# First Class Overloading via Insersection Type Parameters*

Elton Cardoso[2], Carlos Camarão[1], and Lucilia Figueiredo[2]

[1] Universidade Federal de Minas Gerais,
camarao@dcc.ufmg.br
[2] Universidade Federal de Ouro Preto
eltonm.cardoso@gmail.com, luciliacf@gmail.com

**Abstract** The Hindley-Milner type system imposes the restriction that function parameters must have monomorphic types. Lifting this restriction and providing system F "first class" polymorphism is clearly desirable, but comes with the difficulty that inference of types for system F is undecidable. More practical type systems incorporating types of higher-rank have recently been proposed, that rely on system F but require type annotations for the definition of functions with polymorphic type parameters. However, these type annotations inevitably disallow some possible uses of higher-rank functions. To avoid this problem and to promote code reuse, we explore using intersection types for specifying the types of function parameters that are used polymorphically inside the function body, allowing a flexible use of such functions, on applications to both polymorphic or overloaded arguments.

## 1 Introduction

The Hindley-Milner type system [9] (`HM`) has been successfuly used as the basis for type systems of modern functional programming languages, such as Haskell [23] and ML [20]. This is due to its remarkable properties that a compiler can infer the *principal type* for any language expression, without any help from the programmer, and the type inference algorithm [5] is relatively simple. This is achieved, however, by imposing some restrictions, a major one being that function parameters must have monomorphic types.

For example, the following definition is not allowed in the `HM` type system:

```
foo g = (g [True,False], g ['a','b','c'])
```

Since parameter `g` is used with distinct types in the function's body (being applied to both a list of booleans and a list of characters), its type cannot be monomorphic, and this definition of `foo` cannot thus be typed in `HM`.

In contrast, higher-ranked polymorphic type systems such as Girard-Reynolds system `F` [7,25] allow universal quantifiers to appear *within* a type. In a language based on system `F`, `foo` could be assigned, for example, type

* This work is partially sponsored by FAPEMIG.

$$(\forall a.\texttt{[}a\texttt{]} \rightarrow \texttt{[}a\texttt{]}) \rightarrow \texttt{([Bool],[Char])}$$

Function `foo` could then be used, for example, in application (`foo reverse`), where `reverse` computes the reversal of a list, having type $\forall a.\texttt{[}a\texttt{]} \rightarrow \texttt{[}a\texttt{]}$.

The above type for `foo` is a rank-2 type, as it contains quantifiers to the left of the function arrow. Other higher-ranked types could also be assigned to `foo`. For example, `foo` could be assigned type $(\forall a.\texttt{[}a\texttt{]} \rightarrow \texttt{Int}) \rightarrow \texttt{(Int,Int)}$ in an application such as (`foo length`), where `length` has type $\forall a.\texttt{[}a\texttt{]} \rightarrow \texttt{Int}$.

These two types are however incomparable in system `F`. In other words, system `F` lacks *principal types* for expressions: a single expression may be typeable with two or more incomparable types, where neither is more general than the other. As a consequence, type inference cannot always choose a single type and use it throughout the scope of a let-bound definition. In particular, there exists no principal type for the above definiton of `foo`, such that all others follows from it by a sequence of instantiations and generalizations.

Another drawback for the use of system `F` as the basis for a programming language is that complete type inference in this system is undecidable [31].[3]

In order to cope with these problems, more practical type systems have been recently proposed as a basis for the support of higher-ranked polymorphism in programming languages. The main idea is to require the programmer to supply a *type annotation* for the definition of a function with polymorphic type parameters, thus avoiding possible ambiguities on its type and also providing type information that can be used for guiding type inference. Some relevant works along this line are `MLF` [15,16], `FPH` [30] and Flexible Types [18,17]. These type systems differ on the changes that are introduced to the Hindley-Milner type system, particularly on the choice of type annotations required and on the strategy used for type inference.

Lack of principal types and the need for type annotations are not the only issues related to avoiding the restriction of monomorphism of function parameters. The annotation of a higher-ranked polymorphic type for a function (or the annotation of a quantified type for a function parameter) inevitably restricts the contexts where the function may be used. For example, if the type annotated for `foo` is $(\forall a.\texttt{[}a\texttt{]} \rightarrow \texttt{[}a\texttt{]}) \rightarrow \texttt{([Bool],[Char])}$, none of the following applications can be type-checked:

| | | | |
|---|---|---|---|
| `foo length` | where `length` | has type | $\forall a.\texttt{[}a\texttt{]} \rightarrow \texttt{Int}$ |
| `foo head` | where `head` | has type | $\forall a.\texttt{[}a\texttt{]} \rightarrow a$ |
| `foo listMaybe` | where `listMaybe` | has type | $\forall a.\texttt{[}a\texttt{]} \rightarrow \texttt{Maybe}\ a$ |

Modern functional programming languages already include other useful extensions to `HM` and it is desired that higher-ranked functions work well in conjunction with these extensions. In particular, it is desirable that higher-ranked

[3] Kfoury and Tiuryn have indeed also proved undecidability of typeability for any subset of system F with rank $\geq 3$ [13].

functions work well in conjunction with the mechanism of *type classes* [10,8], used in Haskell to support overloading.

Consider, for example, the use of `foo` in the following Haskell examples:

| | | |
|---|---|---|
| `foo allEqual` | where | `allEqual :: ` $\forall a.$ `Eq` $a \Rightarrow$ `[`$a$`] ->` `Bool` |
| `foo sort` | where | `sort :: ` $\forall a.$ `Ord` $a \Rightarrow$ `[`$a$`] ->` `[`$a$`]` |
| `foo nub` | where | `nub :: ` $\forall a.$ `Eq` $a \Rightarrow$ `[`$a$`] ->` `[`$a$`]` |
| `foo fromEnum` | where | `fromEnum :: ` $\forall a.$ `Enum` $a \Rightarrow a \rightarrow$ `Int` |

The type of each of the above arguments is a *constrained polymorphic type* (also called a predicated type). In Haskell, a type class denotes a family of types (instances) on which certain values (the member functions) are defined. For example, the equality operator `(==)` has type $\forall a.$ `Eq` $a \Rightarrow a \rightarrow a \rightarrow$ `Bool`, where the *class constraint* `Eq` $a$ indicates that equality is not parametrically polymorphic, but only works for those types that are an instance of the `Eq` class. In other words, class constraints restrict the possible types to which quantified type variables can be instantiated. Class constraints introduced on the types of overloaded symbols are also propagated to the types of expressions defined in terms of these symbols. For example, the constraint on the type of (`==`) is propagated to the type of function `allEqual`, which checks that all elements of a given list are equal, and also to the type of function `nub`, that removes duplicate elements in a given list.[4]

Type system `QMLF` [19] is, to our knowledge, the only work that investigates how higher-ranked polymorphism can work in conjunction with constrained polymorphism. It extends the `MLF` type system, allowing predicate constraints (such as class constraints) to be specified on higher-ranked types. For example, in this system, `foo` could be annotated with type

$$(\forall a.\texttt{Ord}\ a \Rightarrow \texttt{[}a\texttt{]} \rightarrow \texttt{[}a\texttt{]}) \rightarrow \texttt{([Bool],[Char])}$$

Then, both applications (`foo sort`) and (`foo reverse`) are allowed in `QMLF`, where the type of `reverse` can be instantiated to type $\forall a.$`Ord` $a \Rightarrow$ `[`$a$`] ->` `[`$a$`]`.

But, again, this type annotation for `foo` forbids, for example, all other applications in the list above. In particular, application `foo nub` would not type check, despite the fact that the type of `nub` differs from the type of `foo`'s parameter only on its class constraint. Therefore, allowing type annotation of higher-ranked types with predicate constraints does not solve all problems with respect to a flexible use of higher-ranked functions.

In order to solve these problems and promote code reuse, we adopt a different approach, allowing programmers to annotate *intersection types* for the types of function parameters that are used with different types in the function's body. Intersection types [3] provide a conceptually simple and tractable alternative to the impredicative polymorphism of System `F`, while typing many more programs than the Hindley-Milner type system. The novelty of our work is the combination

[4] A more detailed description of Haskell's type classes may be found, for example, in [23,1,28].

of intersection types with usual Hindley-Milner types *and* constrained polymorphism, in order to allow the definition of higher-ranked functions, with arguments of intersection types, and application of such functions to both polymorphic or overloaded arguments.

Several interesting applications which require higher-ranked polymorphism have been described, for example, in [26,22,29], including the definition of monadic abstractions, data type invariants, dynamic types and generic (or polytypic) functions. In some of these examples, a higher-ranked type annotation serves exactly the purpose of restricting the set of possible arguments for such a function, thus guaranteeing that it has some desired properties (see, for example, chapter 3 on [29]). It should thus be made clear that we do not argue against the usefulness of this kind of polymorphism in a programming language, regarding our work as complementary.

Our type system is named `CTi`, as it combines **c**onstrained polymorphic **t**ypes and **i**ntersection types. The intuitive ideas behind this system are explained in Section 2. Section 3 briefly reviews the two main topics we want to combine — constrained polymorphism and intersection types — and presents a formal definition for the type language of our system. Type system `CTi` is defined in Section 4 and Section 5 presents a corresponding type inference algorithm. Finally, our conclusions are presented in Section 6.

## 2 Intersection parameters and polymorphic arguments

Let us consider what should be the principal (minimal) type of function `foo`, which would allow any of the previously discussed arguments to be applied to this function. Intuitively, an application (`foo f`) should be valid if `f` is a function that can be applied to lists of booleans *and* to lists of chars. Also, (`foo f`) should have type $(\tau_1, \tau_2)$, where $\tau_1$ is the result type of an application of `f` to a list of booleans and $\tau_2$ is the result type of an application of `f` to a list of chars. Using intersection types, this can be written as (for ease of reference, the definition of `foo` is repeated below):

*Example 1.*
```
foo :: ∀b,c.([Bool] → b ∧ [Char] → c) → (b,c)
foo g = (g [True,False], g ['a','b','c'])
```

The use of intersection types seems to us to be a natural choice for expressing the type of a function parameter that is used with different types in the function's body, since only finite uses of this parameter can occur.

Differently from usual intersection type systems [3,14,4], our type system is intended as a conservative extension of (`HM` + constrained polymorphism), maintaining complete compatibility with this system in the absence of type annotations. Type annotations are required for the definition of higher-ranked functions, where the types of parameters that are used polymorphically inside the function's body are specified in the form of intersection types.

An intersection type $(\tau_1 \wedge \tau_2)$ may occur in the type of an expression only to the left of the function type constructor, as, for example, in the above type annotation for `foo`. A type $(\tau_1 \wedge \tau_2)$ may also occur in typing context assignments, being introduced in this context by means of a type annotation.

Since intersection types may only occur to the left of function arrows, intersection type *elimination* is restricted, in our system, to the rule for type derivation of a term variable. Dually, intersection type *introduction* may occur only in the type derivation for the argument of an application: assuming a term $t$ can be assigned type $(\tau_1 \wedge \tau_2) \rightarrow \tau$ in a given typing context, an application $(t\ u)$ will be well typed in this context, if $u$ can be assigned, in this context, a type $\sigma$ that can be instantiated both to $\tau_1$ *and* to $\tau_2$.

For example, application (`foo reverse`) is well typed according to this rule, since the type annotated for `foo` can be instantiated to (`[Bool]` $\rightarrow$ `[Bool]` $\wedge$ `[Char]` $\rightarrow$ `[Char]`) $\rightarrow$ `([Bool],[Char])`, and the type of `reverse` can be instantiated to both `[Bool]` $\rightarrow$ `[Bool]` and `[Char]` $\rightarrow$ `[Char]`.

Analogously, application (`foo sort`) is well typed, in a context where `Bool` and `Char` are both instances of type class `Ord` , since the type of `sort` can be instantiated to both `[Bool]` $\rightarrow$ `[Bool]` and `[Char]` $\rightarrow$ `[Char]`, in this context. Each of the applications of `foo` discussed previously would also be valid, in a context where the constraints on the type of the arguments could be satisfied.

The type for each use of an intersection type parameter in the body of the function is derived by means of intersection type elimination. The appropriate type is inferred according to the type required in each context where the parameter is used, in the same way as it happens for uses of overloaded symbols in a type system for *context-dependent overloading*, such as Haskell's type class system. For example, the type for each use of parameter `g` in the body of `foo` is inferred according to the type of the argument to which `g` is applied. Dually, the type for each use of parameter $x$ in the body of function `f1`, defined below, is infered according to the types of the parameters of the function to which $x$ is given as argument.

*Example 2.*
```
f1 :: (Bool ∧ Int) → Int
f1 x = if x then (x+1) else (x-1)
```

Function `f1` could be applied, for example, to an overloaded symbol, say `o :: C` $a \Rightarrow a$, declared in type class `C`, provided that there exist instance definitions of this class for types `Bool` and `Int`.

Consider now the definition of function `f2` below:

*Example 3.*
```
f2 :: (Int → Int ∧ Bool → Bool) → (Int ∧ Bool) → (Int, Bool)
f2 h y = (h y, h y)
```

The types of parameters `h` and `y` on each application `h y`, in the body of function `f2`, can be determined by the type of the result of this function. Therefore, type inference for function `f2` is defined so as to use the additional information provided by type annotations for selecting the appropriate type for `h` and `y` on each application `h y`. On the other hand, if type annotations were provided only for the parameters of `f2`, then four incomparable types could be derived for `f2`, namely, `(Int,Bool)`, `(Int,Int)`, `(Bool,Int)` and `(Bool,Bool)`; it would not be possible then to determine the types of `h` and `y` on each application. No type could then be infered for `f2`.

As in the case for overloaded symbols, an expression involving a function parameter annotated with an to an intersection type may sometimes be *ambiguous*. As an example, consider application `s z` in the body of function `f3` defined below:

*Example 4.*
```
f3 ::(Int → Int ∧ Bool → Int) → (Int ∧ Bool) → Int
f3 s z = s z
```

In this case, there are two distinct possible type derivations for the body of `f3`, corresponding to the two different possible choices for the types of parameters `s` and `z` in application `s z`.

## 3 Constrained Polymorphism and Intersection Types

Haskell's type class system is based on the more general *theory of qualified types* [10], which extends Hindley-Milner type expressions with type constraints (or predicates). A constrained polymorphic type has the form $\forall\,\overline{a}.\,P \Rightarrow \tau$, where $P$ is a (possibly empty) set of class constraints and $\tau$ is a monomorphic type. Here, and elsewhere, we use the notation $\overline{a}$ for a sequence of type variables $a_1, \ldots, a_n$, for some $n \geq 0$.

Type $\forall\,\overline{a}.\,P \Rightarrow \tau$ denotes the set of instances of $\tau$ that satisfy the predicates in $P$. For example, type `Int → Int → Bool` is an instance of type $\forall a.\,$`Eq` $a \Rightarrow a \rightarrow a \rightarrow$ `Bool`, if constraint `Eq Int` can be satisfied according to the visible class and instance declarations, that is, if there exists an instance definition of type `Int` for class `Eq`. In other words, each class constraint $C\,\tau$ is an assertion that type $\tau$ must be an instance of class $C$.[5]

Satisfiability of class constraints can be described using an entailment relation, denoted by the symbol $\models$. If $P$ and $Q$ are finite sets of type predicates, then the assertion that $P \models Q$ means that predicates in $Q$ are satisfied whenever predicates in $P$ are satisfied.[6]

[5] For simplicity, we only consider here single parameter type classes, since the extension to multiparameter type classes [12,11] is orthogonal to the problem considered in this paper.

[6] See [10] for the general assumptions made about the predicate entailment relation $P \models Q$, and for a definition of the entailment relation induced by Haskell class and instance declarations.

The type language of our system extends the language of constrained polymorphic types, introducing intersection types. The syntax of types and expressions of system `CTi` is given in Figure 1. Types are divided into *monointersection types* ($\tau$), that are used as types of expressions, and *intersection types* ($\iota$), which can be introduced only by a type annotation for a function parameter, and appear in the type of an expression only to the left of a function arrow.

| | | | |
|---|---|---|---|
| Intersection types | $\iota$ | $::= \tau \mid \iota \wedge \tau$ | |
| Monointersection types | $\tau$ | $::= a \mid \iota \to \tau$ | |
| Polymorphic types | $\sigma$ | $::= \forall \overline{a}.\, P \Rightarrow \tau$ | |
| Type constraint | $P, Q$ | $::= \epsilon \mid P,\, \mathtt{C}\,\tau$ | |
| Typing context | $\Gamma$ | $::= \epsilon \mid \Gamma, (x : \sigma) \mid \Gamma, (x : \iota)$ | |
| Terms $t, u$ | $::= x$ | | Variable |
| | $\mid \lambda x.\, t$ | | Functional abstraction |
| | $\mid \lambda (x :: \rho).\, t$ | | Annotated functional abstraction |
| | $\mid t\ u$ | | Function application |
| | $\mid$ `let` $x = u$ `in` $t$ | | Let-binding |
| | $\mid t :: \sigma$ | | Type annotated expression ($\sigma$ closed) |

**Figure 1.** Syntax of types and terms

The intersection type constructor ($\wedge$) is considered, as usual, to be commutative and associative. We write $\tau' \in (\tau_1 \wedge \ldots \wedge \tau_n)$ if $\tau' = \tau_i$, for some $1 \leq i \leq n$.

A typing context $\Gamma$ conveys the typings of in-scope variables; $\Gamma$ binds a term variable, $x$, to its type, either $\sigma$ or $\iota$.

We define $ftv(\sigma)$ to be the set of free type variables of $\sigma$, and extend this function to typing contexts in the obvious way (note that an intersection type $\iota$ does not have bound type variables):

$$ftv(\Gamma) = \bigcup \{ftv(\sigma) \mid (x : \sigma) \in \Gamma\} \ \cup \ \bigcup \{ftv(\iota) \mid (x : \iota) \in \Gamma\}$$

The language of terms of type system `CTi` is the usual language of `HM` augmented with type annotations on both terms ($t :: \sigma$) and lambda-bound variables ($\lambda (x :: \iota).\, t$). We assume that type annotations $\sigma$ are *closed*, that is, they have no free type variables.[7]

[7] Open type annotations, which require lexically-scoped type variables [27], are avoided, because this introduces complications that are out of the scope of this work.

## 4 Typing Rules

The rules of type system `CTi` are defined in Figure 2. A typing judgement has the form

$$P;\ \Gamma \vdash t : \tau$$

meaning that type $\tau$ may be assigned to term $t$, in typing context $\Gamma$, if all constraints in $P$ are satisfied, according to the constraint entailment relation, $P \models Q$, defined by the program class and instance declarations.

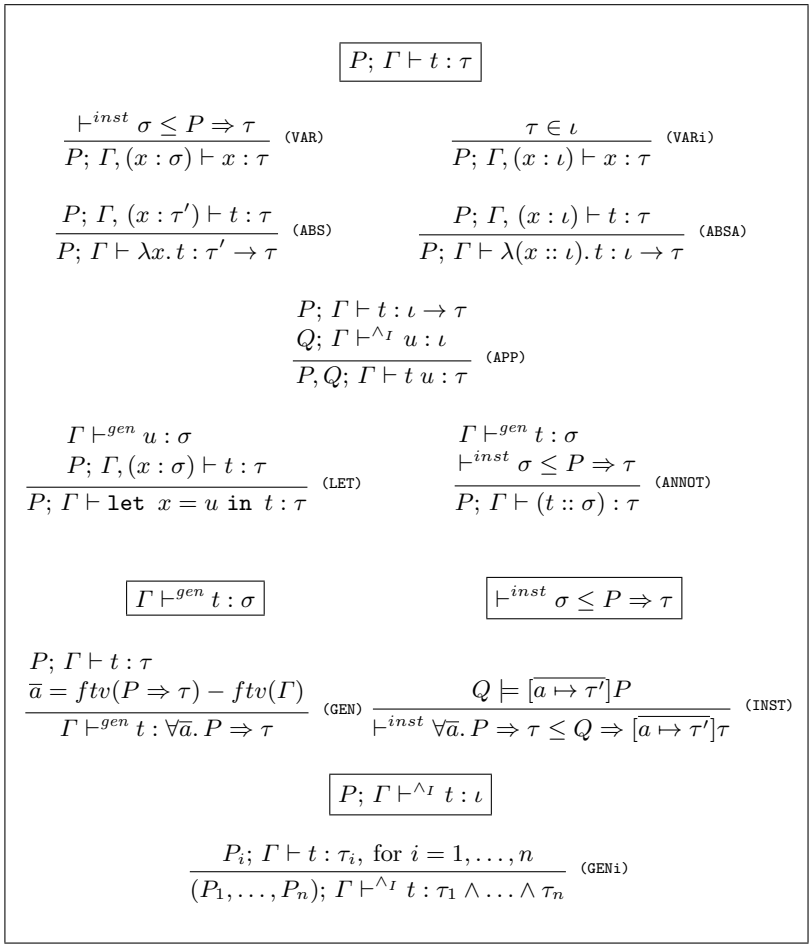


**Figure 2.** Type system `CTi`

The type system is presented in a syntax-directed form, where all type derivations for a given term $t$ (if there are any) have the same structure, uniquely de-

termined by the syntax structure of $t$. A syntax directed-formulation for a type system is closer to type inference, making it easier to derive the type inference algorithm directly from the type system.

The type rules presented in Figure 2 that are related to the handling of constraints are the usual ones in a syntax-directed type system for constrained polymorphism [10]. Type constraint introduction and polymorphic generalisation are restricted to the type derivation rule for let-bindings `(LET)`, and are defined in Figure 2 by means of the auxiliary judgment

$$\Gamma \vdash^{gen} t : \sigma$$

The rule for this judgment essentially says that all constraints on the type variables of a type must be moved from the global context to this type before universal quantification of its type variables.

Dually, polymorphic instantiation and type constraint elimination are restricted to rule `(VAR)`. Constrained polymorphic instantiation is defined in Figure 2 by means of the auxiliary judgement

$$\vdash^{inst} \sigma \leq P \Rightarrow \tau$$

Notation $[\overline{a \mapsto \tau'}]$, used in this rule, is an abreviation for $[a_1 \mapsto \tau'_1, \ldots, a_n \mapsto \tau'_n]$, which represents the capture-avoiding substitution that maps each type variable $a_i$ to monointersection type $\tau_i$, for $i = 1, \ldots, n$.

Rule `(INST)` formalizes constrained polymorphic type instantiation: type $\forall \overline{a}.\, P \Rightarrow \tau$ may be instantiated to type $Q \Rightarrow [\overline{\alpha \mapsto \tau'}]\tau$, provided the set of constraints $[\overline{\alpha \mapsto \tau'}]P$ is entailed by the set of constraints $Q$, according to the assumed constraint entailment relation. For example, the usual class and instance definitions for class `Eq` $a$ in Haskell induce the relation `Eq` $a \models$ `Eq [`$a$`]`, and thus type $\forall a.$`Eq` $a \Rightarrow a \to a \to$ `Bool` can be instantiated to type `Eq` $a \Rightarrow$ `[`$a$`]` $\to$ `[`$a$`]` $\to$ `Bool`.

The only new rules are `(VARi)`, `(ABSA)` and `(APP)`. Rule `(VARi)` defines the type derivation for a term variable that is bound to an intersection type in the typing context, by means of intersection type elimination. Rule `(ABSA)`, very similar to the usual rule `(ABS)`, defines type derivations for lambda-abstractions for which the type of the parameter is annotated.

Type derivation for an application $(t\ u)$ may require intersection type introduction for the derivation of the type of the argument $(u)$, as discussed previously in Section 2. This is dealt with by means of the special judgment

$$P;\ \Gamma \vdash^{\wedge_I} u : \iota$$

The rule for the derivation of this judgment `(GENi)` formalizes the idea of intersection type introduction, also collecting the constraints that apply on each type $\tau \in \iota$ into the global constraint context.

Type system `CTi` is a conservative extension of the *qualified type system* [10], in the sense that, in the absence of type annotations, a judgment $P;\ \Gamma \vdash t : \tau$ is derived in system `CTi` if and only if it is also derived in the qualified type

system. This can be proved by noticing that, in the absence of type annotations, no intersection type can be introduced in the typing context, neither can such a type occur in the type of an expression. Thus, the new rules `(VARi)` and `(ABSA)` of system `CTi` are never used for the derivation of the type of an expression that does not include intersection type annotations. Moreover, the type rule for application `(APP)` of system `CTi` reduces to the usual type rule for application in the qualified type system, when only such expressions are considered.

Type inference can usually be derived almost directly from a syntax-directed formulation for a type system. The idea is to substitute each type that must be guessed in a type derivation rule by a *fresh* type variable, delaying guessing until these types can be later determined, by means of *unification* (see eg. [21]). This technique cannot yet be applied, however, to the rules defined in Figure 2, because a different source of type guessing arises: on intersection type elimination, in rule `(VARi)`, and on intersection type introduction, in the derivation of a judgment $P;\, \Gamma \vdash^{\wedge_I} u : \iota$, used in rule `(APP)`. The idea is to use information provided by type annotations as an additional context for guessing the appropriate type in these cases. Information provided by type annotations therefore must be propagated inwards in a type derivation. The idea of taking advantage of type annotations in this way is called *bidirectional type inference* [22], or *local type inference* [24]. Bidirectional type inference for system `CTi` is presented in the next section.

## 5 Bidirectional Type Inference

The bidirectional type inference rules for system `CTi` are presented in Figure 3. The typing rules defined in this figure express the idea of propagating types inwards, and describe two similar typing judgments:

$$P;\, \Gamma \vdash_{\Uparrow} t : \tau$$

means that, in a typing context $\Gamma$, term $t$ can be inferred to have type $\tau$, provided that predicates in $P$ are satisfiable.

$$P;\, \Gamma \vdash_{\Downarrow} t : \tau$$

means that, in a typing context $\Gamma$, term $t$ can be checked to have type $\tau$, provided that predicates in $P$ are satisfiable. The up-arrow ($\Uparrow$) suggests pulling a type up out of a term, whereas the down-arrow ($\Downarrow$) suggests pushing a type down into a term. The auxiliary judgments for type generalization, $\Gamma \vdash^{gen} t : \sigma$, polymorphic instantiation, $\vdash^{inst} \sigma \leq P \Rightarrow \tau$, and intersection type introduction, $P;\, \Gamma \vdash^{\wedge_I} t : \iota$, are treated in the same way.

The main idea of the bidirectional typing rules is that a term might be typeable in checking mode when it is not typeable in inference mode; for example the term $(\lambda(x :: \mathtt{Int} \wedge \mathtt{Bool}.\, x)$ can be checked with type $(\mathtt{Int} \wedge \mathtt{Bool}) \to \mathtt{Int}$, but is not typeable in inference mode. On the other hand, if we infer the type for a term, we can always check that the term has that type. That is:

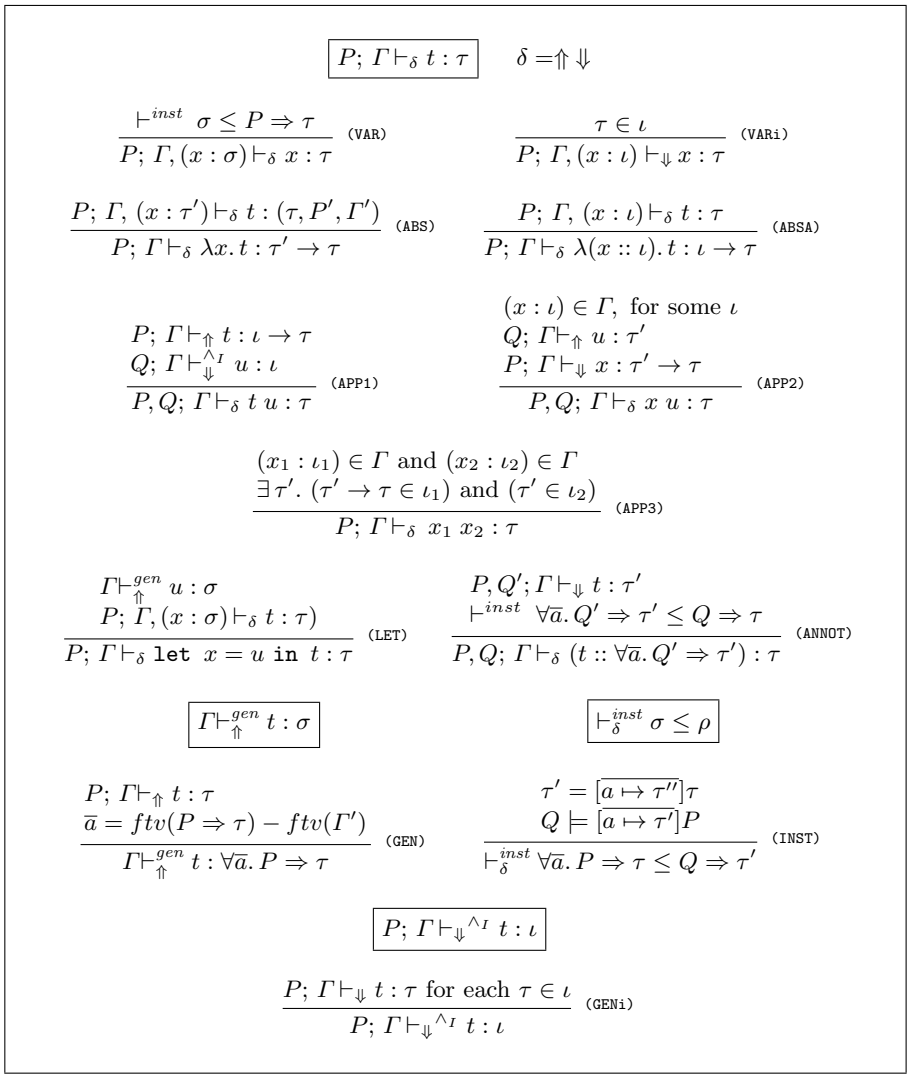


**Figure 3.** Bidirectional type inference for type system `CTi`

$$\text{If } P;\ \Gamma \vdash_\Uparrow t : \tau \text{ then } P; \Gamma \vdash_\Downarrow t : \tau$$

Furthermore, the checking mode allows us to give to a term types that are more specific than its most general type. In contrast, the inference mode may only produce the most general type. For example, if a variable has type $(\forall a.\, a)$, we can check that it has this type and also that it has types `Int`, `Int` $\to$ `Int`, $\forall a.\, [a] \to [a]$ etc. On the other hand, we will only be able to infer $b$, where $b$ is a fresh type variable.

Most of the rules in Figure 3 are the same in any direction $\delta$, namely `(VAR)`, `(ABS)`, `(ABSA)`, `(LET)` and `(ANNOT)`. They can be seen as shorthand for two rules that differ only in the arrow direction.

Intersection type elimination, defined by rule `(VARi)`, is not allowed in the inference mode, since this would imply a nondeterministic choice for the type of the variable, among the component types of the intersection type to which it is bound in the typing context. In type checking mode, on the other hand, this amounts to verifying that the type to be checked for the variable occurs in its intersection type.

Analogously, intersection type introduction, defined by means of judgment $P; \Gamma \vdash_{\Downarrow}{}^{\wedge_I} t : \iota$ is also only allowed in type checking mode, and amounts to checking that each type $\tau \in \iota$ can be derived for term $t$ in context $(P; \Gamma)$.

Type inference and checking for an application $(t\ u)$ is now split into three rules, in order that cases that require intersection type elimination can be treated more properly. Rule `(APP3)` is used only for typing an application $(t\ u)$ where both $t$ and $u$ are variables (intersection type parameters), namely, $x_1$ and $x_2$, bound to intersection types $\iota_1$ and $\iota_2$ in $\Gamma$, respectively. This case requires simultaneous intersection type elimination for derivation of the types for both $x_1$ and $x_2$. This is the case, for example, of each application `h y` in the body of function `f2` of Example 3, and also of application `s z` in the body of function `f3` of Example 4, both discussed in Section 2. Using the information provided by the type annotation in the definition of function `f2`, one can *check* that the first application `h y` in the body of this function may be assigned type `Int`, and the second may be assigned type `Bool`. In a type inference algorithm based on the bidirectional system, application `(s z)`, in the body of function `f3`, can be detected as *ambiguous*: this is done by imposing an additional condition on rule `(APP3)`, for checking that there exists a *unique* type $\tau'$ such that $\tau' \to \tau \in \iota_1$ and $\tau' \in \iota_2$, where $\tau$ is known, in check mode. The implementation of type inference and detection of ambiguity is briefly discussed at the end of this section.

Rule `(APP2)` is used only for typing an application $(t\ u)$ where $t$ is variable, namely, $x$, bound to an intersection type $\iota$ in $\Gamma$. This is the case, for example, of each use of parameter `g` in the body of function `foo` of Example 1. In this case, intersection type elimination is required for inferring the type for $x$, and the type inferred for the argument $u$ provides an additional type context for the intersection type elimination. In other words, a type $\tau'$ is firstly inferred for the argument $u$, and knowledge of this type is then used for checking the type of $x$, that is, for choosing a type $\tau' \to \tau$ among the component types of type $\iota$ bound to $x$, where type $\tau$ is known in checking mode. Note that this choice is guaranteed to be unique in check mode, but not in inference mode. This possible ambiguity on the choice of the type for $x$ also can be detected by a type inference algorithm based on the bidirectional system, by imposing an additional condition on rule `(VARi)`, that requires that this choice must be unique.

Rule `(APP1)` is used in all other applications $(t\ u)$ where the other two rules for application do not apply, that is, when the term $t$ is not a variable, or it is a variable that is not bound to an intersection type in the given typing context.

In this case, we first infer a type $\iota \to \tau$ for function $t$, and then check that the argument $u$ has type $\iota$. In this way we take advantage of the information given by the parameter type of $t$, to provide an additional type context for determining the type for the argument $u$. Notice that, even in checking mode, we ignore the required type $\iota$ when inferring the type for the function $t$, because this additional information is not important in this case.

Rule `(APP1)` accounts indeed for three distinct possible cases:

1. The parameter type for $t$ is indeed an intersection type $\iota$. In this case, type checking for the type of $u$ involves intersection type introduction, which is driven by the required type $\iota$. This is the case, for example, of applications `foo reverse` and `foo sort` discussed in Section 2, where `foo` is the function defined in Example 1.
2. The parameter type for $t$ is not an intersection type, say, the type inferred for $t$ is $\tau' \to \tau$, and $u$ is a variable bound to an intersection type $\iota_u$. This is the case, for example, of each use of parameter $x$ in the body of function `f1` of Example 2. In this case, checking for the type of $u$ involves intersection type elimination, which is driven by the required type $\tau'$.
3. The last case corresponds to usual applications where neither the function parameter type nor the argument type are an intersection type. Note that, in this case, rule `(APP1)` corresponds to the usual inference rule for an application of constrained polymorphic type systems.

The correspondence between the bidirectional type inference defined in Figure 3 and type system `CTi` defined in Figure 2, is stated by the following theorems (proofs are omitted for space reasons and will be provided in a technical report):

**Theorem 1 (Soundness).** *If* $P; \Gamma \vdash_{\Uparrow} t : \tau$ *then* $P; \Gamma \vdash t : \tau$.

**Theorem 2 (Completeness).** *If* $P; \Gamma \vdash t : \tau$ *then* $P; \Gamma \vdash_{\Uparrow} t : \tau$.

A bidirectional type inference algorithm that infers principal types for expressions of type system `CTi` can be derived directly from the rules of Figure 3, using the ideas described in [22], and providing an implementation for the entailment relation $P \models Q$ that is used in these rules. We have developed an implementation for this algorithm, written in Haskell, which is available at `https://github.com/emcardoso/CTi`.

Another important aspect of the type inference is detection of ambiguity. There are two possible sources of ambiguity in system `CTi`: those arising from the use of parameters annotated with an intersection type, and those arising from the use of overloaded symbols. The first source of ambiguity is ruled out by imposing appropriate conditions on rules `(APP3)` and `(VARi)`, for checking that when intersection type elimination is required, the appropriate type is uniquely determined. Ambiguity arising from overloading can be treated as usual, that is, by imposing a suitable syntactic condition over the *principal type* inferred for an expression, that guarantees that this expression is not ambiguous. In the case of a type system with single parameter type classes, a term $t$ with inferred type

$\forall \overline{a}.\, P \Rightarrow \tau$ can be guaranteed to be nonambiguous if $ftv(P) - ftv(\tau) = \emptyset$, as proposed in [10]. For multiparameter type classes a more appropriate condition should be used (see, for example, [2] or [6]). A more detailed discussion of the implementation of the type inference algorithm and the detection of ambiguity will be presented in a further work.

## 6 Conclusion

This paper presents a type system, called `CTi`, that is a conservative extension of the system Hindley-Milner plus type classes with the introduction of a restricted form of higher rank polymorphism that allows to specify parameter types with intersection types. A type inference algorithm that is sound and complete with respect to the type system is also presented. A prototype of the type inference algorithm has been implemented in Haskell. The higher rank polymorphism provided by `CTi` is complementary to the usual higher rank polymorphism of type systems based on system F. There are expressions that can be typed in `CTi` that cannot be typed in these other higher rank systems, and vice-versa.

## References


1. Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2nd edition, 1998.
2. Carlos Camarão, Rodrigo Ribeiro, Lucília Figueiredo, and Cristiano Vasconcellos. A solution to haskell's multiparamemeter type class dilemma. In *Proceedings of 13th Brazilian Symposium on Programming Languages*, pages 19–21, 2009.
3. M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the $\lambda$-calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 1980.
4. M Coppo, M Dezani-Ciancaglini, and B. Veneri. *Principal type schemes and lambda-calculus semantics*. Academic Press, 1980.
5. L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.
6. Simon Peyton Jones et. al. The glorious glasgow haskell compilation system user's guide, version 4.0.6. available at `http://www.haskell.org/ghc/docs/6.12.2/users_guide.pdf`, 1999.
7. Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.
8. Cordelia Hall, Kevin Hammond, Simon Peyton-Jones, and Philip Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.
9. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, 1969.
10. Mark P. Jones. *Qualified Types: theory and practice*. PhD thesis, University of Nottingham, Department of Computer Science, 1994.
11. Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5:1–35, 1995.

12. Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*. ACM Press, June 1997.
13. A. J. Kfoury and J. Tiuryn. Type reconstruction in finite fragments of second order lambda calculus. *Information and Computation*, 98(2):228–257, 1992.
14. A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *POPL'99: ACM Conference on Principles of Programming Languages*, pages 161–174. ACM Press, 1999.
15. Didier Le Botlan and Didier Remy. Mlf: Raising ML to the power of system F. In *ICFP'2003: International Conference on Functional Programming*, pages 27–38. ACM Press, 2003.
16. Didier Le Botlan and Didier Remy. Recasting MLF. *Information and Computaion*, 207(6):726–785, 2009.
17. Daan Leijen. Hmf: Simple type inference for first-class polymorphism. In *ICFP'2008: International Conference on Fucntional Programming*, pages 283–294. ACM Press, 2008.
18. Dann Leijen. Flexible types: robust type inference for first-class polymorphism. In *POPL'2009: International Conference on Principles of Programming Languages*, pages 66–77. ACM Press, 2009.
19. Dann Leijen and Andres Löh. Qualified types for MLF. In *ICFP'2005: International Conference on Fucntional Programming*, pages 144–155. ACM Press, 2005.
20. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML, revised edition*. MIT Press, 1997.
21. John Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
22. S. Peyton-Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.
23. Simon Peyton-Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 4 edition, 2003.
24. Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.
25. John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag.
26. C. Shan. Sexy types in action. *ACM SIGPLAN Notices*, 39(5):15–22, 2004.
27. Mark Shields and Simon Peyton-Jones. Lexically scoped type variables. `http://research.microsoft.com/~simonpj/papers/scoped-tyvars/scoped.ps`, 2004.
28. Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999.
29. Janis Voigtlander. *Types for Programming Reasoning*. PhD thesis, Technische Universität Dresden, 2009.
30. Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. FPH: first-class polymorphism for haskell. In *ICFP08: Proc. of the 13th ACM SIGPLAN Int. Conf. on Functional Programming*, pages 295–306. ACM Press, 2008.
31. J. B. Wells. Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.