Colored Local Type Inference

Martin Odersky Christoph Zenger Matthias Zenger

École Polytechnique Fédéral 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 f is known to have type $(\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$, then $\mathsf{f}\ (\mathbf{fun}\ (\mathsf{x})\ \mathsf{x} + 1)$ would be well-typed, since the type of parameter x can be inferred to be 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 g is known to have type $\forall\mathsf{a}.(\mathsf{Int} \to \mathsf{a}) \to \mathsf{a}$, then

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

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 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 g above.

Idioms like the one of 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, 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.

We represent partial type information by coloring types. A *red* type T is a type which is completely known from the context of the term to be typed and which is propagated down the tree. A *blue* type T is a type which is propagated up the tree[1]. A type can have both red and blue parts. For instance, the list visitor argument for the match method in the code of append would be inferred to have type ListVisitor[c, List[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. In the rest of this paper we develop this idea in a type system for second order lambda calculus with records and subtyping. We first define a subtyping relation

[1] The present version of the paper needs to be read on a color screen or as a printout of a color printer. For the case where colors are not available, we have produced a (less legbible) second version of the paper, where red types are written on light gray background and blue types are written on dark gray background. That version is found at http://lampwww.epfl.ch/papers/clti.

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, a red record type $\{x : a, y : b\}$ cannot be a subtype of the smaller red record type $\{x : a\}$, 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

$$\{x : a, y : b\} \ \leq \ \{x : a\} \ .$$

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

ting 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. Sections 2 and 3 present the internal and external language of our calculus. Section 4 presents an example. Sections 5 and 6 discuss the subtype relation and the type system using colors. Sections 7 and 8 outline the local constraint resolution and the type inference algorithm. Section 9 concudes.

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 in the internal language are given by the grammar

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

A term is a variable x, a function abstraction $\mathbf{fun}[\overline{a}](x : T)$ 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 $a_1, \ldots, a_n$. The presence of records allows us to restrict our language to single-argument functions since polyadic functions can be straightforwardly encoded: $\mathbf{fun}[\overline{a}](x : T, y : T')E$ is encoded as $\mathbf{fun}[\overline{a}](r : \{x : T, y : T'\})[r.x/x, r.y/y]E$ 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 types. Primitive types like Int are treated as free type variables. We identify types that are equivalent up to α-renaming and also identify record types with the same fields, possibly occuring 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 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.

E, F	$=$	...	
	$\mid$	$\mathbf{fun}(x)E$	lightweight abstraction
	$\mid$	$F(E)$	lightweight application

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. The colors indicate which part of the type is known from the context (inherited) and which part is known from the term itself (synthesized). *Blue* types represent synthesized information. Their syntax is:

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

Note that a blue function type contains a red argument type. The complementation of the color reflects the fact that in an application where the type of the function is completely known, its argument type is propagated down the tree, to be used in the type checking of the argument. Types which are propagated down the tree are colored red.
The syntax of *red* types is the dual of the syntax of blue types.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Red and blue types are special cases of general types (written in black), which can have arbitrary red and blue parts. The syntax of general types is:

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

Type constructors are either red or blue. Writing a black type constructor means that we do not assume a particular color. If we write the types T, T, T within a single statement, they are assumed to be all structurally equivalent; differing only in color. If the same constructor occurs in black more than once, as in $\{x_1 : T_1, \ldots, x_n : T_n\} \leq \{x_1 : T_1', \ldots, x_n : T_n'\}$, the occurrences of the constructor are assumed to have the same color. *Substitutions* $[T/a]S$ on types are defined to be color preserving: $[T/a]a = T$ and $[T/a]a = T$. Types in type environments are always blue. This ensures that the type of a variable is always determined by its definition and not influenced by the context of its usage.

$$\Gamma \ = \ x : T \mid \epsilon \mid a \mid \Gamma, \Gamma'$$

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] are defined as follows:

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

Constructors for Option[a] are as follows:

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

Consider now a map function over optional values, which is written in the external language as follows.

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

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

- The formal parameter of function o.match has type $\mathit{OptionVisitor}[c, r]$, which expands to $\{\mathit{caseNone} : \{\} \rightarrow r, \mathit{caseSome} : c \rightarrow r\}$. That is, for the visitor record everything is given from outside,

```
1  map = fun[c,d](f: c → d) fun(o:Option[c])
2    ((o: {match : OptionVisitor[c, r] -r→ r}
3      .match): { caseNone : {} →r
                  caseSome : c →r  } -r→ r
4    {
5      caseNone = (fun()
6        (None: {} -t→ Option[t]())
7          : Option[⊥])
8        : {} → Option[⊥] ,
9      caseSome = (fun(x: c)
10       (Some: t -t→ Option[t](f: (c→d)(x:c)):d))
11         : Option[d])
12       : c → Option[d]
13   } : { caseNone : {} → Option[⊥]
           caseSome : c → Option[d]  }
14   ): Option[d]
15 : (c → d) -c,d→ (Option[c] → Option[d])
```

Figure 2: map

except for the second parameter type of OptionVisitor, which still needs to be instantiated.

- Therefore, function caseSome has type $c \to Option[d]$. The argument type c of the function is given from the outside. It appears blue instead of red because colors are complemented for argument types.
- Therefore, the type of the formal parameter x of caseSome can be reconstructed; no explicit type annotation needs to be given.

5 Subtyping

The subtype relation $\leq$ for colored types is shown in Figure 3. 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 blue to red. The blue type $\mathsf{Int} \to \mathsf{Int}$ for example is a subtype of $\mathsf{Int} \to \top$ and $\mathsf{Int} \to \mathsf{Int} \leq \bot \to \top \leq \top$. But $\mathsf{Int} \to \mathsf{Int}$ is not a subtype of $\top$. Here, the topmost constructors differ, but they have the same color.
This ensures that we never guess types. Inherited information is really coming from the outside, it cannot be discarded by subsumption: If it is red at a certain point in the term, going outward we can discard the red constructor only explicitly in a rule (the origin of the information), not by using subsumption. Similarly, synthesized information is really coming from inside, it cannot be constructed using subsumption. A rule relating e.g. $\bot \leq S \overset{a}{\to} T$ would destroy this property, because it would synthesize a function type which was guessed rather than propagated.
Subtyping's role is to mediate differences when type information about a term is propagated both from the inside and from the outside. A blue type constructor of a given type matches with a red constructor of that same type, or of a supertype.
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, $\bot \leq \mathsf{Int} \to \{x : \mathsf{Int}\}$ via $\bot \leq \top \to \bot$ and $\top \to \bot \leq \mathsf{Int} \to \{x : \mathsf{Int}\}$.
The subtype relations $<:$ for uncolored types of the internal language and $\leq$ for colored types have the following relationship:
Subtyping lemma.

1. $T \leq S$ implies $T <: S$.
2. $T <: S$ implies $T \leq S$,

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 T will typecheck against each supertype of $S :> T$ given completely from outside.
In the typing of map, we use this subtype relation as follows: In the selection o.match we know from outside that o has to have a type of the form $\{match : T \overset{r}{\to} T'\}$. If we look up o in the type environment, we find the blue type $\{match : OptionVisitor[a, r] \overset{r}{\to} r\}$. Subsumption gives us that o also has type $\{match : OptionVisitor[a, r] \overset{r}{\to} r\}$ which is of the required form.

6 Type System

Figure 4 presents the type system for the external language using colored types. Most rules are straightforward. Restricting the (VAR)-rule to blue types is not essential, since Γ only contains blue types.
The rule for function abstraction with an explicit argument type produces a type with a blue arrow as toplevel constructor whereas the rule for lightweight abstractions produces a red 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 black, which tells us that this type can be propagated in either direction.

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

$$a \le a \qquad \bot \le \bot \qquad \top \le \top \qquad \bot \le \top$$

$$\{x_1 : T_1, \ldots, x_n : T_n, x_{n+1} : \top, \ldots, x_m : \top\} \le \{x_1 : T_1, \ldots, x_n : T_n\} \qquad T \xrightarrow{\overline{a}} S \le T \xrightarrow{\overline{a}} S$$

$$\bot \le a \qquad \bot \le \{x_1 : \bot, \ldots, x_n : \bot\} \qquad \bot \le \top \xrightarrow{\overline{a}} \bot$$

$$a \le \top \qquad \{x_1 : \top, \ldots, x_n : \top\} \le \top \qquad \bot \xrightarrow{\overline{a}} \top \le \top$$

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

Figure 3: $T \le S$

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

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

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

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

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

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

In the untyped abstraction, the function's argument type T has to be known from the context (it occurs blue in a contravariant position). Because the formal type parameters are not explicitly mentioned in the source, we disallow their use in the function body in a side condition of rule (ABS). Pierce and Turner do not require this side condition since their system never elides formal type parameters.

Typed function application is also straightforward. The expression's function part needs to have a function type. Only the fact that a function type is required is propagated into the function expression; the rest has to be synthesized. Rule (APP_{tp}) is formulated in a way that makes a separate rule for the case where the function's type is $\bot$ superfluous: Assume that F has type $\bot$. By subsumption it also has type $\top \xrightarrow{\overline{a}} \bot$. Now if E is typable with $\top$ we can conclude with (APP_{tp}) that $F(E)$ has the expected type $\bot$.

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 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 (red) argument type, except for occurrences of type parameters a, where an arbitrary synthesized type is required. This is expressed by the condition $S' \lhd_{\overline{a}} S$ in the premise. The auxiliary relation $\lhd_{\overline{a}}$ is defined as follows, with its complement $\lhd_{\overline{a}}$ being exactly the dual of $\lhd_{\overline{a}}$ where every occurrence of a red type is replaced by the corresponding blue type and vice versa.

$$T \lhd_{\overline{a}} T \qquad\qquad T \lhd_{\overline{a}} a$$

$$\frac{T_1 \lhd_{\overline{a}} T_1' \qquad \dots \qquad T_n \lhd_{\overline{a}} T_n'}{\{x_1 : T_1, \dots, x_n : T_n\} \lhd_{\overline{a}} \{x_1 : T_1', \dots, x_n : T_n'\}}$$

$$\frac{T \lhd_{\overline{a}} T' \qquad S \lhd_{\overline{a}} S'}{T \xrightarrow{\overline{a}} S \lhd_{\overline{a}} T' \xrightarrow{\overline{a}} S'}$$

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 $\overline{a}$, yield a type which is a supertype of the actual argument type S'. Furthermore, when substituted for the function's result type 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 $S \equiv T$, which states that S and T coincide on their red parts. $\equiv$ is defined as follows, with $\equiv$ again being the dual of $\equiv$:

$$\frac{T \equiv S \qquad\qquad T \equiv T}{T_1 \equiv T_1' \qquad \dots \qquad T_n \equiv T_n'}{\{x_1 : T_1, \dots, x_n : T_n\} \equiv \{x_1 : T_1', \dots, x_n : T_n'\}}$$

$$\frac{T \equiv T' \qquad S \equiv S'}{T \xrightarrow{\overline{a}} S \equiv T' \xrightarrow{\overline{a}} S'}$$

As an example, consider the application $\mathsf{f}(\mathsf{x})$ where f has the blue type $a \xrightarrow{a} (a \to a)$ and x has type Int. Since $\mathsf{Int} \lhd_a a$, x is a matching argument. But we will see that $\mathsf{f}(\mathsf{x})$ fails to have a blue type. The high level reason for this is that a occurs co- and contravariantly in the result type and we thus cannot make an optimal choice for R. If we try to give $\mathsf{f}(\mathsf{x})$ the blue type $\mathsf{Int} \to \mathsf{Int}$, we have to choose $R = \mathsf{Int}$. The subtype requirements of the rule are fulfilled, but $R' = \top$ and $T'' = \top \to \top \equiv \mathsf{Int} \to \mathsf{Int}$ provide an alternative which also fulfills the requirements. It falsifies the optimality constraint $\top \to \top \leq \mathsf{Int} \to \mathsf{Int}$, and therefore does not yield a valid typing. Other choices for blue types fail for similar reasons.

On the other hand, $\mathsf{f}(\mathsf{x})$ has the red type $\mathsf{Int} \to \mathsf{Int}$. The choice $R = \mathsf{Int}$ is the same, but here the requirement $T'' \equiv \mathsf{Int} \to \mathsf{Int}$ on T'' is stronger, and the only choice left for R' is Int. $[\mathsf{Int}/a]a \to a \leq [\mathsf{Int}/a]a \to a$ holds and the optimality constraint is satisfied. By analogous reasoning, $\mathsf{f}(\mathsf{x})$ also has the red type $\bot \to \bot$.

The red record type constructor in (SEL) indicates that when determining the qualifier type we know that it has to be a record type with the given field. The field type can be propagated in either direction. Subsumption allows that the argument has additional fields. However, all these additional fields have to have blue types. Allowing red types would be allowing to guess them.

In the record constructor rule the record type constructors are blue, they represent information given explicitly in the term. The field types are black and can be propagated in either direction.

The type system is sound and complete with respect to the internal language.

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

Completeness. If $\Gamma \vdash E : T$ then $\Gamma \vdash^E E : T$

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.

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 we consider again $\mathsf{f}(\mathsf{x})$ with f of type $a \xrightarrow{a} (a \to a)$ and x of type Int. We saw that we can infer the types $\bot \to \bot$ and $\mathsf{Int} \to \mathsf{Int}$ but we will see now that we cannot infer $\bot \to \mathsf{Int}$. If we try to give f the type $\bot \to \mathsf{Int}$, 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),
- 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):

$$S' \leq [\overline{R}'/\overline{a}]S \wedge [\overline{R}'/\overline{a}]T \leq T'' \equiv T' \Rightarrow T' \leq T''.$$

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

$$S' \le [\overline{R}'/\overline{a}]S \wedge [\overline{R}'/\overline{a}]T \le T'' \equiv T' \Rightarrow$$
$$[\overline{R}/\overline{a}]T \le [\overline{R}'/\overline{a}]T \wedge [\overline{R}/\overline{a}]S \le [\overline{R}'/\overline{a}]S$$

This variation requires only minimal changes to the inference algorithm.

7 Constraint Resolution

Type inference needs 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* σ is an idempotent substitution with $\mathrm{dom}(\sigma) = \overline{a}$. An $\overline{a}$-substitution σ 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 $S <: T$. This is stated by the judgement

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

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.

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 σ is a solution of C, then $\sigma S <: \sigma T$.
Completeness. Let σ 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 $\sigma S \le \sigma T$ 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 σ 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 $\sigma_{C,R}$ works also for our system with records.

8 Inference

The type inference algorithm is organized as a recursive algorithm, which does a depth-first traversal of the syntax-tree. The arguments of the inference algorithm are the term E that we want to type, a type environment Γ and a prototype P, which contains partial information on the type of E. The algorithm fills in the information that P was missing and returns the complete type T.
Prototypes P are normal types except for the fact that they may contain an additional type constant ? which indicates that information about this part of the type is missing. We say a type T matches a prototype P if T is obtained from P by replacing ?'s with arbitrary types. Let $(T \nearrow P)$ be the smallest supertype of T which matches P (if such a type exists). Dually, let $(T \searrow P)$ be the greatest subtype of T which matches P. We can regard $(T \nearrow P)$ as a colored type, where the parts which correspond to a ? in P are synthesized, the rest is inherited. For instance

$$((\mathsf{Int} \to \mathsf{Int}) \nearrow (\mathsf{Int} \to ?)) = \mathsf{Int} \to \mathsf{Int}$$

and

$$(\mathsf{Int} \to \bot \nearrow ? \to ? \to ?) = \mathsf{Int} \to \top \to \bot.$$

The type inference algorithm is given as a deduction system for judgements of the form $P, \Gamma \vdash^W E : T$. In

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

the prototype $\mathsf{Int} \to ?$ indicates that only the argument type Int of the function is known from outside. We conclude that x is of type Int and fill it in as a result type. The colors of Int→Int show that the argument type was inherited and the result type was synthesized. Figure 5 shows the inference algorithm. It is presented using colored types, which helps relating it to the type

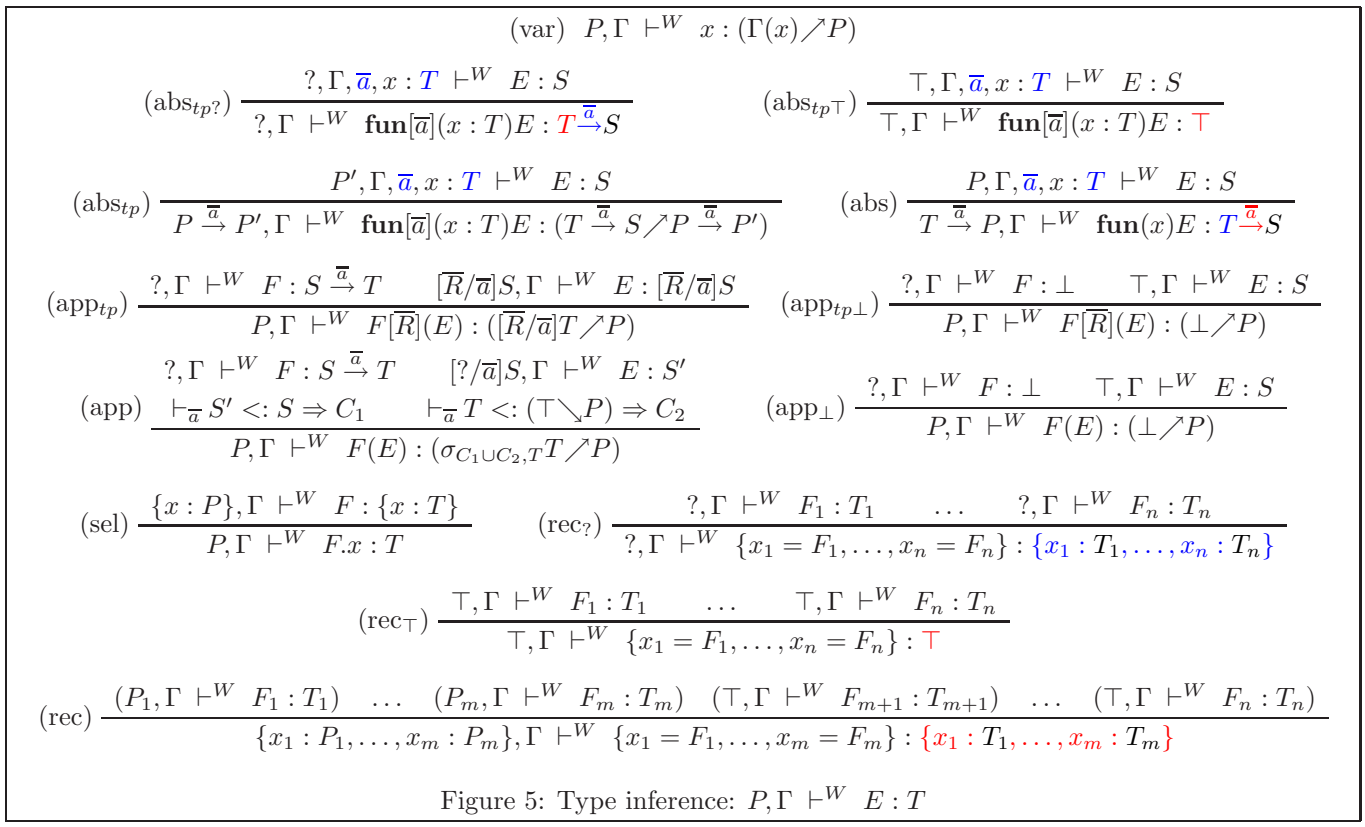

Figure 5: Type inference: $P,\Gamma \vdash^W E : T$

system. In practice the algorithm does not need to compute colors for types, because the colors of intermediate results are only relevant for determining further colors, never for determining the structure of a result type. Moreover the colors of the final result can easily be reconstructed by filling the prototype from the complete type $(T \nearrow P)$.

Most of the algorithmic rules in Figure 5 are straightforward to derive. They are designed such that the completed type T always matches the prototype P.

For example, the typing rule (SEL) tells us that the qualifier of a selection should have a record type with a red constructor. Hence, in the type inference rule (sel) the prototype for the qualifier has a record type containing this field. Further, one can see how the information on the type of the field flows in both directions. First, we pass the partial information in form of the prototype P into the subterm and then we pass the complete information back up in T.

For the record introduction rule (REC), we need to have three separate inference rules. One rule for the case where the prototype is already a record, another rule where the prototype is a simple ? and yet another one for $\top$. The same holds for (ABS_{tp}).

Typed and untyped application both need to consider $\bot$ seperately here. The difficult 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 $S' \lhd_{\overline{a}} S$. then we generate two constraint sets C_1 and C_2 from the two subtype requirements. C_1 guarantees that a solution σ 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 o.match has type $\{caseNone : \{\} \to r, caseSome : c \to r\} \to r$. The type inference checks the actual visitor argument of o.match with prototype $\{caseNone : \{\} \to ?, caseSome : c \to ?\}$. This yields the complete 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.

Soundness. If $P,\Gamma \vdash^W E : T$ then $\Gamma \vdash^E E : T$.

Completeness. If $\Gamma \vdash^E E : T$ 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. Both proofs 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 allows to often leave out 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 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 make annoying type annotations unnecessary.
Since in Funnel the visitors are ubiquitous, it is important that to be able to use lightweight abstractions for them. 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. Colors are not only a good notation for the type system, it turned out that red, blue, and black types are also convenient for spoken language.
As a next step, we will look at the integration of recursive data types and bounded polymorphism. For the language Funnel we have already implemented a prototype for recursive data types, but the details are subject of further research.

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.

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

[Pot98] Francois 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 λ-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. Submitted to The Third ACM SIGPLAN Workshop on Types in Compilation, 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.