Refined Type Inference for ML*

(Abstract)

Manuel Fähndrich† and Alexander Aiken†
University of California, Berkeley‡

1 Introduction

Inclusion constraints over set-expressions [1, 4] provide a general formalism to express a large class of program analyses. Over the past two years, we have experimented with inclusion constraints to model dataflow in type-based analyses. One of our research goals is to determine how to structure and implement precise constraint-based analyses such that they scale to large programs. Program analyses with $O(n^3)$ complexity bounds often exhibit their worst-case complexity in practice and consequently do not scale beyond programs of a few thousand lines. As a result, coarser but faster analyses are usually used [2, 10].

Scaling behavior and precision are intimately connected and in an ideal formalism, one can be traded for the other. Unfortunately, inclusion constraints over set-expressions do not provide enough control over this precision-efficiency tradeoff. As an example, consider Hindley-Milner type inference. The equality constraints arising in the formulation of algorithm $\mathcal{W}$ [8] can be solved as symmetric inclusion constraints using a standard inclusion constraint solver. This approach results however in an algorithm with cubic time complexity, instead of the nearly linear time algorithm based on unification [7].

A key to the efficiency of Hindley-Milner type inference is that types are terms. Terms have unique head constructors, whereas set expressions generally do not. This property—whether a quantity has a unique head constructor—is a prime determinant of the cost of solving type constraints. We have designed an extended inclusion constraint formalism in which values with unique head constructors can be mixed with more general sets. The extended formalism retains the generality of inclusion constraints and set-expressions, but gives control over where this generality is unneeded. Because inclusion constraints over types with unique head constructors can be solved nearly as efficiently as unification, constraints in our extended formalism can be solved more efficiently than pure inclusion constraints.

Not only does the new formalism provide fine control over the precision-efficiency tradeoff, it also enables novel analyses. The rest of this abstract briefly describes one such analysis: Uncaught exception inference for ML [9]. Our formulation of exception inference improves upon earlier work by providing precise information at a cost close to Hindley-Milner type inference.

2 Uncaught Exception Inference

We formulate our uncaught exception inference as an effect system [6] for Mini-ML, a typed lambda calculus with exception constructors and `raise` and `handle` expressions.

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

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.

The type language used has two kinds of types, **u**-types (with unique head constructors) and **s**-types (types based on set-expressions). It differs from Hindley-Milner types only in that exception constructors are differentiated within the type system. The standard ML type system assigns the type `exn` to each exception value. We refine the exception type by making `exn` a unary constructor of signature $\texttt{exn} : \mathbf{s} \rightarrow \mathbf{u}$. The signature specifies that `exn` produces a **u**-type and takes as argument an **s**-type. The argument type describes the set of exception constructors. For example, the `Match` exception is given the type `exn(Match)`, where the name `Match` is reused as a constant type constructor of kind **s**. Function types must also be refined to capture the set of exceptions that may be raised during an application. The function type constructor has the signature $\cdot \stackrel{\cdot}{\rightarrow} \cdot : \overline{\mathbf{u}}\ \mathbf{s}\ \mathbf{u} \rightarrow \mathbf{u}$, and

*This abstract is a short version of a full paper submitted for conference publication.

†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

function types are written $\tau_1 \xrightarrow{\sigma} \tau_2$. The first argument is a **u**-type describing the domain of the function, the second argument is an **s**-type describing the set of exceptions that may be raised during an application, and the third argument is again a **u**-type for the range. The kind for the domain is overlined $\overline{\mathbf{u}}$ to indicate that the function type constructor is contravariant in this field. Variance of type constructors is needed because we use inclusion constraints instead of equalities for type inference.

The type language is described by the following grammar (τ is used for **u**-types, σ for **s**-types):

$$\begin{array}{lcl} \tau & ::= & \alpha \mid b \mid \tau \xrightarrow{\sigma} \tau \mid \mathtt{exn}(\sigma) \\ \sigma & ::= & \epsilon \mid c \mid c(\tau) \mid \sigma \cap \sigma \mid \sigma \cup \sigma \mid \neg\{c_1, \ldots, c_n\} \end{array}$$

Type variables are written α or ϵ, depending on the kind. Base types $b \in B$ are constants for simplicity. 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 $\neg\{c_1, \ldots, c_n\}$ denotes the set of all values except values obtained by applying constructors c_i. Because exceptions can carry values, the two kinds of types are mutually recursive.

Figure 1 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 $\vdash_{\mathrm{p}} p : (\sigma, c, A)$ 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 of the pattern, which is used in forming the filter for unhandled exceptions. The rule for `handle` expressions makes use of the full expressive power of **s**-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.

A few examples illustrate the refined types. First consider the primitive `raise` in ML, which is used to raise an exception. Instead of making it a primitive, we could treat `raise` as a function with type $\mathtt{exn}(\epsilon) \xrightarrow{\epsilon} \alpha$. The type expresses the fact that applying `raise` to an exception of type $\mathtt{exn}(\epsilon)$ causes the observable effect ϵ.

Consider the function `substFail` that calls a function argument, and if the `Fail` exception is raised, raises the alternate exception `e`.

```
exception Fail
fun substFail f e =
              f () handle
                Fail => raise e
```

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

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

$$\frac{\begin{array}{l} A \vdash e_1 : \tau_1 \,!\, \sigma_1 \\ A \vdash e_2 : \tau_2 \,!\, \sigma_2 \\ \tau_1 \subseteq \tau_2 \xrightarrow{\epsilon} \alpha \end{array}}{A \vdash e_1\ e_2 : \alpha \,!\, \sigma_1 \cup \sigma_2 \cup \epsilon} \quad [\text{APP}]$$

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

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

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

$$\frac{\mathrm{typeof}(c) = \tau \longrightarrow \mathtt{exn}(c(\tau))}{\vdash_{\mathrm{p}} c(x) : (c(\tau), c, [x \mapsto \tau])} \quad [\text{PAPP}]$$

Figure 1: Type and exception inference rules.

The types inferred by our type system are

$$\begin{array}{rcl} \mathtt{f} & : & \mathtt{unit} \xrightarrow{\epsilon_1} \alpha \\ \mathtt{e} & : & \mathtt{exn}(\epsilon_2) \\ \mathtt{substFail} & : & (\mathtt{unit} \xrightarrow{\epsilon_1} \alpha) \longrightarrow \mathtt{exn}(\epsilon_2) \\ & & \xrightarrow{\epsilon_1 \cap \neg\{\mathtt{Fail}\} \cup \epsilon_2} \alpha \end{array}$$

The types illustrate the dependencies between the exceptions carried by the function argument `f`, the argument exception `e`, and the exceptions of `substFail`. Given a function $\mathtt{f} : \mathtt{unit} \xrightarrow{\epsilon_1} \alpha$, we know that the expression `f()` has type α and effect ϵ_1. The `handle` expression prevents the `Fail` exception from escaping the body of `substFail`, but contributes the exception ϵ_2 from argument `e`. Consequently, evaluating `substFail` can result in any exceptions from ϵ_2 or exceptions raised by the argument function, except `Fail`.

The theory needed to solve the inclusion constraints arising during type inference is described in a forthcoming paper.

3 Implementation

We have implemented the exception analysis on top of a generic constraint solver written in SML/NJ. The prototype can analyze core SML, which requires the following extensions:

- Let-polymorphism is handled as described in [1].
- 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 **s**-type parameter to capture these exceptions.
- ML has mutable references. We treat inclusion constraints between reference types as equalities.

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 3.9sec 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 [12]. Our running time however improves upon Yi's approach by three orders of magnitude.

4 Related Work

Effect systems [6] naturally contain a mixture of Hindley-Milner types and sets for effects. To our knowledge, all published algorithms for effect systems are based on equality constraints and solved using specialized unification to deal with sets [5, 11]. As a result, effect sets that are joined by dataflow paths have to be equal, an approximation that does not arise in our formulation.

Two earlier approaches to uncaught exception detection for ML are found in [3] and [12]. Guzmán and Suárez describe an extended type system for ML similar to, but less powerful than, the one presented here. Their approach is based on equality constraints, does not treat exceptions as first class values, and ignores value-carrying exceptions. Yi describes a collecting interpretation for estimating uncaught exceptions in ML. His analysis handles the entire ML language and is much finer grained than [3] or the system described here, but is also slow in practice.

References

[1] 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.

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

[3] 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.

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

[5] 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.

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

[7] David McAllester. Inferring Recursive Data Types. http://www.ai.mit.edu/people/dam/rectypes.ps.

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

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

[10] 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.

[11] 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, January 1994.

[12] 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.