# Colored Local Type Inference

Martin Odersky   Christoph Zenger   Matthias Zenger

École Polytechnique Fédérale de Lausanne
{odersky,czenger,zenger}@di.epfl.ch

**Abstract**

We present a type system for a language based on $F_{\leq}$, which allows certain type annotations to be elided in actual programs. Local type inference determines types by a combination of type propagation and local constraint solving, rather than by global constraint solving. We refine the previously existing local type inference system of Pierce and Turner[PT98] by allowing partial type information to be propagated. This is expressed by coloring types to indicate propagation directions. Propagating partial type information allows us to omit type annotations for the visitor pattern, the analogue of pattern matching in languages without sum types.

## 1 Introduction

Many modern programming languages are based on type systems which combine a notion of objects and subtyping with parametric polymorphism [Str91, Mey92, CDG$^+$92, NC97, OW97, PT98, BOSW98]. A popular basis for such type systems is $F_{\leq}$, the second-order lambda calculus with subtyping. While $F_{\leq}$ is an excellent basis for explaining the abstract type structure of programs, it is less suitable as a kernel language for concrete source programs, because of the excessive amount of type information that needs to be written by the programmer.

Most programmers would agree that some kinds of type annotations are useful as a vehicle for program documentation whereas others are annoying because they only repeat information that can easily be deduced from the context. For instance, a type signature for a globally defined function is generally useful, while an explicit type parameter in a function application is often annoying, in particular when the same information can be deduced from the types of the function's actual value parameters.

Local type inference [PT98] aims at eliminating the need for annoying explicit type information. It does this with two techniques. First, type parameters in a function application are inferred from the function's value parameters by solving a constraint system which relates formal with actual argument types. Second, it propagates known types down the syntax tree in order to infer some types of formal value parameters and provide additional guidance to type parameter inference. For instance, if function $\mathsf{f}$ is known to have type $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$, then $\mathsf{f}\ (\textbf{fun}\ (\mathsf{x})\ \mathsf{x} + 1)$ would be well-typed, since the type of parameter $\mathsf{x}$ can be inferred to be $\mathsf{Int}$ by propagating the known type $\mathsf{Int} \to \mathsf{Int}$ down the tree.

This *bidirectional local type inference* can be formalized in a type system where every type rule of $F_{\leq}$ is split into two rules, one for the case where the result type is known, the other for the case where it is unknown. Compared to a type inference algorithm, a formalization of a type system in terms of typing rules is attractive because it provides a contract which can be understood by both users and implementers of a programming language. For this reason, bidirectional local type inference is put forward as the type inference technology of the ML2000 draft proposal [ACF$^+$].

The downward type propagation in bidirectional local type inference works only if the propagated type is completely known. If only some part of a type is known, downward propagation is disabled and types are instead inferred by propagating information from the leaves of the tree upwards. For instance, if $\mathsf{g}$ is known to have type $\forall \mathsf{a}.(\mathsf{Int} \to \mathsf{a}) \to \mathsf{a}$, then

$$\mathsf{g}\ (\textbf{fun}\ (\mathsf{x})\ \mathsf{x} + 1)$$

would *not* be well-typed since the type information known from the outside about the anonymous function is only partial. The outside type information is $\mathsf{Int} \to \mathsf{a}$ where the instance type of the type variable $\mathsf{a}$ has yet to be determined.

In this paper we study a refinement of bidirectional local type inference, where partial as well as total type information can be propagated down the tree. Our approach types essentially all programs which are typable under bidirectional local type inference as well as other programs. For instance, it can type the function application of $\mathsf{g}$ above.

Idioms like the one of $\mathsf{g}$ above arise naturally in Church encodings of parameterized types. Consider for instance the implementation of lists in a language with recursive records, but without sum types. Taking the Church encoding of sum types as a guide, we can implement pattern matching on lists with *visitor* records [GHJV94]. That is, the list type can be represented as a record with a single method, $\mathsf{match}$, which takes a list visitor as argument:

```
type List[a] = {
    match [b] (v: ListVisitor[a, b]): b
}
```

A list visitor is a record which contains two function-valued fields, which are here called caseNil and caseCons. The first function is invoked when the encountered list is Nil, the second when it is not.

```
type ListVisitor[a,b] = {
    caseNil (): b,
    caseCons (x: a, xs: List[a]): b
}
```

The implementations of the list constructors Nil and Cons are then evident:

```
Nil [a] (): List[a] = {
    match v = v.caseNil ()
}
Cons [a] (x: a, xs: List[a]): List[a] = {
    match v = v.caseCons (x, xs)
}
```

As an example of a client using lists and list visitors, consider the standard implementation of the append function.

```
append [c] (xs: List[c], ys: List[c]): List[c] =
    xs.match {
        caseNil () = ys,
        caseCons (x, xs1) = Cons (x, append (xs1, ys))
    }
```

Note the close correspondence between the application of match and a pattern matching case expression in a language with algebraic data types. The methods of the list visitor correspond one-to-one to the branches of a case expression. The same principle can be applied to represent systematically every sum type in a language which has only product types. Most object-oriented languages fall into this category. A systematic application of the visitor pattern can obviate the need for a complication of the type structure in these languages.

That this representation is feasible in practice has been demonstrated in the case of the Pizza [OW97] and GJ [BOSW98] compilers. The Pizza compiler, written in Pizza, made heavy use of algebraic data types and pattern matching. The GJ compiler was derived from the Pizza compiler, but it was written in GJ, which does not have algebraic data types. Most pattern matching expressions in the Pizza compiler were represented by applications of the visitor pattern in the GJ compiler. The switch did not lead to a significant decrease in readability.

Unfortunately, bidirectional local type inference cannot type the body of append above. It requires the parameter types of the caseCons to be given explicitly:

```
append [c] (xs: List[c], ys: List[c]): List[c] =
    xs.match {
        caseNil () = ys,
        caseCons (x : c, xs1 : List[c]) =
            Cons (x, append (xs1, ys))
    }
```

The reason is that the match method of lists is polymorphic. Hence, no type information is propagated into the argument of match and we have to write the parameter types of all methods in the visitor explicitly. This type information is redundant, since it can easily be derived from the type of xs, the receiver of the match application. Most programmers would therefore classify the added type annotations as annoying rather than helpful.

What's required here is that we propagate the knowledge that we are dealing with a ListVisitor into the body of the list visitor record. This knowledge is only partial, since we do not know yet the return type b of the type ListVisitor[a,b] to be propagated. The inference system presented here can deal with such partial type information and can therefore type the first definition of append without auxiliary type annotations in visitor methods.

A type which is completely known from the context of the term to be typed and which is propagated inwards (down the tree) is called "inherited", in analogy to inherited attributes in attribute grammars [Knu68]. By contrast, a type which is propagated outwards (up the tree) is called "synthesized". We write ${}^{\vee}T$ for inherited types and ${}_{\wedge}T$ for synthesized types. In general, a type can have both inherited and synthesized parts. For instance, the list visitor argument for the match method in the code of append would be inferred to have type ${}^{\vee}\mathsf{ListVisitor}[\mathsf{c}, {}_{\wedge}\mathsf{List}[\mathsf{c}]]$. This indicates that we expect from the outside a ListVisitor with first type parameter c, and that the second type parameter is found to be List[c] by typing the visitor record itself.

A type without prefix is allowed to consist of arbitrary inherited or synthesized parts. To embed such a type $T$ in an inherited or synthesized context, we use a $\diamond$ prefix. For example

$$
{}^{\vee}({}_{\wedge}T \to \diamond U)
$$

represents a function type where the function type constructor is inherited, its argument type $T$ is synthesized, and its result type $U$ is arbitrary.

The inherited and synthesized parts of a type can alternatively and more concisely be characterized by coloring them. A second version of this paper intended for color output [OZZ00] uses a red font for inherited parts of a type and a blue font for synthesized parts. Black color is reserved for types with arbitrary inherited or synthesized parts.

In the rest of this paper we develop these ideas in a type system for second order lambda calculus with records and subtyping. We first define a subtyping relation between colored types which reflects the information given in the colors. The subtyping relation is designed such that it allows the definition of a subsumption rule. For instance, an inherited record type ${}^{\vee}\{x : a, y : b\}$ cannot be a subtype of the smaller inherited record type ${}^{\vee}\{x : a\}$, because this would mean that type information about $y$ is "guessed" at a subsumption step in the proof rather than being propagated from further up the tree. On the other hand, it is true that

$$
{}_{\wedge}\{x : a, y : b\} \;\leq\; {}^{\vee}\{x : a\} \;.
$$

We next define a type system that assigns colored types to terms and show how it can be used to infer missing type information for explicitly typed $F_{\leq}$. This type system essentially subsumes the bidirectional local type inference system of Pierce and Turner. A minor deviation is presented at the end of section 6. We finally present a local type inference algorithm which can propagate partial type information down the tree. The algorithm is not formulated with colored types. Instead, we split a colored type into a *prototype* that contains the information which is propagated down the tree, and a type, which represents the completely computed type of a term. Missing information in the prototype is expressed by the special symbol "?". We have shown that the algorithm is sound and complete with respect to the type system.

The type system is presented here with second order lambda calculus as the source language, but its ideas have much wider applicability. For instance, we have used it in the design and implementation of the type system for the functional net language Funnel [Ode00], which is based on join calculus [FG96]. Colored types are useful in general for describing information flow in polymorphic type systems with propagation-based type inference.

**Related Work** There is a long thread of research on type inference for extensions of the Hindley/Milner system or for higher-order lambda calculus. A particularly large body of work is concerned with the extensions we deal with, first-class polymorphism and subtyping. Typically, type inference algorithms for extensions of the Hindley/Milner system are complete, whereas algorithms for variants of second order lambda calculus are incomplete, since the basic type inference problem for $F_2$ is known to be undecidable [Wel94]. Extensions of the Hindley/Milner system with first class polymorphism [OL96, Jon97, GR99] distinguish between polymorphic type schemes and monomorphic types. They differ in the methods how to convert from one to the other. Their type inference algorithm is always based on some form of first-order unification. Similar in motivation to these is Pfenning's work on type inference for $F_2$ [Pfe88], which uses higher-order unification. Extensions of the Hindley/Milner system with subtyping have also been studied [AW93, TS96, EST95, Pot96, Nor98, Pot98]. They are usually based on constrained types [OSW99], which include a set of subtype constraints as part of a type. A problem in practice is that constraint sets can become very large. Trifonov and Smith [TS96] as well as Pottier [Pot98] have proposed schemes to address the problem. A more radical alternative has been proposed by Nordlander [Nor98] and Peyton Jones and Wansbrough [WJ00]. They approximate constrained types with unconstrained types in the generalization step. Nordlander's system [Nor98] heuristically unifies type variables with their bounds. He uses a scheme similar to ours to pass partial type information, treating wildcards (that correspond to our "?") as unique fresh type variables. Roughly similar to subtype polymorphism, but incomparable in expressive power, are Rémy's row variables [Rém89]. A system which combines first-class polymorphism with row variables, as studied in [GR99], can express many aspects of $F_{\leq}$, and admits a complete and decidable type inference algorithm based on unification. But the resulting type system tends to become fragile and complex and the necessary encodings in user programs can be a bit roundabout.

The type inference problem for $F_{\leq}$, which combines subtyping with first-class polymorphism, has been addressed by Cardelli's [Car93] greedy algorithm, which unifies type variables with their bounds, as soon as these are encountered in a subtype constraint. This usually works well in practice, but it does not admit an independent characterization of the output in a type system.

Clearly closest to our work is Pierce and Turner's work on local type inference [PT98]. The main extension over their work is our refinement of type propagation. Whereas their propagation of type information is "all or nothing", we also admit propagation of partial information via colored types. Both forms of type inference rely largely on propagation instead of (global) constraint solving.

A form of local type inference is also used in GJ [BOSW98]. In fact, GJ does not even have a type parameter construct for polymorphic method applications. It relies totally on local type inference for this task. Like Java, GJ requires complete type signatures for all variables and parameters to be given. The techniques discussed here are therefore not directly relevant for type inference in GJ.

Propagation of information along the edges of a syntax tree is also the idea underlying attribute grammars [Knu68]. Our synthesized types correspond to synthesized attributes whereas inherited types correspond to inherited attributes. Attribute grammars cannot express attributes that have inherited as well as synthesized components.

Litvinov [Lit98] develops a type system for Cecil [CT98], which is comparable to ours. Cecil has both structural subtyping and subtyping by name and it can deal with recursive bounds. However, the type inference is heuristic, with no decidability result.

Xi and Pfenning [XP99] also use an "all or nothing" bidirectional algorithm for type-checking. Their setting is an extension of the Hindley-Milner type system by dependent types. They use bidirectionality to infer quantifier elimination and introduction places.

The rest of this paper is structured as follows. Section 2 and 3 present the internal and external language of our calculus. Section 4 presents an example, section 5 and 6 discuss the subtype relation and the type system using colors. Section 7 outlines the local constraint resolution and section 8 the type inference algorithm. Section 9 concludes.

## 2 Internal Language

We have an internal and an external language. The internal language is essentially the fully typed second order lambda calculus with records and subtyping. The external language additionally provides constructs to elide some of the explicit type information needed in the internal language, thus reducing clutter in source programs. The task of type inference is to map the external into the internal language by reconstructing the elided type information.

The internal language is based on $F_{\leq}$ and is almost the same as the one of Pierce and Turner. The one extension with respect to their system is the introduction of record values and types, which help to streamline the encoding of object-oriented programs. The terms, types, and type environments of the internal language are given by the following grammar

$$\begin{array}{llcl}
\textbf{Terms} & E, F & = & x \mid \mathbf{fun}[\overline{a}](x:T)E \\
 & & \mid & F[\overline{T}](E) \mid E.x \\
 & & \mid & \{x_1 = E_1, \ldots, x_n = E_n\} \\
\textbf{Types} & T, S, R & = & a \mid \top \mid \bot \\
 & & \mid & T \xrightarrow{\overline{a}} S \\
 & & \mid & \{x_1 : T_1, \ldots, x_n : T_n\} \\
\textbf{Environments} & \Gamma & = & x : T \mid \epsilon \mid a \mid \Gamma, \Gamma'
\end{array}$$

$$\text{(VAR)}\quad \Gamma \vdash x : \Gamma(x)$$

$$\text{(ABS)}\ \frac{\Gamma, \overline{a}, x : T \vdash E : S}{\Gamma \vdash \textbf{fun}[\overline{a}](x : T)E : T \xrightarrow{\overline{a}} S}$$

$$\text{(APP)}\ \frac{\Gamma \vdash F : S \xrightarrow{\overline{a}} T \qquad \Gamma \vdash E : S' \qquad S' <: [\overline{R}/\overline{a}]S}{\Gamma \vdash F[\overline{R}](E) : [\overline{R}/\overline{a}]T}$$

$$\text{(APP}_\bot\text{)}\ \frac{\Gamma \vdash F : \bot \qquad \Gamma \vdash E : R}{\Gamma \vdash F[\overline{T}](E) : \bot}$$

$$\text{(SEL)}\ \frac{\Gamma \vdash F : \{x_1 : T_1, \ldots, x_n : T_n\}}{\Gamma \vdash F.x_i : T_i}$$

$$\text{(SEL}_\bot\text{)}\ \frac{\Gamma \vdash F : \bot}{\Gamma \vdash F.x_i : \bot}$$

$$\text{(REC)}\ \frac{\Gamma \vdash F_1 : T_1 \qquad \ldots \qquad \Gamma \vdash F_n : T_n}{\Gamma \vdash \{x_1 = F_1, \ldots, x_n = F_n\} : \{x_1 : T_1, \ldots, x_n : T_n\}}$$

$$\bot <: T \quad \text{(BOT)}$$

$$T <: \top \quad \text{(TOP)}$$

$$a <: a \quad \text{(VAR)}$$

$$\frac{T_1 <: T_1' \qquad \ldots \qquad T_n <: T_n'}{\{x_1 : T_1, \ldots, x_n : T_n, x_{n+1} : T_{n+1}, \ldots, x_m : T_m\} <: \{x_1 : T_1', \ldots, x_n : T_n'\}}\ \text{(REC)}$$

$$\frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1' \xrightarrow{\overline{a}} T_2 <: T_1 \xrightarrow{\overline{a}} T_2'}\ \text{(FUN)}$$

Figure 1: $\Gamma \vdash E : T$ and $T <: S$

A term is a variable $x$, a function abstraction $\textbf{fun}[\overline{a}](x : T)$ with formal type parameters $\overline{a}$ and value parameter $x : T$, a function application $F[\overline{T}](E)$, a record constructor $\{x_1 = E_1, \ldots, x_n = E_n\}$ or a record selector $E.x$. The overbar signifies tupling, with $\overline{a}$ equivalent to $a_1, \ldots, a_n$ for an unspecified $n$. The presence of records allows us to restrict our language to single-argument functions since polyadic functions can be straightforwardly encoded: $\textbf{fun}[\overline{a}](x : T, y : T')E$ is encoded as $\textbf{fun}[\overline{a}](r : \{x : T, y : T'\})[r.x/x, r.y/y]E$ and $f[\overline{R}](E, F)$ is encoded as $f[\overline{R}](\{x = E, y = F\})$.
A type is either a type variable $a$, a record type $\{x_1 : T_1, \ldots, x_n : T_n\}$ or a function type $T \xrightarrow{\overline{a}} S$. Only function types are polymorphic, and we write the polymorphic type variables over the function arrow; e.g. $T \xrightarrow{\overline{a}} S$ instead of the more customary $\forall \overline{a}.T \to S$. We also have two types $\bot$, $\top$ which are the least, respectively greatest type. Primitive types like $\mathsf{Int}$ are treated as free type variables. We identify types that are equivalent up to $\alpha$-renaming as well as record types with the same fields, possibly occurring in different order. $\mathrm{tv}(E)$ and $\mathrm{tv}(T)$ are the free type variables of the term $E$ and the type $T$ respectively.
The type system for the internal language and the corresponding subtype relation are presented in Figure 1. As in bidirectional local type inference, the type system for the internal language has no subsumption rule. This is only a matter of presentation, since at the point where we have to match types, in the application rule, we explicitly allow that the argument's type is a subtype of the formal argument type.

## 3 External Language

The external language is a superset of the internal language. There are two additional syntactic constructs, which let one omit type annotations.

$$\begin{array}{rcll} E, F & = & \ldots & \\ & \mid & \textbf{fun}(x)E & \text{lightweight abstraction} \\ & \mid & F(E) & \text{lightweight application} \end{array}$$

Our abstractions are even a bit more lightweight than the ones studied by Pierce and Turner [PT98], since we elide formal type variables as well as argument types, whereas they elide only argument types.
We say, a term $E$ is a *partial erasure* of $F$, if it can be obtained from $F$ by erasing type information, i.e. replacing abstractions by lightweight abstractions and applications by lightweight applications.
We use colors in types to represent the direction from where type information is propagated. Red color indicates a part of the type which is known from the context (inherited), blue color indicates a part of the type which is known from the term itself (synthesized). In black-and-white versions of the paper, colors are represented by up ticks ${}_\wedge$ and down ticks ${}^\vee$.
The syntax of synthesized types ${}_\wedge T$ is:

$$\begin{array}{rcl} {}_\wedge T, {}_\wedge S, {}_\wedge R & = & {}_\wedge a \mid {}_\wedge \top \mid {}_\wedge \bot \\ & \mid & {}_\wedge\{x_1 : {}_\wedge T_1, \ldots, x_n : {}_\wedge T_n\} \mid {}_\wedge({}_\wedge T \xrightarrow{\overline{a}} {}_\wedge S) \end{array}$$

Types which are propagated down the tree are called inherited types. They are written ${}^\vee T$ and their syntax is the dual of synthesized types.

$$\begin{array}{rcl} {}^\vee T, {}^\vee S, {}^\vee R & = & {}^\vee a \mid {}^\vee \top \mid {}^\vee \bot \\ & \mid & {}^\vee\{x_1 : {}^\vee T_1, \ldots, x_n : {}^\vee T_n\} \mid {}^\vee({}^\vee T \xrightarrow{\overline{a}} {}^\vee S) \end{array}$$

Synthesized and inherited types are special cases of general types which can have arbitrary inherited and synthesized parts. They are prefixed with $\diamond$ (In the colored version, they are black). The syntax of general types is:

$$\begin{array}{rcl} \diamond T, \diamond S, \diamond R & = & {}^\vee a \mid {}^\vee \top \mid {}^\vee \bot \\ & \mid & {}^\vee\{x_1 : \diamond T_1, \ldots, x_n : \diamond T_n\} \mid {}^\vee(\diamond T \xrightarrow{\overline{a}} \diamond S) \\ & \mid & {}_\wedge a \mid {}_\wedge \top \mid {}_\wedge \bot \\ & \mid & {}_\wedge\{x_1 : \diamond T_1, \ldots, x_n : \diamond T_n\} \mid {}_\wedge(\diamond T \xrightarrow{\overline{a}} \diamond S) \end{array}$$

Although we will use the adjectives "synthesized" and "inherited" in the following, we will often refer to the type's annotation as its "color".
To prevent too many annotations, the prefixes ${}^\vee$, ${}_\wedge$, and $\diamond$ are interpreted structurally on the type. So ${}_\wedge(T \xrightarrow{\overline{a}} S)$ is

actually identical to ${}_\wedge({}_\wedge T \xrightarrow{\overline{a}} {}_\wedge S)$ and in the following we will always use the former. Also, if we do not annotate the outermost type constructor, we assume it to be $\diamond$.
Type constructors are always inherited or synthesized. A constructor annotated with $\diamond$ means that it is either inherited or synthesized. If we write the types $\diamond T$, ${}_\wedge T$, ${}^\vee T$ within a single statement, they are assumed to be all structurally equivalent, differing only in color. If the same constructor occurs more than once annotated with $\diamond$, as in $\diamond\{x_1 : T_1, \ldots, x_n : T_n\} \leq \diamond\{x_1 : T'_1, \ldots, x_n : T'_n\}$, all occurrences of the constructor are assumed to have the same color.
*Substitutions* $[T/a]S$ on types are defined to be color preserving: $[T/a]{}_\wedge a = {}_\wedge T$ and $[T/a]{}^\vee a = {}^\vee T$.
Types in type environments are always synthesized. This ensures that the type of a variable is always determined by its definition and not influenced by the context of its usage.

$$\Gamma \quad = \quad x : {}_\wedge T \;\mid\; \epsilon \;\mid\; {}_\wedge a \;\mid\; \Gamma, \Gamma'$$

## 4 Example

We demonstrate the application of colored local type inference by means of an example, which details the definition of a map function over the type Option[a] using the visitor technique. The example is analogous to the list and list visitor example presented in the introduction, except that it avoids the use of recursive types (which are not covered here). The types Option[a] and OptionVisitor[a,r] are defined as follows:

```
type Option[a] = {
    match [r] (v: OptionVisitor[a,r]): r
}
type OptionVisitor[a,r] = {
    caseNone (): r,
    caseSome (y: a): r
}
```

Constructors for Option[a] are as follows:

```
None = fun[s](): Option[s] {
    match = fun(v) v.caseNone()
}
Some = fun[t](y: t): Option[t] {
    match = fun(v) v.caseSome(y)
}
```

To make presentation easier, we have added some syntactic sugar to our formal language definition. Type declarations introduce abbreviations for record types. A function abstraction with a return type

$$\mathbf{fun}\ (\mathsf{x}\!: T)\!: T'\ (E)$$

is considered equivalent to a function with an explicity typed body

$$\mathbf{fun}\ (\mathsf{x}\!: T)\!: (E\!: T')\ .$$

An explicitly typed expression such as $E\!: T$ is in turn considered equivalent to $(\mathbf{fun}\ (\mathsf{x}\!: T)\ \mathsf{x})\ E$.
Consider now a map function over optional values, which is written in the external language as follows.

$$\begin{array}{l}
\mathsf{map} = \mathbf{fun}[\mathsf{c,d}](\mathsf{f\!: c} \to \mathsf{d})\ \mathbf{fun}(\mathsf{x\!: Option[c]})\\
\quad (\quad (\mathsf{x}\!: {}^\vee\{match : {}_\wedge OptionVisitor[c,r] \xrightarrow{r} {}_\wedge r\}\\
\qquad\qquad \mathsf{.match})\!: {}^\vee({}_\wedge\left\{\begin{array}{l} caseNone : \{\} \to r\\ caseSome : c \to r\end{array}\right\} \xrightarrow{r} {}_\wedge r)\\
\qquad\quad \{\\
\qquad\qquad \mathsf{caseNone} = (\mathbf{fun}()\\
\qquad\qquad\quad (\mathsf{None}\!: {}^\vee({}_\wedge\{\} \xrightarrow{t} {}_\wedge Option[t])())\\
\qquad\qquad\quad : {}_\wedge Option[\bot])\\
\qquad\qquad : {}^\vee(\{\} \to {}_\wedge Option[\bot])\ ,\\
\qquad\qquad \mathsf{caseSome} = (\mathbf{fun}(\mathsf{y}\!: {}_\wedge c)\\
\qquad\qquad\quad (\mathsf{Some}\!: {}^\vee({}_\wedge t \xrightarrow{t} {}_\wedge Option[t])\\
\qquad\qquad\qquad (\mathsf{f}\!: {}^\vee({}_\wedge c \to {}_\wedge d)(\mathsf{y}\!: {}^\vee c))\!: {}_\wedge d)\\
\qquad\qquad\quad : {}_\wedge Option[d])\\
\qquad\qquad : {}^\vee(c \to {}_\wedge Option[d])\\
\qquad\quad \} : {}^\vee\left\{\begin{array}{l} caseNone : \{\} \to {}_\wedge Option[\bot]\\ caseSome : c \to {}_\wedge Option[d]\end{array}\right\})\\
\quad : {}_\wedge Option[d]\\
: {}_\wedge((c \to d) \xrightarrow{c,d} Option[c] \to Option[d])
\end{array}$$

Figure 2: map

```
map = fun[c,d](f: c → d) fun(x:Option[c])
  x.match {
    caseNone = fun() None(),
    caseSome = fun(y) Some(f(y))
  }
```

Figure 2 presents the same function together with internally computed type information for terms and formal parameters. Note that:

- In applications, we use the function's argument type for typing the actual argument. Here, the formal parameter of function x.match has the synthesized type ${}_\wedge OptionVisitor[c, r]$, which expands to

  $${}_\wedge\{caseNone : \{\} \to r, caseSome : c \to r\}.$$

  So for the actual argument, the visitor record, everything is given from outside, except for the second parameter type $r$ of OptionVisitor, which still needs to be instantiated.

- Therefore, function caseSome has type

  $${}^\vee(c \to {}_\wedge Option[d]).$$

  The argument type ${}^\vee c$ of the function is given from the outside.

- Therefore, the type of the formal parameter x of caseSome can be reconstructed; no explicit type annotation needs to be given.

## 5 Subtyping

Subtyping's role is to mediate differences when type information about a term is propagated both from the inside and

$$T \leq T \qquad\qquad \frac{T_1 \leq T_2 \qquad T_2 \leq T_3}{T_1 \leq T_3}$$

$${}_\wedge a \leq {}^\vee a \qquad {}_\wedge\bot \leq {}^\vee\bot \qquad {}_\wedge\top \leq {}^\vee\top \qquad {}_\wedge\bot \leq {}^\vee\top$$

$${}_\wedge\{x_1 : \diamond T_1, \ldots, x_n : \diamond T_n, x_{n+1} : {}^\vee\top, \ldots, x_m : {}^\vee\top\} \leq {}^\vee\{x_1 : \diamond T_1, \ldots, x_n : \diamond T_n\} \qquad {}_\wedge(\diamond T \xrightarrow{\overline{a}} \diamond S) \leq {}^\vee(\diamond T \xrightarrow{\overline{a}} \diamond S)$$

$${}_\wedge\bot \leq {}^\vee a \qquad {}_\wedge\bot \leq {}^\vee\{x_1 : {}_\wedge\bot, \ldots, x_n : {}_\wedge\bot\} \qquad {}_\wedge\bot \leq {}^\vee({}_\wedge\top \xrightarrow{\overline{a}} {}_\wedge\bot)$$

$${}_\wedge a \leq {}^\vee\top \qquad {}_\wedge\{x_1 : {}^\vee\top, \ldots, x_n : {}^\vee\top\} \leq {}^\vee\top \qquad {}_\wedge({}^\vee\bot \xrightarrow{\overline{a}} {}^\vee\top) \leq {}^\vee\top$$

$$\frac{T_1 \leq T_1' \qquad \ldots \qquad T_n \leq T_n'}{\{x_1 : T_1, \ldots, x_n : T_n\} \leq \{x_1 : T_1', \ldots, x_n : T_n'\}} \qquad \frac{T_1 \overline{<} T_1' \qquad T_2 \leq T_2'}{T_1' \xrightarrow{\overline{a}} T_2 \leq T_1 \xrightarrow{\overline{a}} T_2'}$$

Figure 3: $T \leq S$

$$(\text{VAR})\ \frac{\Gamma(x) = {}_\wedge T}{\Gamma \vdash^c x : {}_\wedge T} \qquad\qquad (\text{SUB})\ \frac{\Gamma \vdash^c E : T \qquad T \leq T'}{\Gamma \vdash^c E : T'}$$

$$(\text{ABS}_{tp})\ \frac{\Gamma, {}_\wedge\overline{a}, x : {}_\wedge T \vdash^c E : S}{\Gamma \vdash^c \mathbf{fun}[\overline{a}](x : T)E : {}_\wedge(T \xrightarrow{\overline{a}} \diamond S)} \qquad (\text{ABS})\ \frac{\Gamma, {}_\wedge\overline{a}, x : {}_\wedge T \vdash^c E : S \qquad \overline{a} \notin \text{tv}(E)}{\Gamma \vdash^c \mathbf{fun}(x)E : {}^\vee(T \xrightarrow{\overline{a}} \diamond S)}$$

$$(\text{APP}_{tp})\ \frac{\Gamma \vdash^c F : {}^\vee({}_\wedge S \xrightarrow{\overline{a}} {}_\wedge T) \qquad \Gamma \vdash^c E : [\overline{R}/\overline{a}]^\vee S}{\Gamma \vdash^c F[\overline{R}](E) : [\overline{R}/\overline{a}]_\wedge T}$$

$$(\text{APP})\ \frac{\begin{array}{c}\Gamma \vdash^c F : {}^\vee({}_\wedge S \xrightarrow{\overline{a}} {}_\wedge T) \qquad \Gamma \vdash^c E : S' \qquad S' \lhd_{\overline{a}} {}^\vee S \\ S' \leq [\overline{R}/\overline{a}]^\vee S \qquad [\overline{R}/\overline{a}]_\wedge T \leq T' \\ \forall \overline{R}', T''.(S' \leq [\overline{R}'/\overline{a}]^\vee S \ \wedge \ [\overline{R}'/\overline{a}]_\wedge T \leq T'' \sim T' \ \Rightarrow \ [\overline{R}/\overline{a}]_\wedge T \leq [\overline{R}'/\overline{a}]^\vee T)\end{array}}{\Gamma \vdash^c F(E) : T'}$$

$$(\text{SEL})\ \frac{\Gamma \vdash^c E : {}^\vee\{x : \diamond T\}}{\Gamma \vdash^c E.x : T} \qquad (\text{REC})\ \frac{\Gamma \vdash^c E_1 : \diamond T_1 \qquad \ldots \qquad \Gamma \vdash^c E_n : \diamond T_n}{\Gamma \vdash^c \{x_1 = E_1, \ldots, x_n = E_n\} : {}_\wedge\{x_1 : \diamond T_1, \ldots, x_n : \diamond T_n\}}$$

Figure 4: $\Gamma \vdash^c E : T$

from the outside. A synthesized type constructor of a given type matches with an inherited constructor of that same type, or of a supertype.

The subtype relation $\leq$ for colored types is shown in Figure 3. Since subtyping is contravariant in the argument type but colors are covariant, we have a second relation $\overline{<}$, which is the same as $\leq$ but with reversed colors.

In this subtype relation a structural change in the type always implies that the different constructors differ also in color. Going from the subtype to the supertype, we always change from synthesized to inherited. The synthesized type ${}_\wedge(\mathsf{Int} \to \mathsf{Int})$ for example is a subtype of ${}_\wedge(\mathsf{Int} \to {}^\vee\top)$ and ${}_\wedge(\mathsf{Int} \to \mathsf{Int}) \leq {}_\wedge({}^\vee\bot \to {}^\vee\top) \leq {}^\vee\top$. But ${}_\wedge(\mathsf{Int} \to \mathsf{Int})$ is not a subtype of ${}_\wedge\top$. Here, the topmost constructors differ, but they have the same color.

This ensures that we never guess types. Synthesized information is really coming from inside, it cannot be constructed using subsumption. A rule relating e.g. ${}_\wedge\bot \leq {}_\wedge(S \xrightarrow{\overline{a}} T)$ would destroy this property, because it would synthesize a function type which was guessed rather than propagated. Similarly, inherited information must really be given from the outside, it cannot be discarded by subsumption: If a type constructor is inherited at a certain point in the term, going outward we can discard the inherited constructor only explicitly in a rule (the origin of the information), not by using subsumption.

The subtype rules in Figure 3 are designed such that they derive smallest possible steps. Therefore we often rely on transitivity for deriving subtype judgements. For instance,

$${}_\wedge\bot \leq {}^\vee(\mathsf{Int} \to \{x : {}_\wedge\bot\})$$

via

$${}_\wedge\bot \leq {}^\vee({}_\wedge\top \to {}_\wedge\bot) \leq {}^\vee(\mathsf{Int} \to \{x : {}_\wedge\bot\}).$$

The subtype relations $<:$ for uncolored types of the internal language and $\leq$ for colored types have the following relationship:

**Lemma 5.1 (Subtyping).**

1. $T \leq S$ implies $T <: S$.
2. $T <: S$ implies ${}_\wedge T \leq {}^\vee S$,

The first statement says that $\leq$ is a restriction of $<:$; i.e. subsumption is sound in the internal language, where we disregard colors when using $<:$ on colored types. The second statement together with subsumption guarantees that a term of type ${}_\wedge T$ will typecheck against each supertype of $T$ given completely from outside.

As an example consider the typing of map. In the selection x.match we know from outside that x has to have a type of the form

$$^{\vee}\{match : {}_{\wedge}T \xrightarrow{r} {}_{\wedge}T'\}.$$

If we look up x in the type environment, we find the synthesized type

$$_{\wedge}\{match : OptionVisitor[a, r] \xrightarrow{r} r\}.$$

Subsumption gives us that x also has type

$$^{\vee}\{match : {}_{\wedge}OptionVisitor[a, r] \xrightarrow{r} {}_{\wedge}r\}$$

which is of the required form.

## 6 Colored Type System

The type system for the external language is formulated in terms of colored types. Typing rules for that system are given in Figure 4. They include a subsumption rule (SUB) which makes use of the subtpying relation defined in section 3. Most rules are straightforward.

Rule (VAR) is the usual tautology rule for variables. Variables in environments are always synthesized; i.e. their type has been completely defined at the point of their definition. Therefore, the (VAR) rule can be restricted to synthesized types $_{\wedge}T$ without loss of generality.

There are two rules for function abstraction. The rule ($\text{ABS}_{tp}$) for function abstraction with an explicit argument type produces a type with a synthesized arrow as top-level constructor, whereas the rule (ABS) for lightweight abstractions produces an inherited arrow. In other words, lightweight abstractions require a function type to be passed from the outside into the abstraction. The function's result type is in each case prefixed with a $\diamond$, which tells us that this type can be propagated in either direction.

In the untyped abstraction (ABS), the type $T$ of the function's argument is inherited. So it has to be known from the context. Here is an example for this:

$$\mathbf{fun}(\mathsf{x})\ \mathsf{x}\ +\ \mathsf{1} : {}^{\vee}(\mathsf{Int} \rightarrow {}_{\wedge}\mathsf{Int}).$$

This function type indicates that the argument type must be known from the context, whereas the result type is determined by the function itself.

Because formal type parameters are not explicitly mentioned in the term, we disallow their use in the function body (see the side condition of rule (ABS)). Pierce and Turner do not require this side condition since their system never elides formal type parameters.

If the context does not provide the required information on the argument type, we have to annotate it in the function definition. Example:

$$\mathbf{fun}(\mathsf{x} : \mathsf{Int})\ \mathsf{x}\ +\ \mathsf{1} : {}_{\wedge}(\mathsf{Int} \rightarrow \mathsf{Int})$$

As a consequence, that function has a purely synthesized type.

There are also two rules for function application. The rule ($\text{APP}_{tp}$) for typed function application is straightforward. The expression's function part $F$ needs to have a function type. This is the only requirement propagated into the function expression; the rest of the function type has to be synthesized. With this information we also know the type of the argument $E$, so it gets a purely inherited type, which just needs to be checked.

For example, a function f of type $_{\wedge}((\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int})$ can be applied to the term $\mathbf{fun}(\mathsf{x})\ \mathsf{x} + \mathsf{1}$, yielding

$$\mathsf{f}\ (\mathbf{fun}(\mathsf{x})\ \mathsf{x}\ +\ \mathsf{1}) : {}_{\wedge}\mathsf{Int}.$$

The type derivation of this judgement uses the subsumption step

$$^{\vee}(\mathsf{Int} \rightarrow {}_{\wedge}\mathsf{Int}) \leq {}^{\vee}(\mathsf{Int} \rightarrow \mathsf{Int})$$

for typing the function argument. This step weakens the synthesized result type $_{\wedge}\mathsf{Int}$ of the argument to the same type $^{\vee}\mathsf{Int}$ in inherited form.

Rule ($\text{APP}_{tp}$) is formulated in a way, which makes a separate rule for the case where the function's type is $\perp$ superfluous: Assume that the expression $F$ has type $_{\wedge}\perp$. By subsumption it also has type $^{\vee}({}_{\wedge}\top \xrightarrow{\overline{a}} {}_{\wedge}\perp)$. Now if the argument $E$ is typable with $^{\vee}\top$, we can conclude with ($\text{APP}_{tp}$) that $F(E)$ has the expected type $_{\wedge}\perp$.

By far the most complicated rule in our system is rule (APP) for function application without explicit type parameters. This is not surprising, since this rule plays the role of four different rules in Pierce and Turner's system. The premise of the rule requires again the function part $F$ of the expression to have a function type. The argument expression is then checked to have a type which coincides with the function's (inherited) argument type, except for occurrences of type parameters $a_i$, where an arbitrary synthesized type is required. This is expressed by the condition $S' \triangleleft_{\overline{a}} {}^{\vee}S$ in the premise. The auxiliary relation $\triangleleft_{\overline{a}}$ expresses a replacement of every occurrence of a variable in $\overline{a}$ by an arbitrary type. We do not try to take account of sharing at this point. Therefore different occurrences of the same type variable can be associated with different types. The relation is defined as follows.

$$^{\vee}T \triangleleft_{\overline{a}} {}^{\vee}T \qquad\qquad {}_{\wedge}T \triangleleft_{\overline{a}} {}^{\vee}a_i$$

$$\frac{T_1 \triangleleft_{\overline{a}} T_1' \quad \ldots \quad T_n \triangleleft_{\overline{a}} T_n'}{^{\vee}\{x_1 : \diamond T_1, \ldots, x_n : \diamond T_n\} \triangleleft_{\overline{a}} {}^{\vee}\{x_1 : \diamond T_1', \ldots, x_n : \diamond T_n'\}}$$

$$\frac{T \triangleleft_{\overline{a}} T' \qquad S \triangleleft_{\overline{a}} S'}{^{\vee}(\diamond T \xrightarrow{\overline{a}} \diamond S) \triangleleft_{\overline{a}} {}^{\vee}(\diamond T' \xrightarrow{\overline{a}} \diamond S')}$$

In other words, when type checking function arguments, we use a weaker constraint than the one implied by the function type, since different occurrences of the same type variable can be matched with different types. The next two conditions in the premise of rule (APP) tighten the constraint. They require the existence of a tuple of types $\overline{R}$, which, when substituted for the type variables $\overline{a}$ in the formal argument type $^{\vee}S$, yield a type which is a supertype of the actual argument type $S'$. Furthermore, when substituted for $\overline{a}$ in the function's result type $_{\wedge}T$, we require a type which is a subtype of the whole rule's result type $T'$.

The final premise of rule (APP) requires that the tuple of types $\overline{R}$ minimizes the function's result type when compared to any other solution which satisfies the same constraints. The premise makes use of the auxiliary relation $S \sim T$, which states that $S$ and $T$ coincide on their inherited parts. $\sim$ is defined as follows:

$$_{\wedge}T \sim {}_{\wedge}S \qquad\qquad {}^{\vee}T \sim {}^{\vee}T$$

$$\frac{T_1 \sim T_1' \qquad \dots \qquad T_n \sim T_n'}{{}^{\vee}\{x_1 : \diamond T_1, \dots, x_n : \diamond T_n\} \sim {}^{\vee}\{x_1 : \diamond T_1', \dots, x_n : \diamond T_n'\}}$$

$$\frac{T \sim T' \qquad S \sim S'}{{}^{\vee}(\diamond T \xrightarrow{\overline{a}} \diamond S) \sim {}^{\vee}(\diamond T' \xrightarrow{\overline{a}} \diamond S')}$$

For example, suppose we have a function g of type ${}_{\wedge}((\mathsf{Int} \to a) \xrightarrow{a} a)$ which is again applied to our term **fun**(x) x + 1. To show that

$$\mathsf{g}\ (\mathbf{fun}(\mathsf{x})\ \mathsf{x}\ +\ 1) : {}_{\wedge}\mathsf{Int},$$

we choose $R = \mathsf{Int}$ in the rule (APP). ${}^{\vee}(\mathsf{Int} \to {}_{\wedge}\mathsf{Int}) \triangleleft_a {}^{\vee}(\mathsf{Int} \to a)$ shows that the type of g provides enough information to type the argument, ${}^{\vee}(\mathsf{Int} \to {}_{\wedge}\mathsf{Int}) \leq {}^{\vee}(\mathsf{Int} \to \mathsf{Int})$ shows that actual and formal argument type match, and trivially $[\mathsf{Int}/a]_{\wedge}a \leq {}_{\wedge}\mathsf{Int}$ turns ${}_{\wedge}\mathsf{Int}$ into an appropriate result type. Clearly, Int is also the optimal choice here.

However, with a function $\mathsf{h}{:}(a \to a) \xrightarrow{a} a$ we cannot find a type for h (**fun**(x) x + 1). The condition $S' \triangleleft_a {}^{\vee}(a \to a)$, which requires that argument and result type of $S'$ are synthesized, fails for $S' = {}^{\vee}(\mathsf{Int} \to {}_{\wedge}\mathsf{Int})$ and all its supertypes, since they are all of the form ${}^{\vee}(\mathsf{Int} \to S'')$, where the argument type is inherited. This is not unexpected, since neither h nor **fun**(x) x + 1 have information on the type of x.

As a further example for the use of the lightweight application rule (APP), consider the function application f(x), where f has the synthesized type

$${}_{\wedge}(a \xrightarrow{a} (a \to a))$$

and x has type ${}_{\wedge}\mathsf{Int}$. Since ${}_{\wedge}\mathsf{Int} \triangleleft_a {}^{\vee}a$, x is a matching argument. But f(x) fails to have a purely synthesized type. The high level reason for this is that $a$ occurs co- and contravariantly in the result type and thus we cannot make an optimal choice for $R$. If we try to give f(x) the synthesized type ${}_{\wedge}(\mathsf{Int} \to \mathsf{Int})$, we have to choose $R = \mathsf{Int}$. The subtype requirements of the rule (APP) are fulfilled, but $R' = \top$ and

$$T'' = {}_{\wedge}(\top \to \top) \sim {}_{\wedge}(\mathsf{Int} \to \mathsf{Int})$$

provide an alternative which also fulfills the requirements. This falsifies the optimality constraint

$${}_{\wedge}(\top \to \top) \leq {}^{\vee}(\mathsf{Int} \to \mathsf{Int}),$$

and therefore does not yield a valid typing. Other choices for synthesized types fail for similar reasons.

On the other hand, we can verify that f(x) does have the inherited type

$${}^{\vee}(\mathsf{Int} \to \mathsf{Int}).$$

The choice $R = \mathsf{Int}$ is the same, but here the requirement

$$T'' \sim {}^{\vee}(\mathsf{Int} \to \mathsf{Int})$$

on $T''$ is stronger, and the only choice left for $R'$ is Int. Indeed,

$$[\mathsf{Int}/a]_{\wedge}(a \to a) \leq [\mathsf{Int}/a]^{\vee}(a \to a)$$

holds and the optimality constraint is satisfied. By analogous reasoning, f(x) also has the inherited type

$${}^{\vee}(\bot \to \bot).$$

Rule (SEL) specifies the typing of record selection. The premise of this rule uses an inherited record type constructor, which reflects the fact that when typing expression $E$ in a selection $E.x$, we know that $E$ must be a record with an $x$ field. The field's type can be propagated in either direction. Subsumption allows that the argument may have additional fields. However, all these additional fields have to have synthesized types. Allowing inherited types would enable us to guess them.

Finally, rule (REC) specifies the typing of record construction. The record type constructor appears in synthesized form in the conclusion of the rule, reflecting the fact that in a record formation $\{x_1 = E_1, \dots, x_n = E_n\}$ we know the shape of the constructed record without further context information. The types of the fields $x_1, \dots, x_n$, on the other hand, can again be propagated in either direction.

For example, assume we want to type

$$\{\ \mathsf{u}\ =\ \mathbf{fun}(\mathsf{x})\ \mathsf{x}\ +\ 1,\ \mathsf{v}\ =\ 3\ \}.\mathsf{u}$$

The record expression itself has type

$${}_{\wedge}\{u : {}^{\vee}(\mathsf{Int} \to {}_{\wedge}\mathsf{Int}), v : {}_{\wedge}\mathsf{Int}\}.$$

The selection rule expects that the qualifier has a record type with one $u$ component. We apply the subsumption rule to get the record type into the proper form

$${}^{\vee}\{u : \mathsf{Int} \to {}_{\wedge}\mathsf{Int}\}.$$

Here it was essential that the $v$ component had a synthesized type, because for subsumption we needed

$${}_{\wedge}\mathsf{Int} \leq {}^{\vee}\top.$$

We would not have been able to type

$$\{\ \mathsf{u}\ =\ \mathbf{fun}(\mathsf{x})\ \mathsf{x}\ +\ 1,\ \mathsf{v}\ =\ 3\ \}.\mathsf{v}$$

since the type's $u$ component is not purely synthesized and

$${}^{\vee}(\mathsf{Int} \to {}_{\wedge}\mathsf{Int}) \not\leq {}^{\vee}\top.$$

### Soundness and Completeness

We have shown, that the type system is sound and complete with respect to the internal language.

**Theorem 6.1 (Soundness)** If $\Gamma \vdash^c E : T$ then there exists a term $F$ such that $E$ is a partial erasure of $F$ with $\Gamma \vdash F : S$ and $S <: T$.

**Theorem 6.2 (Completeness)**. If $\Gamma \vdash E : T$ then $\Gamma \vdash^c E : {}_{\wedge}T$

Completeness is easy to show, as the rules for the lightweight terms need not be considered. For soundness we insert the derived argument types, formal type parameters, and actual type parameters into $E$ and show that the resulting term $F$ always has a unique type which is a subtype of $T$.

Note that the external language does not have a subject reduction property. For instance

$$(\mathbf{fun}(\mathsf{x}{:}\ \mathsf{Int}{\to}\mathsf{Int})\ \mathsf{x})\ (\mathbf{fun}(\mathsf{y})\ \mathsf{y} + 1)\ 3$$

has type ${}_{\wedge}\mathsf{Int}$. But, assuming standard reduction semantics, this expression reduces to (**fun**(y) y + 1) 3

which does not have a type.

One can still show type soundness of the external language by using type soundness for the internal language $F_{\leq}$ together with the soundness theorem 6.1, which relates the two languages.

### Variants

The optimality constraint in rule (APP) minimizes the instantiated version of the function's result type. Several other variants of this constraint are also possible.

*1. Taking account of inherited information.* There is one kind of term that bidirectional local type inference can type in checking mode, but colored local type inference cannot. To see this, consider again f(x) with f of type

$$_{\wedge}(a \xrightarrow{a} (a \to a))$$

and x of type $_{\wedge}\mathsf{Int}$. We saw that we can infer the types $^{\vee}(\bot \to \bot)$ and $^{\vee}(\mathsf{Int} \to \mathsf{Int})$, but we will explain now that we cannot infer $^{\vee}(\bot \to \mathsf{Int})$. If we try to give f the type $^{\vee}(\bot \to \mathsf{Int})$, we find with $R = \mathsf{Int}$ and $R = \bot$ two solutions for the substitution $[R/a]$, none of which is better than the other. Bidirectional local type inference infers $\bot \to \mathsf{Int}$ in checking mode.

This case can only appear in a polymorphic application where

- a type variable occurs co- and contravariantly in the function's result type ($a$ in the example),
- this type variable can be chosen in different ways ($\bot$ and Int), and
- the type is given completely from outside.

The last two points mean in particular that the type given from outside replaces co- and contravariant occurrences of the type variable with different types ($\bot$ and Int in the example).

We could solve this problem by using another optimality criterion for (APP):

$$S' \leq [\overline{R}'/\overline{a}]^{\vee}S \;\;\wedge\;\; [\overline{R}'/\overline{a}]_{\wedge}T \leq T'' \sim T' \;\;\Rightarrow\;\; _{\wedge}T' \leq {}^{\vee}T''.$$

Using this criterion, one could infer the type $^{\vee}(\bot \to \mathsf{Int})$ for f(x). But implementing the new criterion comes at a considerable cost in algorithmic complexity.

*2. Extending optimality for function arguments.* In practice, one might often want to restrict (APP) further instead of generalizing it. The problem is that rule (APP) as well as bidirectional local type inference do not always instantiate all type variables to unique types. The case of f(x) above is one where bidirectional local type inference succeeds, but does not instantiate the type variable $a$. Another case is the application g(x) where g has type $_{\wedge}(a \xrightarrow{a} \{\})$ and x has type $_{\wedge}\mathsf{Int}$. Here, both colored and bidirectional local type inference would succeed without determining an instance type for $a$. In fact, both Int and $\top$ would be possible instantiations. This indeterminacy is not a problem for languages which are parametric [Wad89], because in these languages the particular instantiation of a type variable cannot affect the result of a computation.

But most real world languages are not parametric – overloading, dynamic type casts, or inheritance with overriding all destroy parametricity. If parametricity does not hold, it is mandatory that all type variables are instantiated to unique types. In our case, this could be achieved by a strengthened optimality constraint in rule (APP), which requires that the argument type as well as the result type is minimized:

$$\begin{array}{l} S' \leq [\overline{R}'/\overline{a}]^{\vee}S \;\;\wedge\;\; [\overline{R}'/\overline{a}]_{\wedge}T \leq T'' \sim T' \;\;\Rightarrow \\ [\overline{R}/\overline{a}]_{\wedge}T \leq [\overline{R}'/\overline{a}]^{\vee}T \;\;\wedge\;\; [\overline{R}/\overline{a}]_{\wedge}S \leq [\overline{R}'/\overline{a}]^{\vee}S \end{array}$$

This variation requires only minimal changes to the inference algorithm.

*3. Dealing gracefully with non-variant result types.* The optimality criteria given so far rely on a minimization of a function's result type. Hence, if the function's result type is non-variant in the type variable(s) to be instantiated, a best solution often does not exist and type parameters have to be given explicitly. As an example, consider a version of the List type augmented by an append function:

```
type List[a] = {
    match [b] (v: ListVisitor[a, b]): b
    append (ys: List[a]): List[a]
}
```

This type is non-variant in the type variable a, since a appears covariantly and contravariantly in the type of append.[1] Consider now a function singleton which creates a one-element list.

```
singleton [a] (x: a): List[a] = Cons (x, Nil[a]())
```

Intutitively, one would expect

```
singleton("abc"): List[String]
```

but with the optimality criteria given so far we get instead an ambiguity type error. The problem is that both [String/a] and [$\top$/a] are legal instantiations of singleton's type parameter, a. The two instantiations lead to two result types List[String] and List[$\top$], with neither of the two being better than the other.

Ambiguities like these can be avoided by arbitrarily picking one instantiation over the others. Our current implementation always picks minimal instance types. That is, it instantiates type variables which are non-variant in the function result type to the smallest type which is consistent with the local constraints. With this modification, we get the expected type List[String] for singleton("abc"). Our experience indicates that the modification greatly reduces the number of required type annotations in programs which deal with non-variant types.

Even after the modification, there is in practice one more rough edge in the treatment of non-variant types. Consider an occurrence of a parameter-less constructor of a non-variant type, such as Nil for non-variant List, and assume that there is no inherited type information. With our original optimality constraint, Nil() is ambiguous, since it has types List[String] and List[$\top$], among others. With our modified optimality constraint, we get instead

[1] A purely co-variant version of append could be written if type variables with lower bounds were permitted: append[b >: a](ys: List[b]): List[b]. But a type system with bounds like these is beyond the scope of the present paper.

$\mathsf{Nil}()\colon \mathsf{List}[\bot]$.

The problem is that $\mathsf{List}[\bot]$ is not a very useful type for $\mathsf{Nil}()$, because it is not a subtype of any other list type (assuming that lists are non-variant). If one wanted a list of String, one would need either a type parameter as in Nil[String](), or an explicit type annotation, such as Nil(): List[String]. The local type inference algorithm for GJ has a neat solution to this problem by adding an "unknown type" $*$ to the internal type language. Types with $*$-parameters can be implicitly widened to types with arbitrary types at corresponding positions. Some syntactic restrictions prevent duplication of $*$-types and thus guarantee the soundness of the widening rule. It is not clear yet, whether the GJ solution can be generalized to the setting considered here.

## 7 Constraint Resolution

In the type inference we need to find the locally best solution for actual type parameters of polymorphic functions in lightweight applications. This requires solving a set of subtype constraints. We can use the techniques introduced in bidirectional local type inference.

An *$\overline{a}$-constraint set* $C$ is a set of inequations $T <: a <: T'$, where $(\mathrm{tv}(T) \cup \mathrm{tv}(T')) \cap \overline{a} = \emptyset$. We abbreviate $T <: a <: \top$ by $T <: a$ and $\bot <: a <: T'$ by $a <: T'$.

An *$\overline{a}$-substitution* $\sigma$ is an idempotent substitution with $\mathrm{dom}(\sigma) = \overline{a}$. An $\overline{a}$-substitution $\sigma$ is a *solution* for an $\overline{a}$-constraint set $C$ if we have $T <: \sigma a <: T'$ for each $T <: a <: T'$ in $C$.

During type inference we generate $\overline{a}$-constraint sets from subtype constraints containing $\overline{a}$. Given types $T$ and $S$ where only one of them contains $\overline{a}$, the constraint generation algorithm algorithm computes the minimal $\overline{a}$-constraint set $C$ which guarantees $S <: T$. The judgement

$$\vdash_{\overline{a}} S <: T \Rightarrow C$$

which describes this is directly taken from [PT98]. We have for example:

$$\vdash_{a,b} \mathsf{Int} \to \mathsf{Int} <: a \to b \Rightarrow \{a <: \mathsf{Int}, \mathsf{Int} <: b\}$$

The soundness and completeness properties shown by Pierce and Turner [PT98] carry over to our system with records.

**Theorem 7.1 (Soundness).** Suppose that either $\mathrm{tv}(S) \cap \overline{a} = \emptyset$ or $\mathrm{tv}(T) \cap \overline{a} = \emptyset$. If $\vdash_{\overline{a}} S <: T \Rightarrow C$ and $\sigma$ is a solution of $C$, then $\sigma S <: \sigma T$.

**Theorem 7.2 (Completeness).** Let $\sigma$ be an $\overline{a}$-substitution and let $S$ and $T$ be types such that either $\mathrm{tv}(S) \cap \overline{a} = \emptyset$ or $\mathrm{tv}(T) \cap \overline{a} = \emptyset$. If $\sigma S \leq \sigma T$ then $\vdash_{\overline{a}} S <: T \Rightarrow C$ for some $C$.

Now, given a constraint set $C$ and a type $R$, $\sigma_{C,R}$ is a solution of $C$, which makes the type $\sigma_{C,R}R$ minimal, i.e. for each solution $\sigma$ of $C$: $\sigma_{C,R}R <: \sigma R$. $\sigma_{C,R}$ is undefined, if such a solution does not exist.

For the example above we get $\sigma_{\{a<:\mathsf{Int},\mathsf{Int}<:b\}}\{x : a, y : b\} = \{x : \bot, y : \mathsf{Int}\}$. The algorithm of Pierce and Turner [PT98] to compute $\sigma_{C,R}$ also works for our system with records.

## 8 Type Inference

Type inference is organized as a recursive algorithm, which does a depth-first traversal of the syntax-tree. The parameters of the inference algorithm are the term $E$ that we want to type, a type environment $\Gamma$, and a prototype $P$, which contains partial information on the type of $E$. The algorithm fills in the information that $P$ was lacking and returns the complete type $T$. The type inference algorithm is given as a deduction system for judgements of the form $P, \Gamma \vdash^w E : T$ (see Figure 5).

*Prototypes* $P$ are regular types except that they may contain an additional type constant "?" which indicates that information about a part of the type is lacking. We say a type $T$ matches a prototype $P$ if $T$ is obtained from $P$ by replacing "?"'s with arbitrary types.

For example in the judgement

$$\mathsf{Int} \to ?, \epsilon \vdash^w \mathbf{fun}(x)x : \mathsf{Int} \to \mathsf{Int}$$

the prototype $\mathsf{Int} \to ?$ indicates that only the argument type Int of the function is known from outside. But from this argument type we conclude that $x$ is of type Int and fill it in as a result type.

For each inference judgement we can find a corresponding judgement in the colored system by combining the information of $P$ and $T$ into a single colored type. For instance, the above judgement corresponds to

$$\epsilon \vdash^c \mathbf{fun}(x)x : {}^{\vee}(\mathsf{Int} \to {}_{\wedge}\mathsf{Int})$$

in the colored system. The parts present in the prototype are now inherited, the rest is synthesized.

To reconstruct the colored type from the prototype $P$ and the type $T$ in general, we use the operation $T \nearrow P$. If $T$ matches $P$, then $T \nearrow P$ is structurally equal to $T$. It is inherited on the parts given in $P$, and it is synthesized, where $P$ has a ?. If $T$ does not match the prototype $P$, we use the smallest supertype of $T$ which does. If such a type does not exist, $T \nearrow P$ is undefined. Dually, $T \searrow P$ is the greatest subtype of $T$ which matches $P$, is inherited on the parts given in $P$, and is synthesized elsewhere.

For instance

$$\mathsf{Int} \to \mathsf{Int} \nearrow \mathsf{Int} \to ? = {}^{\vee}(\mathsf{Int} \to {}_{\wedge}\mathsf{Int})$$

and

$$\mathsf{Int} \to \bot \nearrow ? \to ? \to ? = {}^{\vee}({}_{\wedge}\mathsf{Int} \to {}_{\wedge}\top \to {}_{\wedge}\bot).$$

It is crucial for the inference to keep the rule system deterministic. Therefore there must not be a subsumption rule in the inference system. Since for the inference the prototype is given and since there can be at most one supertype of $T$ which matches a given prototype $P$, namely $T \nearrow P$, we can always choose this one. Consequently, every derivable inference judgement $P, \Gamma \vdash^w E : T$ satisfies the invariant that $T$ matches $P$.

Most of the rules in Figure 5 are now straightforward to derive. Where above we combined the prototype $P$ and the type $T$ into a single colored type, we now have to do it the other way and split a colored type into a prototype and a regular type. For instance, in the rule (sel) we split the colored type $T$ from the (SEL) rule into $P$ and $T'$. Then ${}^{\vee}\{x : \diamond T\}$ splits into $\{x : P\}$ and $\{x : T'\}$.

$$\text{(var)}\quad P, \Gamma \vdash^w x : \Gamma(x) \nearrow P$$

$$(\mathrm{abs}_{tp,?})\ \frac{?, \Gamma, \overline{a}, x : T \vdash^w E : S}{?, \Gamma \vdash^w \mathbf{fun}[\overline{a}](x : T)E : T \xrightarrow{\overline{a}} S} \qquad (\mathrm{abs}_{tp,\top})\ \frac{\top, \Gamma, \overline{a}, x : T \vdash^w E : S}{\top, \Gamma \vdash^w \mathbf{fun}[\overline{a}](x : T)E : \top}$$

$$(\mathrm{abs}_{tp})\ \frac{P', \Gamma, \overline{a}, x : T \vdash^w E : S}{P \xrightarrow{\overline{a}} P', \Gamma \vdash^w \mathbf{fun}[\overline{a}](x : T)E : T \xrightarrow{\overline{a}} S \nearrow P \xrightarrow{\overline{a}} P'} \qquad (\mathrm{abs})\ \frac{P, \Gamma, \overline{a}, x : T \vdash^w E : S}{T \xrightarrow{\overline{a}} P, \Gamma \vdash^w \mathbf{fun}(x)E : T \xrightarrow{\overline{a}} S}$$

$$(\mathrm{app}_{tp})\ \frac{?, \Gamma \vdash^w F : S \xrightarrow{\overline{a}} T \qquad [\overline{R}/\overline{a}]S, \Gamma \vdash^w E : [\overline{R}/\overline{a}]S}{P, \Gamma \vdash^w F[\overline{R}](E) : [\overline{R}/\overline{a}]T \nearrow P} \qquad (\mathrm{app}_{tp,\bot})\ \frac{?, \Gamma \vdash^w F : \bot \qquad \top, \Gamma \vdash^w E : S}{P, \Gamma \vdash^w F[\overline{R}](E) : \bot \nearrow P}$$

$$(\mathrm{app})\ \frac{\begin{array}{cc} ?, \Gamma \vdash^w F : S \xrightarrow{\overline{a}} T & [?/\overline{a}]S, \Gamma \vdash^w E : S' \\ \vdash_{\overline{a}} S' <: S \Rightarrow C_1 & \vdash_{\overline{a}} T <: \top \searrow P \Rightarrow C_2 \end{array}}{P, \Gamma \vdash^w F(E) : \sigma_{C_1 \cup C_2, T} T \nearrow P} \qquad (\mathrm{app}_{\bot})\ \frac{?, \Gamma \vdash^w F : \bot \qquad \top, \Gamma \vdash^w E : S}{P, \Gamma \vdash^w F(E) : \bot \nearrow P}$$

$$(\mathrm{sel})\ \frac{\{x : P\}, \Gamma \vdash^w F : \{x : T\}}{P, \Gamma \vdash^w F.x : T} \qquad (\mathrm{rec}_?)\ \frac{?, \Gamma \vdash^w F_1 : T_1 \quad \ldots \quad ?, \Gamma \vdash^w F_n : T_n}{?, \Gamma \vdash^w \{x_1 = F_1, \ldots, x_n = F_n\} : \{x_1 : T_1, \ldots, x_n : T_n\}}$$

$$(\mathrm{rec}_\top)\ \frac{\top, \Gamma \vdash^w F_1 : T_1 \quad \ldots \quad \top, \Gamma \vdash^w F_n : T_n}{\top, \Gamma \vdash^w \{x_1 = F_1, \ldots, x_n = F_n\} : \top}$$

$$(\mathrm{rec})\ \frac{(P_1, \Gamma \vdash^w F_1 : T_1) \ \ldots \ (P_m, \Gamma \vdash^w F_m : T_m) \ (\top, \Gamma \vdash^w F_{m+1} : T_{m+1}) \ \ldots \ (\top, \Gamma \vdash^w F_n : T_n)}{\{x_1 : P_1, \ldots, x_m : P_m\}, \Gamma \vdash^w \{x_1 = F_1, \ldots, x_n = F_n\} : \{x_1 : T_1, \ldots, x_m : T_m\}}$$

Figure 5: $P, \Gamma \vdash^w E : T$

The resulting rule (sel) illustrates the information flow used in the type checking of (SEL). From outside we get information about the type in $P$. From this we conclude that the type of the record must match $\{x : P\}$. The type inference for the record will tell us that its final type is $\{x : T\}$. From that we know that the final type for the selection expression is $T$.

For the record introduction we need three separate rules, (rec), ($\mathrm{rec}_?$), and ($\mathrm{rec}_\top$), because we have to consider three different kinds of prototypes, $\{x_1 : P_1, \ldots, x_m : P_m\}$, ?, and $\top$. The reason is that in rule (REC) the types $T_i$ may be inherited or may have inherited components, although the record type constructor is synthesized. Similarly, we have to construct three rules ($\mathrm{abs}_{tp}$), ($\mathrm{abs}_{tp,?}$) and ($\mathrm{abs}_{tp,\top}$) for ($\mathrm{ABS}_{tp}$).

The most important case is again the untyped application rule (app), where we have to infer type parameters. Here, the descriptive premises in the type system are replaced by local constraint resolution. First of all, passing $[?/\overline{a}]S$ as a prototype for the actual argument guarantees $S' \triangleleft_{\overline{a}}{}^{\vee} S$. Then we generate two constraint sets $C_1$ and $C_2$ from the two subtype requirements. $C_1$ guarantees that a solution $\sigma$ fulfills $S' <: \sigma S$, which ensures that the type of the actual argument is a subtype of the function's argument type. $C_2$ ensures $\sigma T <: \top \searrow P$, so that a solution will always have a supertype matching $P$. Each solution of $C_1 \cup C_2$ satisfies both constraints. Choosing $\sigma_{C_1 \cup C_2, T}$ guarantees that we have a solution also satisfying the optimality constraint.

In the map example of Figure 2 x.match has type

$$\{caseNone : \{\} \to r, caseSome : c \to r\} \xrightarrow{r} r.$$

The type inference checks the actual visitor argument of x.match with prototype

$$\{caseNone : \{\} \to ?, caseSome : c \to ?\}.$$

This yields the final type

$$\{caseNone : \{\} \to Option[\bot], caseSome : c \to Option[d]\}.$$

Thus, the resulting constraint system is

$$\{Option[\bot] <: r, Option[d] <: r\}.$$

The second constraint implies the first, so the optimal solution for result type $r$ is

$$\sigma_{C,r} = [Option[d]/r].$$

Therefore, $Option[d]$ is the complete type for the visitor application.

We have shown soundness and completeness of the type inference with respect to the colored type system.

**Theorem 8.1 (Soundness).** If $P, \Gamma \vdash^w E : T$ then $\Gamma \vdash^c E : T \nearrow P$.

**Theorem 8.2 (Completeness).** If $\Gamma \vdash^c E : T$ and $T \leq T \nearrow P$ then $P, \Gamma \vdash^w E : T \nearrow P$.

The condition $T \leq T \nearrow P$ in the completeness theorem requires that the prototype $P$ contains at least the information which is present in the inherited part of $T$. For the special case that $T$ is purely inherited or purely synthesized, completeness simplifies to the following corollary.

**Corollary 8.3 (Completeness).** If $\Gamma \vdash^c E : {}_\wedge T$ then $?, \Gamma \vdash^w E : T$. If $\Gamma \vdash^c E : {}^\vee T$ then $T, \Gamma \vdash^w E : T$.

The proofs for soundness and completeness proceed by induction on the derivation. In the proof for completeness we always regard the last non-subsumption step together with all following subsumption steps.

## 9 Conclusion

When designing the type system for the functional net language Funnel [Ode00], we were looking for a type system with deep subtyping and polymorphic records. Further, we wanted to have a source language that avoided unnecessary clutter due to type annotations.

First, we were looking into type systems with unification-based type inference. Since deep subtyping and polymorphic records together do not allow complete type inference, one has to find restrictions on these two properties. The problem with this approach is that every new language construct, or even just a slight change of an existing language construct is a possible threat to the decidability or tractability of the type inference. It is even often difficult to see whether a specific change leads to undecidability.

Using $F_{\leq}$ and a local type inference scheme proved to be more robust and flexible in this respect. Here, we always have the internal language as a starting point and fallback. On top of this we can introduce lightweight versions of our syntactic constructs that obviate the need for many type annotations.

Since in Funnel programs the visitor pattern is used pervasively, it is important to be able to express visitors with lightweight abstractions. The first main contribution of this paper is a local type inference algorithm that is able to propagate partial type information. This is essential for visitors, but it is also helpful in eliding type information for other language constructs.

Although our type inference algorithm can type the visitor pattern, the local constraint resolution algorithm is the same as the one by Pierce and Turner [PT98] and the complexity of the algorithm is similar. The added power of our inference is hence solely derived from a more refined propagation scheme.

The second main contribution of this paper is the presentation of the type system as a colored deduction system. Information flow during type inference is no longer explicit, it is encoded as the color of the types. This yields a very compact notation with little redundancy. Looking at the discussion at the end of chapter 6, we can see that we often have to change very little when regarding different versions of the type system.

## References


[ACF+] Andrew Appel, Luca Cardelli, Kathleen Fisher, Carl Gunter, Robert Harper, Xavier Leroy, Mark Lillibridge, David B. MacQueen, John Mitchell, Greg Morrisett, John H. Reppy, Jon G. Riecke, Zhong Shao, and Christopher A. Stone. Principles and preliminary design for ML 2000.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 31–41, New York, June 1993. ACM Press.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proc. OPPSLA'98*, October 1998.

[Car93] Luca Cardelli. An implementation of $F_{<:}$. Technical Report 97, DEC Systems Research Center, February 1993.

[CDG+92] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[CT98] Craig Chambers and Cecil Team. The Cecil language, specification and rationale, December 1998.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proc. MFPS '95, Eleventh Conference on the Mathematical Foundations of Programming Semantics*, March 1995.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, January 1996.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GR99] Jacques Garrigue and Didier Rémy. Semi-explicit first-class polymorphism for ML. *Information and Computation*, 155:134–171, 1999.

[Jon97] Mark P. Jones. First-class polymorphism with type inference. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 483–496, Paris, Jan 1997. ACM Press.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, February 1968.

[Lit98] Vassily Litvinov. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, languages and applications*, October 1998.

[Mey92] Bertrand Meyer. *Eiffel, The Language*. Object Oriented Series. Prentice Hall, Engelwood Cliffs, 1992.

[NC97] Johan Nordlander and Magnus Carlsson. Reactive objects in a functional language - an escape from the evil I. In *Proceedings of the Haskell Workshop*, June 1997.

[Nor98] Johan Nordlander. Pragmatic subtyping in polymorphic languages. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, September 1998.

[Ode00] Martin Odersky. Functional nets. In *European Symposium on Programming*, Lecture Notes in Computer Science. Springer Verlag, 2000. Invited Paper.

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 54–67, January 1996.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[OZZ00] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. `http://lampwww.epfl.ch/papers/clti-color.ps.gz`, 2000.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, July 1988.

[Pot96] François Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 122–133, January 1996.

[Pot98] François Pottier. A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 228–238, September 1998.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 252–265, January 1998.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989.

[Str91] Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991.

[TS96] Valery Trifonov and Scott Smith. Subtyping constraint types. In *International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. Springer, September 1996.

[Wad89] Philip Wadler. Theorems for free! In *Fourth Symposium on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, September 1989. London.

[Wel94] J.B. Wells. Typability and type checking in the second order $\lambda$-calculus are equivalent and undecidable. In *Proc. 9th IEEE Symposium on Logic in Computer Science*, pages 176–185, July 1994.

[WJ00] Keith Wansbrough and Simon Peyton Jones. Simple usage polymorphism. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, September 2000.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.