# Discriminative Sum Types Locate the Source of Type Errors

Matthias Neubauer Peter Thiemann
Universität Freiburg
{neubauer,thiemann}@informatik.uni-freiburg.de

## Abstract

We propose a type system for locating the source of type errors in an applied lambda calculus with ML-style polymorphism. The system is based on discriminative sum types—known from work on soft typing—with annotation subtyping and recursive types. This way, type clashes can be registered in the type for later reporting. The annotations track the potential producers and consumers for each value so that clashes can be traced to their cause.

Every term is typeable in our system and type inference is decidable. A type derivation in our system describes all type errors present in the program, so that a principal derivation yields a principal description of all type errors present. Error messages are derived from completed type derivations. Thus, error messages are independent of the particular algorithm used for type inference, provided it constructs such a derivation.

## Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Polymorphism*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type Structure*

## General Terms

Languages, Theory

## Keywords

type inference, polymorphism, type errors

## 1 Introduction

Many functional programming languages have type systems which are derived from ML's type system with parametric polymorphism [22]. ML-style polymorphism has proved to be a practical compromise which allows for expressive polymorphic definitions while keeping type inference decidable.

However, despite 25 years of experience with ML-style typing and numerous implementations of type inference algorithms for this kind of type system, programmers are still struggling with the error messages reported when the inference algorithm fails. Virtually every ML programmer can tell stories about type errors where it took hours to identify the actual problem with the program. While initiated functional programmers seem to accept this as a fact of life to be endured in return for the wonderful type soundness guarantee, it makes life hard for beginners, in fact, too hard for some.

The root of the problem lies in the operational way in which type inference algorithms produce error messages. Most algorithms are based on Milner's algorithm $\mathcal{W}$ [22] that traverses the syntax tree of an expression and composes the type of the expression bottom-up. At each function application $e\ e'$, the algorithm recursively computes the types $t$ of $e$ and $t'$ of $e'$. Next, it has to make sure that the type of $e$ really is a function type. Hence, the algorithm attempts to unify $t$ with $t' \to \beta$, where $\beta$ is a fresh type variable. It reports a type error if this unification fails. (Similar unifications happen at elimination expressions for other type constructors.)

Unfortunately, the point in the expression where the unification fails may not even be close to the point of the actual mistake in the program. For example, when processing

$$(\lambda f.f\ 1)\ (\lambda y.\texttt{if}\ y\ 1\ 0) \tag{1}$$

algorithm $\mathcal{W}$ first computes the types of the subexpressions as

$$\lambda f.f\ 1 : (\texttt{int} \to \alpha) \to \alpha \qquad \lambda y.\texttt{if}\ y\ 1\ 0 : \texttt{bool} \to \texttt{int}$$

and then tries to unify

$$(\texttt{int} \to \alpha) \to \alpha \doteq (\texttt{bool} \to \texttt{int}) \to \beta$$

which results (in the worst case) in the error message "Failed to unify `int` with `bool`" issued at the outermost application in expression (1). Only indirectly does this message point to the actual problem, namely the mismatch of the type of $f$'s argument and its use in the body of $f$:

$$(\lambda f.f\ \boxed{1}^{+})\ (\lambda y.\texttt{if}\ \boxed{\texttt{y}}^{-}\ 1\ 0)$$

That is, $f$ is applied to an argument of type `int` which is *produced* by the expression $\boxed{1}^{+}$ (the annotation $^{+}$ flags a producer expression, which is the source for some value) whereas the function bound to $f$ *consumes* its argument as a `bool` in the context

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*ICFP'03,* August 25–27, 2003, Uppsala, Sweden.
Copyright 2003 ACM 1-58113-756-7/03/0008 ...$5.00

$\texttt{if}\ \boxed{\phantom{x}}^{-}\ 1\ 0$ (the annotation $^{-}$ indicates an expression used in an elimination context or a consumer position).

Numerous attempts have been made at explaining type errors and locating their actual source. They range from instrumentations of algorithm $\mathcal{W}$ through alternative type inference algorithms to approaches relying on principal typings. Section 6 discusses a representative sample of this related work.

## 1.1 Locating Errors with Multivocal Types

In the present work, we pursue an approach based on the theory of discriminative sum types, which has been developed for soft typing [7, 18, 38], and which is closely related to systems with row types [35, 29]. While row types are usually indexed with record or variant labels, discriminative sum types are indexed with *type constructors* and the component types are the argument types of the constructors.

Anticipating the formal definition in Section 3.4, the idea of a discriminative sum type is that at each node of a type each type constructor may be used at most once. The "unused" type constructors are hidden in a *row variable* $\rho$ and we use ";" to separate different type constructors. The separator ";" binds weaker than every type constructor. For example, in the node $(\texttt{int};\rho_a \to \rho_b;\rho_1)$ of a type there are two components, one for the type (constructor) int and another for $\to$.

The most interesting operation on discriminative sum types is unification, because it must preserve the above invariant. The idea here is that types unified componentwise and by substituting for the row variables if there is no corresponding component (as with row types). For example, unification of $(\texttt{int};\rho_a \to \rho_b;\rho_1)$ with $(\texttt{bool};\rho_2)$ results in the substitution $\rho_1 \mapsto (\texttt{bool};\rho')$ and $\rho_2 \mapsto (\texttt{int};\rho_a \to \rho_b;\rho')$ yielding the substituted term $(\texttt{int};\rho_a \to \rho_b;\texttt{bool};\rho')$.

An alternative view would consider each node in a discriminative sum type as a finite map $\nu$ from the set of type constructors to a list of type nodes, so that $\nu(\chi)$ are the arguments of type constructor $\chi$.

Each type in a system of discriminative sum types may be *multivocal*, that is, it may have more than one top-level type constructor. The type $(\texttt{int};\texttt{bool};\rho)$ is an example for such a multivocal type where $\rho$ is a (row) type variable. A type like $(\texttt{bool};\rho_2)$ with at most one type constructor is *univocal*.

For the subexpressions of our example, type inference now yields

$$\begin{array}{rcl} \lambda f.f\ 1 & : & (((\texttt{int};\rho_1) \to \rho_2;\rho_3) \to \rho_2;\rho_4) \\ \lambda y.\texttt{if}\ y\ 1\ 0 & : & ((\texttt{bool};\rho_5) \to (\texttt{int};\rho_6);\rho_7) \end{array}$$

and unification of the two types

$$\begin{array}{rl} & (((\texttt{int};\rho_1) \to \rho_2;\rho_3) \to \rho_2;\rho_4) \\ \doteq & (((\texttt{bool};\rho_5) \to (\texttt{int};\rho_6);\rho_7) \to \rho_8;\rho_9) \end{array}$$

succeeds with the multivocal type

$$(((\texttt{bool};\texttt{int};\rho_{10}) \to (\texttt{int};\rho_6);\rho_3) \to (\texttt{int};\rho_6);\rho_4)$$

by substituting $\rho_1 \mapsto (\texttt{bool};\rho_{10})$, $\rho_5 \mapsto (\texttt{int};\rho_{10})$, $\rho_2 \mapsto (\texttt{int};\rho_6)$, $\rho_7 \mapsto \rho_3$, and $\rho_9 \mapsto \rho_4$. In fact, type inference succeeds for the entire expression and computes the result type $(\texttt{int};\rho_6)$ (which is *univocal*, that is, there is at most one type constructor present).

The gain in doing so is that we can inspect the type derivation after the inference algorithm has completed its work:

$$\begin{array}{rcl} \lambda f.f\ 1 & : & (((\texttt{bool};\texttt{int};\rho_{10}) \to (\texttt{int};\rho_6);\rho_3) \\ & & \qquad \to (\texttt{int};\rho_6);\rho_4) \\ \lambda y.\texttt{if}\ y\ 1\ 0 & : & ((\texttt{bool};\texttt{int};\rho_{10}) \to (\texttt{int};\rho_6);\rho_3) \\ (\lambda f.f\ 1)\ (\lambda y.\texttt{if}\ y\ 1\ 0) & : & ((\texttt{bool};\texttt{int};\rho_{10}) \to (\texttt{int};\rho_6);\rho_3) \end{array}$$

In particular, the two program points that actually caused the problem also have multivocal types:

$$\begin{array}{rcl} f : ((\texttt{bool};\texttt{int};\rho_{10}) \to (\texttt{int};\rho_6);\rho_3) \vdash 1 & : & (\texttt{bool};\texttt{int};\rho_{10}) \\ y : (\texttt{bool};\texttt{int};\rho_{10}) \vdash y & : & (\texttt{bool};\texttt{int};\rho_{10}) \end{array}$$

However, the typing still does not relate enough information what exactly happened and why/if these two program points are the culprits for the type error.

For that reason, each type constructor in a discriminative sum type must carry a *flow set* annotation. A flow set is a set of program labels that indicate the potential sources and sinks of a value whose type carries that flow set. Since each type constructor carries a separate flow set, each component's flow is traced separately. There is a further distinction between *source labels* with superscript $^{+}$ that indicate potential producers of a value and *sink labels* with superscript $^{-}$ that indicate potential consumers of a value.

Returning to our example, we first label each source or sink subexpression appropriately:

$$[[\lambda f.[f]_{3^-}\ [1]_{4^+}]_{2^+}]_{1^-}\ [\lambda y.\texttt{if}\ [y]_{6^-}\ [1]_{7^+}\ [0]_{8^+}]_{5^+}$$

Looking again at *any* of the above types involving the multivocal part $(\texttt{bool};\texttt{int};\rho_{10})$, we find the following annotations:

$$y : (\texttt{bool}^{6^-};\texttt{int}^{4^+};\rho_{10})$$

This annotated type indicates that the error stems from the fact the value of type int produced at program point $4^{+}$ may be consumed as a value of type bool by the if expression at program point $6^{-}$.

It will turn out that a multivocal type $t$ is a sufficient indicator for a type error, but further machinery is required to report such an error. As a first step, we consider the set of subexpressions whose type contains $t$ or that consume a value whose type contains $t$. Following Haack and Wells [14], we visualize such a set as an *expression slice* where subexpressions and subcontexts that do not contribute to the multivocal type are blanked out using ellipses (..). In the example, the expression slice associated with $(\texttt{bool}^{6^-};\texttt{int}^{4^+};\rho_{10})$ is

$$[[\lambda f.[f]_{3^-}\ [1]_{4^+}]_{2^+}]_{1^-}[\lambda y.\texttt{if}\ [y]_{6^-}\ (..)\ (..)]_{5^+}$$

The proposed error message for this expression is the above slice with additional highlighting of the offending consumers and producers as detailed in Section 2.4. In Section 2, we elaborate on the additional features, annotation subtyping and recursive types, required to turn type inference for discriminative sum types into a practically useful framework for locating type errors. We motivate these extensions with examples.

## 1.2 Contributions

We have designed a type system with discriminative sum types that enables the precise localization of the causes of type errors. The type system is a conservative extension of the Hindley/Milner type system with parametric polymorphism. It is inspired by work on soft typing and row types and it inherits many of its properties:

- **type inference is decidable**, even in the presence of annotation subtyping and recursive types;
- the type inference algorithm can produce a principal type derivation; we view this principal type derivation as a **principal description of the type errors** in the term;
- **type errors** are reported by interpreting the completed type derivation, hence they **are independent of the type inference algorithm** used to compute them.

The technical contributions of the paper are the following. Starting from an applied lambda calculus, we define a type system with discriminative sum types and ML-style polymorphism, which is parameterized over a certain style of constraint systems, and prove subject reduction. Then we extend the calculus with labeling and establish the correctness of the data flow information in the flow set annotations by proving subject reduction for the labeled calculus in the type system with simple annotations and with annotation subtyping. We show that this information is valid also for the original, unlabeled calculus by relating the labeled and the unlabeled calculus via a reduction correspondence. Further, we prove that our system is a conservative extension of ML and, conversely, characterize those type derivations that give rise to an ML type derivation.

There is a prototype implementation of the type inference algorithm with annotation subtyping for an applied lambda calculus.

## 1.3 Overview

In the following Section, we explain the additional features annotation subtyping and recursive types. Section 3 contains the theoretical basis of our system. It introduces the labeled calculus $\lambda L$, a type system with discriminative sum types and ML-style polymorphism, and states some technical results. Section 4 discusses extensions that are required to make the system amenable to a full programming language. Section 5 contains some notes on our prototype implementation. Finally, we discuss related work in Section 6 and conclude.

# 2 Locating Type Errors

The example in the introduction has given some flavor of the kind of error messages that can be extracted from our type system. However, the example is chosen so that it does not exert the full power of the system. Further features, in particular annotation subtyping and recursive types are needed. The following examples motivate them and show how error messages can be extracted from typings in the respective extended system.

## 2.1 Annotation Subtyping

In the flow sets attached to each type constructor, the type system performs a flow analysis that tracks the sources (producers, introductions) and the sinks (consumers, eliminations) of all values. Annotation subtyping increases the precision of that analysis by modeling the direction of the data flow. For the monomorphic case, similar type systems have been shown to be equivalent in power to 0CFA [16, 26].

### 2.1.1 Flowing Forwards

Consider the following example expression:

$$\lambda x.(x, \texttt{if true (if true } [\texttt{true}]_{1^+}\ [0]_{2^+})\ x) \tag{2}$$

Both conditionals have type $(\texttt{int}^{2^+};\texttt{bool}^{1^+};\rho)$ indicating that their value is either the boolean introduced at point $1^+$ or the integer introduced at point $2^+$. Unfortunately, the same type (including the annotation) is also inferred for the variable $x$ which cannot assume either of these values in the given expression.

This so-called "poisoning" is a well-known phenomenon of equation-based flow analysis. It is due to the analysis equating the result types of both branches of the outer conditional. These equations induce an artificial backwards flow that can never happen during execution of the program.

The remedy is to move from an equation-based system to a subtyping-based system. To this end, we rely on a subtyping relation of flow-annotated types that preserves the structure of the types and only affects the annotations. In the example expression (2), this approach leads to the following typings:

$$\begin{array}{rcl}
[\texttt{true}]_{1^+} & : & (\texttt{int};\texttt{bool}^{1^+};\rho) \\
[0]_{2^+} & : & (\texttt{int}^{2^+};\texttt{bool};\rho) \\
\texttt{if true } [\texttt{true}]_{1^+}\ [0]_{2^+} & : & (\texttt{int}^{2^+};\texttt{bool}^{1^+};\rho) \\
x & : & (\texttt{int};\texttt{bool};\rho) \\
\texttt{if true (if true } [\texttt{true}]_{1^+}\ [0]_{2^+})\ x & : & (\texttt{int}^{2^+};\texttt{bool}^{1^+};\rho)
\end{array}$$

These typings seem to indicate that—while $x$ is affected by the type error—it actually does not contribute to it because its multivocal type is *not inhabited* (its flow sets are empty). However, this inference is not correct in general as we shall see in Section 2.1.3 below. Hence, we report the type error as the slice

$$\lambda x.(..)(x, \texttt{if}\ (..)\ (\texttt{if}\ (..)\ [\texttt{true}]_{1^+}\ [0]_{2^+})\ x).$$

### 2.1.2 Flowing Backwards

The above example demonstrates an error caused by an expression that might evaluate to values of different type. The typing correctly tracks the values flowing "forward" from their introduction to the offending expression.

However, the dual situation where two subexpressions want to consume the same value at different types also gives rise to a type error. Since there may be no producer in the expression, it is necessary to track the consumers, too. Contrary to information about producers of values, the information about consumers of values *flows backwards*, towards the source of the value. Hence, the subtyping relation transports consumer labels "from right to left", so that the subtype always has more consumer labels than the supertype.

Here is an example that demonstrates the backward flow of consumer labels in action:

$$\lambda x.(\texttt{if}\ [x]_{2^-}\ [0]_{4^+}\ [1]_{5^+}, [x]_{6^-}\ [1]_{8^+})$$

In this expression, $x$ is used in two different elimination contexts, once as a boolean and once as a function. Hence, its typing is

$$x \quad : \quad (\texttt{bool}^{2^-};((\texttt{int}^{8^+};\rho_1) \to^{6^-} \rho_2);\rho_0).$$

Since only the top-level of this type is multivocal, the only subexpressions concerned with the error are $2^-$ and $6^-$ and the occurrences of the variable $x$. Hence, the offending slice is

$$\lambda x.(..)(\texttt{if}\ [x]_{2^-}\ (..)\ (..), [x]_{6^-}\ (..)).$$

### 2.1.3 Inhabitation

The next example demonstrates that inhabitation of an expression's multivocal type is not always a good indicator for the role of that expression in a type error.

$$\lambda x.\lambda y.(\texttt{if true}\ x\ [0]_{4^+}, \texttt{if true}\ x\ y, \texttt{if true}\ y\ [\texttt{false}]_{5^+})$$

Here are the typings of the interesting subexpressions:

$$\begin{array}{rcl} x & : & (\texttt{bool};\texttt{int};\rho_x) \\ y & : & (\texttt{bool};\texttt{int};\rho_y) \\ \texttt{if true}\ x\ [0]_{4^+} & : & (\texttt{bool};\texttt{int}^{4^+};\rho_1) \\ \texttt{if true}\ x\ y & : & (\texttt{bool};\texttt{int};\rho_2) \\ \texttt{if true}\ y\ [\texttt{false}]_{5^+} & : & (\texttt{bool}^{5^+};\texttt{int};\rho_3) \end{array}$$

Although the type of the second subexpression, $\texttt{if true}\ x\ y$, is uninhabited, this subexpression is clearly contributing to the type error in an essential way. In fact, all subexpressions, except the conditions, are essential for obtaining a type error in this case.

Hence, we refrain from using inhabitation of a multivocal type as an indicator whether a subexpression participates in a type error. However, it makes sense to indicate the degree of inhabitation visually, since observing changes in inhabitation can be helpful in certain cases (cf. the example in Section 2.1.1).

## 2.2 Recursive Types

The standard implementation of Hindley/Milner-style type inference fails at programs like

$$[f]_{0^+}\ [x:xs]_{1^-} = [add\ ([f]_{2^-}\ xs)\ ([f]_{3^-}\ x)]_{5^+}$$

The reason for this failure is that $x$ must have type $\alpha$ and type $\texttt{list}\ \alpha$ at the same time. However, $x$ could be assigned the recursive type $\mu\alpha.\texttt{list}\ \alpha$, which is theoretically sound. Although ML-style type inference with recursive types is feasible, the resulting types are often unintuitive. For that reason, the standard algorithm rules out recursive types by using a unification algorithm with "occur-check".

However, a type system to investigate type errors must be able to give a type to the above term so that a suitable error message can be extracted from the typing. Hence our system includes recursive types and it assigns the type

$$f : (\rho \to^{0^+,2^-,3^-} (\texttt{int}^{5^+};\rho_2);\rho_0)\ \textit{where}\ \rho = (\texttt{list}^{1^-}\ \rho;\rho_1).$$

How much information can be extracted from this typing?

1. Since $\rho$ refers to itself, we can deduce that the standard inference algorithm reports an occurs-check error.
2. The type constructor `list` that is involved in the recursive definition is annotated with $1^-$, indicating that the pattern matching on $f$'s argument contributes to the problem.
3. By examining the rest of the typing derivation, we find that $x:\rho$ and $xs:\rho$, that is, only the typing of the context $add\ [\,]\ [\,]$ does not involve the type $\rho$.

Hence, the offending slice in the definition of $f$ is

$$[f]_{0^+}\ [x:xs]_{1^-} = (..)\ ([f]_{2^-}\ xs, [f]_{3^-}\ x)$$

which exactly pinpoints the source of the problem. In this case, the typing does not involve a multivocal type. Hence, we must *define* which node of the type causes the error. Our choice is the header node of the loop, indicated by the variable $\rho$ in the example.

In the example, function $f$ only serves as a mediator because the recursive type does not appear at the top-level of $f$'s type. Our reporting phase will indicate this difference visually.

## 2.3 Flow Classes

The examples above only contain one type error at a time. When more than one error is present in a program, it is not obvious which occurrences of multivocal types belong together. For example, consider an expression that contains two copies of expression (2). Each copy has a number of subexpressions of type $(\texttt{int};\texttt{bool};\rho)$, where the association to a particular consumer or producer is not obvious because the flow sets in the type are empty.

What is needed is an additional classification of type nodes into equivalence classes. To this end, each expression and each type node is equipped with a *flow-class label*. The expression $[e]^\ell$ attaches its flow-class label $\ell$ to the top-level node of $e$'s type. A flow-class label behaves differently than a source or a sink label. It is propagated equationally, like the type structure itself, ignoring the direction of data flow.

With this set-up a flow-class label $\ell$ records in the type that a value of this type may be passed through by an expression with this label. For example, in

$$[\texttt{let}\ x = [42]_{1^+}\ \texttt{in}\ [x]^2]^3$$

the $[\,]^2$ and the $[\,]^3$ attach flow-class labels to the type constructor `int`, so that the type of $[x]^2$ is $(\texttt{int}^{1^+,2,3};\rho)$ (remember that flow-class labels are propagated equationally, that is, they go forwards and backwards), which is also the type of the whole expression.

In this way, each node in a type has a set of flow-class labels attached to it. Since propagation of these labels is equation-based, these sets are either disjoint or equal for any given pair of nodes. Hence, the sets of flow-class labels induce a partition on the set of type nodes. Moreover, all members of the same partition represent the same underlying type (after erasure of all flow annotations).

## 2.4 Collecting and Reporting Type Errors

Our proposed overall procedure for a type error reporting tool is derived from the above discussion. It works in two phases, a collecting phase and a reporting phase. The collection phase has the following tasks:

1. Decorate the expression with producer and consumer labels as well as with flow-class labels. All these labels must be distinct so that there is a mapping from the set of labels to the set of subexpressions occurrences in the original expression.
2. Perform type inference for the system proposed in this paper. The algorithm does not matter as long as it computes a mapping that maps each label to the type of the subexpression at that label.
3. During a traversal of all types of all subexpressions, collect the following set $E$ of *sets of flow-class labels*: if a type that contains a node $t$ so that either $t$ is multivocal or $t$ is the header of a recursive type, then $R \in E$ where $R$ is the set of all flow-class labels in the node $t$. The set $R$ is the *scope* of the error and $t$ the *offender*.

At this point, each element of $E$ corresponds to a type error that must be fixed separately. The reporting phase picks an element $R \in E$ and extracts a slice from the original expression that contains all subexpressions

- whose type contains a node marked with $R$, or
- that are consumer expressions where the consumed type contains a node marked with $R$.

With the slice it shows a general description, for example, "recursive type", "consumer/consumer conflict", "consumer/producer conflict", etc, that is derived from the offenders in the type of the slice. Furthermore, it highlights each subexpression $e$ of the slice by taking into account the following questions (depending on the programmer's settings):

- Does the offender occur at the top-level of $e$'s type? If not, this indicates that $e$ is merely transmitting the offense.
- What is the degree of inhabitation of the offender's type? That is, how many sources and sinks appear in its flow sets? Important hints can be drawn from changes in inhabitation.
- Is $e$ a consumer or a producer involved in the top-level of the offending type?

All this information can be readily extracted from the flow sets in a typing in our system.

In each of the examples in this section, the procedure leads exactly to the slices reported.

# 3 Formal System

Our term language consists of $\lambda$-terms with constants and a `let`-expression.

$$\begin{array}{llll} \textit{Variables} & x & \in & \textit{Var} \\ \textit{Constants} & c & \in & \textit{Const} \\ \textit{Terms} & e & ::= & c \mid x \mid e\ e \mid \lambda x.e \mid \texttt{let}\ x = e\ \texttt{in}\ e \end{array}$$

We adopt Barendregt's variable convention [5] and identify $\alpha$-equivalent terms $e$ and $e'$ by writing $e \equiv e'$. The notation $FV(e)$ denotes the set of free variables in $e$, and $e[x := e']$ denotes the result of substituting the free occurrences of $x$ in $e$ by $e'$.

## 3.1 The calculi $\lambda_{let}$ and $\lambda_{let,bool}$

We generate calculi by *notions of reduction*. The following two relations define the notions of $\beta$- and *let*-reduction:

$$\begin{array}{lrcl} (\beta) & (\lambda x.\ e)\ e' & \longrightarrow & e[x := e'] \\ (\textit{let}) & \texttt{let}\ x = e'\ \texttt{in}\ e & \longrightarrow & e[x := e'] \end{array}$$

For each notion of reduction $r$, $\longrightarrow_r$ denotes the compatible one-step reduction of $r$, $\twoheadrightarrow_r$ is the reflexive, transitive closure of $\longrightarrow_r$, and $=_r$ is the smallest equivalence relation generated by $\twoheadrightarrow_r$ [5].

The $\lambda_{let}$-calculus has no constants and is generated by $R = \{\beta, let\}$.

The calculus $\lambda_{let,bool}$ extend the calculus $\lambda_{let}$ by adding the three constants `true`, `false`, and `if` representing the introduction and elimination constructs for booleans. The following two additional reduction rules define their operational behavior:

$$\begin{array}{lrcl} (\textit{if.1}) & \texttt{if true}\ e_2\ e_3 & \longrightarrow & e_2 \\ (\textit{if.2}) & \texttt{if false}\ e_2\ e_3 & \longrightarrow & e_3 \end{array}$$

We recall the following known property of $\lambda_{let}$ and $\lambda_{let,bool}$.

FACT 1. *The calculi $\lambda_{let}$ and $\lambda_{let,bool}$ are confluent.*

## 3.2 The labeled calculus $\lambda L$

To identify certain subterms of a term and trace their flow during a sequence of reductions as discussed above, we introduce a new labeled $\lambda$-calculus inspired by labeled reductions of Barendregt [5] and Abadi et al. [1]. We distinguish different kinds of labels:

$$\begin{array}{llll} \textit{Source Labels} & \ell^+ & \in & \textit{Lab}^+ \\ \textit{Sink Labels} & \ell^- & \in & \textit{Lab}^- \\ \textit{Class Labels} & \ell & \in & \textit{Lab} \\ \textit{Sets of Class Labels} & L & \in & \mathcal{P}(\textit{Lab}) \\ \textit{Type Constructors} & \chi & \in & \textit{TyCon} \end{array}$$

The sets $Lab^+$, $Lab^-$, and $Lab$ are disjoint. Source labels, $\ell^+ \in Lab^+$, are attached to introduction expressions (like $\lambda$ or `true`) and trace the flow of produced values. Sink labels, $\ell^- \in Lab^-$, are attached to elimination contexts (like the first subexpression of function application or the condition of a conditional) and trace the attraction towards a consumer. Class labels are propagated equationally, that is, they flow both forwards and backwards. Sets of class labels are denoted by $L$, and type constructors are denoted by $\chi \in TyCon$ where we assume that $\texttt{bool}, \rightarrow \in TyCon$.

The calculus $\lambda L$ has three additional families of labeling constants.

- The constant $\lfloor\_\rfloor_{\chi,\ell^+}$ labels an expression that introduces a type constructor $\chi$ with the source label $\ell^+$.
- The constant $\lfloor\_\rfloor_{\chi,\ell^-}$ labels a context that eliminates a type constructor $\chi$ with the sink label $\ell^-$.
- The constant $\lfloor\_\rfloor^L$ annotates an expression with a set of class labels $L$.

We usually write $\lfloor e\rfloor^L$ instead of the juxtaposition $\lfloor\_\rfloor^L\ e$. Sometimes we use the meta variable $a$ to indicate an annotated term.

Six notions of reduction deal with labeling constants:

$$\begin{array}{lrcl} (\textit{L.union}) & \lfloor\lfloor e\rfloor^{L_1}\rfloor^{L_2} & \longrightarrow & \lfloor e\rfloor^{L_1 \cup L_2} \\ (\textit{L.swap}) & \lfloor\lfloor e\rfloor_{\chi,\ell^+}\rfloor^{L} & \longrightarrow & \lfloor\lfloor e\rfloor^{L}\rfloor_{\chi,\ell^+} \\ (\textit{L.elim}) & \lfloor\lfloor e\rfloor_{\chi,\ell_1^+}\rfloor_{\chi,\ell_2^-} & \longrightarrow & e \\ (\textit{L.lam}) & \lfloor\lambda x.e\rfloor^{L} & \longrightarrow & \lambda x.e \\ (\textit{L.true}) & \lfloor\texttt{true}\rfloor^{L} & \longrightarrow & \texttt{true} \\ (\textit{L.false}) & \lfloor\texttt{false}\rfloor^{L} & \longrightarrow & \texttt{false} \end{array}$$

The *L.union* rule collapses two consecutive labelings of an expression into one labeling by merging their label sets. The *L.swap* rule lets source labels $\ell^+$ travel outwards over a set of other labels $L$. If a source labeling $\ell_1^+$ with constructor annotation $\chi$ hits a sink labeling $\ell_2^-$ with the same constructor annotation $\chi$, they cancel each other by the *L.elim* rule. The rules *L.lam*, *L.true*, and *L.false* move a labeling out of the way by removing it completely.

The labeled $\lambda L$-calculus enjoys the same fundamental theorem as the unlabeled calculi:

PROPOSITION 1. *The calculus $\lambda L$ is confluent.*

PROOF. All critical pairs are joinable. Hence, the lemma follows using Theorem 6.2.4 of [4] and Proposition 3.3.5 of [5]. $\square$

An unlabeled term $e$ can be labeled in many ways, but not all label-

$$x \hookrightarrow [x]^L$$

$$\frac{e_1 \hookrightarrow e_1' \quad e_2 \hookrightarrow e_2'}{e_1\ e_2 \hookrightarrow [[e_1']_{\rightarrow,\ell^-}\ e_2']^L}$$

$$\frac{e \hookrightarrow e'}{\lambda x.e \hookrightarrow [[\lambda x.e']_{\rightarrow,\ell^+}]^L}$$

$$\frac{e_1 \hookrightarrow e_1' \quad e_2 \hookrightarrow e_2'}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \hookrightarrow [\mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2']^L}$$

$$\mathtt{true} \hookrightarrow [[\mathtt{true}]_{\mathtt{bool},\ell^+}]^L$$

$$\mathtt{false} \hookrightarrow [[\mathtt{false}]_{\mathtt{bool},\ell^+}]^L$$

$$\frac{e_1 \hookrightarrow e_1' \quad e_2 \hookrightarrow e_2' \quad e_3 \hookrightarrow e_3'}{\mathtt{if}\ e_1\ e_2\ e_3 \hookrightarrow [\mathtt{if}\ [e_1']_{\mathtt{bool},\ell^-}\ e_2'\ e_3']^L}$$

**Figure 1. Labeling**

ings make sense. The relation $e \hookrightarrow a$ (defined in Figure 1) specifies how an unlabeled term $e$ can be annotated with labels resulting in a sensibly labeled counterpart $a$. The relation guarantees that

- every subterm of $e$ carries a flow-class label in $a$,
- every introduction of type constructor $\chi$ is annotated with a source label for $\chi$, and
- every elimination of type constructor $\chi$ is annotated with a sink label for kind $\chi$.

LEMMA 1. *(Basic Properties of Labeling)*

- *If $e \hookrightarrow a$, then $a \equiv [a']^L$ for some $a'$ and $L$.*
- *If $e \hookrightarrow [a]^{L_1}$, then $e \hookrightarrow [a]^{L_2}$.*
- *If $e_1 \hookrightarrow a_1$, and $e_2 \hookrightarrow a_2$, then $e_1[x := e_2] \hookrightarrow a_1[x := a_2]$.*

With $erase(a)$, we denote the unlabeled term obtained by removing all labelings from $a$. The "unlabeled" calculus $\lambda_{let,bool}$ and its labeled counterpart $\lambda L$ correspond in the following way:

PROPOSITION 2. *(Simulation)*

- *If $e \longrightarrow e'$, and $e \hookrightarrow a$, then $a \longrightarrow\!\!\!\!\!\!\rightarrow a'$ for some $a'$ with $e' \hookrightarrow a'$.*
- *If $e \hookrightarrow a$, and $a \longrightarrow a'$, then $e \longrightarrow\!\!\!\!\!\!\rightarrow erase(a')$.*

## 3.3 The Damas-Milner Type System

We first recapitulate Damas and Milner's type system for Mini-ML [11, 9]. The types, type schemes, and type assumptions are:

| | | | | |
|---|---|---|---|---|
| *Type Variables* | $\alpha$ | $\in$ | *TyVar* | |
| *Types* | $t$ | $::=$ | $\alpha \mid \chi\, t_1 \ldots t_n$ | |
| *Type Schemes* | $s$ | $::=$ | $\forall\overline{\alpha}.t$ | |
| *Type Assumptions* | $A$ | $::=$ | $\cdot \mid A, x : t$ | $x \notin A$ |

where $\alpha$ ranges over a set of type variables. Type schemes are identified modulo $\alpha$-equivalence, and we write $A(x)$ for the type assigned to $x$ in $A$.

The type system of Mini-ML is defined by the deduction system in Figure 2. It constructs proofs for the type judgment $A \vdash_{DM} e : t$ stating that the term $e$ has type $t$ under type assumptions $A$.

$$\frac{T(c) \succ t}{A \vdash_{DM} c : t}$$

$$\frac{A(x) \succ t}{A \vdash_{DM} x : t}$$

$$\frac{A \vdash_{DM} e_1 : t_2 \rightarrow t_1 \quad A \vdash_{DM} e_2 : t_2}{A \vdash_{DM} e_1\ e_2 : t_1}$$

$$\frac{A, x : t_2 \vdash_{DM} e : t_1}{A \vdash_{DM} \lambda x.e : t_2 \rightarrow t_1}$$

$$\frac{A \vdash_{DM} e_1 : t_1 \quad (A, t_1) \vdash_{DM}^{gen} s \quad A, x : s \vdash_{DM} e_2 : t_2}{A \vdash_{DM} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : t_2}$$

**Figure 2. Typing rules of Mini-ML**

The deduction rules rely on the notions of *type instantiation* and *type generalization*:

DEFINITION 1. *(Instantiation) A type $t$ is an* instance *of a type scheme $s$, written $s \succ t$, if there is a substitution for the bound variables of $s$ yielding $t$.*

DEFINITION 2. *(Generalization) A type scheme is a* generalization *of type $t$ under some type assumptions $A$, written $(A, t) \vdash_{DM}^{gen} \forall\overline{\alpha}.t$, if for all $\alpha \in \overline{\alpha}$, $\alpha \notin FV(A)$.*

The type system is parameterized over a function $T$ that maps each constant $c \in Const$ to a closed type scheme. For example, for $\lambda_{let,bool}$ the function $T$ should map the constants involving booleans as follows:

$$\begin{array}{rcl} \mathtt{true} & : & \mathtt{bool} \\ \mathtt{false} & : & \mathtt{bool} \\ \mathtt{if} & : & \forall\alpha.\mathtt{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \end{array}$$

## 3.4 Discriminative Sum Types with Constraints

This section explains the technical foundations of our type system based on discriminative sum types, flow-set annotations, and recursive types. Constraints are also needed to describe the propagation of flow-set annotations. Since two different constraint languages are required, we start by abstracting over the constraint system. This yields a general framework for program analysis using constrained types with discriminative sums. We rely on an abstract notion of *constraint system* which is similar to Jones's predicates [19] and Odersky et al.'s notion of constraints [25].

DEFINITION 3. *(Constraint System) A constraint system over a type language is a structure $(\Omega, \vdash)$ where $\Omega$ is a constraints language extending a type language and $\vdash$ is an entailment relation $\vdash \subset \mathcal{P}(\Omega) \times \mathcal{P}(\Omega)$, such that the following holds:*

*(i)* $C \cup D \vdash C$,

*(ii)* *if $C_1 \vdash C_2$, and $C_2 \vdash C_3$, then $C_1 \vdash C_3$, and*

*(iii)* *If $C \vdash D$, then $\phi(C) \vdash \phi(D)$.*

*where $C$ and $D$ are sets of constraints, and $\phi$ is a substitution of type variables.*

Hence, the entailment relation associated with a constraint system must be monotone, transitive, and closed under substitutions. For notational convenience we usually write $C,D$ for the union (conjunction) of two constraint sets $C$ and $D$.

Discriminative sum types are a variation of row types introduced by Wand [35] and Remy [29] originally intended for the purpose of typing records and variants. While row types are usually indexed with record or variant labels, discriminative sum types are indexed with *type constructors* and the component types are the argument types of the constructors. Similar constructions have been used for soft typing systems [7, 18, 38]. For performing the flow analysis required for error reporting, we annotate each type constructor in a discriminative type with a *flow set*.

| | | | | |
|---|---|---|---|---|
| *Row Variables* | $\rho$ | $\in$ | *RowVar* | |
| *Set Variables* | $\upsilon$ | $\in$ | *SetVar* | |
| *Sets of Type Constr's* | $\Theta$ | $\in$ | $\mathcal{P}(TyCon)$ | |
| *Flow Set Ann's* | $u^L$ | $::=$ | $\upsilon^L \mid \ell; u^{L\cup\{\ell\}}$ | $\ell \notin L$ |
| *Types* | $t^\Theta$ | $::=$ | $\rho^\Theta \mid \chi^u\, t_1 \ldots t_n; t^{\Theta\cup\{\chi\}}$ | $\chi \notin \Theta$ |
| *Type Schemes* | $s$ | $::=$ | $\forall\overline{\rho\upsilon}.C \Rightarrow t$ | |
| *Type Assumptions* | $A$ | $::=$ | $\cdot \mid A, x:t$ | $x \notin A$ |

A type $t^\Theta$ is either a row variable $\rho^\Theta$ or it consists of variants indexed by type constructors $\chi$ which are not mentioned in $\Theta$. Each variant consists of a type term $\chi^u\, t_1 \ldots t_n$ where the type constructor is annotated with a flow set, followed by further variants which are restricted so that $\chi$ cannot appear again. The superscript on a row variable *restricts* the types that may be substituted for the variable. We sometimes omit the superscript, when the restriction is obvious from the context.

Each type constructor carries a flow set annotation $u$. A flow set annotation $u$ is a set of labels and a set variable $\upsilon$. Again, a superscript on a flow set annotations,$u^L$, indicates that certain labels cannot appear in the flow set.

A type scheme $\forall\overline{\rho\upsilon}.C \Rightarrow t$ represents the sets of types that may be obtained from $t$ by applying substitutions for the row variables $\overline{\rho}$ and the set variables $\overline{\upsilon}$, restricted by a set of constraints $C$ of a given constraint system $(\Omega,\vdash)$. The following definition of substitution serves to formalize the instantiation relation between a type scheme and its instance types.

DEFINITION 4. *(Substitution) A substitution $\phi$ is a finite mapping from set variables to flow set annotations, and from row variables to types. We extend substitutions to total mappings on constraints, flow set annotations, types, type schemes, and type assumptions in the usual capture-avoiding manner.*

DEFINITION 5. *(Instantiation) A pair of a set of constraints and a type $(C',t')$ is an instance of the type scheme $\forall\overline{\rho\upsilon}.C \Rightarrow t$, written $s \succ t$, if there is a substitution $\phi$ with domain $\{\overline{\rho\upsilon}\}$ so that $(C',t') = (\phi C, \phi t)$.*

The next relation describes *generalizations* of a type $t$ with respect to a type assignment $A$ and constraints $D$.

DEFINITION 6. *(Generalization) A pair of a constraint set and a type scheme, $(C_1,s)$, is a generalization of a type $t$ with respect to a type assignment $A$ and constraints $D$, written $(D,A,t) \vdash_{gen} (C_1,s)$ with $s \equiv \forall\overline{\rho\upsilon}.C_2 \Rightarrow t$, if the following holds*

*(i) for all $\alpha \in \overline{\rho} \cup \overline{\upsilon}$, $\alpha \in FV(t) \setminus (FV(C_1) \cup FV(A))$,*

*(ii) $C_1, C_2 \vdash D$ and $D \vdash C_1, C_2$.*

Similar to the previous variant defined for the Damas-Milner type system, the generalization relation specifies which type variables may be quantified over. In addition, it splits up the given constraint set taking into account that only a part without quantified variables stays outside the type scheme.

The following system of syntax-directed deduction rules defines a general framework for constrained type systems with discriminative sum types. An instance of the framework is determined by a constraint system $(\Omega,\vdash)$ and a function $T$ that defines closed type schemes for all constants.

$$\frac{T(c) \succ (D,t) \quad C \vdash D}{C \mid A \vdash c : t}$$

$$\frac{A(x) \succ (D,t) \quad C \vdash D}{C \mid A \vdash x : t}$$

$$\frac{C \mid A \vdash e_1 : (t_2 \to^u t_1; t^{\to}) \quad C \mid A \vdash e_2 : t_2}{C \mid A \vdash e_1\, e_2 : t_1}$$

$$\frac{C \mid A, x : t_2 \vdash e : t_1}{C \mid A \vdash \lambda x.e : (t_2 \to^u t_1; t^{\to})}$$

$$\frac{D \mid A \vdash e_1 : t_1 \quad (D,A,t_1) \vdash_{gen} (C,s) \quad C \mid A, x : s \vdash e_2 : t_2}{C \mid A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2}$$

The conclusion of a deduction using those rules yields a type judgment $C \mid A \vdash e : t$ meaning that in the context of constraints $C$ and type assumptions $A$, the term $e$ has type $t$.

The rules for function application and for abstraction are different from the usual presentations because they only determine that the component for the function type constructor $\to$ is defined in the type. However, they do **not** rule out the presence of other type constructors, *i.e.*, multivocal types. The `let` rule differs from the one used in HM(X) [25], by having a less restrictive generalization relation. In particular, HM(X) restricts generalization to abstracting only solvable constraints.[1] Since the intended use of the system is prescriptive (that is, a program analysis setting), we can safely restrict ourselves to constraints that are always solvable. The rules for constants and variables additionally require that the constraints resulting from the instantiation of the type scheme are satisfied under the constraints present in the context.

## 3.5 Structural Properties

This section explores basic properties of the type system that are independent of the stated operational semantics. The lemmas developed here will later help proving several subject reduction results. The first lemma establishes that typing is stable under substitutions.

LEMMA 2. *(Type Instantiation) Let $\phi$ be a substitution. If $C \mid A \vdash e : t$, then $\phi(C) \mid \phi(A) \vdash e : \phi(t)$.*

The next two lemmas establish the admissibility of weakening of typing contexts, first for type assignments, then for constraints.

LEMMA 3. *(Weakening of Type Assignments) If $C \mid A \vdash e : t$, and $A, A'$ is a valid type assumption, then $C \mid A, A' \vdash e : t$.*

LEMMA 4. *(Weakening of Constraints) If $C \mid A \vdash e : t$, and $D \vdash C$, then $D \mid A \vdash e : t$.*

[1] Their generalization relation is $(D,A,t) \vdash_{gen} (C_1 \wedge \exists\overline{\rho\upsilon}C_2, s)$ in the notation of Definition 6.

The following Substitution Lemma is essential to show subject reduction for reductions involving substitution. The lemma also has to take into account the generalization relation because type assignments register type schemes, whereas type judgments only assign types to terms.

LEMMA 5. *(Substitution Principle) If $C \mid A, x : s, A' \vdash e : t$, and it also holds that $D_1 \mid A \vdash e' : t'$, $(D_1, A, t') \vdash_{gen} (D_2, s)$, and $C \vdash D_2$, then $C \mid A, A' \vdash e[x := e'] : t$.*

## 3.6 Subject Reduction for $\lambda_{let}$ and $\lambda_{let,bool}$

Subject Reduction holds for our generic type system if reduction preserves typings. For the calculus $\lambda_{let}$ subject reduction can be established independently of the choice of a constraint system.

THEOREM 1. *(Subject Reduction for $\lambda_{let}$) If $C \mid A \vdash e : t$, and $e \longrightarrow e'$, then $C \mid A \vdash e' : t$.*

Assuming the following type bindings for booleans

$$\begin{array}{rcl} \texttt{true} & : & \forall \rho\upsilon.(\texttt{bool}^{\upsilon};\rho^{\{\texttt{bool}\}}) \\ \texttt{false} & : & \forall \rho\upsilon.(\texttt{bool}^{\upsilon};\rho^{\{\texttt{bool}\}}) \\ \texttt{if} & : & \forall \rho\rho'\upsilon.(\texttt{bool}^{\upsilon};\rho^{\{\texttt{bool}\}}) \rightarrow \rho' \rightarrow \rho' \rightarrow \rho' \end{array}$$

the corresponding property also holds for the $\lambda_{let,bool}$-calculus.

THEOREM 2. *(Subject Reduction for $\lambda_{let,bool}$) If $C \mid A \vdash e : t$, and $e \longrightarrow e'$, then $C \mid A \vdash e' : t$.*

## 3.7 Subject Reduction for $\lambda L$ with Simple Flow Constraints

The first constraint system, called *simple flow constraints*, traces class labels using the following constraint language $\Omega^S$:

$$\textit{Simple Flow Constraints} \quad cs \quad ::= \quad \ell \rightsquigarrow t \mid \ell \rightsquigarrow u$$

A constraint $\ell \rightsquigarrow t$ expresses a shallow labeling of $t$ with $\ell$. It means that a certain type $t$ must at least have $\ell$ in every set attached to its top-level type constructor. A constraint $\ell \rightsquigarrow u$ means that $\ell$ is member of the flow set $u$. This is formalized by the two relations $C \vdash \ell \rightsquigarrow u$ (for annotations) and $C \vdash \ell \rightsquigarrow t$ (for types) defined as the smallest relation satisfying the following rules:

$$C \vdash \ell \rightsquigarrow \ell;u \qquad \frac{C \vdash \ell \rightsquigarrow u}{C \vdash \ell \rightsquigarrow \ell';u}\ell \neq \ell'$$

$$\frac{\ell \rightsquigarrow \rho \in C}{C \vdash \ell \rightsquigarrow \rho} \qquad \frac{C \vdash \ell \rightsquigarrow u \quad C \vdash \ell \rightsquigarrow t}{C \vdash \ell \rightsquigarrow \chi^u\, t_1 \ldots t_n; t}$$

Whenever the type contains a row variable, the judgment is turned into a constraint. In contrast, set variables can always be instantiated to satisfy the judgment. This formulation of constraint entailment leaves substitution implicit. In an implementation, we have to add equality/substitution constraints of the form $\upsilon = u$.

The next two lemmas state some properties about the entailment relation for simple flow constraints. They are needed to prove subject reduction for the type system equipped with simple flow constraints as constraint language. We use the short-hand notation $L \rightsquigarrow t$ for the set of constraints $\ell_1 \rightsquigarrow t, \ldots, \ell_n \rightsquigarrow t$ with $L \equiv \{\ell_1, \ldots, \ell_n\}$.

LEMMA 6. *If $C \vdash L_1 \rightsquigarrow t$, and $C \vdash L_2 \rightsquigarrow t$, then $C \vdash L_1 \cup L_2 \rightsquigarrow t$.*

LEMMA 7. *If $C \vdash L \rightsquigarrow (t_1 \rightarrow^u t_2; t)$, then $C \vdash L \rightsquigarrow t_1$, $C \vdash L \rightsquigarrow u$, $C \vdash L \rightsquigarrow t_2$, and $C \vdash L \rightsquigarrow t$.*

$$\frac{C \vdash u \leq u'}{C \vdash \ell;u \leq \ell;u'}$$

$$\frac{\ell_1^-;\ldots;\ell_n^-;\upsilon \leq \ell_1^+;\ldots;\ell_m^+;\upsilon' \in C \quad \ell_i^- \in Lab^- \quad \ell_j^+ \in Lab^+}{C \vdash \ell_1^-;\ldots;\ell_n^-;\upsilon \leq \ell_1^+;\ldots;\ell_m^+;\upsilon'}$$

$$C \vdash \rho \leq \rho \qquad \frac{\rho \leq \rho' \in C}{C \vdash \rho \leq \rho'}$$

$$\frac{\begin{array}{c} p_1 \ldots p_n = polarity(\chi) \quad (\forall 1 \leq i \leq n)\; C \vdash t_i \leq^{p_i} t_i' \\ C \vdash u \leq u' \quad C \vdash t \leq t' \end{array}}{C \vdash \chi^u\, t_1 \ldots t_n; t \leq \chi^{u'}\, t_1' \ldots t_n'; t'}$$

**Figure 3. Entailment of Refined Flow Constraints**

We define the following type bindings for labeling constants. They make use of simple flow constraints to express the occurrence of the appropriate labels in flow set annotations of the involved types.

$$\begin{array}{rcl} [\_]^L & : & \forall \rho\rho'\upsilon.\{L \rightsquigarrow \rho\} \Rightarrow (\rho \rightarrow^{\upsilon} \rho);\rho'^{\{\rightarrow\}} \\ [\_]_{\chi,\ell} & : & \forall \rho\overline{\rho}\rho'\upsilon\upsilon'.(\chi^{\ell;\upsilon}\,\overline{\rho};\rho^{\{\chi\}}) \rightarrow^{\upsilon'} (\chi^{\ell;\upsilon}\,\overline{\rho};\rho^{\{\chi\}});\rho'^{\{\rightarrow\}} \end{array}$$

In the first type scheme, the constraint guarantees that the set of labels $L$ is attached to the top-level node of the type substituted for $\rho$. The second type scheme does not have a constraint, it just installs the (source or sink) label $\ell$ on top of the type constructors $\chi$.

Given the previous development, we may now define a type system for $\lambda L$ that expresses flow information by using the constraint system of *simple flow constraints* and the above type bindings. The following subject reduction result shows that typing is preserved under labeled reductions of $\lambda L$.

THEOREM 3. *(Subject Reduction for $\lambda L$) If $C \mid A \vdash a : t$, and $a \longrightarrow a'$, then $C \mid A \vdash a' : t$.*

## 3.8 Subject Reduction for $\lambda L$ with Refined Flow Constraints

As explained in Section 2.1.1, the flow information gained from typings of the type system using simple flow constraints can be rather imprecise. It shares flow information between all occurrences of expressions with the same type, hence the flow information of these expressions coincide. We improve on this shortcoming by introducing a second, subtyping-based constraint system.

The second constraint system, called *refined flow constraints*, is an extension of the first one. Its constraint language $\Omega^R$ extends $\Omega^S$.

$$\textit{Refined Flow Constraints} \quad cr \quad ::= \quad cs \mid u \leq u' \mid t \leq t'$$

In addition to the constraints from $\Omega^S$, there are two other forms of constraints. The constraint $u \leq u'$ expresses that (i) every positive label that occurs in $u$ also occurs in $u'$ and (ii) every negative label that occurs in $u'$ also occurs in $u$. That is, it is "subset" for positive labels and "superset" for negative labels. The constraint $t \leq t'$ expresses that the flow sets in the annotations of the types $t$ and $t'$ relate by $\leq$. However, the structure of the types $t$ and $t'$ is identical, the "subsetting" happens only on the annotation level.

The entailment relation $\vdash$ for the constraint system $\Omega^R$ is the least relation defined by the rules shown in Figure 3 (again leaving substitution implicit). It guarantees that negative labels cannot travel from left to right, and positive labels cannot travel from right to left. Subtyping on a row variable is either suspended (and turned into a constraint) or immediately satisfiable in the reflexive case. Subtyping on a type constructor demands that subsetting on the annotations holds and that all arguments of the type constructor are in the subtype relation according to the polarity of the type constructor. The polarity of an argument of a type constructor indicates whether the constructor is covariant $\oplus$ or contravariant $\ominus$ in this argument. The polarity of the constructor, $polarity(\chi) \in \{\oplus, \ominus\}^*$, is a sequence of such indicators. Hence, We define $t \leq^{\oplus} t'$ as $t \leq t'$ and $t \leq^{\ominus} t'$ as $t' \leq t$.

Satisfiability of subtyping constraints is transitive.

LEMMA 8. *If $C \vdash t_1 \leq t_2$, and $C \vdash t_2 \leq t_3$, then $C \vdash t_1 \leq t_3$.*

The following lemma enables us to prove subject reduction.

LEMMA 9. *If $C \vdash (t_1 \rightarrow^u t_2; t) \leq (t_1' \rightarrow^{u'} t_2'; t')$, then $C \vdash t_1' \leq t_1$, $C \vdash t_2 \leq t_2'$, $C \vdash u \leq u'$, and $C \vdash t \leq t'$.*

Using the subtyping constraints we can refine the type bindings for the labeling constants.

$$\begin{array}{rcl} [\_]^L & : & \forall \rho\rho'\rho''\upsilon.\{L \rightsquigarrow \rho, \rho \leq \rho'\} \Rightarrow (\rho \rightarrow^{\upsilon} \rho'); \rho''^{\{\rightarrow\}} \\ [\_]_{\chi,\ell} & : & \forall \rho\overline{\rho}\rho'\upsilon\upsilon'.(\chi^{\ell;\upsilon}\ \overline{\rho}; \rho^{\{\chi\}}) \rightarrow^{\upsilon'} (\chi^{\ell;\upsilon}\ \overline{\rho}; \rho^{\{\chi\}}); \rho' \end{array}$$

The type for labeling constants for positive and negative labels is exactly the same as before. However the constraint given in the type schemes of the labeling with sets of flow-class labels has changed. Instead of simply using the same type variable for argument and result, the argument now needs to have a subtype of the result.

For proving subject reduction in this calculus, we must be able to weaken a type assumption to a subtype. This property only holds for properly labeled terms.

LEMMA 10. *If $e \hookrightarrow a$, $C \mid A, x : t', A' \vdash a : t$, and $C \vdash t'' \leq t'$, then $C \mid A, x : t'', A' \vdash a : t$.*

The second type system for $\lambda L$ with flow information uses the second constraint system of *refined flow constraints* and the new type bindings for the labeling constants.

Subject reduction also holds for labeling reductions of $\lambda L$ in this setting provided we are dealing with a sensibly labeled term obtained by the labeling relation $\hookrightarrow$. The labeling relation guarantees that we can always coerce to supertypes.

THEOREM 4. *(Subject Reduction) If $e \longrightarrow e'$, $e \hookrightarrow a$, and $C \mid A \vdash a : t$, then there exists an $a'$ such that $C \mid A \vdash a' : t$, and $e' \hookrightarrow a'$.*

## 3.9 Conservativity over Mini-ML

In this section, we investigate the question how derivability in our system relates to derivability in the Damas-Milner system, and vice versa. Clearly, our system should be conservative in the sense that any expression which has an ML type is also typeable in our system. Since this property is trivial (recall that every expression is typeable), we show that an ML typeable expression has a type derivation without multivocal types in our system. Formally, the judgment $V \vdash_U t$ (defined in Figure 4) characterizes such univocal types.

Basically, univocal types have only one defined type constructor in the discriminative sum. Further, we must guarantee that each row variable $\rho$ appearing besides a single type constructor does not interfere with other free variables occurring elsewhere. Hence, the judgment $V \vdash_U t$ states that $t$ is a univocal type assuming there is the set of fresh row variables $V$ available.

$$V \vdash_U \rho \quad \rho \notin V$$

$$\frac{V_1 \vdash_U t_1 \quad \ldots \quad V_n \vdash_U t_n}{V_1 \cup \ldots \cup V_n \cup \{\rho\} \vdash_U \chi^u\ t_1 \ldots t_n; \rho^{\{\chi\}}}\ \rho \notin FV(t_1) \cup \ldots \cup FV(t_n)$$

**Figure 4. Univocal types**

Using the judgment for univocal types we specialize the rules of the generic type system with multivocal type to be able to recognize those deductions where only univocal types occur.

$$\frac{T(c) \succ (D, t) \quad C \vdash_U D \quad V \vdash_U t \quad V \cap (FV(C) \cup FV(A)) = \emptyset}{C \mid A \vdash_U c : t}$$

$$\frac{A(x) \succ (D, t) \quad C \vdash_U D \quad V \vdash_U t \quad V \cap (FV(C) \cup FV(A)) = \emptyset}{C \mid A \vdash_U x : t}$$

$$\frac{C \mid A \vdash_U e_1 : (t_2 \rightarrow^u t_1; \rho^{\rightarrow}) \quad C \mid A \vdash_U e_2 : t_2}{C \mid A \vdash_U e_1\ e_2 : t_1}$$

$$\frac{C \mid A, x : t_2 \vdash_U e : t_1 \quad \rho \notin (FV(C) \cup FV(A))}{C \mid A \vdash_U \lambda x.e : (t_2 \rightarrow^u t_1; \rho^{\rightarrow})}$$

$$\frac{D \mid A \vdash_U e_1 : t_1 \quad (D, A, t_1) \vdash_{gen} (C, s) \quad C \mid A, x : s \vdash_U e_2 : t_2}{C \mid A \vdash_U \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : t_2}$$

By construction, each term that is typeable only with univocal types is also typeable under the less restrictive system with multivocal types. This property is expressed by the following lemma.

LEMMA 11. *If $C \mid A \vdash_U e : t$ then $C \mid A \vdash e : t$.*

Completeness is captured as follows: if there is a Damas-Milner-typing for a term $e$, then there is also a typing with univocal types of a corresponding labeled term $a$.

THEOREM 5. *(Completeness) If $A \vdash_{DM} e : t$, and $e \hookrightarrow a$, then there exists some $C, A', t'$ such that $C \mid A' \vdash_U a : t'$.*

Finally, the following Soundness Theorem shows that the systems $\vdash_U$ allows us to identify those terms that are also typeable under the Damas-Milner type system.

THEOREM 6. *(Soundness) If $e \hookrightarrow a$ and $C \mid A \vdash_U a : t$, then there exists some $A', t'$ such that $A' \vdash_{DM} e : t'$.*

Taken together these results imply that our procedure for identifying type errors by looking for multivocal types is correct.

## 4 Extensions

A number of extensions are required to make the system amenable to a full-blown programming language. Sum types, product types, and recursive data types are easy to add. For example, to analyze programs with lists, we only need to find a sound type scheme for the list constructors. These type schemes look daunting but they are straightforward to generate automatically using the $V \vdash_U t$ judgment from Section 3.9:

$$\begin{array}{rcl} \mathtt{nil} & : & \forall \rho_0\rho_1\upsilon_0.\mathtt{list}^{\upsilon_0}\ \rho_0; \rho_1 \\ \mathtt{cons} & : & \forall \rho_0\rho_1\rho_2\rho_3\rho_4\rho_5\rho_6\upsilon_0\upsilon_1\upsilon_2\upsilon_3. \\ & & (\rho_1 \rightarrow^{l_1^+;\upsilon_0} \\ & & \quad ((\mathtt{list}^{l_3^-;\upsilon_1}\ \rho_2; \rho_5) \rightarrow^{l_2^+;\upsilon_2} (\mathtt{list}^{l_4^+;\upsilon_3}\ \rho_3; \rho_6); \rho_4); \rho_0) \end{array}$$

The generation principle of the extended type schemes for constructors can be used to generate extended type schemes from arbitrary ML type schemes (which is necessary in the presence of modules, libraries, etc). First, fresh row variables are added according to the judgment $V \vdash_U t$. Then, each type constructor in the ML type is assigned a fresh label (possibly reflecting the name of the function and its defining module) with polarity defined by its occurrence in the type.

# 5 Implementation

At present, there is a prototype implementation of the type inference engine for refined flow constraints (with annotation subtyping) but without recursive types. The prototype relies on a naive implementation of row unification and is very inefficient.

However, there are a number of efficiently implemented type inference algorithms available that should be fairly straightforward to adapt to our system. Good starting points would be the system of Henglein and Rehof [18], the Wallace framework of Pottier [28], and the soft typing implementation of Wright [38]. The Wallace framework is probably the most advanced of these, it provides conditional constraints, rows, and subtyping. However, it only supports polymorphic lets at the top-level.

# 6 Related Work

## 6.1 Alternative Type Inference Algorithms

Lee and Yi [20] compare the error reporting capabilities of Milner's bottom-up algorithm $\mathcal{W}$, which performs unification only at function applications, and a folklore variation called $\mathcal{M}$, which passes the expected type in a top-down manner and hence requires unification at lambda abstractions and variables. They refute the (previous) folklore that $\mathcal{M}$ is better at error reporting than $\mathcal{W}$ by showing that the error behavior of both is incomparable.

McAdam [21] tries to avoid the bias of a particular traversal order of the syntax tree by unifying at each expression the substitutions collected in traversing the subexpression.

Mitchell's type inference algorithms [23] return an typing judgment with a type environment and the computed type. The judgment is computed bottom-up by unifying the types (as usual) and the type environments.

Chitil [8] identifies the lack of compositionality of the usual inference algorithms as one important reason, why type error messages are hard to comprehend. He proposes a type inference algorithm that is essentially similar to Mitchell's but which also treats `let`-bound variables in a compositional way and computes principal typings. Based on the results of the inference algorithm Chitil defines an explanation graph and provides the means to navigate it. A similar algorithm has also be proposed by Damas as algorithm $\mathcal{T}$ [10]. Principal typings have been investigated separately, Wells [37] gives a good overview of their use in theory and practice.

## 6.2 Type Errors

Wand [34] instruments unification in a type inference algorithm for an implicitly typed lambda calculus. Each substitution created during a unification is annotated with the program point (a function application or some other elimination construct) that caused the unification. When a clash between two different type constructors is detected, the list of program points that led to each constructor is printed along with the location where the clash occurred.

Walz and Johnson [33] also base their approach on inspecting the unification procedure. By looking for maximum flows in graphs representing the unification problems they locate program points for each clash occurring during unification. With their approach the order how unification proceeds affects the final result of program points that are reported.

Beaven and Stansifer [6] provide detailed explanations of the types of expressions in a (partially instantiated) typing derivation. In case, the typing derivation is incomplete due to a unification failure, they propose to investigate the two types that led to the failure.

Duggan and Bent [12] criticize earlier attempts to locate the source of type errors. They propose to provide an explanation for the unification steps taken by the type inference algorithm prior to the error. The form of this explanation is a graph of subexpressions with their associated effect on the substitution constructed during type inference. They describe an efficient unification algorithm extended by gathering the information required for generating the explanation.

Heeren and others [15] describe a type inference engine for a Haskell subset that is geared towards generating good error messages. Instead of committing to a particular strategy of solving the equality constraints arising, they create a constraint graph during an initial traversal of the syntax tree. In varying the way that this graph is traversed and simplified, they can simulate a number of type inference algorithms, including $\mathcal{W}$ and $\mathcal{M}$. They propose a number of heuristics that generate error messages from the constraint graph. We believe that our type system provides a formal justification for their constraint graph and their heuristics may be adaptable to extract error messages from our typings.

Haack and Wells [14] define a slicing-based approach to finding the source of a type error. They separate type inference in two phases, generation of equality constraints between types and constraint solution. Each equality is annotated with the program point the caused the generation of the constraint. The solver propagates the annotation when decomposing equalities. For a program with a type error, the generated constraint set is not solvable. The authors show that each minimal unsolvable subset determines a minimal program slice that exhibits the type error. The problem with their approach is that the notion of a minimal unsolvable subset is not unique and they do not give an algorithm that enumerates all such minimal unsolvable subsets. In contrast, our approach yields a principal description of the type errors via the type inference algorithm for multivocal types. Our slices are not determined by annotations on constraints, but rather by flow labels inferred by a flow analysis.

## 6.3 Row Types

Rémy and Wand [36, 30, 31] introduce row types, the heart of our approach, for modeling record and variant types. They give sound and complete type inference algorithms for their respective systems.

Pottier [28] considers type inference for constrained type systems with subtyping. His constraint logic includes conditional constraints and rows, which clearly subsumes the facilities required for inferring multivocal types. The system is phrased as an instance of the HM(X) framework [25], which provides the actual type inference algorithm for free.

Skalka and Smith [32] have specialized row types to set types, which are record types with "just the labels". They employ conditional constraints to infer precise types for the usual operations on sets. Our application does not require this precision.

## 6.4 Soft Typing

Soft typing, or dynamic typing, is a type theory geared at establishing typing properties of programs in dynamically typed languages like Scheme. In such a language, each value is tagged with its type and each operation is guarded by a dynamic check that stops execution if an argument does not have the required type. The use of soft typing is twofold. On the one hand, the inferred types provide documentation to the programmer. On the other hand, an implementation can make use of inferred types to omit dynamic type checks and potentially optimize data representations by omitting type tags.

Our type system with multivocal types is similar to a soft typing system. However, there is no attempt to reconcile typings by inserting dynamic check operations. Instead, we aim at finding the locations that would cause the insertion of a dynamic check. Hence, we view our work as complementary to the work on soft typing in that it could provide guidance to the users of such a system.

The techniques used to infer soft typings are similar to the ones employed in this work. However, since one goal of soft typing is the ability to omit checks, such systems also try to infer information about the *absence* of type constructors where our system is only concerned about their presence.

Cartwright and Fagan [7] is the pioneering work in that area. Their type system includes discriminative unions, recursive types, and parametric polymorphism. The system and their algorithm are inspired by earlier work on subtyping [24] and record typing [29].

Aiken et al. [3] propose an extended soft typing system that drops the restriction on unions and adds intersection types and conditional types. Their system provides a simpler formalism with more accurate typings.

Henglein [17] defines a dynamic typing discipline which precisely formalizes the role of the dynamic tagging and tag-check operations. He defines an equational theory for these operations and defines a weaker rewriting theory that gives rise to optimally placed tagging and tag-check operations in *minimal safe completions*).

Henglein and Rehof [18] give a framework that enables a translation of Scheme programs to ML programs. Their work extends soft typing by not just considering the elimination of tag checks but also the introduction of tags. It also extends it towards modularity in that program fragments can be type checked and translated separately. Furthermore, it guarantees minimal safe completions in the presence of polymorphism. A main difference of our approach is that we need not worry about the placement of tag operations because we are not interested in running the code as long as some types are multivocal. However, it might be revealing to show where the tag operation *would be put* to narrow down the cause of the error.

Wright's system [38] is very similar to our multivocal types. It also extends Hindley/Milner-style polymorphism with recursive types and discriminative unions (called *tidy types*). Type constructors carry a flag that indicates presence or absence of the constructor. In contrast, we are only tracking the presence of constructors but our set annotations track the sources and sinks of computed values.

Flanagan and Felleisen [13] use set-based analysis to discover programming errors in a dynamically typed language (Scheme). The cited work provides a framework for simplifying constraints for set-based analysis to the point where the analysis can be performed component-wise. The goals of that work are similar, but the tools are different.

## 6.5 Flow Analysis

From the vast literature on flow analysis, the following papers exhibit the correspondence between flow analysis (using abstract interpretation or constraints) and annotated type systems. This correspondence was noted independently by Palsberg and O'Keefe [26] and Heintze [16]. The first work shows the equivalence of a type system with recursive types, subtyping, and $\top$ and $\perp$ types with a constraint-based flow analysis. Heintze's work relates abstract interpretation-based flow analysis with annotated type systems. It characterizes 0CFA in terms of an annotated type system with simple types and subtyping and exhibits the difference to simple typing (without subtyping) which performs an equational flow analysis. Similar observations can be made for polyvariant flow analysis and intersection types [27].

# 7 Conclusion

We have designed a type system based on discriminative sum types with recursive types and annotation subtyping. This system provides the essential information for discovering the sources of type errors in programs with ML-style type inference. The paper establishes the theoretical framework for the system and reports initial experiments with a prototype implementation. These experiments are encouraging. In fact, all examples have been checked with the implementation. Further work is needed to scale the implementation to a full language and to address advanced features like overloading, existential types, and rank-*n* polymorphism.

### *Acknowledgments*

Thanks are due to Robert Cartwright, Olaf Chitil, Olivier Danvy, Fritz Henglein, John Matthews, and Didier Remy for discussion and suggestions on the work reported in this paper.

# 8 References

[1] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In Kent Dybvig, editor, *Proceedings of the 1996 International Conference on Functional Programming*, pages 83–91, Philadelphia, PA, May 1996. ACM Press, New York.

[2] ACM. *Proc. of the 13th Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, 1986.

[3] Alexander Aiken, Edward L. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Proceedings of the 1994 ACM SIGPLAN Symposium on Principles of Programming Languages*, Portland, OR, January 1994. ACM Press.

[4] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[5] H. P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.

[6] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(4):17–30, March 1993.

[7] Robert Cartwright and Mike Fagan. Soft typing. In *Proc. Conference on Programming Language Design and Implementation '91*, pages 278–292, Toronto, Canada, June 1991. ACM.

[8] Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In Xavier Leroy, editor, *Proceedings of the 2001 International Conference on Functional Programming*, Florence, Italy, September 2001. ACM Press, New York.

[9] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 13–27, 1986.

[10] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, Computer Science Department, Edinburgh University, 1985. report CST-33-85.

[11] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 1982 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[12] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, July 1996.

[13] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, March 1999.

[14] Christian Haack and Joe Wells. Type error slicing in implicitly typed, higher-order languages. In *Proc. 12th European Symposium on Programming*, Lecture Notes in Computer Science, Warsaw, Poland, April 2003. Springer-Verlag.

[15] Bastiaan Heeren, Johan Jeuring, Doaitse Swierstra, and Pablo Azero Alcocer. Improving type-error messages in functional languages. Technical Report UU-CS-2002-009, Institute of Information and Computing Science, University Utrecht, Netherlands, February 2002. Technical Report.

[16] Nevin Heintze. Control-flow analysis and type systems. In Alan Mycroft, editor, *Proceedings of the 1995 International Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 189–206, Glasgow, Scotland, September 1995. Springer-Verlag.

[17] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.

[18] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In Simon Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, La Jolla, CA, June 1995. ACM Press, New York.

[19] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, UK, 1994.

[20] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, 1998.

[21] Bruce J. McAdam. On the Unification of Substitutions in Type Inference. In Kevin Hammond, Anthony J.T. Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98), London, UK*, number 1595 in Lecture Notes in Computer Science, pages 139–154. Springer-Verlag, September 1998.

[22] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[23] John Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[24] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, July 1991.

[25] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[26] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 367–378, San Francisco, CA, January 1995. ACM Press.

[27] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In Luca Cardelli, editor, *Proc. 25th Annual ACM Symposium on Principles of Programming Languages*, pages 197–208, San Diego, CA, USA, January 1998. ACM Press.

[28] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[29] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, pages 77–88, Austin, Texas, January 1989. ACM Press.

[30] Didier Rémy. Projective ML. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, San Francisco, California, USA, June 1992.

[31] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.

[32] Christian Skalka and Scott Smith. Set types and applications. *Electronic Notes in Theoretical Computer Science*, 75, 2003.

[33] Janet A. Walz and Gregory F. Johnson. A maximium flow approach to anomaly isolation in unification-based incremental type inference. In POPL1986 [2], pages 44–57.

[34] Mitchell Wand. Finding the source of type errors. In POPL1986 [2], pages 38–43.

[35] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the 1989 IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989. IEEE Computer Society Press. To appear in *Information and Computation*.

[36] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, July 1991.

[37] Joseph B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, number 2380 in Lecture Notes in Computer Science, pages 913–925. Springer-Verlag, 2002.

[38] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.