad A\{\mathbf{let}\ x : \sigma'\} \vdash e : \tau}{A \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau} \quad \left(\begin{array}{l} \sigma' = \forall \vec{\alpha}.\tau' \text{ where} \\ \vec{\alpha} = FTV(\tau') - FTV(A) \end{array} \right)$$

Here a $\sigma \preceq \tau$ means that τ must be a substitution instance of σ: $\sigma \preceq \tau$ if and only if $\sigma = \forall \vec{\alpha}.\tau'$ and $\tau = S(\tau')$ where $domain\, S \subseteq \vec{\alpha}$. E.g., $\forall \alpha.\alpha \to \alpha \preceq int \to int$.

$FTV(\tau')$ and $FTV(A)$ denote the set of free type variables ocurring in τ' and in A, respectively. A' in the (VAR-x) rules must not contain a type assumption for x.

12.2 Constraint generation

By renaming all types in the inference rules to be unique type variables we arrive at the following constraint formulation:

$$\text{(VAR-M)} \qquad A, \lambda x : \alpha_x, A' \vdash x : \alpha'_x \quad (\alpha_x = \alpha'_x)$$

$$\text{(VAR-P)} \qquad A, \mathbf{let}\ x : \alpha_x, A' \vdash x : \alpha'_x \quad (\alpha_x \preceq \alpha'_x)$$

$$\text{(VAR-F)} \qquad A, \mathbf{fix}\ x : \alpha_x, A' \vdash x : \alpha'_x \quad (\alpha_x = \alpha'_x)$$

$$\text{((ABSTR)} \qquad \frac{A, \lambda x : \alpha_x \vdash e : \alpha_e}{A \vdash \lambda x.e : \alpha_{\lambda x.e}} \quad (\alpha_{\lambda x.e} = \alpha_x \to \alpha_e)$$

$$\text{(APPL)} \qquad \frac{A \vdash e : \alpha_e \quad A \vdash e' : \alpha_{e'}}{A \vdash ee' : \alpha_{ee'}} \quad (\alpha_e = \alpha_{ee'} \to \alpha_{e'})$$

$$\text{(FIX-M)} \qquad \frac{A, \mathbf{fix}\ x : \alpha_x \vdash e' : \alpha_{e'}}{A \vdash \mathbf{fix}\ x.e' : \alpha_{fix\ x.e'}} \quad (\alpha_x = \alpha_{e'} \wedge \alpha_x = \alpha_{fix\ x.e'})$$

$$\text{(LET)} \qquad \frac{A \vdash e' : \alpha_{e'} \quad A, \mathbf{let}\ x : \alpha_x \vdash e : \alpha_e}{A \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \alpha_{let\ x=e'\ in\ e}} \quad \left(\begin{array}{l} \alpha_x = \mathit{Close}(\alpha_{e'}, A) \wedge \\ \alpha_e = \alpha_{let\ x=e'\ in\ e} \end{array} \right)$$

To satisfy the constraint $\alpha_x = \mathit{Close}(\alpha_{e'}, A)$ a substitution must satisfy $S(\alpha_x) = \forall \vec{\beta}.S(\alpha_{e'})$ where $\vec{\beta} = FTV(S(\alpha_{e'})) - FTV(S(A))$.

12.3 Quantifier-free constraints

The occurrence of *Close* in a constraint is awkward. In Algorithm W style algorithms they are avoided by interleaving constraint generation and constraint solving: When processing a **let** -construct the constraints generated for it are solved and the *Close*-operation is performed on the result type of e' instead of a generating a formal constraint for it. (Indeed in Algorithm W this immediate solving is done for each construct, not just for **let** -expressions.)

Close can be avoided in a different fashion (Hen88; Hen93), without requiring eager solving based on the following observations. Consider the rule for **let** .

- A type scheme $\forall \vec{\alpha}.\tau'$ consists of a simple type (τ') and an explicit

specification ($\vec{\alpha}$) of which type variables in τ are *generic* (can be instantiated). The same information can be represented by τ together with the type variables that are *nongeneric* (must *not* be instantiated).

- The nongeneric type variables are exactly those that occur free in A, the bindings in whose scope the **let** -expression occurs.
- The nongeneric type variables are exactly those that occur in λ- and **fix** -bindings in A.

So all we need to do is associate a **let**-bound variable with its *unquantified* simple type and remember to prevent instantiation of the type variables in λ- and **fix** -bindings containing the **let** -binding.

$$\text{(VAR-M)} \qquad A, \lambda x : \alpha_x, A' \vdash x : \alpha'_x \quad (\alpha_x = \alpha'_x)$$

$$\text{(VAR-P)} \qquad A, \mathbf{let} x : \alpha_x, A' \vdash x : \alpha'_x \quad \left(\begin{array}{l} (\alpha_x, \vec{\beta}) \sqsubseteq (\alpha'_x, \vec{\beta}) \\ \text{where } \vec{\beta} = LTV(A) \end{array} \right)$$

$$\text{(VAR-F)} \qquad A, \mathbf{fix}\, x : \alpha_x, A' \vdash x : \alpha'_x \quad (\alpha_x = \alpha'_x)$$

$$\text{((ABSTR)} \qquad \frac{A, \lambda x : \alpha_x \vdash e : \alpha_e}{A \vdash \lambda x.e : \alpha_{\lambda x.e}} \quad (\alpha_{\lambda x.e} = \alpha_x \to \alpha_e)$$

$$\text{(APPL)} \qquad \frac{A \vdash e : \alpha_e \quad A \vdash e' : \alpha_{e'}}{A \vdash ee' : \alpha_{ee'}} \quad (\alpha_e = \alpha_{ee'} \to \alpha_{e'})$$

$$\text{(FIX-M)} \qquad \frac{A, \mathbf{fix}\, x : \alpha_x \vdash e' : \alpha_{e'}}{A \vdash \mathbf{fix}\, x.e' : \alpha_{fix\, x.e'}} \quad (\alpha_x = \alpha_{e'} \wedge \alpha_x = \alpha_{fix\, x.e'})$$

$$\text{(LET)} \qquad \frac{A \vdash e' : \alpha_{e'} \quad A, \mathbf{let}\, x : \alpha_x \vdash e : \alpha_e}{A \vdash \mathbf{let}\, x = e'\, \mathbf{in}\, e : \alpha_{let\, x=e'\, in\, e}} \quad \left(\begin{array}{l} \alpha_x = \alpha_{e'} \wedge \\ \alpha_e = \alpha_{let\, x=e'\, in\, e} \end{array} \right)$$

Here $LTV(A)$ denotes the sequence of types (actually type variables) bound to the λ- and **fix** -bindings in A. E.g., $LTV(\lambda x : \alpha_x, \mathbf{let}\, y : \alpha_y, \mathbf{fix}\, z : \alpha_z) = (\alpha_x, \alpha_z)$.

The constraint $(\alpha_x, \vec{\beta}) \sqsubseteq (\alpha'_x, \vec{\beta})$ is a *semi-unification constraint* (Hen88): A substitution S satisfies it if there exists R such that $S(\alpha_x, \vec{\beta}) = R(S(\alpha'_x, \vec{\beta}))$. Note that associating $\vec{\beta}$ with both α_x and α'_x ensures that R is the identity on all type variables that occur in $S(\vec{\beta})$. These are the nongeneric variables in since they occur free in λ- and **fix** -bindings in A.

Exercise 10 The above system is equivalent to a system with monomorphic recursion (FIX-M). Change the constraint-based inference systems above include polymorphic excursion (FIX-P). □

References

[CF58] H. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[CHS72] H. Curry, J. Hindley, and J. Seldin. *Combinatory Logic*, volume II of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1972.

[Cur69] H. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.

[Dam84] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984. Technical Report CST-33-85 (1985).

[DKM84] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *J. Logic Programming*, 1:35–50, 1984.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, January 1982.

[Fre02] Alexandre Frey. Satisfying subtype inequalities in polynomial space. *Theoretical Computer Science*, 277(1-2):105–117, April 2002.

[Gir71] J. Girard. Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–92, 1971.

[GLT89] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[Gom91] C. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *Transactions on Programming Languages and Systems*, 1991. To appear in TOPLAS special issue with selected papers from IEEE Computer Society 1990 International Conference on Computer Languages.

[Hen88] Fritz Henglein. Type inference and semi-unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 184–197, New York, NY, USA, July 1988. ACM.

[Hen89] Fritz Henglein. *Polymorphic Type Inference and Semi-Unification*. PhD thesis, Rutgers University, April 1989. Available as NYU Technical Report 443, May 1989, from New York University, Courant Institute of Mathematical Sciences, Department of Computer Science, 251 Mercer St., New York, N.Y. 10012, USA.

[Hen90a] Fritz Henglein. A lower bound for full polymorphic type inference: Girard-Reynolds typability is DEXPTIME-hard. Ruu-cs-90-14, Utrecht University, April 1990.

[Hen90b] Fritz Henglein. A simplified proof of DEXPTIME-completeness of ML typing. Manuscript, March 1990.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, April 1993.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, December 1969.

[Hin83] R. Hindley. The completeness theorem for typing lambda-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[HM94] Fritz Henglein and Harry Mairson. The complexity of type inference for higher-order typed lambda calculi. *Journal of Functional Programming (JFP)*, 4(4):435–477, October 1994.

[KM89] T. Kuo and P. Mishra. Strictness analysis: A new perspective based on type inference. In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 260–272. ACM Press, September 1989.

[KMM91] P. Kanellakis, H. Mairson, and J. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic — Essays in Honor of Alan Robinson*. MIT Press, 1991.

[KTU89] A. Kfoury, J. Tiuryn, and P. Urzyczyn. Computational consequences and partial solutions of a generalized unification problem. In *Proc. 4th IEEE Symposium on Logic in Computer Science (LICS)*, June 1989.

[KTU90a] A. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is DEXPTIME-complete. In *Proc. 15th Coll. on Trees in Algebra and Programming (CAAP), Copenhagen, Denmark*, pages 206–220. Springer, May 1990. Lecture Notes in Computer Science, Vol. 431.

[KTU90b] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proc. 22nd Annual ACM Symp. on Theory of Computation (STOC), Baltimore, Maryland*, pages 468–476, May 1990.

[LM92] P. Lincoln and J. Mitchell. Algorithmic aspects of type inference with subtypes. In *Proc. 19th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programmin Languages (POPL), Albuquerque, New Mexico*, pages 293–304. ACM Press, January 1992.

[Mai90] H. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proc. 17th ACM Symp. on Principles of Programming Languages (POPL)*. ACM, January 1990.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Mit91] J. Mitchell. Type inference with simple subtypes. *J. Functional Programming*, 1(3):245–285, July 1991.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Conf. on Programming, LNCS 167*, 1984.

[PW78] M. Paterson and M. Wegman. Linear unification. *J. Computer and System Sciences*, 16:158–167, 1978.

[RDR88] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.

[Rey83] J. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

[Tha88] S. Thatte. Type inference with partial types. In *Proc. Int'l Coll. on Automata, Languages and Programming (ICALP)*, pages 615–629, 1988. Lecture Notes in Computer Science.

[Tof92] M. Tofte. Principal signatures for higher-order program modules. In *Proc. 19th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico*, pages 189–199. ACM Press, January 1992.

[Urz97] Pawel Urzyczyn. Type reconstruction in f_ω. *Mathematical Structures in Computer Science (MSCS)*, 7:329–358, 1997.

[Wad89] P. Wadler. Theorems for free! In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 347–359. ACM Press, September 1989.

[Wan87] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.

[Wel93] J.B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. Preliminary Draft, August 1993.