Program Analysis Using Mixed Term and Set Constraints

Manuel Fähndrich* and Alexander Aiken*

EECS Department
University of California, Berkeley**

Abstract. There is a tension in program analysis between precision and efficiency. In constraint-based program analysis, at one extreme methods based on unification of equality constraints over terms are very fast but often imprecise. At the other extreme, methods based on the resolution of inclusion constraints over set expressions are quite precise, but are often inefficient in practice. We describe a parameterized framework for constraint-based program analyses that allows the analysis designer to embed terms and set expressions within each other. Constraints over these mixed expressions are partially between equality and inclusion, which enables an entire spectrum of program analyses with varying degrees of precision and efficiency to be expressed. We also show that there are interesting analyses that take advantage of this mixture. In particular, we report on the design and implementation of an uncaught exception analysis for core ML. Our results show that the analysis approaches the efficiency of algorithm $\mathcal{W}$.

1 Introduction

The Hindley-Milner polymorphic type inference system [Mil78] is the classical example of a constraint-based program analysis. It uses *equality constraints* over a term algebra to infer types for functional programming languages such as ML [MTH90]. This system has inspired many other analyses based on equality constraints (e.g. [Hen92, Ste96]). Such systems are appealing because they yield concise results and because the equality constraints can be solved using unification in nearly linear time (in the monomorphic case). The disadvantage of equality constraints is that they cannot model the *direction* of value flow within a program. Information always flows in all directions, causing a loss of precision.

Another class of program analyses is based on *set-inclusion constraints* [Hei92, AW92, AW93]. Because inclusion constraints can model the direction of value flow within a program quite accurately, inclusion constraints yield more precise results than equality-based systems. Examples of such analyses are [Shi88, Hei94, AWL94, FFK+96, MW97]. The disadvantage of inclusion constraints is that they are more expensive to solve than equality constraints. The best known algorithm for solving the simplest inclusion constraints is dynamic transitive closure which requires cubic time, and for more expressive constraints solvability becomes at least EXPTIME-hard [AKVW93, MH97].

To make inclusion constraint-based analyses practical, a lot of effort is spent on tuning the representation and manipulation of constraints. Most importantly, inclusion constraints must be simplified to obtain more concise representations of solutions [Pot96, FA96, TS96, FF97]. While experimenting with a type-based program analysis system entirely based on inclusion constraints over the past two years, we have noticed that constraints describing the inferred types have many uninteresting solutions. Simplification does not help with this problem, because by definition constraint simplification preserves the set of solutions. The unneeded solutions contribute directly to the size of and the expense of manipulating inferred types.

To describe precisely interesting solutions of our analyses, we have invented a new kind of constraint which lies in between equality and inclusion. This technique opens up a new avenue to making certain program analyses based on inclusion constraints practical. By making explicit where in an analysis the generality of sets is needed, and where structural constraints are sufficient, solutions can be described more concisely and computed more efficiently.

In this paper we describe a parameterized constraint formalism that combines inclusion constraints over terms with inclusion constraints over sets. Before proceeding, we fix some terminology. We view the abstract

* Supported in part by NSF Young Investigator Award CCR-9457812 and NSF Grant CCR-9416973

** Authors' address: EECS Department, University of California, Berkeley, 387 Soda Hall #1776, Berkeley, CA 94720-1776 Email: {manuel,aiken}@cs.berkeley.edu

properties computed in program analyses as types. Constraints in our formalism are between *type expressions*, and solutions of the constraints are *types*. To a first approximation, types can be thought of as sets of values. We refer to Hindley-Milner style types as *term types* and to types based on set expressions [AW93] as *set types*. The key property of term types is that they have unique head constructors.

Our formalism covers an entire spectrum of program analyses with varying degrees of precision and efficiency. On one extreme, the formalism can express Hindley-Milner type inference, and on the other extreme, it can express complete inclusion-based analyses. Most importantly, we can express analyses in between these extremes, in particular analyses that consist mostly of term types but also include set types where precision is needed.

The formalism is based on a 2-sorted algebra of type expressions (Section 3). Inclusion constraints between these types are solved using unification and rewrite techniques (Section 4). At the Hindley-Milner end of the spectrum, the implementation of the constraint solving process essentially yields algorithm $\mathcal{W}$ [Mil78]. (Note that the system described in [AW93] can also express Hindley-Milner type inference using only inclusion constraints, but the inference algorithm still requires cubic time in this case.) To make our ideas concrete, we instantiate our framework to a particular analysis near the Hindley-Milner end of the spectrum, namely uncaught exception inference for a subset of ML (Sections 2.2 and 5) and present preliminary results from an implementation showing that the analysis approaches the efficiency of algorithm $\mathcal{W}$. We start our exposition in Section 2 with a description of the problem that motivated the formalism described in this paper, and our running example analysis: Type and exception inference for a subset of ML.

2 Motivation

This section motivates our framework with an example application: Uncaught exception inference for a subset of the ML language. Exception inference is an interesting problem for our formalism because we can express it as a minimal refinement of standard Hindley-Milner type inference, and it makes essential use of set types. Furthermore, exception inference is an interesting problem in its own right, because in practice large ML programs can unexpectedly terminate with uncaught exceptions.

2.1 The Problem

We begin by illustrating the problem of types that are more general than needed. In ML, the type of an exception value v is simply `exn`—no indication is given of the possible exception constructors of v. Consider a refinement of the ML type system that models exception types with an explicit annotation of the set of exception constructors. For example, we model the type of the exception constructor `Subscript` as `exn(Subscript)`. A possible inference rule for `if` expressions based on inclusion constraints is

$$\frac{\begin{array}{l} A \vdash p : \text{bool} \\ A \vdash e_1 : \tau_1 \\ A \vdash e_2 : \tau_2 \\ \tau_1 \subseteq \alpha \qquad \alpha \text{ fresh} \\ \tau_2 \subseteq \alpha \end{array}}{A \vdash \texttt{if } p \texttt{ then } e_1 \texttt{ else } e_2 : \alpha} \quad [\text{IF}]$$

The rule says that the result type must contain the types of both branches. The conditional expression

```
if p then Subscript else x
```

returns either the exception value `Subscript` (exceptions are first-class), or the value of the program variable `x`. Assuming `x` has type β, applying the inference rule to this expression gives the type α along with two lower bounds, written

$$\alpha \texttt{ where } \quad \texttt{exn(Subscript)} \subseteq \alpha \ \ \wedge \ \ \beta \subseteq \alpha$$

There are many solutions for α and β satisfying these constraints. One possible solution is

$$\begin{array}{l} \beta \mapsto \texttt{int} \\ \alpha \mapsto \texttt{exn(Subscript)} \cup \texttt{int} \end{array}$$

For many programming languages (and in particular for ML), this solution is uninteresting, because the union of an integer and an exception cannot be used anywhere. We are really only interested in solutions where the type of the `else` branch is also an exception. However, we cannot simply require both branches to have the same type as in a standard ML type system, because the `else` branch may contribute an exception other than `Subscript`. For example, if the `else` branch can return `exn(Match)`, we would like to infer that the entire `if` can return `exn(Subscript)` or `exn(Match)`. Thus we have two conflicting goals: On one hand we need the generality of inclusion constraints to allow different exception constructors in the branches of the conditional, and on the other hand we do not want the full generality of inclusion constraints, since they admit many uninteresting solutions. In our example, the interesting solutions all have the form

$$\begin{aligned}\beta &= \mathtt{exn}(\gamma)\\ \alpha &= \mathtt{exn}(\mathtt{Subscript} \cup \gamma)\end{aligned}$$

which clarifies that the `if` expression and both branches return exceptions and that the set of exception constructors of the result includes the `Subscript` exception and any exceptions contributed by the `else` branch. In summary, the example illustrates two points:

- For particular applications, inclusion constraints may admit more solutions than required.
- Set types are needed to express sets of values with more than one head constructor (e.g. $\mathtt{Subscript} \cup \gamma$).

2.2 Sample Application

This subsection sets the stage for our framework by proposing a type and exception inference system for a subset of ML. Here we describe only the type language and some examples. The type rules and an implementation are discussed in Section 5.

The standard ML type system gives no information about the set of exceptions that an expression may raise. Knowing only the types, a programmer must assume that each expression e has the worst possible effect: Every imaginable exception may be raised during evaluation of e. The exception inference we describe here gives the programmer more precise information about possible exceptions. We present our analysis for Mini-ML but discuss an implementation for core SML in Section 5.

As an aside, note that for first-order languages, exceptions can be inferred separately from types. In ML however, functions and exceptions are first-class values, and as a result, exception inference cannot be separated from type inference.

The syntax of Mini-ML is a typed lambda calculus with exception constants and `raise` and `handle` expressions.

$$\begin{aligned}e &::= x \mid \mathtt{fn}\ x\ \mathtt{=>}\ e \mid e_1\ e_2 \mid e\ \mathtt{handle}\ p_i\ \mathtt{=>}\ e_i \mid \mathtt{raise}\ e\\ p &::= c \mid c(x)\end{aligned}$$

Handle expressions use pattern matching to provide specialized handlers for different exceptions. The set of constants and primitives of the language are accessed through identifiers in an initial environment.

Every exception in ML has the type `exn`. In order to distinguish among different exception constructors, we refine this type to $\mathtt{exn}(\sigma)$, where σ is a set type capturing the set of exception constructors. Furthermore, we need to refine the type of functions to include information about the possible exceptions raised during an application. Function types are written $\tau_1 \xrightarrow{\sigma} \tau_2$, where τ_1 describes the domain of the function, τ_2 the range of the function, and σ the possible exceptions raised by the function (notation borrowed from effect systems, see Section 5). The resulting type language has two sorts, set types and term types, given by the following grammar

$$\begin{aligned}\tau &::= \alpha \mid B \mid \tau \xrightarrow{\sigma} \tau \mid \mathtt{exn}(\sigma)\\ \sigma &::= \epsilon \mid c \mid c(\tau) \mid \sigma \cap \sigma \mid \sigma \cup \sigma \mid \neg\{c\}\end{aligned}$$

We use τ for term types and σ for set types. Type variables are written α or ϵ, depending on the sort. The set B denotes a set of base types. We use $c \in \mathit{ExnCons}$ for exception constructors. Note that exceptions may be constants or carry a value. An exception c carrying a value of type τ has type $\mathtt{exn}(c(\tau))$. Set types can furthermore be formed by intersection, union, and complement. The type $\neg\{c\}$ is the set of all values except values obtained by applying constructor c. Because exceptions can carry values, the two sorts of types are mutually recursive. Note that the term types used here are ML types with embedded set types.

A few examples illustrate the refined types. First consider the primitive `raise` in ML, which is used to raise an exception. Its ML type is $\texttt{raise} : exn \to \alpha$. Using our refined types, the type becomes $\texttt{raise} : \texttt{exn}(\epsilon) \xrightarrow{\epsilon} \alpha$, capturing the fact that applying `raise` to an exception of type $\texttt{exn}(\epsilon)$ causes the observable effect ϵ.

Consider a function `catchFail` that calls a function argument, and if the `Fail` exception is raised, returns the default value `d`.

```
exception Fail
fun catchFail f d = f () handle Fail => d
```

We assign the type

$$\texttt{catchFail} : (\texttt{unit} \xrightarrow{\epsilon} \alpha) \xrightarrow{0} \alpha \xrightarrow{\epsilon \cap \neg\{\texttt{Fail}\}} \alpha$$

to this function. The type illustrates the dependencies between the exceptions carried by the function argument `f` and the exceptions of `catchFail`. Given a function $\texttt{f} : \texttt{unit} \xrightarrow{\epsilon} \alpha$ which may raise an exception from the set ϵ, we know that the expression `f()` has type α and effect ϵ. The `handle` expression prevents the `Fail` exception from escaping the body of `catchFail`. As a result, we know that evaluating `catchFail` can result in any exceptions raised by the argument function, except `Fail` (written $\epsilon \cap \neg\{\texttt{Fail}\}$). Set expressions are crucial for describing such types.

We are aware of two earlier approaches to uncaught exception detection for ML. In [GS94], Guzmán and Suárez describe an extended type system for ML similar to, but less powerful than, the one presented here. They do not treat exceptions as first class values, and they ignore value-carrying exceptions. In [Yi94], Yi describes a collecting interpretation for estimating uncaught exceptions in ML. His analysis is presented as an abstract interpretation [CC77] and is much finer grained than [GS94] or the system described here, but is also slow in practice.

3 Types and Domains

Section 2.2 outlined a mixed type language for expressing ML types with refined exception information. This is a particular example of a class of analyses that is expressible in our framework. This section presents the general case: We introduce the parameterized type language over which inclusion constraints are solved, show the relationship between types and appropriate semantic domains, and define what constitutes a solution to a system of inclusion constraints over types. Section 4 describes how to compute the solutions of constraints.

3.1 Type Language

We introduce two sorts of types, **u**-types and **s**-types, which are similar to term types and set types, except that **u**-types and **s**-types can be embedded within one another. The type language and resolution rules for inclusion constraints between **u**-types or **s**-types are parameterized by a set of constructor signatures Σ. Let $S = \{\mathbf{u}, \mathbf{s}\}$, then each signature is of the form:

$$c : \iota_1 \ldots \iota_n \to S$$

and each ι is one of $\{\mathbf{u}, \overline{\mathbf{u}}, \mathbf{s}, \overline{\mathbf{s}}\}$. The overlined sorts mark contravariant arguments of c, the rest are covariant arguments. Let $\mathcal{V}$ be an S-sorted set of type variables, i.e. $\mathcal{V} = (\mathcal{V}_\mathbf{u}, \mathcal{V}_\mathbf{s})$. We use greek lowercase letters $\alpha, \beta, \epsilon, \ldots$ to denote type variables. Let Σ^+ be the extension of Σ with the signatures

$$\begin{aligned} \cup &: \mathbf{s}\ \mathbf{s} \to \mathbf{s} \\ \cap &: \mathbf{s}\ \mathbf{s} \to \mathbf{s} \\ \neg\{c_1, \ldots, c_n\} &: \mathbf{s} \qquad \text{for any set of } \mathbf{s}\text{-constructors } c_i \in \Sigma \\ 0 &: \mathbf{s} \\ 1 &: \mathbf{s} \end{aligned}$$

for the set-operations *union*, *intersection*, *complement* of constructors, and constants 0 and 1 for the least **s**-type and the greatest **s**-type respectively. The language of *type expressions* is formed by the sorted term algebra $\mathbf{T}_{\Sigma^+}(\mathcal{V})$. We use letters $T_1, T_2, \ldots$ to refer to **u**-types and **s**-types.

To illustrate the type languages that can be formed, we give three example signature sets Σ. Figure 1 gives the signatures of type constructors for ordinary Hindley-Milner types. All types are $\mathbf{u}$-types in this case and the set of type constructors includes, for example, nullary constructors such as `int` and unary constructors such as $\texttt{list} : \mathbf{u} \to \mathbf{u}$. Figure 2 shows the signatures of type constructors corresponding to the set types of [AW93]; all types are $\mathbf{s}$-types. Finally, Figure 3 contains the signatures for the type language given in Section 2.2 for our ML exception inference. There are constant type constructors of sort $\mathbf{u}$ for a set of non-parameterized base types B, constructors of sort $\mathbf{s}$ for all exception constructors, some with argument types of sort $\mathbf{u}$. The type constructor `exn` simply embeds exception $\mathbf{s}$-types as $\mathbf{u}$-types. Finally the function constructor $\cdot \stackrel{\cdot}{\longrightarrow} \cdot$ contains a contravariant $\overline{\mathbf{u}}$ field for the domain, a covariant $\mathbf{u}$ field for the range, and a covariant $\mathbf{s}$ field for the exceptions that the function may raise.

$$
\begin{array}{ll}
c : \mathbf{u}_1 \ldots \mathbf{u}_n \to \mathbf{u} & \text{(for all } n\text{-ary ML type constructors)} \\
\cdot \longrightarrow \cdot : \overline{\mathbf{u}}\,\mathbf{u} \to \mathbf{u} & \text{(Function type)}
\end{array}
$$

Fig. 1. Signature Σ for Hindley-Milner types.

$$
\begin{array}{ll}
c : \mathbf{s}_1 \ldots \mathbf{s}_n \to \mathbf{s} & \text{(for all } n\text{-ary data constructors)} \\
\cdot \longrightarrow \cdot : \overline{\mathbf{s}}\,\mathbf{s} \to \mathbf{s} & \text{(Function type)}
\end{array}
$$

Fig. 2. Signature Σ for set types [AW93].

$$
\begin{array}{ll}
c : \mathbf{u} & \text{(for all } c \in B \text{ basetypes)} \\
c : \mathbf{s} & \text{(for all constant exception constructors } c\text{)} \\
c : \mathbf{u} \to \mathbf{s} & \text{(for all exception constructors } c \text{ with arguments)} \\
\texttt{exn} : \mathbf{s} \to \mathbf{u} & \\
\cdot \stackrel{\cdot}{\longrightarrow} \cdot : \overline{\mathbf{u}}\,\mathbf{s}\,\mathbf{u} \to \mathbf{u} & \text{(Function type)}
\end{array}
$$

Fig. 3. Signature Σ for exception inference.

For completeness, we conclude this section with two technical comments. First, type constructors with *non-variant* (neither co- nor contravariant) fields cannot be modeled directly in our formalism. An example of such a type constructor is the ML datatype

```
datatype 'a identity = Id of ('a -> 'a)
```

However, we can represent this type constructor by doubling the non-variant field, one copy being contravariant, the other covariant.

$$\texttt{identity} : \overline{\mathbf{u}}\,\mathbf{u} \to \mathbf{u}$$

Any ML type T `identity` can then be represented as $\texttt{identity}(T, T)$ with the desired non-variance in T. The second comment concerns strictness of constructors. We allow each constructor to be strict or non-strict in any of its covariant fields (contravariant fields are always non-strict). To keep notation to a minimum we omit strictness annotations from constructor signatures. Where necessary, constructor strictness will be mentioned explicitly.

3.2 Semantics of Types

We give semantics to types using a variation on the standard ideal model [MPS84]. The semantic domain $\mathcal{D}$ contains a least element $\perp$ and is equipped with a complete partial order $\leq$, where $\perp \leq t$ for all $t \in \mathcal{D}$. Types are *downward-closed* subsets of $\mathcal{D}$. A set $X \subseteq \mathcal{D}$ is downward-closed, iff $\forall t \in X - \{\perp\}, t' \leq t \implies t' \in X$.

Similarly, a type X is *upward-closed*, iff $\forall t \in X - \{\bot\}, t \leq t' \implies t' \in X$. We assume $\mathcal{D}$ is lifted (i.e. $\lambda x.\bot \neq \bot$).

A suitable domain $\mathcal{D}$ is described in [AW93] where the authors use it to give meaning to set types. We first review the semantics of set types and then describe the necessary generalizations for our framework. Given a *type assignment* $\sigma_{\mathcal{V}}$ mapping type variables in $\mathcal{V}$ to types, the meaning function σ mapping set types into the semantic domain $\mathcal{D}$, is

$$\begin{aligned}
\sigma(\alpha) &= \sigma_{\mathcal{V}}(\alpha) \\
\sigma(0) &= \{\bot\} \\
\sigma(1) &= \mathcal{D} \\
\sigma(c(T_1, \ldots, T_n)) &= \{c(t_1, \ldots, t_n) \mid t_i \in \sigma(T_i)\} \cup \{\bot\} \\
\sigma(T_1 \to T_2) &= \{f \mid t \in \sigma(T_1) \implies f(t) \in \sigma(T_2)\} \cup \{\bot\} \\
\sigma(T_1 \cap T_2) &= \sigma(T_1) \cap \sigma(T_2) \\
\sigma(T_1 \cup T_2) &= \sigma(T_1) \cup \sigma(T_2)
\end{aligned}$$

The function σ interprets types $c(T_1, \ldots, T_n)$ as sets of tuples labeled by c. Function types cannot be modeled as sets of tuples and are treated specially. Other standard type constructors, e.g. `list` also need to be modeled as special cases in such an interpretation. For example, the standard interpretation for a type $\texttt{list}(T)$ is

$$\begin{aligned}
&\sigma(\texttt{list}(T)) = X \quad \text{where } X \text{ is defined by the equation} \\
&X = \{\texttt{nil}, \bot\} \cup \{\texttt{cons}(t_1, t_2) \mid t_1 \in \sigma(T), t_2 \in X\}
\end{aligned}$$

which is very different from a set of tuples labeled by `list`.

Having numerous special cases is impractical and interpreting all type constructors as labeled tuples would severely limit the applicability of the formalism we are developing. Our solution is to parameterize the semantics of types with mappings $\phi_c : (2^{\mathcal{D}})^n \to 2^{\mathcal{D}}$ for each n-ary constructor c in a given signature set Σ. Each ϕ_c gives meaning to type expressions with head constructor c by mapping argument types (downward-closed sets) to types. Given a type assignment $\theta_{\mathcal{V}}$, we now define the meaning θ of type expressions by

$$\begin{aligned}
\theta(\alpha) &= \theta_{\mathcal{V}}(\alpha) \\
\theta(0) &= \{\bot\} \\
\theta(1) &= \mathcal{D} \\
\theta(c(T_1, \ldots, T_n)) &= \phi_c(\theta(T_1), \ldots, \theta(T_n)) \\
\theta(T_1 \cap T_2) &= \theta(T_1) \cap \theta(T_2) \\
\theta(T_1 \cup T_2) &= \theta(T_1) \cup \theta(T_2) \\
\theta(\neg\{c_1, \ldots, c_n\}) &= (\mathcal{D} - \textstyle\bigcup_i (\phi_{c_i}(\overline{\mathcal{D}}, \ldots, \overline{\mathcal{D}})) \cup \{\bot\}
\end{aligned}$$

We have added a case for set-complement with the intuitive meaning. The notation $\phi_c(\overline{\mathcal{D}}, \ldots, \overline{\mathcal{D}})$ denotes the largest type with head constructor c. Since c can be contravariant in certain fields, we use $\overline{\mathcal{D}}$ to denote $\mathcal{D}$ if it occurs in a covariant field, and $\{\bot\}$ if it occurs in a contravariant field.

The meaning functions ϕ_c must satisfy certain properties for the interpretation to be sensible and to guarantee the soundness of constraint resolution (see Section 4.2). These properties are summarized by the axioms below:

1. $\phi_c : (2^{\mathcal{D}})^n \to 2^{\mathcal{D}}$ if $c \in \Sigma$ is of arity n.
2. The variance and strictness of ϕ_c agrees with the declared variance and strictness of c.
   (For covariant fields, $X \subseteq Y \implies \phi_c(A_1, \ldots, A_n, X, B_1, \ldots, B_m) \subseteq \phi_c(A_1, \ldots, A_n, Y, B_1, \ldots, B_m)$.)
3. $\phi_c(X_1, \ldots, X_n) \supset \{\bot\}$ forall $X_1, \ldots, X_n$, st. $\begin{cases} X_i \supset \{\bot\} \text{ if the } i\text{th field is strict} \\ X_i \supseteq \{\bot\} \text{ otherwise} \end{cases}$
4. $\phi_c(\overline{\mathcal{D}}, \ldots, \overline{\mathcal{D}}) \cap \phi_d(\overline{\mathcal{D}}, \ldots, \overline{\mathcal{D}}) = \{\bot\}$ for all $c \neq d$. (i.e. constructor meanings are disjoint)
5. $\left(\bigcup_{c \in \Sigma} \phi_c(\overline{\mathcal{D}}, \ldots, \overline{\mathcal{D}})\right) \subset \mathcal{D}$
6. $\phi_c(X_1, \ldots, X_n) \cap \phi_c(Y_1, \ldots, Y_n) = \phi_c(X_1 \diamond_1 Y_1, \ldots, X_n \diamond_n Y_n)$
   for any upward-closed type $\phi_c(Y_1, \ldots, Y_n)$
   where $\diamond_i = \begin{cases} \cap \text{ if the } i\text{th field is covariant} \\ \cup \text{ if the } i\text{th field is contravariant} \end{cases}$

Axioms 1 and 2 require that the meaning function and the declared constructors agree in arity and variance. Axiom 3 requires that for all arguments each constructor c has at least one value besides bottom in its interpretation, as long as the arguments avoid the strictness of c. Axiom 4 says that distinct constructors map to disjoint meanings (besides bottom). Axiom 5 states that the domain $\mathcal{D}$ is larger than the union of all the constructor meanings. This axiom guarantees that no finite union is $\mathcal{D}$ and no complement is $\{\bot\}$[1], a fact that simplifies the resolution of constraints involving set-complement. Finally, Axiom 6 is a distributive law required to simplify intersections syntactically.

To illustrate that standard interpretations satisfy these axioms, consider the function type constructor $\cdot \longrightarrow \cdot : \overline{\mathbf{s}}\ \mathbf{s} \to \mathbf{s}$ with the usual interpretation

$$\phi_{\longrightarrow}(X,Y) = \{f \mid t \in X \implies f(t) \in Y\} \cup \{\bot\}$$

The interpretation agrees with the signature in arity and variance: As Y grows, more functions satisfy the implication, and as X grows, fewer functions satisfy the implication (Axioms 1 and 2). Axiom 3 is satisfied since even for the smallest function type $X = \mathcal{D}$, and $Y = \{\bot\}$ the interpretation contains the least function $\lambda x.\bot$, which is different from $\bot$. Axiom 4 implies that no other constructors $d \in \Sigma$ map to function values, since $\{\bot\} \to \mathcal{D}$ contains every function. Axiom 5 is not relevant for a single constructor. Finally, Axiom 6 is satisfied since the only upward-closed function type is $\phi_{\longrightarrow}(\{\bot\}, \mathcal{D})$ (all functions):

$$\begin{aligned}\phi_{\longrightarrow}(X,Y) \cap \phi_{\longrightarrow}(\{\bot\},\mathcal{D}) &= \phi_{\longrightarrow}(X,Y) \\ &= \phi_{\longrightarrow}(X \cup \{\bot\}, Y \cap \mathcal{D})\end{aligned}$$

Before we define the solutions to systems of constraints in Section 3.3, we need the notion of a *contour*. Contours capture semantically the property that **u**-types have a single head constructor. A semantic notion of a single head constructor is needed to define the solutions for **u**-variables. Intuitively, contours correspond to the largest **u**-types in the domain $\mathcal{D}$.

Definition 1 (Contour). A set $X \subseteq \mathcal{D}$ is a contour, iff $\exists c : \iota_1 \ldots \iota_n \to \mathbf{u} \in \Sigma$ such that

$$X = \phi_c(Y_1, \ldots, Y_n)$$

and for all $k = 1 \ldots n$,

$$Y_k = \begin{cases} \text{a contour} & \text{if } \iota_k = \mathbf{u} \text{ or } \iota_k = \overline{\mathbf{u}} \\ \mathcal{D} & \text{if } \iota_k = \mathbf{s} \\ \{\bot\} & \text{if } \iota_k = \overline{\mathbf{s}} \end{cases}$$

Note that by Axiom 3 above, contours are strictly larger than $\{\bot\}$. Given a *contour assignment* $\Theta_{\mathcal{V}_\mathbf{u}}$ mapping **u**-variables to contours, we define an alternative interpretation of **u**-types as contours:

$$\begin{aligned}\Theta(\alpha_\mathbf{u}) &= \Theta_{\mathcal{V}_\mathbf{u}}(\alpha_\mathbf{u}) \\ \Theta(c(T_1,\ldots,T_n)) &= \phi_c(Y_1,\ldots,Y_n) \quad \text{where } c : \iota_1 \ldots \iota_n \to \mathbf{u} \text{ and} \\ Y_k &= \begin{cases} \Theta(T_k) & \text{if } \iota_k = \mathbf{u} \text{ or } \iota_k = \overline{\mathbf{u}} \\ \mathcal{D} & \text{if } \iota_k = \mathbf{s} \\ \{\bot\} & \text{if } \iota_k = \overline{\mathbf{s}} \end{cases}\end{aligned}$$

3.3 Constraints and Admissible Solutions

Constraints are formed between pairs of **u**-types or pairs of **s**-types, written $T_1 \subseteq_\mathbf{u} T_2$ and $T_1 \subseteq_\mathbf{s} T_2$ respectively. Consider again the example constraint $\mathtt{exn}(\mathtt{Match}) \subseteq \alpha$ from Section 2.1. Given the semantics of ML, we want to rule out solutions such as $\alpha = \mathrm{int} \cup \mathtt{exn}(\mathtt{Match})$. The only solutions we are interested in are of the form $\alpha = \mathtt{exn}(\beta)$, where $\mathtt{Match} \subseteq \beta$, i.e. α has a single head constructor in all solutions. We can achieve exactly this effect using $\subseteq_\mathbf{u}$ constraints. The meaning of $\subseteq_\mathbf{s}$ and $\subseteq_\mathbf{u}$ constraints is given below.

Definition 2 (Solution). A pair $(\theta_\mathcal{V}, \Theta_{\mathcal{V}_\mathbf{u}})$ consisting of a type assignment $\theta_\mathcal{V}$ and a contour assignment $\Theta_{\mathcal{V}_\mathbf{u}}$ is a solution to a system of constraints S if and only if

[1] Different from [AW93].

- $\theta(T_1) \subseteq \theta(T_2)$ for every constraint $T_1 \subseteq_\iota T_2 \in S$.
  ($\iota = \mathbf{u}$ or $\iota = \mathbf{s}$)
- $\Theta(T_1) = \Theta(T_2)$ for every constraint $T_1 \subseteq_\mathbf{u} T_2 \in S$.
  (The contours of $\mathbf{u}$-constrained types must be equal.)
- $\theta_\mathcal{V}(\alpha) \subseteq \Theta_{\mathcal{V}_\mathbf{u}}(\alpha)$ for every $\mathbf{u}$-variable α in S.
  (The solution for α agrees with the contour assigned to α.)

We illustrate Definition 2 with a few examples. Consider the signatures Σ of strict constructors

$$\mathtt{c} : \mathbf{s} \to \mathbf{u}$$
$$\mathtt{d} : \mathbf{s} \to \mathbf{u}$$

and the constraint $\mathtt{c}(0) \subseteq_\mathbf{u} \mathtt{d}(1)$. Since $\mathtt{c}$ is strict, $\mathtt{c}(0) = 0$, so $\theta(\mathtt{c}(0)) \subseteq \theta(\mathtt{d}(1))$ is satisfied for any θ. For an inclusion $\subseteq_\mathbf{s}$ this would be enough. However, the contours $\Theta(\mathtt{c}(0))$ and $\Theta(\mathtt{d}(1))$ are strictly larger than $\{\bot\}$ (Axiom 3) and thus cannot be equal by Axiom 4. Therefore, this constraint is unsatisfiable.

Next consider the constraint $\alpha \subseteq_\mathbf{u} \mathtt{d}(1)$. This constraint is satisfied by any assignment for α of the form $\mathtt{d}(\beta)$. To see why α must be of this form, note that the contour $\Theta(\alpha)$ must be equal to $\Theta(\mathtt{d}(1))$ in all solutions. Since any solution for α must agree with this contour, no solutions are lost by equating $\alpha = \mathtt{d}(\beta)$ (β fresh). These equations make it possible to solve parts of the constraints using unification. By the same reasoning, one can show that the constraints $\alpha \subseteq_\mathbf{u} \mathtt{c}(1) \ \wedge \ \alpha \subseteq_\mathbf{u} \mathtt{d}(1)$ have no solution.

4 Computing Solutions

This section describes how to compute the solutions of a system of $\mathbf{u}$ and $\mathbf{s}$-constraints. Section 4.1 describes why adapting the resolution of [AW93] to our type language is difficult due to our parameterized interpretation of types and how this problem can be solved, and Section 4.2 describes the resolution rules.

The following terminology is used in the next sections. Type expressions on the left of constraints are said to occur in *positive positions*, and type expressions on the right of constraints occur in *negative positions*. Type sub-expressions occur in the same position (positive or negative) as their immediately enclosing expression, unless the sub-expression is a contravariant field of a constructor, in which case its position is inverse w.r.t. the enclosing expression. For example, assuming the signature for a function type constructor is $\cdot \to \cdot : \overline{\mathbf{s}}\ \mathbf{s} \to \mathbf{s}$, then in the constraint $T_1 \to T_2 \subseteq_\mathbf{s} T_3 \to T_4$, the sub-expressions T_1 and T_4 occur negatively and T_2 and T_3 occur positively.

4.1 Upward-closed Monotypes and Type Complement

We first review the theory developed in [AW93] to solve inclusion constraints and then adapt it to our new formalism. The simple part is to extend the resolution rules to solve $\mathbf{u}$-constraints (Section 4.2). Here, we deal with the more serious problem, namely adapting the resolution to our parameterized interpretations ϕ_c of type constructors.

The resolution rules given by Aiken and Wimmers require the ability to compute two type expressions $\overline{T}$ and $\neg\overline{T}$ for any type expression T. The type expression $\overline{T}$ is both ground (has no variables) and has the property that it denotes the smallest upward-closed set s.t. $\sigma(\overline{T}) \supseteq \sigma(T)$ for all assignments σ. For example $\overline{T_1 \to T_2} = 0 \to 1$ (the set of all functions) for any T_1 and T_2. The type expression $\neg\overline{T}$ denotes $(\mathcal{D} - \sigma(\overline{T})) \cup \{\bot\}$, which is the type complement of $\overline{T}$. (To see this, note that $\overline{T} \cup \neg\overline{T} = 1$ and $\overline{T} \cap \neg\overline{T} = 0$.)

The algorithms for computing $\overline{T}$ and $\neg\overline{T}$ given in [AW93] are syntax-directed and depend crucially on the fixed interpretation of constructors described in Section 3.2. Since we parameterize the interpretation of constructors, we have no hope of giving an algorithm that computes upward-closed types and complement syntactically.

Before we outline our solution to this problem, we need to delve deeper into the reasons why upward-closure and type complement are needed for constraint resolution. Constraint resolution involves systematically rewriting constraints into simpler forms. There are two forms of constraints that are difficult to decompose during resolution: $T_1 \cap T_2 \subseteq_\mathbf{s} T_3$ and $T_1 \subseteq_\mathbf{s} T_2 \cup T_3$. In pure set theory, the constraint $T_1 \subseteq T_2 \cup T_3$ is equivalent to $T_1 \cap \neg T_2 \subseteq T_3$. However, we interpret set expressions as types (downward-closed sets of

values [MPS84]) and the complement of a type is not necessarily a downward-closed set, and thus not a type. For example, the complement of the function type $1 \to 0$ contains every function except the least function $\lambda x.\bot$. Only the complements of upward-closed types are themselves types.

Aiken and Wimmers show how to solve these problematic constraints under the following restrictions

- Unions $T_1 \cup T_2$ in negative positions must be disjoint, i.e. $T_1 \cap T_2 = 0$ in all solutions.
- Intersections in positive positions must be of the form $T \cap M$, where M denotes an upward-closed *monotype* (ground type). (In the rest of the paper, M stands for upward-closed monotypes.)

The problematic constraints are then simplified using the following two rules:

$$T_1 \subseteq T_2 \cup T_3 \Leftrightarrow T_1 \cap \neg\overline{T_2} \subseteq T_3 \ \wedge \ T_1 \cap \neg\overline{T_3} \subseteq T_2$$
$$T_1 \cap M \subseteq T_2 \Leftrightarrow T_1 \subseteq (T_2 \cap M) \cup \neg M$$

The resolution rules are to be read as left-to-right rewrite rules. Observe that the right-hand sides of the rules introduce upward-closed types and complement types not present on the left.

These resolution rules are unusable in our framework since we cannot form the upward-closure and complement of types during resolution. Fortunately, inspection of the rewrite rules shows that the set of upward-closed monotypes required during resolution is fixed by the initial constraints. Therefore, we can circumvent the problem by putting the constraints in a form that makes all required upward-closed monotypes explicit in the initial system of constraints. To make the necessary upward-closed monotypes explicit in the constraints, we define an abbreviation Pat as follows.[2] Let M be an upward-closed monotype. Define

$$\mathrm{Pat}(T, M) = (T \cap M) \cup \neg M$$

With Pat we can reformulate the resolution rules for intersections in positive positions as follows

$$T_1 \cap M \subseteq T_2 \Leftrightarrow T_1 \subseteq \mathrm{Pat}(T_2, M)$$

Note that the right-side of the equivalence uses only type expressions present on the left. Representing arbitrary disjoint unions in negative positions is more complex. Let T_1 and T_2 be disjoint types. Observe that

$$\begin{aligned}
&\mathrm{Pat}(T_1, \overline{T_1}) \cap \mathrm{Pat}(T_2, \overline{T_2}) \cap \mathrm{Pat}(0, \neg(\overline{T_1} \cup \overline{T_2})) \\
&= (T_1 \cap \overline{T_1} \cup \neg\overline{T_1}) \cap (T_2 \cap \overline{T_2} \cup \neg\overline{T_2}) \cap (\overline{T_1} \cup \overline{T_2}) && \text{def. of Pat} \\
&= (T_1 \cup \neg\overline{T_1}) \cap (T_2 \cup \neg\overline{T_2}) \cap (\overline{T_1} \cup \overline{T_2}) && T \cap \overline{T} = T \\
&= T_1 \cap T_2 \cap \overline{T_1} \ \cup \ T_1 \cap T_2 \cap \overline{T_2} \ \cup \ T_1 \cap \neg\overline{T_2} \cap \overline{T_1} \ \cup \ T_1 \cap \neg\overline{T_2} \cap \overline{T_2} \cup && \text{distribute} \\
&\quad \neg\overline{T_1} \cap T_2 \cap \overline{T_1} \ \cup \ \neg\overline{T_1} \cap T_2 \cap \overline{T_2} \ \cup \ \neg\overline{T_1} \cap \neg\overline{T_2} \cap \overline{T_1} \ \cup \ \neg\overline{T_1} \cap \neg\overline{T_2} \cap \overline{T_2} \\
&= T_1 \cap \neg\overline{T_2} \cap \overline{T_1} \ \cup \ \neg\overline{T_1} \cap T_2 \cap \overline{T_2} && T \cap \neg\overline{T} = 0, \ T_1 \cap T_2 = 0 \\
&= T_1 \cap \neg\overline{T_2} \ \cup \ \neg\overline{T_1} \cap T_2 && T \cap \overline{T} = T \\
&= T_1 \ \cup \ T_2 && \overline{T_1} \cap \overline{T_2} = 0
\end{aligned}$$

To represent $T_1 \cup T_2$ in negative positions, we need the upward-closures $\overline{T_1}$, $\overline{T_2}$, and the complement $\neg(\overline{T_1} \cup \overline{T_2})$. Below, we show the resolution of a constraint involving $T_1 \cup T_2$:

$$\begin{aligned}
T \subseteq T_1 \cup T_2 \ &= \ T \subseteq \mathrm{Pat}(T_1, \overline{T_1}) \cap \mathrm{Pat}(T_2, \overline{T_2}) \cap \mathrm{Pat}(0, \neg(\overline{T_1} \cup \overline{T_2})) \\
&\Leftrightarrow T \subseteq \mathrm{Pat}(T_1, \overline{T_1}) \ \wedge \ T \subseteq \mathrm{Pat}(T_2, \overline{T_2}) \ \wedge \ T \subseteq \mathrm{Pat}(0, \neg(\overline{T_1} \cup \overline{T_2})) \\
&\Leftrightarrow T \cap \overline{T_1} \subseteq T_1 \ \wedge \ T \cap \overline{T_2} \subseteq T_2 \ \wedge \ T \cap \neg(\overline{T_1} \cup \overline{T_2}) \subseteq 0
\end{aligned}$$

[2] Pat stands for *pattern*, since it is used most frequently in constraints generated for pattern matching.

$$S \cup \{0 \subseteq_{\mathbf{s}} T\} \equiv S \tag{1}$$
$$S \cup \{T \subseteq_{\mathbf{s}} 1\} \equiv S \tag{2}$$
$$S \cup \{c(T_1, \ldots, T_n) \subseteq_\iota c(T'_1, \ldots, T'_n)\} \equiv S \cup \{T_1 \subseteq_{\iota_1} T'_1, \ldots, T_n \subseteq_{\iota_n} T'_n\} \quad \text{where } c : \iota_1 \ldots \iota_n \to \iota \tag{3}$$
$$S \cup \{T_1 \cup T_2 \subseteq_{\mathbf{s}} T\} \equiv S \cup \{T_1 \subseteq_{\mathbf{s}} T, T_2 \subseteq_{\mathbf{s}} T\} \tag{4}$$
$$S \cup \{T \subseteq_{\mathbf{s}} T_1 \cap T_2\} \equiv S \cup \{T \subseteq_{\mathbf{s}} T_1, T \subseteq_{\mathbf{s}} T_2\} \tag{5}$$
$$S \cup \{\alpha \subseteq_\iota \alpha\} \equiv S \tag{6}$$
$$S \cup \{\alpha \cap M \subseteq_{\mathbf{s}} \alpha\} \equiv S \tag{7}$$

$$S \cup \{T_1 \subseteq_{\mathbf{s}} \mathrm{Pat}(T_2, M)\} \equiv S \cup \{T_1 \cap M \subseteq_{\mathbf{s}} T_2\} \qquad \text{if } T_1 \neq \alpha \tag{8}$$
$$S \cup \{\alpha \cap M \subseteq_{\mathbf{s}} T\} \equiv S \cup \{\alpha \subseteq_{\mathbf{s}} \mathrm{Pat}(T, M)\} \tag{9}$$
$$S \cup \{\neg\{c_1, \ldots, c_n\} \subseteq_{\mathbf{s}} \neg\{d_1, \ldots, d_m\}\} \equiv S \qquad \text{if } \{d_1, \ldots, d_m\} \subseteq \{c_1, \ldots, c_n\} \tag{10}$$
$$S \cup \{c(\ldots) \subseteq_{\mathbf{s}} \neg\{d_1, \ldots, d_m\}\} \equiv S \qquad \text{if } c \notin \{d_1, \ldots, d_m\} \tag{11}$$
$$S \cup \{T_1 \subseteq_{\overline{\mathbf{s}}} T_2\} \equiv S \cup \{T_2 \subseteq_{\mathbf{s}} T_1\} \tag{12}$$
$$S \cup \{T_1 \subseteq_{\overline{\mathbf{u}}} T_2\} \equiv S \cup \{T_2 \subseteq_{\mathbf{u}} T_1\} \tag{13}$$
$$S \cup \{\alpha \subseteq_{\mathbf{u}} c(T_1, \ldots, T_n)\} \equiv S \cup \{\alpha = c(\alpha_1, \ldots, \alpha_n), \alpha_i \subseteq_{\iota_i} T_i\} \tag{14}$$
$$\alpha_i \text{ fresh}, c : \iota_1 \ldots \iota_n \to \mathbf{u}$$
$$S \cup \{c(T_1, \ldots, T_n) \subseteq_{\mathbf{u}} \alpha\} \equiv S \cup \{\alpha = c(\alpha_1, \ldots, \alpha_n), T_i \subseteq_{\iota_i} \alpha_i\} \tag{15}$$
$$\alpha_i \text{ fresh}, c : \iota_1 \ldots \iota_n \to \mathbf{u}$$

Fig. 4. Resolution rules for constraints.

Instead of using unions in negative positions, constraints need to be written as intersections of Pat. Only unions of upward-closed monotypes remain, and these can only appear in second positions of Pat, where they never need to be decomposed.

To summarize, we replace the two rules of [AW93] for simplifying intersections in positive positions and unions in negative positions with a single rule, along with an abbreviation Pat containing an implicit complement. The new resolution rule uses only sub-expressions of the original constraints, and does not require the formation of upward-closed and complement types during resolution. Because an implementation of our system cannot know the intended interpretation of constructors, our approach effectively requires the analysis designer to provide the necessary upward-closed and complement types explicitly to the system.

4.2 Constraint Resolution

Having dealt with the necessary changes to accommodate the parameterized semantics of constructors, we can now focus on the resolution of constraints between **u**-types. Figure 4 shows the resolution rules. Due to the sorted algebra of types the resolution rules preserve sorts, i.e. given constraints between types of the same sort, the resolution rules only produce constraints between equal sorts. The rules should be read as left-to-right rewrite rules. Rules 1–7 are from [AW93]. Below the line are the new rules. Rules 8 and 9 are discussed above. Note the side condition on Rule 8, which, along with intersection simplification (Figure 5), avoids a cycle in the rewrite rules. Rules 10 and 11 deal with complement types. Rules 12 and 13 flip the inclusion for constraints arising from contravariant fields. Rules 14 and 15 instantiate **u**-variables to satisfy contour equalities (similar to the approach of [HM97, Mos96]); these rules introduce fresh variables.

We must ensure that the resolution process terminates. The simple constraint

$$\alpha \subseteq_{\mathbf{u}} c(\alpha)$$

produces the sequence of constraints $\alpha = c(\alpha_1)$, $\alpha_1 \subseteq_{\mathbf{u}} \alpha$, $\alpha_1 \subseteq_{\mathbf{u}} c(\alpha_1)$, etc. ad infinitum. The problem is essentially the same as in unification and can be solved with an occurs check that ensures the instantiated variable α does not appear in the instantiation.

$$
\begin{aligned}
T \cap_\iota T &= T && \\
T \cap_{\mathbf{s}} 0 &= 0 && \\
T \cap_{\mathbf{s}} 1 &= T && \\
T \cap_{\mathbf{u}} 1 &= T && \\
T \cap_{\overline{\mathbf{s}}} 0 &= T && \text{(see text page 11)} \\
T \cap_{\overline{\mathbf{u}}} 0 &= T && \text{(see text page 11)} \\
c(T_1, \ldots, T_n) \cap_\iota c(T'_1, \ldots, T'_n) &= c(T_1 \cap_{\iota_1} T'_1, \ldots, T_n \cap_{\iota_n} T'_n) && c : \iota_1 \ldots \iota_n \to \iota \in \Sigma \\
c(\ldots) \cap_{\mathbf{s}} d(\ldots) &= 0 && c \neq d \\
c(\ldots) \cap_{\mathbf{u}} d(\ldots) &= \text{Inconsistent} && c \neq d \\
(\alpha \cap_{\mathbf{s}} T_1) \cap_{\mathbf{s}} T_2 &= \alpha \cap_{\mathbf{s}} (T_1 \cap_{\mathbf{s}} T_2) && \\
(T_1 \cup T_2) \cap_{\mathbf{s}} T_3 &= (T_1 \cap_{\mathbf{s}} T_3) \cup (T_2 \cap_{\mathbf{s}} T_3) && \\
c(\ldots) \cap_{\mathbf{s}} \neg\{c_1, \ldots, c_n\} &= \begin{cases} c(\ldots) & \text{if } c \notin \{c_1, \ldots, c_n\} \\ 0 & \text{otherwise} \end{cases} && \\
\neg\{c_1, \ldots, c_n\} \cap_{\mathbf{s}} \neg\{d_1, \ldots, d_m\} &= \neg(\{c_1, \ldots, c_n\} \cup \{d_1, \ldots, d_m\}) && \\
\alpha \cap_{\mathbf{u}} c(T_1, \ldots, T_n) &= c(\alpha_1 \cap_{\iota_1} T_1, \ldots, \alpha_n \cap_{\iota_n} T_n) && \alpha = c(\alpha_1, \ldots, \alpha_n) \\
& && c : \iota_1 \ldots \iota_n \to \mathbf{u} \in \Sigma
\end{aligned}
$$

Fig. 5. Simplifying intersections $T \cap M$ where M is an upward-closed monotype.

Constraints are solved by applying the resolution rules until no rule applies.[3] The resulting system is either inconsistent (meaning it has a constraint listed in Figure 6 or fails an occurs check), in which case it has no solutions, or the the constraints are all on variables and the solutions can be characterized in the same way as described in [AW93][4]

We briefly discuss the soundness of the resolution rules. Rules 1, 2 and 6 are obviously sound. Rule 3 follows from the variance of constructors (Axiom 2), Rules 4 and 5 from standard set theory. Rules 7–9 follow from set theory, disjointness (Axiom 4), and Axiom 6. Rules 10 and 11 are sound by Axioms 4 and 5.

While the resolution rules are sound, they may be incomplete. If the ϕ_c are not injective (e.g. consider strict constructors), the resolution of constructors may impose stronger constraints on arguments than necessary, and more system may be rejected as inconsistent.

As mentioned above, intersections in positive positions must be simplified to guarantee that rule 9 need only be applied to a left-hand side of the form $\alpha \cap M$. We briefly discuss the intersection simplification in Figure 5. Intersections can in principle only be formed between **s**-types. However, it is convenient to express $\cap$-simplification using intersections between **u**-types that are always eliminated as part of the simplification process. We use the operators $\cap_{\mathbf{s}}$ and $\cap_{\mathbf{u}}$ to distinguish between intersections on **s**-types and **u**-types respectively. The notation $\cap_{\overline{\mathbf{s}}}$ and $\cap_{\overline{\mathbf{u}}}$ is used for unions that appear due to contravariant fields. Since contravariant fields in upward-closed monotypes must be 0, the only forms involving $\cap_{\overline{\mathbf{s}}}$ and $\cap_{\overline{\mathbf{u}}}$ are $T \cap_{\overline{\mathbf{s}}} 0 = T$ and $T \cap_{\overline{\mathbf{u}}} 0 = T$. Intersections involving **u**-variables $\alpha \cap c(\ldots)$ are simplified by instantiating $\alpha = c(\alpha_1, \ldots, \alpha_n)$. The variable α must be of this form in order for the intersection to have a contour (see Section 3.2). Furthermore, note that in upward-closed monotypes appearing in intersections we allow the type 1 in covariant (resp. 0 in contravariant) fields of sort **u**. Symmetric cases are omitted from Figure 5.

Our motivation for the presented framework was to use unification to solve constraints between **u**-types in nearly linear time. Due to the generation of fresh variables during resolution of $\subseteq_{\mathbf{u}}$ constraints, solving may actually require exponential time in the size of the constraints. The reason is that the constraints can

[3] Note that we have omitted the rules for transitive constraints, which are standard.

[4] Additionally, the meaning functions ϕ_c must satisfy $\phi_c(X_1, \ldots, X_n) \subseteq \mathcal{D}_i \implies X_1 \ldots X_n \subseteq \mathcal{D}_{i-1}$, where $\mathcal{D}_i$ are elements of the increasing sequence of approximations $\mathcal{D}_0 \subseteq \mathcal{D}_1 \subseteq \mathcal{D}_2 \ldots$, that make up the domain $\mathcal{D}$.

$$
\begin{array}{ll}
S \cup \{1 \subseteq_{\mathbf{s}} 0\} & \\
S \cup \{1 \subseteq_{\mathbf{s}} c(\ldots)\} & \\
S \cup \{1 \subseteq_{\mathbf{s}} \neg\{c_1, \ldots, c_n\}\} & n > 0 \\
S \cup \{c(\ldots) \subseteq_{\mathbf{s}} 0\} & \\
S \cup \{c(\ldots) \subseteq_{\iota} d(\ldots)\} & c \neq d \\
S \cup \{c(\ldots) \subseteq_{\mathbf{s}} \neg\{d_1, \ldots, d_m\}\} & c \in \{d_1, \ldots, d_m\} \\
S \cup \{\neg\{\ldots\} \subseteq_{\mathbf{s}} 0\} & \\
S \cup \{\neg\{\ldots\} \subseteq_{\mathbf{s}} c(\ldots)\} & \\
S \cup \{\neg\{c_1, \ldots, c_n\} \subseteq_{\mathbf{s}} \neg\{d_1, \ldots, d_m\}\} \; \{d_1, \ldots, d_m\} \not\subseteq \{c_1, \ldots, c_n\} &
\end{array}
$$

Fig. 6. Inconsistent constraint systems.

describe types of exponential tree size and that fresh copies of such types may be formed. In unification based type inference, types can also be of exponential size, but their shared graph representation is always linear in the original constraints and never copied. In practice, the complexity of our approach depends on the application. As long as the inferred types are relatively small (as e.g. in the case of ML [HM97]), the practical complexity appears to be close to linear.

5 Exception Inference for ML

We now instantiate the developed framework to our motivating example from Section 2.2. The signatures of type constructors appear in Figure 3 and have already been described in Section 3.1.

We cast the type and exception inference for ML as an effect inference system [LG88]. In this model, every expression has a type and an effect. The type of an expression describes the set of possible unexceptional values of the expression, whereas the effect describes the set of exceptions that may be raised during evaluation. Note that exception inference never fails for well-typed ML programs.

Figure 7 shows the type rules for exception inference. Types for constants, exceptions and primitive operators are assumed to be defined in an initial type environment. Judgments have the form $A \vdash e : \tau \,!\, \sigma$, meaning that under the type assumptions A, expression e has type τ and may raise the exceptions σ. There are also judgments for exception patterns $\vdash_{\mathrm{p}} p : (\sigma, c, \overline{\sigma}, A)$ meaning that pattern p matches exception σ and binds variables x in the domain of A to the type $A(x)$. Furthermore, the judgment infers the exception constructor c and upward-closed monotype $\overline{\sigma}$ of the pattern. Observe that the rule for `handle` expressions makes use of the full expressive power of set types. It uses intersection and complement to form the set of exceptions that pass through the handler, and union is used to combine the exceptions of all the handlers. The constraint $\sigma_0 \subseteq_{\mathbf{s}} \mathrm{Pat}(\sigma_i, \overline{\sigma_i},)$ intuitively separates σ_0 into those values that match the pattern (σ_i) and those that do not ($\neg\overline{\sigma_i}$) (see definition Section 4.1).

Some remarks about extending the described exception inference to core SML are in order.

- Let-polymorphism is handled as described in [AW93].
- Exception declarations in SML produce new exceptions at every evaluation. Exception declarations within `let` expressions can therefore give rise to an unbounded number of distinct exceptions, all sharing the same name. Consequently, only exceptions declared at toplevel can safely be filtered by name in `handle` expressions. In practice, we find that the vast majority of exceptions are declared at toplevel. This problem does not arise in the CAML dialect of ML.
- Datatypes hide the internal structure of values. We must ensure that exceptions do not "disappear" into datatypes. To this end, we extend datatypes containing exception values (directly or through functions) with a single extra type parameter to capture these exceptions.
- ML has mutable references. We treat these as special cases in our implementation.

$$A \vdash x : A(x) \,!\, 0 \qquad \text{[VAR]}$$

$$\frac{A[x \mapsto \alpha] \vdash e : \tau \,!\, \sigma}{A \vdash \texttt{fn}\ x\ \texttt{=>}\ e : (\alpha \xrightarrow{\sigma} \tau) \,!\, 0} \qquad \text{[ABS]}$$

$$\frac{\begin{array}{c} A \vdash e_1 : \tau_1 \,!\, \sigma_1 \\ A \vdash e_2 : \tau_2 \,!\, \sigma_2 \\ \tau_1 \subseteq_{\mathbf{u}} \tau_2 \xrightarrow{\epsilon} \alpha \end{array}}{A \vdash e_1\ e_2 : \alpha \,!\, \sigma_1 \cup \sigma_2 \cup \epsilon} \qquad \text{[APP]}$$

$$\frac{A \vdash e : \texttt{exn}(\epsilon) \,!\, \sigma}{A \vdash \texttt{raise}\ e : \alpha \,!\, \sigma \cup \epsilon} \qquad \text{[RAISE]}$$

$$\frac{\begin{array}{l} A \vdash e_0 : \tau_0 \,!\, \sigma_0 \\ \vdash_{\mathrm{p}} p_i : (\sigma_i, c_i, \overline{\sigma_i}, A_i) \quad \text{for } i = 1 \ldots n \\ A + A_i \vdash e_i : \tau_i \,!\, \sigma_i' \quad \text{for } i = 1 \ldots n \\ \sigma_{\mathrm{pass}} = \sigma_0 \cap \neg\{c_1, \ldots, c_n\} \\ \sigma_0 \subseteq_{\mathbf{s}} \mathrm{Pat}(\sigma_i, \overline{\sigma_i}) \quad \text{for } i = 1 \ldots n \\ \tau_i \subseteq_{\mathbf{u}} \alpha \quad \text{for } i = 0 \ldots n \end{array}}{A \vdash e_0\ \texttt{handle}\ p_i\ \texttt{=>}\ e_i : \alpha \,!\, (\sigma_{\mathrm{pass}} \cup \bigcup_{i=1\ldots n} \sigma_i')} \qquad \text{[HANDLE]}$$

$$\vdash_{\mathrm{p}} c : (c, c, c, []) \qquad \text{[PCON]}$$

$$\frac{\mathrm{typeof}(c) = \tau \rightarrow \texttt{exn}(c(\tau))}{\vdash_{\mathrm{p}} c(x) : (c(\tau), c, c(1), [x \mapsto \tau])} \qquad \text{[PAPP]}$$

Fig. 7. Type and exception inference rules for Mini-ML.

We have implemented the ML exception inference using an untuned prototype implementation of the general constraint framework. The largest program we have tested so far is the lexer generator ml-lex (1200 lines of ML). The analysis time for ml-lex is 2.8sec on a 200MHz Pentium with 64MB of main memory. This compares well to the 0.9sec the SML/NJ compiler requires to type-check the same program. The analysis infers the following type for the main function `lexGen`:

```
lexGen : string -(Match \/ eof \/ error \/ lex_error \/ Subscript)-> unit
```

The five uncaught exceptions correspond exactly to the results reported by Yi [Yi94].

6 Related Work

Work on set-based program analysis [Hei92, FF97] and inclusion constraint-based type inference [AWL94, Pot96, FA96, TS96, MW97] has mostly focused on how to simplify constraints to achieve scalability. The developed techniques and heuristics are orthogonal to our approach of restricting the interesting solutions of constraints. We deem constraint simplification still necessary on the regular inclusion constraints that arise in our approach.

In [MNP97] the authors describe INÈS, a system for solving inclusion constraints over non-empty sets of trees. They give an algorithm for computing the largest solution of the constraints and show that equal-

ity constraints between set expressions can be solved using unification. Their constraint language is less expressive than ours and they only compute a particular solution of the constraints.

In type disciplines based on primitive subtyping [Mit84, FM88], the base types form a partial order. This order induces a partial order on types by structural extension over function type constructors, tuples, etc. Subtype constraints can be solved structurally until only atomic constraints (between atoms) remain. There is a strong parallel to our approach in that we can solve inclusion constraints between **u**-types structurally until we are left with inclusion constraints between **s**-types. Our approach differs however in that the constraints between **s**-types may induce new constraints between **u**-types, whereas atomic subtyping constraints can never induce new structural constraints.

Effect systems [Luc87] naturally contain a mixture of Hindley-Milner types and sets for effects. In [LG88] Lucassen and Gifford describe type and effect inference rules using a subset relation on types induced by the subset relation of effect sets contained in the types. However, they do not show how to solve such constraints and, in fact, in a later paper drop the subset constraints for equality constraints which they solve with generalized unification [JG91]. Similarly, Tofte and Talpin [TT94] use a mixture of types and sets in an effect system to infer allocation and deallocation points of memory regions at compile-time. But their inference rules are based on equality constraints which they solve using a generalized unification procedure.

Henglein's work on efficient binding time analysis [Hen91] and tag inference [Hen92] also combines subtyping and equality constraints. His algorithms runs in nearly linear time. They can unfortunately not be directly expressed in our framework.

In [SH97] the authors study points-to analysis w.r.t. the precision–efficiency tradeoff. They contrast an algorithm based on inclusion constraints [And94] with the equality based algorithm of [Ste96], and then describe a spectrum of algorithms in between. We are currently using the same analyses to tune and validate our framework.

7 Conclusion

We described a parameterized constraint formalism that combines inclusion constraints over terms and sets. The formalism covers an entire spectrum of program analyses with varying degrees of precision and efficiency, ranging from Hindley-Milner type inference to complete inclusion based analysis.

We instantiated the framework with an example analysis for inferring types and exceptions for a subset of ML. Preliminary timing measurements are very encouraging. The running time of our type inference with exceptions is roughly within a factor of three of standard type inference on medium-size programs.

8 Acknowledgments

We would like to thank John Boyland and Rowan Davies for helpful comments on drafts of this paper.

References

[AKVW93] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Computer Science Logic '93*, volume 832 of *Lect. Notes in Comput. Sci.*, pages 1–17. Eur. Assoc. Comput. Sci. Logic, Springer, September 1993.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[AW92] A. Aiken and E. Wimmers. Solving Systems of Set Constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[AW93] A. Aiken and E. Wimmers. Type Inclusion Constraints and Type Inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[AWL94] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, January 1994.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by contruction or approximation of fixed points. In *Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[FA96] Manuel Fähndrich and Alex Aiken. Making set-constraint based program analyses scale. In *First Workshop on Set Constraints at CP'96*, Cambridge, MA, August 1996. Available as Technical Report CSD-TR-96-917, University of California at Berkeley.

[FF97] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In PLDI'97 [PLD97].

[FFK+96] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching Bugs in the Web of Program Invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[FM88] Y. Fuh and P. Mishra. Type inference with subtypes. In *Proceedings of the 1988 European Symposium on Programming*, pages 94–114, 1988.

[GS94] Juan Carlos Guzmán and Ascánder Suárez. An extended type system for exceptions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 127–135, June 1994.

[Hei92] N. Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, 1992.

[Hei94] Nevin Heintze. Set Based Analysis of ML Programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–17, June 1994.

[Hen91] F. Henglein. Efficient Type Inference for Higher-Order Binding-Time Analysis. In *5th ACM Conference Proceedings on Functional Programming Languages and Computer Architecture*, pages 448–72, 1991.

[Hen92] F. Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, July 1992.

[HM97] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In PLDI'97 [PLD97].

[ICF97] *Proceedings of the International Conference on Functional Programming (ICFP '97)*, June 1997.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 303–310, January 1991.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[Luc87] John M. Lucassen. *Types and Effects —Towards the Integration of Functional and Imperative Programming.* Ph.D. thesis, MIT Laboratory for Computer Science, August 1987.

[MH97] David McAllester and Nevin Heintze. On the complexity of set-based analysis. In ICFP'97 [ICF97], pages 150–63.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit84] J. Mitchell. Coercion and type inference (summary). In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[MNP97] Martin Müller, Joachim Niehren, and Andreas Podelski. Inclusion constraints over non-empty sets of trees. In *Proceedings of the Seventh International Joint Conference on the Theory and Practice of Software Development (TAPSOFT'97)*, April 1997.

[Mos96] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[MPS84] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymophic types. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 165–174, January 1984.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In ICFP'97 [ICF97].

[PLD97] *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

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

[SH97] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[Shi88] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[TS96] Valery Trifonov and Scott Smith. Subtyping Constrained Types. In *Proceedings of the 3rd International Static Analysis Symposium*, pages 349–365, September 1996.

[TT94] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[Yi94] Kwangkeun Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Proceedings of the 1st International Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science.* Springer, 1994.