Exception Analysis for Non-Strict Languages

Kevin Glynn*, Peter J. Stuckey, Martin Sulzmann† and Harald Søndergaard
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
{keving, pjs, sulzmann, harald}@cs.mu.oz.au

Abstract

In this paper we present the first exception analysis for a non-strict language. We augment a simply-typed functional language with exceptions, and show that we can define a type-based inference system to detect uncaught exceptions. We have implemented this exception analysis in the GHC compiler for Haskell, which has been recently extended with exceptions. We give empirical evidence that the analysis is practical.

Categories and Subject Descriptors

D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) languages; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—Control primitives

General Terms

Languages, Theory

Keywords

Exceptions, non-strict functional programming languages, type inference, effect systems, Boolean constraints

1 Introduction

Ever since their introduction in PL/I and Ada, exceptions have been recognised as a useful mechanism for dealing with abnormal results occurring during the execution of a program. Exceptions can make programs more robust, that is, able to recover from abnormal situations without halting, in a way that preserves program modularity.

Keywords

Exceptions, non-strict functional programming languages, type inference, effect systems, Boolean constraints

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.

Copyright 2002 ACM 1-58113-487-8/02/0010 ...$5.00

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.

Copyright 2002 ACM 1-58113-487-8/02/0010 ...$5.00
the program allows an appropriate \textit{Exception} type to be constructed. Our analysis can do this on the fly, but to simplify the presentation we assume that the exceptions in a program are already known.

Exception values are part of the denotation of all types (much as \(\bot\) is a part of the denotation of all pointed types). A primitive function, \textit{raise}, throws an exception. Since an exception can be of any type, we have a family of type-indexed \textit{raise} primitives:

\[
\text{raise}_0 :: \text{Exception} \to t
\]

For simplicity, in the rest of this paper we use the unadorned \textit{raise} statement instead of the appropriate type-correct instance.

We do not support first class exceptions nor exceptions with arguments. Instead, we insist that in a well-formed program, the argument to \textit{raise} must be a manifest (explicit) Exception constructor. Exception valued expressions can appear nowhere else.

We follow [14] and accept non-determinism in the exceptions being raised. Non-determinism is a consequence of not having a fixed evaluation order throughout the language. For example, which exception is raised in the following expression?

\[
\text{(raise BadArgument)} + \text{(raise DivideByZero)}
\]

Operationally, we will get the first exception encountered during evaluation, so the result depends upon which argument is evaluated first. We solve this problem by allowing the denotation of an expression to be a set of exceptions, and the expression above denotes the set \{\text{BadArgument}, \text{DivideByZero}\}. This allows an implementation to evaluate the arguments in any order.

Exceptions are caught by a new language construct, \textit{try}.

\[
\begin{align*}
\text{try } & \text{expr} \\
\text{match } & \text{excep\_name}_1 \to \text{expr}_1 \\
& \text{excep\_name}_2 \to \text{expr}_2 \\
& \ldots \\
& \text{excep\_name}_n \to \text{expr}_n
\end{align*}
\]

Informally, \textit{expr} is evaluated to weak head normal form. If that raised no exceptions then the result of \textit{expr} is the result of the \textit{try} expression as well. Otherwise one of the raised exceptions is chosen randomly. It will matches one of the \textit{exception patterns} then we evaluate and return that pattern’s right hand side expression. If the exception does not match any pattern then it is re-raised by the \textit{try} expression.

Operationally, of course, the implementation does not raise sets of exceptions. Instead, the first exception to be encountered is returned to the innermost \textit{try} statement, where it is matched against the patterns as above. However, this behaviour is consistent with the above described semantics.

Note that non-determinism of exceptions can leak into the language proper. For example, the following expression can evaluate to either 0 or 1, and the semantics should reflect this.

\[
\begin{align*}
\text{try } & \text{(raise BadArgument)} + \text{(raise DivideByZero)} \\
\text{match } & \text{BadArgument} \to 0 \\
& \text{DivideByZero} \to 1
\end{align*}
\]

The main contribution of this paper is a novel exception analysis for a non-strict language. It is presented as a type-based inference system which makes novel use of Boolean constraints. Apart from supporting well-known algorithms for implementation, the use of Boolean constraints allows an elegant formulation for the analysis of \textit{try} expressions. It also makes the extension to a polymorphically typed language easier (we do not show the extension here, but it carries over from our Binding-time Analysis [8]). We give a formal correctness result and show some empirical results of an implementation of our analysis in GHC.

In Section 2 we introduce a simply-typed, higher-order functional language extended with constructs for raising and catching exceptions. Section 3 presents our Exception Logic including a formal correctness result. An inference algorithm is described in Section 4. In Section 5 we consider our implementation in the GHC compiler and present some empirical analysis results. Related work is discussed in Section 6. Section 7 concludes.

## 2 Source Language

Types and expressions are given by the grammar

\[
\begin{align*}
\text{Types } t & ::= \text{Int} \mid \text{Bool} \mid \text{Exn} \mid t \to t \\
\text{Expressions } e & ::= x \mid \lambda x . e \mid e \mid \text{fix} \cdot x \cdot \text{in} \cdot e \\
& \quad \text{if } e \text{ then } e \text{ else } e \\
& \quad \text{let } x = e \text{ in } e \\
& \quad \text{raise } e_1 | e_2 | \ldots | e_n \\
\text{Exceptions } ex & ::= Ex_1 | \ldots | Ex_n
\end{align*}
\]

The language has a standard syntax extended with primitives to raise and catch exceptions. There are a fixed number of exception constructors, \(Ex_1, \ldots, Ex_n\). We denote the set of exception constructors \(\mathcal{E}\). We use the notation \([e_1 \to e_i]_{i \in I}\) to indicate a sequence of the form

\[
[e_1 \to e_1, \ldots, e_m \to e_m].
\]

(\(I\) is a sequence 1 \(\ldots\) \(m\) for some positive integer \(m\)). Each \(Ex_i\) is a pattern exception constructor matching an element of \(Ex_1, \ldots, Ex_n\).

The meaning of \textit{raise} and \textit{try} has been discussed in the previous section. Our \textit{try} statement is representative of those found in most other languages. Peyton Jones et al. [14] have a slightly different catch statement (\textit{getException}), but this is easy to write using our \textit{try} statement.

The type language is also standard and the type logic is not given here. As we shall see, the standard denotation of each type is extended to include all possible exceptions. For the \textit{try} expression, the type of \(e\) and all the \(e_i\)’s must be the same, and this is also the type of the \textit{try} expression.

We can extend the exception escape analysis to a language with let-style polymorphism using a technique from our previous work [8]. Then \textit{raise} becomes a primitive in the initial environment of type \(\forall a.\text{Exn} \to a\). Our implementation uses this approach and does handle let-polymorphism. However, in the interest of clarity we shall omit let-polymorphism in the following.

The addition of exceptions to a non-strict language has considerable ramifications. As noted in Section 1 determinism is the most prominent victim, and we discuss other consequences in Section 6. Peyton Jones et al. [14] limit the semantic impact of non-determinism by putting it in the IO monad. This is just sweeping the non-determinism under the carpet, and there is no difference between their exception semantics and ours. All the examples in this paper can be written (with small modifications) in their system and will have the same behaviour. For the purposes of this paper we consider that the (entire) computation is non-deterministic, not just parts in the IO monad. This simplifies the semantics, leading to a simple correctness proof for the analysis.
The meaning of a closed expression is a finite, non-empty set of possible values. In addition to normal values we also have abnormal values. We denote by $V$ the least solution of the following equation:

$$V = B_\bot + \mathcal{E}_\bot + V \rightarrow V$$

Base domains $B$, denoting values of type Int and Bool, are ordered by equality, function spaces have pointwise ordering, and the $+$ is coalesced sum. The partial ordering $\preceq$ is thus defined as lifted to finite non-empty sets by defining (Smyth Ordering)

$$A \sqsubseteq B \text{ iff } \forall y \in B, \exists x \in A . \ x \leq y$$

With this, there is no distinction between possible and definite non-termination, which is adequate for our purpose. (On sets of function values, the Smyth ordering is in fact a pre-order, so strictly we should consider equivalence classes with respect to the induced equivalence relation, but the details are of little interest here.)

Figure 1 gives a denotational definition of the language. The semantics is conservative in the sense that it forces the normal-order evaluation of a program. An environment $\rho$ maps a free variable to a set of possible denotations. As an example of an initial environment, here is a suitable semantic equation for the built-in operator $\mathbf{+}$.

Note that the arguments to $\rho_{\text{init}}(\mathbf{+})$ are sets of denotations, as is the result.

$$\rho_{\text{init}}(\mathbf{+}) = \lambda x. \lambda y. \{ \bot \mid \bot \in (x \cup y) \} \cup (\mathcal{E} \cap (x \cup y))$$

$$\cup \{ i + j \mid i \in x \land j \in y \land \mathcal{E} \}$$

A raise expression has only one (abnormal) value. A lambda expression is always considered a normal value, but in general, a value of function type may have an abnormal value, witness

$$\text{if } (\text{raise } E x) \text{ then } + \text{ else } +$$

The value of this expression is $E x \bot$, and so is the result of applying the function to some argument. Similarly no distinction is made between $\bot$ and $\lambda x. \bot$, although in Haskell they can be distinguished by the seq operator.

## 3 Exception Logic

The structure of exception types reflects the structure of the underlying type system.

### Annotations

$$b ::= \delta$$

### Exception Types

$$\tau ::= b | \tau \mathrel{\downarrow} \tau$$

### Type Schemes

$$\eta ::= \tau | \forall \delta. C \Rightarrow \tau$$

Each possible exception $ex \in \mathcal{E}$ is allocated a unique variable $\delta_{ex}$.

We write $\mathcal{E} = \mathcal{E}_\bot$ to refer to the free variables in an exception type $\eta$. The constraint component $C$ will be described shortly.

We are often interested in the topmost annotation of a type. The function $\| \cdot \|$ returns a type’s topmost annotation:

$$\|\delta\| = \delta$$

$$\|\tau\| = b$$

Exception variable relationships are captured through Boolean constraints. A variable $\delta$ stands for a Boolean variable. We denote by $true$ the always true constraint. If variable $\delta_2$ represents at least the exceptions that variable $\delta_1$ represents then we express that by Boolean implication $\delta_1 \rightarrow \delta_2$. For example, if an expression of type $\delta$ can raise exception $Ex3$, then this is expressed by a constraint of $\delta_{Ex3} \rightarrow \delta$. In addition to implication, constraints may contain conjunction and existential quantification.

### 2IMP

$$C ::= \top \mid \delta \rightarrow \delta \mid C \land C \mid \exists \delta . C$$

We allow $\exists \delta . C$ as shorthand for $\exists \delta_1 . \exists \delta_2 . \ldots . \exists \delta_n . C$. We write $C_1 \models C_2$ to denote model-theoretic entailment among constraints $C_1$ and $C_2$. It is convenient to define a structural ordering, $\leq s$ between exception types. These translate via function $\llbracket \cdot \rrbracket_{\text{sub}}$ to a conjunction of primitive constraints:

$$\llbracket \delta_1 \leq s \delta_2 \rrbracket_{\text{sub}} = \delta_1 \rightarrow \delta_2$$

$$\llbracket \tau_1 \downarrow s \tau_1 \leq s \tau_2 \downarrow s \tau_2 \rrbracket_{\text{sub}} = \delta_1 \rightarrow \delta_2 \land \llbracket \tau_1 \leq s \tau_1 \rrbracket_{\text{sub}} \land \llbracket \tau_2 \leq s \tau_2 \rrbracket_{\text{sub}}$$

To relate an expression’s valid exception type to its underlying type, we employ a “shape” system. This system ensures that an expression’s exception type has the same shape as its underlying type. The judgement $\eta \uparrow_s t$ states that $\eta$ is well-shaped with respect to underlying type $t$. A judgement $\eta \uparrow_s t$ is valid if it can be derived by the following shape rules:

### (Base)

$$\delta \vdash \top, \text{ Bool} \vdash \delta, \text{ Int} \vdash \delta \mapsto \text{Exn}$$

### (Arrow)

$$\tau \vdash \delta \rightarrow \tau \vdash \tau_1 \vdash \tau_1 \rightarrow \tau_2$$

### (∀)

$$\forall \delta. D \Rightarrow \tau \vdash \tau \rightarrow t$$

We require a valid exception type to be well-shaped.

### 3.1 The Logic

We now present an Exception Logic which identifies escaping exceptions. Note that we maintain a phase distinction between the process of type inference (of underlying types) and exception analysis. We will always assume that expressions are well-typed and fully type-annotated.

Exception Typing judgements are of the form $C, \Gamma \vdash (e :: t) : \eta$ where $C$ is the global constraint, $\Gamma$ contains the exception types of free variables and primitive functions in $e$, $(e :: t)$ is an expression, type-annotated with its underlying type, and $\eta$ is an exception type scheme. We denote by $\Gamma_s$ the type environment obtained from $\Gamma$ by excluding the variable $x$. We denote by $\Gamma_s(x :: t) : \eta$ the extension of environment $\Gamma_s$ with $(x :: t) : \eta$. We will always assume that $(e :: \text{Exn$_s$}) : \delta_{ex} \in \Gamma_s$ for each $ex \in E$.

Figure 2 defines the typing rules. We maintain the invariant that types in the environment and types of expressions are “shape” correct. In rule (Sub) we translate structural constraints into Boolean constraints via the $\llbracket \cdot \rrbracket_{\text{sub}}$ function. Rules (VI) and (III) restrict quantification to the “free” variables. Note that $\delta_s \in \mathcal{E}$ for each $ex \in \mathcal{E}$, therefore quantification over $\delta_{Ex1}, \ldots , \delta_{Exn}$ is prohibited. In rule (App) we require that every exception arising from evaluating the function component flows to the result of the application, that is, $\delta \rightarrow \tau_2$. Note that we do not have the constraint $\tau_1 \rightarrow \tau_2$, which would be necessary for a strict language. Rule (Try) employs the power of Boolean constraints to “catch” matching exceptions by existentially quantifying “away” (the $\exists \delta_{Ex} C_0$ part) information about exceptions raised in the sub-expression that would be caught by the try. Only exceptions that flow to the top-most annotation of expression $e$’s type will be caught. Exceptions that flow to other parts of $e$’s type must be preserved. This is achieved through the constraint
\[
\begin{align*}
\llbracket x \rrbracket \rho & = \rho(x) \\
\llbracket \text{raise } e \rrbracket \rho & = \{ e \} \\
\llbracket \lambda x . e \rrbracket \rho & = \{ \lambda y . [e]_\rho[x := y] \} \text{ where } y \text{ fresh} \\
\llbracket e \rrbracket \rho & = \bigcup \{ ([e]_\rho) r \mid r \in [e']_\rho \land [e]_\rho \not\in \mathcal{E}_t \} \cup (\mathcal{E}_t \cap [e]_\rho) \\
\llbracket \text{try } e \text{ match } [x_i \to e_i]_{i \in I} \rrbracket \rho & = \bigcup_{i \in I} \{ [e_i]_\rho \mid e_i \in [e]_\rho \} \cup ([e]_\rho \setminus \bigcup_{i \in I} [e_i]_\rho) \\
\llbracket \text{let } x = e \text{ in } e' \rrbracket \rho & = [e']_\rho[x := [e]_\rho] \\
\llbracket \text{fix } e \rrbracket \rho & = \text{fix}(\lambda x . [e]_\rho[x := v]) \\
\llbracket \text{if } e \text{ then } e_1 \text{ else } e_2 \rrbracket \rho & = (\mathcal{E}_t \cap [e]_\rho) \cup \{ v_1 \mid v_1 \in [e_1]_\rho \land \text{True} \in [e]_\rho \} \cup \{ v_2 \mid v_2 \in [e_2]_\rho \land \text{False} \in [e]_\rho \}
\end{align*}
\]

Figure 1. Semantics of the Source Language

![](https://example.com/figure1.png)

\[
\begin{align*}
\text{(Raise)} & \quad \frac{\tau \vdash_s t \quad C \models \delta \xrightarrow{\text{Abs}} \tau}{C, \Gamma \vdash ((\text{raise}_0 : \text{Exn} \rightarrow t) (\text{ex} : \text{Exn}) :: t) : \tau} \\
\text{(Fix)} & \quad \frac{C, \Gamma, (x :: t) : \eta \vdash (e :: t) : \eta \quad \eta \vdash_s t}{C, \Gamma \vdash ((\text{fix} (\lambda x :: t . (e :: t)) :: t) :: t) : \eta} \\
\text{(Var)} & \quad \frac{\frac{C, \Gamma \vdash (x :: t) : \eta \quad (x :: t) : \eta \in \Gamma}{C, \Gamma \vdash (x :: t) : \eta}} \\
\text{(Sub)} & \quad \frac{C, \Gamma \vdash (e :: t) : \tau_2 \quad \tau_1 \vdash_s t \quad C \models [\tau_2 \leq_s \tau_1]_{\text{sub}}}{C, \Gamma \vdash (e :: t) : \tau_1} \\
\text{(App)} & \quad \frac{C, \Gamma \vdash (e :: t) : \tau \quad C, \Gamma \vdash ((e_1 :: t_1) \rightarrow t_2) : \tau \quad C, \Gamma \vdash (e_2 :: t_2) :: t_2}{C, \Gamma \vdash ((e_1 :: t_1) \rightarrow t_2) (e_2 :: t_2) :: t_2} \\
\text{(Let)} & \quad \frac{C, \Gamma \vdash (e_1 :: t_1) : \eta \quad C, \Gamma \vdash (e_2 :: t_2) : \tau \quad C, \Gamma \vdash (\text{let } x = (e_1 :: t_1) \text{ in } e_2 :: t_2) :: t}{C, \Gamma \vdash ((x :: t) : \eta \quad \eta \vdash_s t) \quad C, \Gamma \vdash (x :: t) : \tau} \\
\text{(If)} & \quad \frac{C, \Gamma \vdash (e_1 :: \text{Bool}) \in \delta \quad C, \Gamma \vdash (e_2 :: t) : \tau \quad C, \Gamma \vdash (e_3 :: t) : \tau \quad C \models \delta \rightarrow \tau}{C, \Gamma \vdash ((\text{if } e_1 :: \text{Bool} \text{ then } (e_2 :: t) \text{ else } (e_3 :: t)) :: t) : \tau} \\
\text{(Try)} & \quad \frac{C_0, \Gamma \vdash (e :: t) : \tau_0 \quad C_0, \Gamma \vdash (e :: t) : \tau \quad \tau \vdash_s t \quad C_0 = [\tau_0 \leq_s \tau]_{\text{sub}} \land C_0'}{C, \Gamma \vdash ((\text{try } (e :: t) \text{ match } [(e_i :: \text{Exn}) \rightarrow (e_i :: t)]_{i \in I} :: t) : \tau}
\end{align*}
\]

Figure 2. Exception Logic
Example 1

\[
C_0 = \delta_{E_{x_1}} \rightarrow \delta_0 \land \delta_{E_{x_2}} \rightarrow \delta_0
\]

\[
C_0, \Gamma \vdash \text{(raise } E_{x_1}) : \delta_0
\]

\[
C_0, \Gamma \vdash \text{(raise } E_{x_2}) : \delta_0
\]

\[
\delta_{E_{x_1}} \rightarrow \delta_1, \Gamma \vdash \text{(raise } E_{x_1}) : \delta_1
\]

\[
\delta_1 \rightarrow \delta' \land \delta_{E_{x_2}} \rightarrow \delta_1 \land \exists \delta_0, C_0 \land \exists \delta_{E_{x_1}}, C_0 \land \delta_0 \rightarrow \delta', \Gamma \vdash \text{(try ...)} : \delta'
\]

\[
\exists \delta_0, \exists \delta_1 (\delta_1 \rightarrow \delta' \land \delta_{E_{x_1}} \rightarrow \delta_1 \land \exists \delta_0, C_0 \land \exists \delta_{E_{x_1}}, C_0 \land \delta_0 \rightarrow \delta'), \Gamma \vdash \text{(try ...)} : \delta'
\]

Example 2

\[
C_0 = \delta_{E_{x_1}} \rightarrow \delta_2
\]

\[
C_0, \Gamma \vdash \lambda \cdot \text{raise } E_{x_1} : \delta_1 \rightarrow \delta_2, \quad \text{true}, \Gamma \vdash \lambda \cdot \text{true} : \delta_3 \rightarrow \delta_3
\]

\[
\delta_{E_{x_1}} \rightarrow \delta_2 \land \llbracket \delta_1 \rightarrow \delta_2 \rrbracket \otimes \llbracket \delta_3 \rightarrow \delta_3 \rrbracket \llbracket \delta_{E_{x_1}} \otimes \delta_{E_{x_1}} \rrbracket \llbracket \delta_1 \rightarrow \delta_2 \rrbracket \llbracket \delta_3 \rightarrow \delta_3 \rrbracket \llbracket \delta_{E_{x_1}} \otimes \delta_{E_{x_1}} \rrbracket
\]

\[
\exists \delta_0, \exists \delta_1 (\delta_1 \rightarrow \delta' \land \delta_{E_{x_1}} \rightarrow \delta_1 \land \exists \delta_0, C_0 \land \exists \delta_{E_{x_1}}, C_0 \land \delta_0 \rightarrow \delta'), \Gamma \vdash \text{(try ...)} : \delta'
\]

\[
\exists \delta_0, \exists \delta_1 (\delta_1 \rightarrow \delta' \land \delta_{E_{x_1}} \rightarrow \delta_1 \land \exists \delta_0, C_0 \land \exists \delta_{E_{x_1}}, C_0 \land \delta_0 \rightarrow \delta'), \Gamma \vdash \text{(try ...)} : \delta'
\]

Figure 3. Example Type Derivations

(\exists \tau_0, C_0) \land (\exists E_{x_2}, C_0).

Lemma 4 in the appendix formally states that this constraint only removes primitive constraints between \(\delta_{E_{x_1}}\) and \(\tau_0\).

Example 1. Consider the expression

\[
\text{try (if } x > y \text{ then raise } E_{x_1} \text{ else raise } E_{x_2})
\]

\[
\text{match } E_{x_1} \rightarrow \text{raise } E_{x_2}
\]

We can show that exceptions \(E_{x_2}\) and \(E_{x_3}\) can escape this expression, using the derivation in Figure 3. For simplicity, we have omitted underlying types and uninteresting deductions. Note that \(\exists \delta_0, C_0 \land \exists \delta_{E_{x_1}}, C_0 = \delta_{E_{x_2}} \rightarrow \delta_0\). The last step of the derivation is a simplification of the constraints in the previous deduction.

Example 2. Consider another expression

\[
\text{try } \lambda \cdot \text{raise } E_{x_1}
\]

\[
\text{match } E_{x_1} \rightarrow \lambda \cdot \text{true}
\]

Note that exception \(E_{x_1}\) which appears in the body of a function in the try statement won’t be caught. A derivation can be found in Figure 3. In the (Try) step we find that \((\exists \delta_0, \delta_{E_{x_1}} \rightarrow \delta_2) \land (\exists \delta_{E_{x_2}}, \delta_{E_{x_1}} \rightarrow \delta_2) = \delta_{E_{x_1}} \rightarrow \delta_2\). Therefore, in the conclusion we have that \(\delta_{E_{x_1}} \rightarrow \delta_0\) as desired.

Our Exception Logic preserves well-shapedness. Let \(\Gamma\) be an exception type environment such that for each \((x : t) : \eta \in \Gamma\) we have that \(\eta \vdash _x t\) and \(C, \Gamma \vdash \eta : \eta' \sigma\). The Exception Logic is a conservative extension of the underlying type system. That is, all expressions that can be typed in the underlying type system can be given a valid Exception Type by our logic.

3.2 Correctness of Logic

In this section we prove the correctness of our Exception Logic. We show that any exceptions which appear in the denotation of an expression are predicted by any valid derivation of that expression’s Exception Type in our logic.

In a first step, we enrich the term language of annotations by introducing set types:

**Annotations**

\[
b ::= \delta \setminus E
\]

where \(E \subseteq \mathcal{E}\). Set types represent solutions of constraints in 2IMP.

In addition, we extend our constraint language by introducing a subset relation among set types.

\[
C ::= \text{true} \setminus \delta \setminus C \cup \exists \mathcal{E} C \cup E \subseteq \mathcal{E}
\]

Let \(C\) be a constraint and let \(\phi\) be a closing substitution (i.e., domain(\(\phi\) = \(\text{fv}(C)\)), mapping annotation variables to set types. Then we obtain the instantiated constraint \(\phi C\) by the following transformation function:

\[
\llbracket \phi(\delta_1 \rightarrow \delta_2) \rrbracket_{\text{con}} = \phi(\llbracket \delta_1 \rrbracket_{\text{con}}) \subseteq \phi(\llbracket \delta_2 \rrbracket_{\text{con}})
\]

\[
\llbracket \phi(E_1 \subseteq E_2) \rrbracket_{\text{con}} = E_1 \subseteq E_2
\]

\[
\llbracket \phi(C_1 \land C_2) \rrbracket_{\text{con}} = \llbracket \phi C_1 \rrbracket_{\text{con}} \land \llbracket \phi C_2 \rrbracket_{\text{con}}
\]

Without loss of generality we assume that constraints are free of existential quantifiers.

We say a closing substitution \(\phi\) is a solution to a constraint \(C\) iff \(\phi\) is a mapping from annotation variables to set types, and \(\models [ \llbracket \phi C \rrbracket ]_{\text{con}}\). We often write \(\models C\) instead of \(\models [ \llbracket \phi C \rrbracket ]_{\text{con}}\). We will always assume that substitutions \(\phi\) map variables to set types such that \(\phi(\delta_{x_0}) = \{e x\}\) for \(e x \in \mathcal{E}\).

Example 3. The substitution

\[
[ E_{x_1} ]_{\delta_{E_{x_1}}} / [ E_{x_2} ]_{\delta_{E_{x_2}}} / [ E_{x_3} ]_{\delta_{E_{x_3}}} \otimes \delta_{E_{x_1}} \otimes \delta_{E_{x_2}} \otimes \delta_{E_{x_3}}
\]

is a solution to the constraint

\[
(\delta_{E_{x_1}} \rightarrow \delta_2) \land (\delta_{E_{x_2}} \rightarrow \delta_3) \land (\delta_{E_{x_3}} \rightarrow \delta_3).
\]

Constraint solutions allow us to give meaning to exception types. Our exception types identify groups of denotations with similar exception raising behaviour. We assume that substitutions operate on types as usual. This extends naturally to type environments. Then,
in the above example \( \phi(\delta_1, \delta_2, \delta_3) \) is the exception type of any functional expression that can only raise exception \( Ex_1 \) and when applied can raise only exceptions \( Ex_2, Ex_3 \) and \( Ex_4 \).

We say a type \( \tau \) is a monotype iff \( \phi(\tau) = 0 \). A type scheme \( \eta \) is closed iff \( \phi(\eta) = 0 \).

We are now in a position to give a meaning to types. The meaning function \( \llbracket \cdot \rrbracket \) maps monotypes and closed type schemes to ideals, i.e., non-empty, downward-closed and limit-closed subsets of \( V \).

\[
\begin{align*}
\llbracket E \rrbracket & = E \cup (V \setminus E) \\
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket & = \{ v \in V \mid (v \in E) \lor (v \llbracket \tau_1 \rrbracket \subseteq \llbracket \tau_2 \rrbracket) \} \\
\llbracket \exists \delta. C \Rightarrow \tau \rrbracket & = \{ \eta \subseteq \{ \{ \phi(\tau) \mid \phi(\tau) = 0 \} \} \} \\
\end{align*}
\]

**Lemma 1.** Let \( \eta \) be a closed exception type scheme, then \( \llbracket \eta \rrbracket \) is an ideal.

Let \( d \) be a value in \( V \) and \( \eta \) a type scheme. We write \( d \models^+ \eta \) iff \( d \in [\llbracket \eta \rrbracket] \).

This relationship can be extended to sets of denotations. Let \( ds \) be a set of values in \( V \). We write \( ds \models^+ \eta \) iff for all \( d \in ds \) we have that \( d \models^+ \eta \).

We let \( \rho \) denote variable environments, mapping variables to sets of values in \( V \). Let \( \Gamma \) be a type environment. We say \( \rho \) models \( \Gamma \), written \( \rho \models^+ \Gamma \), iff for all \( x : \eta \in \Gamma \), \( \rho(x) \models^+ \eta \).

In our system a program is evaluated in the context of an initial environment which provides mechanisms for primitive operations which are not definable within the language, e.g., the addition of two integers. The initial type environment describing these primitive operations must correctly approximate the behaviour of the real primitives in the sense defined above.

**Theorem 1 (Correctness).** Let \( C, \Gamma \vdash (\cdot : \cdot) : \eta \) be a valid typing judgement, \( \Phi \) be a substitution and \( \rho \) a variable environment such that \( \rho \models^+ \Phi \), \( \Phi \Gamma \) is closed, and \( \rho \models^+ \Phi\Gamma \). Then \( [\cdot : \cdot] \rho \models^+ \Phi \eta \).

A proof of this theorem is in the Appendix.

We extend function \( \models^+ \) which returns a type's topmost annotation as follows: \( [\{Ex_1, \ldots, Ex_n\}] = \{Ex_1, \ldots, Ex_n\} \).

**Corollary 1.** Let \( C, \Gamma \vdash (\cdot : \cdot) : \tau \) be a valid typing judgement, \( \Phi \) be a substitution, \( \rho \in \mathcal{E} \) be an exception and \( \delta \) be a variable environment such that \( \models^+ \Phi \), \( \Phi \Gamma \) is closed, \( \rho \models^+ \Phi\Gamma \) and \( \exists x \in \{\cdot\} \rho \). Then, we have that \( \rho(\delta) \subseteq [\rho(\Phi\delta)] \).

### 4 Exception Type Inference

We employ a standard constraint-based type inference process \([8, 7]\). Constraints generated are in 2IMP which allows for efficient constraint manipulation.

We assume that we are given a well-typed program where each subexpression is annotated with its underlying type. Type inference computes the missing exception-type information. The inference algorithm shown in Figure 4, is formulated as a deduction system over clauses of the form

\[ \Gamma, (\cdot : \cdot) \vdash_{\text{Inf}} (C, \tau) \]

with a type environment \( \Gamma \) and a type-annotated expression \( (\cdot : \cdot) \) as input, and a constraint \( C \) and a type \( \tau \) as output. In the inference rules we use \( \tau \models^+ t \) to denote the creation of a fresh exception type \( \tau \) of shape \( t \).

All inference rules are syntax-directed except rule (\( \exists \) Intro). We assume that rule (\( \exists \) Intro) is applied aggressively, so that useless variables do not appear in the resulting constraints.

In rule (Let) we integrate quantification over free variables. We define a generalisation function giving the generalised type scheme and the generalised constraint. Let \( \delta \) be a constraint, \( \Gamma \) a type environment and \( \tau \) a type. Then \( \text{gen}(C, \Gamma, \tau) = (\exists \delta. C, \forall \delta. C \Rightarrow \tau) \) where \( \delta = \text{fv}(C, \tau) \setminus \text{fv}(\Gamma) \). Note that \( \text{fv}(\Gamma) \) only returns the free variables of exception types. We could be more efficient by pushing in only the affected constraints. Note that we assume \((\cdot : Ex) \models^+ \delta x \in \Gamma \) for each \( x \in E \). Therefore, quantification over \( \delta_{Ex_1}, \ldots, \delta_{Ex_4} \) is prohibited in rules (Let) and (\( \exists \) Intro).

In rule (Fix) we perform a Kleene-Mycroft iteration until a fixed point is found. We define \((\forall \delta. C \Rightarrow \tau) \models^+ (\forall \delta'. C' \Rightarrow \tau') \) iff \( C' = \exists \delta'. (C \land [\llbracket \tau \leq \tau' \rrbracket]_{\text{sub}}) \) where w.l.o.g. there are no name clashes between \( \delta \) and \( \delta' \). The fixed point operator \( \delta \) takes an environment and a type-annotated expression as input and returns a constraint and a type as output.

\[
\begin{align*}
\delta & = \text{fix}(C, \tau) \\
\delta & = \text{fix}(\Gamma, (\cdot : \cdot), \cdot, \cdot) : \eta_i < \eta_{i+1} \\
& = (C, \tau) \quad \text{if } \eta_i = \eta_{i+1}
\end{align*}
\]

where \( \Gamma, (\cdot : \cdot) : \eta_i, e : \cdot \vdash_{\text{Inf}} (C, \tau) \) and \( (\cdot : \cdot) : \eta_{i+1} \). Note that the sequence \( \eta_0 \leq \ldots \leq \eta_{n} \) is necessarily finite. The quantifier is over annotation variables only, which have a finite number of instances.

Soundness states that every deduction in the inference system is a valid deduction in the logical system.

**Theorem 2 (Soundness of Inference).** Let \( \Gamma, (\cdot : \cdot) \vdash_{\text{Inf}} (C, \tau) \). Then \( \Gamma \models (\cdot : \cdot) : \tau \).

Completeness states that every deduction derivable in the logical system is subsumed by a deduction in the inference system.

**Theorem 3 (Completeness of Inference).** Assume that

- \( C, \Gamma \vdash (\cdot : \cdot) : \forall \delta_1. C_1 \Rightarrow \tau_1 \), and
- \( \Gamma, (\cdot : \cdot) \vdash_{\text{Inf}} (C_2, \tau_2) \)

and let \( \delta_2 = \text{fix}(C_2, \tau_2) \setminus \text{fv}(\Gamma) \). Then

\( C \land C_1 \models (\exists \delta_2, (C_2 \land [\tau_2 \leq \tau_1]_{\text{sub}}) \).

It is not too difficult to perform a simplistic complexity analysis of the inference algorithm. If \( n \) is the size of the (underlying) type annotated program, then the number of annotation variables is \( O(n) \), and the size of any constraint is \( O(n^2) \). The basic constraint operations are; conjunction, which is linear in the sum of its arguments hence \( O(n^2) \); and projection (existential quantification) which naively creates \( O(n^2) \) constraints eliminating one variable, and hence is \( O(n^3) \). In any fixed point calculation there can be at most \( O(n^2) \) iterations, since each iteration must add one new constraint, otherwise we have reached a fixed point. By restarting intermediate fixed point calculations from the previously computed value, we can ensure that each intermediate fixed point calculation also involves at most \( O(n^2) \) steps in its lifetime. (This is the “accelerated Kleene-Mycroft iteration” of [2]). In total we have \( O(n) \) operations (even assuming aggressive use of the (\( \exists \) Intro) rule), each at
Figure 4. Exception Type Inference
worst $O(n^3)$, performed at most $O(n^2)$ times for a total complexity of $O(n^6)$.

After exception type inference, a compiler will need to use an entailment operation to extract information about escaping exceptions. However, entailment is $O(n^2)$, so the total cost is dominated by the cost of inference.

5 Evaluation Results

We have an implementation of our exception analysis for GHC. GHC desugars a well-typed program to an explicitly typed, System F style functional language Core. Core is very similar to the source language used in this paper. In addition to the analysis described here our implementation has precise support for polymorphism and structured, recursive data.

We have used the constraint solver library from our previous work on strictness analysis [7]. The solver is written in C and statically linked into the compiler. The constraints are represented by a directed graph, variables are nodes and an implication is a connection in the graph.

To evaluate the practical effectiveness of our inference algorithm we have run our analysis on the programs in the spectral component of the nafib suite. The nafib suite is available from the GHC implementers. Programs in the spectral component are mainly small, key components of real programs, and consists of almost 100 Haskell modules.

The programs raise exceptions both directly by calls to the Haskell error function and indirectly by compiler inserted error calls for pattern-match failures.

None of the programs catch exceptions, but we argue that the try construct can be implemented efficiently and always reduces the size of constraints. The constraint $(\exists \tau_0.C_0) \land (\exists \delta_0.C_0)$ from the (Try) rule can be implemented without copying $C_0$ by simply removing any direct connections from $\delta_0$ to $\tau_0$.

Overall, for the 100 modules in the spectral suite the average cost of our analysis was 28% of the compilation time without our analysis.

In Figure 5 we show analysis times for the 25 spectral modules that take GHC over 5 seconds to compile. The Lines column shows lines of code. Note the excessive time for the Main module in the nucleic2 suite. Our analysis has a fine-grained treatment of structured data. The data structures used in nucleic2 are complex, causing the constraints and $\delta$s generated for intermediate expressions to contain a large number of variables. A lot of the code in nucleic2’s Main module consists of building and destructing data with such types. Such code can be handled efficiently by reducing the precision of the analysis. A detailed investigation is left as future work.

6 Related Work and Discussion

Previous work [13, 4, 16, 9] on exception analysis for ML-style languages either employed a type and effect system [10] or an abstract interpretation based approach to approximate the set of exceptions that can be raised during an expression’s evaluation. They all share the assumption that the language is strict, whereas we introduce for the first time an exception analysis for a non-strict language.

The analysis of Leroy and Pessaux [13] uses a type and effect system based on equality constraints and refinement types. Equality constraints allow for an efficient solving of exception constraints, but equality constraints might cause loss of precision in some cases.

Consider the following example

\[
\begin{aligned}
  \text{let } id &= \lambda x.\mathit{in} \\
  \text{let } app0 &= \lambda f.\{(0)\} + (\try f 0) \text{ match } \mathit{NotReally} \rightarrow 0 \text{ in} \\
  \text{let res } &= \text{app0 id in} \\
\end{aligned}
\]

In the type and effect system the try expression in app0’s right hand side ‘inflicts’ the deduced type for argument $f$. It is given the exception type of a function that can raise exception NotReally. Since $f$ is required for the result of app0, this further injects app0’s type. Now any use of app0, such as the innocent application to id in the definition of res, will have exception NotReally escaping. The problem is caused by the use of unification constraints which force equality.

In this situation we use an inequality constraint $(\leq \tau)$.

On the other hand, there are also examples where Leroy and Pessaux’s use of refinement types allows for a result which is more precise than that which we obtain. Moreover, they combine their analysis with a data flow analysis, which allows them to give useful results for programs using first class exceptions and exceptions that carry arguments.

A more precise analysis based on inclusion constraints over set-expressions is considered by Fähndrich and Aiken. Their approach relies on a parametrised formalism that combines inclusion constraints over terms and sets [3].

Yi [16] describes an exception analysis based on abstract interpretation. This analysis is very fine-grained and there is an associated high cost and high precision.

Our development and implementation of this exception analysis has gained significantly from previous experiences with a binding-time analysis. Exception and binding-time analysis are essentially forms of control-flow analysis. The analyses therefore share the same constraint domain 2IMP. In [8], we experimented with more powerful Boolean domains. For example, if we allow for constraints of the form $(\delta \rightarrow \delta') \rightarrow C$ we are able to give a more precise description of the (Try) rule.

\[
\begin{aligned}
C_0, \Gamma \vdash (e \vdash t) : \tau_0 \\
C_0, \Gamma \vdash (e_1 \vdash t) : \tau \quad t \vdash_\mathcal{S} \tau \\
C'_0 = (\delta_{ex} \rightarrow [\tau_0]) \rightarrow ([\tau, \leq \tau]_{\mathcal{S}} \land C_0) \\
C = C'_0 \land \ldots \land C'_n \land (\exists \delta_0.C_0) \land (\exists \delta_{ex}.C_0) \land [\tau_{ex} \leq \tau]_{\mathcal{S}} \\
C, \Gamma \vdash (\try (e \vdash t) \text{ match } [ex_{id} \leftarrow (e_1 \vdash t)_{i \leftarrow \mathcal{S}}] : \tau)
\end{aligned}
\]

The result of the branches will affect the final result only if the corresponding exception will actually be raised. In our experience [8], the gain of a more precise analysis does not justify the cost of slower analysis times.

In Java, ML and Haskell exceptions are first class values, that is, they can be passed to and returned from functions, stored in data structures, passed to polymorphic functions and so on. Consider the following function:

\[
\begin{aligned}
\text{anyButC } ::& \quad \texttt{Exn \rightarrow Int} \\
\text{anyButC ex} &= \text{try (raise ex)} \\
\text{match ExC } \rightarrow 1
\end{aligned}
\]

We would like to give this function a type that says it raises any exception in the argument except ExC. In addition, the argument may evaluate to an exceptional value and ExC must be stripped from this too.
<table>
<thead>
<tr>
<th>Program</th>
<th>Module</th>
<th>Lines</th>
<th>GHC Only</th>
<th>Exception Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>boyer</td>
<td>Main</td>
<td>263</td>
<td>3.52</td>
<td>47%</td>
</tr>
<tr>
<td>circsim</td>
<td>Main</td>
<td>402</td>
<td>7.51</td>
<td>29%</td>
</tr>
<tr>
<td>comp.lab_zift</td>
<td>Main</td>
<td>882</td>
<td>6.73</td>
<td>31%</td>
</tr>
<tr>
<td>constraints</td>
<td>Main</td>
<td>163</td>
<td>3.43</td>
<td>14%</td>
</tr>
<tr>
<td>eliza</td>
<td>Main</td>
<td>228</td>
<td>3.55</td>
<td>26%</td>
</tr>
<tr>
<td>event</td>
<td>Main</td>
<td>449</td>
<td>3.72</td>
<td>12%</td>
</tr>
<tr>
<td>fft</td>
<td>Main</td>
<td>410</td>
<td>5.16</td>
<td>17%</td>
</tr>
<tr>
<td>fibheaps</td>
<td>Main</td>
<td>116</td>
<td>3.39</td>
<td>45%</td>
</tr>
<tr>
<td>fish</td>
<td>Main</td>
<td>86</td>
<td>3.99</td>
<td>23%</td>
</tr>
<tr>
<td>genfft</td>
<td>Main</td>
<td>500</td>
<td>4.48</td>
<td>16%</td>
</tr>
<tr>
<td>ida</td>
<td>Main</td>
<td>488</td>
<td>4.71</td>
<td>32%</td>
</tr>
<tr>
<td>nucleic2</td>
<td>Main</td>
<td>650</td>
<td>7.97</td>
<td>788%</td>
</tr>
<tr>
<td>RA</td>
<td>Main</td>
<td>587</td>
<td>5.33</td>
<td>93%</td>
</tr>
<tr>
<td>RC</td>
<td>Main</td>
<td>565</td>
<td>5.12</td>
<td>100%</td>
</tr>
<tr>
<td>RU</td>
<td>Main</td>
<td>604</td>
<td>5.23</td>
<td>104%</td>
</tr>
<tr>
<td>para</td>
<td>Main</td>
<td>274</td>
<td>3.43</td>
<td>21%</td>
</tr>
<tr>
<td>power</td>
<td>Main</td>
<td>89</td>
<td>6.04</td>
<td>91%</td>
</tr>
<tr>
<td>rewrite</td>
<td>Main</td>
<td>395</td>
<td>6.13</td>
<td>160%</td>
</tr>
<tr>
<td>sched</td>
<td>Main</td>
<td>553</td>
<td>3.93</td>
<td>42%</td>
</tr>
<tr>
<td>simple</td>
<td>Main</td>
<td>859</td>
<td>24.22</td>
<td>142%</td>
</tr>
<tr>
<td>solid</td>
<td>Main</td>
<td>1242</td>
<td>19.94</td>
<td>5%</td>
</tr>
<tr>
<td>sphere</td>
<td>Main</td>
<td>289</td>
<td>6.45</td>
<td>55%</td>
</tr>
<tr>
<td>typecheck</td>
<td>Main</td>
<td>656</td>
<td>5.78</td>
<td>52%</td>
</tr>
<tr>
<td>wang</td>
<td>Main</td>
<td>355</td>
<td>4.16</td>
<td>13%</td>
</tr>
<tr>
<td>wave4main</td>
<td>Main</td>
<td>597</td>
<td>5.51</td>
<td>13%</td>
</tr>
</tbody>
</table>

Figure 5. Indicative Cost of Exception Analysis

Briefly, this could be supported by our constraint system if we replace our annotation variables by a vector of annotation variables, one per possible exception. An exception would be represented by forcing the corresponding variable to be true. When matching an exception we would use existential qualification to “disconnect” any constraints on matched exceptions. A more formal discussion of this is in the first author’s forthcoming PhD thesis.

Fähndrich and Rehof have recently proposed a new method for flow analysis [5] for the efficient analysis of recursive definitions in the presence of polymorphic flow types. In the absence of recursive types, the method offers improved worst-case time complexity for flow analysis, namely cubic time analysis. Although our analysis has a higher worst-case complexity, our empirical results clearly show the feasibility of our approach.

Our semantics differs in some respects from that proposed by Peyton Jones et al. [14]. Our definition is entirely denotational, whereas [14] uses a mixture of denotational and operational semantics. More importantly, the semantics of [14] serves a different purpose, and is generally more slack than ours. For a function application e e′, where both e and e′ may give abnormal results, we utilise the non-strictness of e to disregard the abnormal values from e′, that is, the result is just the abnormal values of e. Similarly, for a conditional expression if e then e1 else e2, we do not care about abnormal values from e1 and e2 if it turns out that e only produces abnormal results. In contrast, the semantics of [14] stipulate that the branches of the conditional still be scrutinised for abnormal results.

The reason why Peyton Jones et al. [14] prefer the less precise semantics is a desire to have the semantics justify a number of program transformations that a compiler may try, such as strictness transformations and GHC’s case-case transformation. For example, consider the following function which is strict in both its arguments

\[
addPair : (\text{Int}, \text{Int}) \rightarrow (\text{Int}, \text{Int})
\]

\[
addPair \ xp \ yp = \begin{cases} 
\text{case } xp \ of & \\
(x1, x2) \rightarrow & \begin{cases} 
\text{case } yp \ of & \\
(y1, y2) \rightarrow (x1 + y1, x2 + y2)
\end{cases}
\end{cases}
\]

Clearly an abnormal value E₁, from the first argument will be raised rather than one (say, E₂) from the second argument. Changing the order in which the arguments are scrutinised can therefore change the (abnormal) result of an application of addPair. Yet an optimising Haskell compiler is in the habit of making exactly such transformations, and it would be unfortunate to ban them. The answer in [14] is that in either case, E₁ and E₂ should both be included.

We make no judgement about the usefulness of this or that system of exceptions for a non-strict functional language. The semantic issues are far from straightforward. We saw in Section 1 how non-determinism in exceptions can leak into the language proper, since we can use try (or any similar exception handling mechanism) to construct an expression amb which may evaluate to either 0 or 1. This raises the well-known problem that either beta reduction is lost, or else variables become indefinite, as is the case for the semantics we have proposed. More specifically, given the semantics proposed for ‘+’, the value of amb + amb may be odd: it can be 0, 1, or 2.

A different anomaly is that changing evaluation order may return a result where a non-strict language’s usual order does not. The intention with the meta-language used in Figure 1 is that function application is non-strict, so if loop is an expression whose evaluation does not terminate, then \((\lambda x. \text{loop})(\text{raise } Ex)\) also fails to terminate. Hence, so does

\[
\text{try } (\lambda x. \text{loop}) (\text{raise } Ex) \quad \text{match } Ex \rightarrow 1
\]

With strict function application, however, the value of \((\lambda x. \text{loop})(\text{raise } Ex)\) is the exception Ex, so the try expression has
value 1.

7 Conclusion

We have presented an analysis for escaping exceptions in a non-strict language, and we have established its correctness. The analysis discovers exceptions that could possibly be raised and not caught within an expression. It is formulated as a constrained-type inference system, using Boolean constraints to capture the (sometimes intricate) dependencies that are needed to reason about control flow. The work is part of a bigger effort to build a general constraint-based analysis framework for non-strict functional languages with polymorphic programs and structured data, much in the spirit of [6]. We have found that Boolean constraints have many useful features in the context of “analysis as constrained-type inference”.

Boolean constraints make the formulation of a polyvariant analysis easy (by polyvariance we mean “property polymorphism”, that is, an analyser’s ability to infer different properties for a function at separate applications of a function, in a manner similar to type inference for a polymorphic language). They also help achieve modularity of analysis, that is, the ability to produce analysis results that are context-independent, so as to support separate compilation. Although we have not described the extension to a polymorphically typed language. They also help achieve modularity of analysis, that is, the ability to produce analysis results that are context-independent, so as to support separate compilation. Although we have not described the extension to a polymorphically typed language, in this paper, our implementation does handle this, and also structured data, and in fact the Boolean constraint approach supports such extensions very well.

Many important subclasses of Boolean functions are closed under existential quantification. This is useful, since in many contexts existential quantification is the logical counterpart to the principle of eliminating “irrelevant” variables (thus restricting attention to “relevant” variables). In Section 3.1 we pointed out instances where existential quantification gave us an elegant formulation of variable elimination, for example in the rule for try.

Our evaluation results, for a version of GHC with our analysis incorporated, indicate that the analysis is practical, and a rough complexity analysis shows that the analysis is \(O(n^6)\), a bound which can possibly be improved.

Acknowledgements

We thank the reviewers for their helpful comments.

8 References


APPENDIX: CORRECTNESS PROOF

First let us introduce some useful lemmas.

Lemma 2. Let \( \vdash \delta\varphi \vdash \tau \) and \( \vdash \phi \psi \) for some substitution \( \phi \) such that \( \phi(\delta\varphi) = \{ \text{ex} \} \). Then \( \text{ex} \in \{ \text{ex} \} \).

Lemma 3. Let \( \tau_1 \) and \( \tau_2 \) be two monotypes such that \( \vdash \{ \tau_1 \leq_x \tau_2 \} \). Then \( \{ \tau_1 \} \subseteq \{ \tau_2 \} \).

Lemma 4. Let \( C \) be a constraint, \( \delta_0, \delta_1, \ldots, \delta_n \) be variables such that \( C \) consists of only primitive constraints of the form \( \delta \rightarrow \delta' \), \( \delta_0 \) does not appear on the left-hand side and \( \delta_1, \ldots, \delta_n \) do not appear on the right-hand side of an implication. Let \( \delta \) and \( \delta' \) be variables such that \( \delta \) differs from \( \delta_1, \ldots, \delta_n \), \( \delta' \) differs from \( \delta_0 \) and \( C \). Then \( \{ \text{ex}_0 \}, \{ \text{ex}_n \} \} \) \( \vdash \delta \rightarrow \delta' \).
such that $\exists\delta \subseteq \phi_\Gamma$ is closed and $\rho \models \phi_\Gamma$. Then $[e :: t]p \models^2 \phi_\eta$. We prove correctness by induction over type derivations.

We note that applying substitutions $\phi$ to type schemes $\eta$ might result in type schemes of the form $\forall \delta_1, [E_1] : \delta_1 \Rightarrow \ldots$. Clearly, this type scheme is not a valid member of any of the syntactic domains presented. The important point is that eventually we will substitute $\delta_1$ by a ground value. Therefore, the translation function $[\[]$ will not get stuck and will work correctly.

We will also assume that the meaning function $[\[]$ for expressions in Figure 1 also applies to type-annotated expressions (simply by erasing the underlying types). For convenience we will sometimes omit some of the underlying type annotations.

**Case (Var):** We find the following situation

$$\phi \eta \models (x :: t) : (x :: t) : \eta$$

From the assumption $\rho \models \phi_\Gamma$ so $\rho(x) \models^2 \phi_\eta$. From the semantic equations $[x :: t]p = \rho(x)$ so $[x :: t]p \models^2 \phi_\eta$ as required.

**Case (Raise):** We find the following situation

$$\phi_\eta \models (\text{raise}_\phi :: \text{Exn} \Rightarrow t) (\text{ex} :: \text{Exn}) :: t : \tau$$

From the assumption we have that $\models \forall \phi_\Gamma$. We can immediately follow that $[\text{raise}_\phi :: \text{Exn} \Rightarrow t] (\text{ex} :: \text{Exn}) :: t \models^2 \phi_\tau$.

**Case (Abs):** We find the following situation

$$\phi_\eta \models (\lambda (x :: t_1). (e :: t_2) :: t_2) : \tau_1 \Rightarrow (e :: t_2) : \tau_2$$

We want to show $[\lambda x.e]p \models \phi_\tau_1 \Rightarrow \phi_\tau_2$. Note that w.l.o.g. $\phi(\delta) = \emptyset$.

We find the following equivalences.

$$[\lambda x.e]p \models^2 \phi_\tau_1 \Rightarrow \phi_\tau_2$$

iff

$$[\lambda x.e]p \subseteq [\phi_\tau_1 \Rightarrow \phi_\tau_2]$$

iff

$$[\lambda x.e]p \subseteq \{v \mid v \models [\phi_\tau_1] \in [\phi_\tau_2]\}$$

iff (def. of $[\lambda x.e]p$)

$$\{v \models [\phi_\tau_1]/x\} \subseteq \{v \models [\phi_\tau_2]/x\}$$

Let $\rho' = \rho([\phi_\tau_1]/x)$ then application of the induction hypothesis to the premise yields $[e]p' \models^2 \phi_\tau_2$. The last equivalence is fulfilled. Therefore, we can establish the induction step and we are done.

**Case (App):** We find the following situation

$$\phi_\eta \models (e :: t_1 \Rightarrow t_2) : (t_1 \Rightarrow \tau_2)$$

We apply the induction hypothesis to the two premises (we silently extend $\phi$ such that $\phi_\tau_1$ is ground) yields $[e_1 :: t_1 \Rightarrow t_2]p \models^2 \phi(t_1 \Rightarrow \tau_2)$ and $[e_2 :: t_1]p \models^2 \phi_\tau_1$.

Recall the semantic equation

$$[e_1 e_2 :: t_2]p = \bigcup \{ [e_1]p \cap [e_2]p \land [e_1]p \not\subseteq \{x_1 x_2 \mid x_2 \in [e_2]p \land \exists x_1 [e_1]p \not= \emptyset \}\}$$

We want to show $[e_1 e_2 :: t_2]p \models^2 \phi_\tau_2$, that is, $[e_1 e_2 :: t_2]p \subseteq [\phi_\tau_2]$. We distinguish among the following cases:

1. $e \in [\phi_1]p$.

   In such a situation, we know that $\phi_\tau_1 \models \phi_\tau_2$. From $\phi_\tau_1 \models \phi_\tau_2$ it follows that $\phi_\tau_1 \models \phi_\tau_2$. Therefore, $\phi \models \phi_\tau_2$ (by Lemma 2).

2. $\bot$ is in every exception type.

3. $w \in [e_1 e_2 :: t_2]p$ where $f \in ([\phi_1]p \setminus \{x_2 \mid x_2 \not\in [e_2]p\})$ and $w \in f [e_2]p$.

   It follows that $f \models [\phi_\tau_1 \Rightarrow \phi_\tau_2]$. By definition of $[\phi_\tau_1 \Rightarrow \phi_\tau_2]$, we find that $w \not\in [\phi_\tau_2]$ and we are done.

**Case (Let):** We find the following situation

$$\phi_\eta \models (e :: t_1 :: t_2 :: t)\ : \ t$$

We apply the induction hypothesis to the first premise and find that $[e_1 :: t_1]p \models^2 \phi_\eta$. We apply the induction hypothesis to the second premise (there is a double induction going on and we would need to include the type environment in our induction argument, but the details are straight-forward and omitted here) and find that $[e_2 :: t_2]p \models^2 \phi_\tau$ which establishes the induction step.

**Case (Nil):** We find the following situation

$$\phi_\eta \models (e :: t) : \tau$$

We apply the induction hypothesis to the premise and find that $[e :: t]p \models^2 \phi_\eta$. We apply the induction hypothesis to the second premise (there is a double induction going on and we would need to include the type environment in our induction argument, but the details are straight-forward and omitted here) and find that $[e :: t_1]p \models [\phi_\tau]$. We are done.

**Case (If):** We find the following situation

$$\phi_\eta \models (e :: t_1 :: t) : \tau$$

We apply the induction hypothesis to the premise and find that $[e :: t]p \models^2 \phi_\eta$. We are done.

**Case (App):** We find the following situation

$$\phi_\eta \models (e_1 :: t_1) : \tau$$

We apply the induction hypothesis to the premise and find that $[e :: t_1]p \models^2 \phi_\eta$. We are done.

**Case (Nil):** We find the following situation

$$\phi_\eta \models (e :: t) : \tau$$

We apply the induction hypothesis to the premise and find that $[e :: t]p \models^2 \phi_\eta$. We are done.

**Case (App):** We find the following situation

$$\phi_\eta \models (e_1 :: t_1 :: t) : \tau$$

We apply the induction hypothesis to the premise and find that $[e :: t]p \models^2 \phi_\eta$. We are done.

We distinguish among the following cases:
1. \( v_2 \in [[e_2]]p \land True \in [[e_1]]p \):
   We can apply the induction to the then branch and immediately find that \( v_2 \vdash ^v \phi \).

2. \( v_3 \in [[e_3]]p \land False \in [[e_1]]p \):
   Similarly, we apply the induction hypothesis to the else branch and find that \( v_3 \vdash ^v \phi \).

3. \( \bot \) is in every exception type.

4. \( ex \in [[e_1]]p \):
   \( ex \in [[x]]p \), by the constraint \( \vdash \delta \rightarrow \tau \) and Lemma 2.

**Case (VI):** We find the following situation

\[
\begin{align*}
C \land D, \Gamma & \vdash (e :: t) : \tau \\
\exists \delta & \subseteq \text{fr}(D, \tau) \backslash \text{fr}(\Gamma, C)
\end{align*}
\]

C, \exists \delta, D, \Gamma \vdash (e :: t) : \forall \delta. D \Rightarrow \tau

By assumption, \( \vdash \phi C \land \phi(\exists \delta. D) \) and \( p \vdash ^v \phi \Gamma \) for some substitution \( \phi \). W.l.o.g., we find \( \psi \) such that \( \vdash \psi D \) and \( \phi \subseteq \psi \) (that is, there exists \( \phi' \) such that \( \phi' \circ \phi = \psi \)). The induction hypothesis applied to the premise yields \([e :: t]p \in [[\psi(\exists \delta. D \Rightarrow \tau)]]\). We immediately find that \([e :: t]p \in [[\psi(\delta) \Rightarrow \tau]]\) and we are done.

**Case (VI):** We find the following situation

\[
\begin{align*}
C, \Gamma & \vdash (x :: t) : \forall \delta. D \Rightarrow \tau \\
\exists \delta & \subseteq \text{fr}(D, \tau) \backslash \text{fr}(\Gamma, C)
\end{align*}
\]

C, \Gamma \vdash (x :: t) : \forall \delta. D \Rightarrow \tau

By assumption, we have that \( \vdash \phi C \), \( p \vdash ^v \phi \Gamma \), \( \phi \Gamma \) and \( \psi(\exists \delta. D \Rightarrow \tau) \) closed for some substitution \( \phi \). The induction hypothesis applied to the premise yields \([x :: t]p \in [[\psi(\exists \delta. D \Rightarrow \tau)]]\). We note that \([\psi(\exists \delta. D \Rightarrow \tau)] \subseteq [[\psi \tau]]\) where \( \psi' = \phi \circ \psi \). This establishes the induction step.

**Case (Fix):** We find the following situation

\[
\begin{align*}
C, \Gamma & \vdash (x :: t) : \eta \\
\exists \eta & \subseteq t
\end{align*}
\]

C, \Gamma \vdash (\text{fix} x :: t : in e) :: t : \eta

Note that \([\text{fix} x :: t]p = \text{fix}(\lambda \nu. [[e]]p[x := v]) = \bigcup f \downarrow\)

where \( f = \lambda \nu. [[e]]p(x := v) \).

We show, by induction, that \( f^i \perp \subseteq [[\eta]] \) for any \( i \geq 0 \).

Case \( i = 0 \): By definition, \( \perp \subseteq [[\eta]] \).

Case \( i \Rightarrow i + 1 \): We have that \( f^{i+1} \perp = f(f^i \perp) = [[e]]p[x := f^i \perp] \). By induction \( f^i \perp \subseteq [[\eta]] \), therefore \( p' = p[x := f^i \perp] \) is well-defined. We find that \( p' \vdash ^v \phi \Gamma \cdot \eta \). We can apply the induction hypothesis to the first premise. We find that \( [[e]]p' \subseteq [[\eta]] \), therefore \( f^{i+1} \perp \subseteq [[\eta]] \) which concludes the induction step.

Note that ideals are limit closed, hence \( \text{fix}(f) \subseteq [[\eta]] \).

**Case (Try):** We find the following situation:

\[
\begin{align*}
C_0, \Gamma & \vdash (e :: t) :: \tau_0 \\
C, \Gamma & \vdash (x :: t) : \tau \vdash t \\
C' & = \frac{}{\exists t \subseteq \tau \perp \land C} \\
C & = C_1 \land \ldots \land C_n \land (\exists \tau_0). C_0 \land (\exists \delta_{a_1} C_0 \land \ldots \land \exists \delta_{a_n} C_0 \land \exists \tau_0 \leq \tau) \perp \land C
\end{align*}
\]

C, \Gamma \vdash \text{try} (e :: t) : match \( \{ ex \rightarrow (e :: t) \} : \mu t :: t \rangle \)

By assumption, \( \vdash \phi C \), \( \phi \Gamma \) closed, \( p \vdash ^v \phi \Gamma \). We want to show that

\[
\begin{align*}
\text{try} (e :: t) & \vdash \text{match} \{ ex :: \text{Exn} \rightarrow (e :: t) \} : \phi \tau \\
& \text{iff} \quad \text{try} (e :: t) \vdash \text{match} \{ ex :: \text{Exn} \rightarrow (e :: t) \} : \phi \tau \subseteq \phi \tau
\end{align*}
\]

where

\[
\begin{align*}
\text{try} (e :: t) & \vdash \text{match} \{ ex :: \text{Exn} \rightarrow (e :: t) \} : \phi \tau = \\
& \bigcup_{i \in I} \{ [e]p_i :: ex \subseteq [[e]]p_i \cup \bigcup_{i \in I} \{ [e_i]p_i \}
\end{align*}
\]

We distinguish between the following cases:

1. Case \( v \in [[e]]p \) and \( v \not\subseteq \delta \): Let \( \delta_0 = \left[ \tau_0 \right] \). Let \( \psi' \) be a substitution such that \( \psi'(\delta_0) = \psi(\delta_0) \cup \delta_0 \), \( \psi'(\delta_0) = \psi(\delta_0) \) and \( \phi \) and \( \phi' \) agree on all variables except those in \( \delta_0 \cup \{ \delta_0 \} \).

We prove that \( \vdash \phi C_0 \). Assume the contrary. That is, there exists \( \delta \not\subseteq \delta' \in C_0 \) such that \( \psi'(\delta) \not\subseteq \psi'(\delta') \). Consider the following three cases.

2. Assume \( \delta = \delta_0 \) then again we find a contradiction because \( \psi'(\delta_0) \) represents the largest possible set.

3. Assume \( \delta = \delta_{a_i} \) for some \( i \) and \( \delta' \) differs from \( \delta_0 \). Note that \( \vdash \phi C \). Application of Lemma 4 leads immediately to a contradiction.

4. Case \( \psi(\delta) \not\subseteq \psi(\delta') \) which contradicts our assumption.

We conclude that \( \vdash \phi C_0 \). Application of the induction hypothesis to \( C_0, \Gamma \vdash (e :: t) :: \tau_0 \) yields \( \text{match} \{ ex :: \text{Exn} \rightarrow (e :: t) \} : \phi \tau' \). We also have that \( \forall_{i \in I} \{ [e_i]p_i \subseteq [[e]]p_i \perp \land C \} \). We can apply the induction hypothesis to \( C, \Gamma \vdash (e :: t) : \tau_0 \) and obtain that \( \text{match} \{ ex :: \text{Exn} \rightarrow (e :: t) \} : \phi \tau \subseteq \phi \tau_0 \). By assumption \( v \not\subseteq \delta \). We can conclude that \( v \in [[\delta \tau]] \) and we are done.

2. Case \( e \in [[e_1]]p \land e_1 \in [[e]]p \):

Again we extend \( \phi \) to \( \Psi' \) such that \( \vdash \phi C_0 \). By assumption \( e_1 \in [[e]]p \). Application of the induction hypothesis to \( C_0, \Gamma \vdash (e :: t) :: \tau_0 \) and applying the Corollary yields \( \psi'(\delta_{a_i}) \subseteq \psi'[\tau_{a_i}] \). We know that \( C \vdash \left[ \tau \right] , \subseteq \tau \land C \). We find that \( \vdash \phi[\left[ \tau \right] \subseteq \tau \land C] \). We can apply the induction hypothesis to \( C, \Gamma \vdash (e :: t) : \tau_0 \) and obtain that \( [[e']]p = \psi' \phi \). Then by Lemma 3, \( [\psi \tau] \subseteq [[\eta]] \). Hence, \( [[e]]p = \psi' \phi \). Finally, we can conclude that \( v \in [[\delta \tau]] \) and we are done.