Universal Types

Matthias Horbach

April 30, 2003

Abstract

In this paper we will describe how an improtant restriction of simply typed λ–calculus, namely that we cannot write polymorphic terms, can be overcome by means of an extension called System F.

Based mainly on the book *Types and Programming Languages* [1] by Benjamin Pierce, we will discuss important properties of this system, but also its limitations.

Contents

1 Polymorphism

1.1 Problems with Simply Typed λ–Calculus

The simply typed λ–calculus is sufficient to express monomorphic functions. However it cannot adequately model polymorphic functions.

Consider the function

```
double_Nat = λf:Nat→Nat. λy:Nat. f(f y)
```

Simply typed λ–calculus requires `double` to be unnecessarily restricted to functions from integers to integers when we would have liked to define it polymorphically for functions $\alpha \to \alpha$ from an arbitrary type α to itself. The identity function can similarly be defined only for specific types such as integers:

```
id_Nat = λx:Nat. x
id_Bool = λx:Bool. x
id_NatToNat = λx:Nat→Nat. x
```

We could of course write an identity function for every argument type we need, but when the code grows longer, e.g. if we write sorting functions for lists of natural numbers, for lists of booleans, and so on, then this will soon become annoying. And if we want to change an algorithm, then we have to do this for each and every type we need. This is certainly not what we want.

There is still another, more theoretical aspect: It seems that we cannot capture the fact that the form of the identity function does not depend on any specific type. We cannot express the idea of a functional form that is the same for a variety of types, and we must explicitly bind variables and values to a specific type at a time when such binding may be premature.

1.2 Varieties of Polymorphism

As we have just seen, simply typed λ–calculus needs to be extended. There are many possibilities to do so, hence we will first take a look at different ways in which programming languages introduce values with varying types.

Conventional typed languages, such as Pascal, are based on the idea that functions, and hence their operands, have a unique type. Such languages are said to be *monomorphic*, in the sense that every value and variable can be interpreted to be of one and only one type. Monomorphic programming

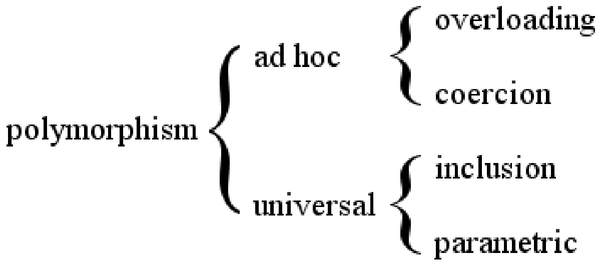

Figure 1: Varieties of polymorphism

languages may be contrasted with *polymorphic* languages in which some values and variables may have more than one type. Polymorphic functions are functions whose operands and results can have more than one type.

Strachey [2], and later on Cardelli and Wegner [3] distinguished, informally, between two major kinds of polymorphism.[1] *Universal polymorphism* is obtained when a function works uniformly on a range of types: these types normally exhibit some common structure. *Ad-hoc polymorphism* is obtained when a function works on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type.

There are two major kinds of ad hoc polymorphism. In *overloading* the same variable name is used to denote different functions, and the context is used to decide which function is denoted by a particular instance of the name. We may imagine that a preprocessing of the program will eliminate overloading by giving different names to the different functions; in this sense overloading is just a convenient syntactic abbreviation. A *coercion* is instead a semantic operation which is needed to convert an argument to the type expected by a function, in a situation which would otherwise result in a type error. Coercions can be provided statically, by automatically inserting them between arguments and functions at compile time, or may have to be determined dynamically by runtime tests on the arguments.

There are also two major kinds of universal polymorphism, i.e. two major ways in which a value can have many types. In *inclusion polymorphism*[2] an object can be viewed as belonging to many different classes which need not be disjoint, i.e. there may be inclusions of classes. In *parametric polymorphism*, a polymorphic function has an implicit or explicit type parameter, which determines the type of the argument for each application of that function.

In the next chapter, we will see how parametric polymorphism can help

[1] Their listing is not complete, and it is simplified, but this will suffice for our purpose.

[2] Inclusion polymorphism was added to Strachey's original model by Cardelli and Wegner.

solve the problems from 1.1, and we will give an overview over important properties of this kind of polymorphism.

2 System F

2.1 Historical Remark

Although our motivation comes from a programmer's point of view, the first one to "invent" universal polymorphism was *Jean-Yves Girard*, who worked on proof theory in logic and tried to extend the Curry-Howard correspondence to predicate logic. In his doctoral thesis from 1972 [4], he called his system *System F*.

Only two years later, *John Reynolds* [5] independently developed the same system, motivated by computer science. He used the term *polymorphic λ–calculus*, which already shows that it can be seen as a direct extension of simply typed λ–calculus, a construction that we will discuss below.

2.2 An Extension of Simply Typed λ–Calculus

Let us reconsider our example of the identity function. We want to avoid to write the same piece of code over and over, so we would like to be able to "abstract out" the type annotations. In simply typed λ–calculus, we already have a mechanism for doing this with terms: λ–abstractions. The way to extend the abstraction of terms out of terms to an abstraction of types out of terms is just reusing the already known notation for terms. Such type abstractions and type applications are written as

$$\lambda\mathtt{X.t}$$

and

$$\mathtt{t'\ [T]}$$

In these terms `X` is a *type variable*, `t` and `t'` are both terms, and `T` is some type.[3]

Concerning typing we get a new class of types when doing type abstraction, corresponding to the arrow type of term abstractions: the universal types $\forall\mathtt{X.T}$. The new typing rule is

[3]Type application is often written `t [T]`, and we will adopt this notation for better readability, though `t T` would be more consistent. We will use $\lambda\mathtt{X.t}$ as notation for type abstractions, whereas some autors prefer $\Lambda\mathtt{X.t}$.

$$\frac{\Gamma\texttt{,X} \vdash \texttt{t : T}}{\Gamma \vdash \lambda\texttt{X.t : }\forall\texttt{X.T}} \quad \text{T-TAbs}$$

When a type argument is given, the type of the application is computed as known from term applications:

$$\frac{\Gamma \vdash \texttt{t : }\forall\texttt{X.T}}{\Gamma \vdash \texttt{t [T'] : [T'/X]T}} \quad \text{T-TApp}$$

The context Γ,X in the rule T-TAbs denotes that a type variable X has been bound by a type abstraction. Note that we only want to evaluate terms that are closed (regarding the respective contexts), so all free occurences of (type) variables have to be bound in Γ. In addition, we assume that α–renaming is done whenever needed to avoid capturing.

When term abstraction and term application meet during evaluation of a simply typed term, the pair forms a redex. This is canonically extended to type abstraction and type application:

$$\frac{\texttt{t} \longrightarrow \texttt{t'}}{\texttt{t [T]} \longrightarrow \texttt{t' [T]}} \quad \text{E-TApp}$$

$$(\lambda\texttt{X.t) [T]} \longrightarrow \texttt{[T/X]t} \quad \text{E-TappTabs}$$

A complete list of the syntax and all evaluation and typing rules of System F is given in table 1. Additions with respect to simply typed λ–calculus are highlighted.

2.3 Examples

Now that we have the machinery, we will discuss the examples from 1.1, starting with the identity function. We abstract out the type information to create a single polymorphic function[4]

```
id = λX. λx:X. x;

> id: ∀X. X → X
```

and instantiate it to receive any concrete identity function that may be required:

```
id [Nat];

> <fun> : Nat → Nat
```

[4]Written in ML, id would look similar: `val id = fn x:'a => x`, with the automated typing output `'a id: 'a → 'a`

Syntax

`t ::=`	terms
`x`	variable
`λx:T.t`	abstraction
`t t`	application
`λX.t`	type abstraction
`t [T]`	type application
`v ::=`	values
`λx:T.t`	abstraction
`λX.t`	type abstraction
`T ::=`	types
`X`	type variable
`T→T`	function type
`∀X.T`	universal type
`Γ ::=`	contexts
`∅`	empty context
`Γ,x:T`	term variable binding
`Γ,X`	type variable binding

Evaluation

$$\frac{\texttt{t1} \longrightarrow \texttt{t1'}}{\texttt{t1 t2} \longrightarrow \texttt{t1' t2}} \quad \textsc{E-App1}$$

$$\frac{\texttt{t2} \longrightarrow \texttt{t2'}}{\texttt{v1 t2} \longrightarrow \texttt{v1 t2'}} \quad \textsc{E-App2}$$

$$(\lambda\texttt{x:T.t})\ \texttt{v} \longrightarrow [\texttt{v/x}]\texttt{t} \quad \textsc{E-AppAbs}$$

$$\frac{\texttt{t} \longrightarrow \texttt{t'}}{\texttt{t [T]} \longrightarrow \texttt{t' [T]}} \quad \textsc{E-TApp}$$

$$(\lambda\texttt{X.t})\ [\texttt{T}] \longrightarrow [\texttt{T/X}]\texttt{t} \quad \textsc{E-TappTabs}$$

Typing

$$\frac{\texttt{x:T} \in \Gamma}{\Gamma \vdash \texttt{x : T}} \quad \textsc{T-Var}$$

$$\frac{\Gamma,\texttt{x:T1} \vdash \texttt{t : T2}}{\Gamma \vdash \lambda\texttt{x:T1.t : T1→T2}} \quad \textsc{T-Abs}$$

$$\frac{\Gamma \vdash \texttt{t1 : T1→T2} \quad \Gamma \vdash \texttt{t2 : T1}}{\Gamma \vdash \texttt{t1 t2 : T2}} \quad \textsc{T-App}$$

$$\frac{\Gamma,\texttt{X} \vdash \texttt{t : T}}{\Gamma \vdash \lambda\texttt{X.t : ∀X.T}} \quad \textsc{T-TAbs}$$

$$\frac{\Gamma \vdash \texttt{t : ∀X.T}}{\Gamma \vdash \texttt{t [T'] : [T'/X]T}} \quad \textsc{T-TApp}$$

Table 1: The polymorphic λ-calculus (System F)

The evaluation of a term works exactly as one would expect:

```
id [Nat] 5
=    (λX. λx:X. x) [Nat] 5
⟶  [Nat/X](λx:X. x) 5
=    (λx:Nat. x) 5
⟶  [5/x](x)
=    5
```

A more useful example is the doubling function:

```
double = λX. λf:X→X. λx:X. f(f x);

> double: ∀X. (X→X) → X → X
```

The abstraction on the type `X` allows us to obtain doubling functions for specific types by instanciating `double` with different type arguments:

```
doubleNat = double [Nat];

> doubleNat: (Nat→Nat) → Nat → Nat

doubleBool = double [Bool];

> doubleBool: (Bool→Bool) → Bool → Bool

doubleNatToNat = double [Nat→Nat];

> doubleNatToNat: ((Nat→Nat)→Nat→Nat) →
                    (Nat→Nat) → Nat → Nat)
```

Once instancitated with a type argument, `double` can be further applied to an actual function and argument of appropriate types:

```
doubleNat (λx:Nat. succ(x)) 3;

> 5: Nat
```

Here is a last but trickier example: self–application. In simply typed λ–calculus there is no way to type `λx.x x`. In System F, on the other hand, this term becomes typable, if we give `x` a polymorphic type and instantiate it appropriately:

```
selfapp = λx:∀X.X→X. (x [∀X.X→X] x);

> selfapp: (∀X.X→X) → (∀X.X→X)
```

In a concrete example, one does not always need an explicit `selfapp` function. A self application of `double` for example looks like that:

```
quadruple = λX. double [X→X] (double [X]);

> quadruple: ∀X. (X→X) → X → X
```

2.4 Basic Properties

By now, we have developed a system that seems to be well behaved in many examples, but we do not yet have any results on whether this impression is justifiably.

Reconsidering simply typed λ–calculus, we want to ensure that at least some basic properties (e.g. the orthogonality of evaluation and typing) are not destroyed by our extension. The most important properties will be stated here, along with ideas how to prove them.

Theorem 1 (Uniqueness of Types). *Every well–typed System F term has exactly one type.*

Proof. Straightforward induction. □

Theorem 2 (Type Preservation). *If* $\Gamma \vdash \texttt{t : T}$ *and* $\texttt{t} \longrightarrow \texttt{t'}$*, then also* $\Gamma \vdash \texttt{t':T}$.

Proof idea. As for simply typed λ–calculus, the proof works by induction on the derivations of $\Gamma \vdash \texttt{t : T}$. At each step of the induction, we assume that the desired property holds for all subderivations (i.e. if $\Gamma \vdash \texttt{s : S}$ and $\texttt{s} \longrightarrow \texttt{s'}$, then $\Gamma \vdash \texttt{s' : S}$ whenever $\Gamma \vdash \texttt{s : S}$ is proved as a subderivation of the present one) and proceed by case analysis on the last rule used in the derivation.
The only part that is not straightforward is the type application rule E-TappTabs, for which one needs the additional substitution lemma

$$\Gamma,\texttt{X},\Delta \vdash \texttt{t : T} \Longrightarrow \Gamma,\texttt{[S/X]}\Delta \vdash \texttt{[S/X]t : [S/X]T}$$

The extra environment Δ is used to gain a strong enough induction hypothesis. □

Theorem 3 (Progress). *If* `t` *is a closed, well–typed term, then either* `t` *is a value or else there is some well–typed term* `t'` *such that* $\texttt{t} \longrightarrow \texttt{t'}$.

Proof idea. Again straightforward induction on typing derivations. The only interesting cases are the ones for applications, i.e. `t = t1 t2`.

- One possibility is ⊢ `t1: T1→T2` and ⊢ `t2:T1`. By induction hypothesis, either `t1` is a value or else it can be further evaluated. The same holds for `t2`. If either `t1` or `t2` can be evaluated, either E-App1 or E-App2 applies to `t`. If both `t1` and `t2` are values, then `t1` has the form `λx:T1.t'`, so either rule E-AppAbs applies to `t`.

- If otherwise ⊢ `t1: ∀X.T2` and `t2=[T1]`, then the same thoughts show that either `t1` can be evaluated and hence E-Tapp applies to `t`, or `t1` is a value (necessarily of form `λX.t'`) and rule E-TappTabs applies to `t`.

□

Theorem 4 (Strong Normalization). *Every well–typed System F term* `t` *is strongly normalizing, i.e. every maximal chain of evaluations of* `t` *is finite and ends with the same term* `t'`*.*

Unlike the theorems above, the last one is difficult to prove. Girard proved the normalization property as part of his doctoral thesis [4]. In fact, it is not obvious that normalization does hold at all. We will see in 2.6 that one can code powerful functions like a sorting function in System F. So one might well ask whether it is possible to write well–typed System F terms that do not evaluate to a normal form. The normalization theorem refutes this.

2.5 Erasure, Typability and Type Reconstruction

There are some astonashing implications of the normalization theorem. One is that there are untypable terms, e.g.

```
(λx.x x) (λx.x x)
```

This term has no normal form, and hence it cannot be typable. Observe that this term is valid in the pure λ–calculus (without typing). So what terms can be typed? And in how far can this typing be automized?

To answer these questions, we will formalize the notions of typing and type reconstruction. First of all, we need a connection between System F and untyped λ–calculus. To do so, we inductively define a mapping `erase`:

```
erase(x)        = x
erase(λx:T.t)   = λx.erase(t)
erase(t t')     = erase(t) erase(t')
erase(λX.t)     = erase(t)
erase(t [T])    = erase(t)
```

A term m in the untyped λ–calculus is said to be *typable*, if there is some well–typed term t such that `erase(t) = m`. The *type reconstruction* problem then asks, given an untyped term m, whether we can find a well–typed term t that erases to m.

It took until 1994 to solve the problem whether type reconstruction is possible, and finally *Wells* [6] showed that in general it is not:

Theorem 5. *Type reconstruction for System F is undecidable.*

But complete erasure brings along some severe problems as soon as we consider a complete programming language that allows for side effects like references or exceptions. For example if we add an exception raising primitive `error`, then the term

```
let f = λX.error in 0
```

evaluates to `0`, because `λX. error` is a value and the `error` in its body is never evaluated, while the erasure

```
let f = error in 0
```

raises an exception when evaluated. This can only be prevented if the abstraction is somehow maintained in the erasure.

So one might argue that full type reconstruction is perhaps too strong a demand, and that at least the positions of type abstraction and type application should be known beforehand, yielding[5]

```
erase(x)        = x
erase(λx:T.t)   = λx.erase(t)
erase(t t')     = erase(t) erase(t')
erase(λX.t)     = λ_.erase(t)
erase(t [T])    = erase(t) []
```

But in 1985 *Boehm* [7] proved type reconstruction for this partial erasure to be undecidable as well.

At first glance, this seems to make System F a toy for theorists, which is not suitable for real programming languages, as automatic type checking and reconstruction is a very convenient tool for a programmer. But Boehms results inspired others to invent partial type reconstruction techniques that will sometimes fail but work well in practice.

[5]The "_" in the rule for `erase(λX.t)` denotes any unused variable. One might as well insert the variable `X` from the left hand side.

These considerations show that, although the elegance and power of System F make it ideal for theoretical studies of polymorphism, the loss of type reconstruction is a drawback, whenever a real programming language is designed. The good news is that the full power of System F usually remains unused, so that one often needs only fragments of the system.

The most popular such fragment is the *let-polymorphism* of ML, in which type variables may only be instantiated by monotypes (i.e. types that are not universally quantified). There are System F terms that cannot be adequately modeled in ML (cf. 2.7), but ML type reconstruction is known to be decidable.

2.6 Parametricity and its Implications

Let us have a look at terms of some special types, e.g. terms of type `∀X.X→X`:

```
id = λX. λx:X.x
λX. λx:X.id [X] x
λX. λx:X.(λy:X->X. y) (id [X]) x
```

These terms look different, but their bodies share the same normal form with respect to evaluation, and regarded as functions, they all behave the same way: For any type X, they expect an argument of type X and return this argument. In fact, every term of type `∀X.X→X` will be such a function, because if the type of an argument is unknown, then we cannot apply any functions changing this argument.

Other systems might allow terms like

```
λX. λx:X. if X=Nat then x+1 else x,
```

but System F prevents this. This fact, that a polymorphic program behaves uniformly, i.e. independent of the argument types, is known as *parametricity.* This principle was first discovered by Reynolds in 1974 [5].

We can only give a short glance at the effects of this powerful principle. An introduction with many examples can be found in [8]. Consider terms of type `∀X.X→X→X`. A term of this type must request a type and two arguments of this type, so it has to start with `λX. λt:X. λf:X`. In fact, there are only two of these behaving differently:

```
λX. λt:X. λf:X. t
λX. λt:X. λf:X. f
```

So if we adopt the naive model of types as sets, this type has exactly two elements.[6] But all types with two elements can be regarded as the type Bool. So System F contains hidden structure and we can model booleans within this System:

```
CBool = ∀X.X→X→X

tru = λX. λt:X. λf:X. t
fls = λX. λt:X. λf:X. f
```

This kind of representation is called *Church encoding*. All of the usual functions on booleans can be modeled as well, e.g.

```
not = λb:CBool. (λX. λt:X. λf:X. b [X] f t)
and = λb:CBool. λc:CBool.
        λX. λt:X. λf:X. b [X] (c [X] t f) f
or  = λb:CBool. λc:CBool.
        λX. λt:X. λf:X. b [X] t (c [X] t f)
```

Even case distinctions are easily included:

```
if  = λX. λb:CBool. λc1:X. λc2:X. b [X] c1 c2
```

So System F really contains a representation for the booleans. The type ∀X.X→X from above corresponds to the type Unit, and uninhabited types as ∀X.X or ∀X.∀Y.X→Y to the empty type Null. But even more complex types can be encoded. The natural numbers can be inductively defined as $\{0, s(0), s(s(0)), \ldots\}$, based on a successor function and one starting constant. This can be encoded as well:

```
CNat = ∀X. (X→X) → X → X

c0 = λX. λs:X→X. λz:X. z
c1 = λX. λs:X→X. λz:X. s z
succ = λn:CNat. (λX. λs:X→X. λz:X. s (n [X] s z))
plus = λm:CNat. λn:CNat. m [CNat] succ n
times = λm:CNat. λn:CNat. λX. λx:X→X. n [X] (m [X] s)
exp = λm:CNat. λn:CNat. λX. n [X→X] (m [X])
```

A bit more complicated, but even more interesting, is the encoding of lists:

[6] There is a small problem here: As we will see in 2.7, there is no naive set–theoretic model of System F. But the essential ideas do not change in more suitable models.

```
List X = ∀R.(X→R→R) → R → R

nil = λX. (λR. λc:X→R→R. λn:R. n)
cons = λX. λhd:X. λtl:List X.
         (λR. λc:X→R→R. λn:R. c hd (tl [R] c n))
```

We will not go further into the understanding of lists, but give an example of the expressivity of System F: a sorting function in the pure language, without extras as fixpoint operators or loops. We use a let construct and pairs and omit some type annotations to simplify the code.

```
insert = λX. λleq:X→X→bool. λl:List X. λe:X.
   let res = l [List X * List X]
      (λhd:X. λacc:List X * List X.
         let rest = acc.1
            newrest = hd::rest
            restwithe = acc.2
            newrestwithe =
               if leq e hd
               then e::hd::rest
               else hd::restwithe
         in  (newrest, newrestwithe) )
         (nil[X], e::nil[X])
   in res.2

sort = λX. λleq:X→X→bool. λl:List X.
   l [List X] (λhd:X. λtl:List X. insert [X] leq tl hd)
      (nil [X])
```

The possibility to encode all this follows from the parametricity of System F, so if we design a full–scale programming language with System F as its core, we can add these features as primitives without changing the expressivity or disturbing the properties of the core language.

2.7 Impredicativity

Before learning what impredicativity means, we have to do a short excursion into set theory. Its naive version (where sets were defined by predicates, and every predicate defined a set) tripped over *Russel's Paradox*. The problem out of which the paradox arises is that the membership condition may make a reference to the set that is being defined. For instance in Russel's Paradox this reference is non–self–membership, a quantification over a set of elements that includes the very element that is defined. Such membership conditions were called *impredicative*. So impredicativity is the stumbling block of naive set theory.

When looking at polymorphic systems, the distinction between impredicativity and predicativity is once again done by a distinction of how quantification works.

When a System F type is universally quantified, e.g. `T = ∀X.X→X`, then the type variable ranges over *all* types, thus also over `T` itself. We can actually instantiate a term of type `T` with `T`, which yields a function of type `T→T`. Hence System F is an impredicative system, and its types cannot be modeled as sets.

In contrast to that, languages like ML are predicative: There is type quantification in ML, but this quantification ranges only over quantifier–free types, i.e. arguments of a function can only be monomorphic.

To clearify the difference, let us look at the doubling function.

The functions

```
double = λX. λf:X→X. λx:X. f(f x);

> double: ∀X. (X→X) → X → X

id = λX. λx:X. x;

> id: ∀X. X → X
```

have an analogue in ML:

```
fun double f x = f (f x);

> 'a double: ('a→'a) → 'a → 'a

fun id x = x;

> 'b id: 'b → 'b
```

The application of both also works similarly: A term like `double [Nat] id` (where `id` is a polymorphic term), is invalid in System F; instead we must write

```
double [Nat] (id [Nat]);

> <fun>: Nat → Nat double [Nat] (id [Nat]) 3;

> 3: Nat
```

to receive well–typed terms. Here the argument `id [Nat]` is of monomorphic type `Nat→Nat`. If we write

```
double id 3;
```

```
> 3: Nat;
```

in an ML-like language, then `id` is instantiated in this term to a monomorphic function:

```
fn x:Nat => x
```

So far so good. But System F also allows us to write the doubling function in another way:

```
double2 = λY. λf:∀X. X→X. λx:Y. f [Y] (f [Y] x);
```

```
> double2: ∀Y. (∀X. X→X) → Y → Y
```

such that we can apply the polymorphic identity function:

```
double2 [Nat] id;
```

```
> <fun>: Nat → Nat
```

There is no way to imitate this in a predicative system.

3 Summary

Even simple examples show that simply typed λ–calculus is not as expressive as we would like. Starting from the wish to write polymorphic functions, which is impossible in simply typed λ–calculus, we have explored polymorphic λ–calculus, or System F, an extension in which these problems do no longer exist. We have seen that in many ways this calculus behaves like the base calculus, but there are additional advantages implied by the parametricity and the impredicativity of the system, that show its expressiveness.

Nevertheless important problems such as (full) type reconstruction are not decidable, so System F is not perfect, and language designers have to make a trade-off between the convenience of automated type checkers and the potentially available expressiveness.

References

[1] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[2] Christopher Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, 1967.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[4] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[5] John C. Reynolds. Towards a theory of type structure. *Proc. Colloque sur la Programmation*, pages 408–425, 1974.

[6] Joe B. Wells. Typability and type checking in the 2nd order λ–calculus are equivalent and undecidable. *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 176–185, 1994.

[7] Hans Böhm. Partial polymorphic type inference is undecidable. *26th Annual Symposium on Foundations of Computer Science*, pages 339–345, 1985.

[8] Philip Wadler. Theorems for free! *Functional Programming Languages and Computer Architecture*, pages 347–359, 1989.