Diagnosis of Ill-typed Programs

Venkatesh Choppella and Christopher T. Haynes
Indiana University, USA

{choppell,chaynes}@cs.indiana.edu

Abstract

A framework, based on syntactic and type constraints, is provided for defining program slices that contribute to a given type error or similar syntactic property. We specify soundness, minimality and completeness criterion for these slices and outline an algorithm for their lazy generation.

1 Introduction

Error diagnosis in current type-reconstruction algorithms either misses information that is relevant, presents irrelevant details, or both. We propose a general framework that describes a type error in terms of the error's "symptom," type constraints logically deriving the symptom, and program slices contributing the type constraints that derive the symptom.

We present two frameworks for diagnostic reasoning that subsume the analysis of type errors. In the first framework, based on propositional logic, a program slice is a set of program subforms that contribute to a given type constraint. We call these slices *location sets*. The second, based on first-order logic, is a more fine-grained framework in which a program slice is a set of relations (syntactic constraints) between subforms that lead to a given type constraint. We call these program slices *partial syntactic descriptions*.

Our diagnostic error analysis is achieved using a variant, due to Port [17], of the standard unification algorithm, which we call *diagnostic unification*. This algorithm provides information from which it is possible to directly obtain the set of all minimal proofs of the derivation of a given type constraint in a deduction system consisting of type rules and type constraints. A proof is minimal if the result of deleting any step or set of steps is ill-formed. A type error is signaled by a type constraint that is trivially unsatisfiable.

Section 2 reviews related work. Section 3 presents an extended example of our technique based on the simply-typed lambda calculus with constants and if. Section 4 formalizes diagnostic analysis of type errors in the framework of propositional logic, including soundness, minimality and completeness criteria for location sets. A sound and complete algorithm that computes location sets for the simply-typed lambda calculus is presented. Section 5 formalizes the notions of syntactic constraints, partial syntactic descriptions, and soundness and completeness criteria for judging algorithms that generate partial syntactic descriptions. Section 6 adapts these techniques to ML-style polymorphism and polymorphic recursion. Section 7 concludes with directions for further work, including a discussion of how the framework of this paper may be useful in arriving at a general framework for diagnostic analysis of program properties.

2 Related work

Wand [21] was among the first to address the problem of reporting type errors. He modified the traditional unification algorithm to accumulate "reason lists" that "explained" the introduction and binding of type variables. He speculated that a "completeness result" might be provable for his algorithm, but did not formulate one. We formalize soundness criteria which Wand's algorithm fails to satisfy. For example, for $\lambda x \lambda y.f(y\ x)(y\ 3)(\mathsf{not}\ x)$, Wand's algorithm returns the reason list $\{(y\ 3), (\mathsf{not}\ x)\}$. This set does not imply enough type constraints to derive the type error.

The network flow approach of Johnson and Walz [9, 20] attempts to derive all "multiply contradictory hypotheses" of the type of a program variable and assign them relative frequencies. The authors claim that their implementation associates text with type errors, but the details are not published. They provide no correctness criterion or characterization of their complex algorithm.

Gomard [6] isolates the untypable parts of a program with a two-level syntax and type rules that employ a special type called "untyped." When an expression with well-typed subexpressions is ill-typed, the entire expression is flagged with no indication of which parts of the subexpressions contributed to the error. In other words, although his technique

identifies untypable subprograms, it does not identify minimal slices of these subprograms.

Maruyama et al. [13] use a tracing technique for type error analysis, but they employ a flawed heuristic. Only parse tree nodes that are adjacent to the node where unification fails are considered. We demonstrate that more distant nodes may contribute to the error. Our algorithm employs adjacency in a type-constraint graph rather than the parse tree.

Beaven and Stansifer [1] and Soosaipillai [19] develop a method of explaining the flow of type information in a parse tree. Duggan et al. [5] have developed a similar method. While these techniques are useful for building a type debugging environment that guides the user by "explaining" the process of type inference, they are not designed to identify the source of a type error.

Program slicing techniques have been studied in the context of "automatic" run-time program debugging [23]. We demonstrate how program slices may be used to diagnose static program properites, including type error analysis.

Our approach is related to the more general characterization of conflict sets and diagnosis developed by Reiter [18]. Our location sets can be viewed as Reiter's conflict sets.

3 Example

In this section we informally introduce the key notions in our diagnosis of type errors by analysing a small ill-typed program in λ_c^{if}: the simply-typed lambda calculus with constants and if.

The (pre-term) expressions in our mini-language are given by the grammar

$$e ::= x \mid \lambda x.e \mid @\,e\,e \mid \mathsf{if}\,e\,e\,e \mid c^\tau \tag{1}$$

where @ denotes combination (application), $x \in V$ is a denumerable set of variable symbols, and $c^\tau \in C$ is a set of typed constants (primitives). The types $\tau \in T$ of λ_c^{if} are given by the grammar

$$\tau :: t \mid \tau \rightarrow \tau' \mid \mathtt{bool} \mid \mathtt{int} \tag{2}$$

where $t \in TV$ ranges over type variables.

Let e_0 be the expression

$$\lambda x.\mathsf{if}\ x\ (@\mathsf{inc}\ x)\ x \tag{3}$$

where $\mathsf{inc}^{\mathtt{int}\rightarrow\mathtt{int}}$ is the increment constant with type $\mathtt{int}\rightarrow\mathtt{int}$. The parse tree for e_0 is given in Figure 1(a), in which each parse tree node is given a unique index, called its *location*.

The type inference process may be thought of as generating *type constraints* (equations) as the expression is traversed [14, 2]. With each type constraint, we associate the location at which it was generated and the syntactic constraint involved. Figure 2 lists the type constraints associated with our sample expression, e_0, along with their associated locations, syntactic constraints (expressed in informal English), and an identifying label (a letter in the range a through i).

These *initial type constraints* are represented by a term graph; for our example the solid-edge subgraph of Figure 1(b). Type reconstruction involves attempting to solve these equations by term unification implemented as rewrite rules on the term graph. The rewrite rules may add additional *derived type constraints*. In our example, the initial type constraints f and g, along with the "subterm unification" rule result in addition of the following derived type constraints, represented by broken edges in Figure 1(b).

$\mathtt{int} \doteq t_7 : j$
$\mathtt{int} \doteq t_4 : k$

The type-error symptoms are explained via connectivity between nodes in the term graph. In our example, a proof for the equation $\mathtt{int} \doteq \mathtt{bool}$ can be generated directly from any path connecting the nodes `int` and `bool`. This contradiction leads to the conclusion that e_0 is ill-typed.

A minimal path connecting two nodes in a type graph corresponds to a minimal proof. The minimal paths from `int` to `bool` in Figure 1(b) are $jiec$ and $kdgec$.

The equations j and k are not associated with any location or syntactic constraint in e_0. Instead, they owe their existence to the subterm unification rule and to the connectivity of the $\rightarrow$ nodes via the edges f and h. Hence the minimal subsets of the initial edges (type constraints) that derive $\mathtt{int} \doteq \mathtt{bool}$ are $E_1 = \{h, f, i, e, c\}$ and $E_2 = \{h, f, d, g, e, c\}$. These correspond to two proofs of the untypability of e_0, which are independent and both minimal.

By collecting the locations at which the type constraints were generated, we get the program slices illustrated in Figure 3. We may obtain more refined program slices, which we term "partial syntactic descriptions," by collecting the sets of syntactic constraints corresponding to sets of type constraints. From E_1 and E_2 we obtain the slices S_1 (Figure 4(a)) and S_2 (Figure 4(b)). Each slice is minimal in the sense that no proper subset of either will imply enough type constraints to derive the type error. Further, any other slice of e_0 that generates enough type constraints to cause the type error $\mathtt{int} \doteq \mathtt{bool}$ will properly contain either S_1 or S_2. Each slice S_1 and S_2 can be thought of as a schema representing a set of programs, any member of which is guaranteed to be ill-typed with the same symptom and proof.

In S_1, the absence of an edge from the if node to the @ node suggests that the @ node need not be a direct subexpression of the if node to cause the type error. It may occur anywhere so long as the variable at location 7 is within the scope of the formal at location 1. In S_2, the presence of two if nodes indicates that the if in which x occurs in the test part does not have to be the same if in which the nodes 4, 5, and 6 occur to cause the type error symptom to be derived.

These examples illustrate that each partial syntactic description generates a maximally weakened precondition on

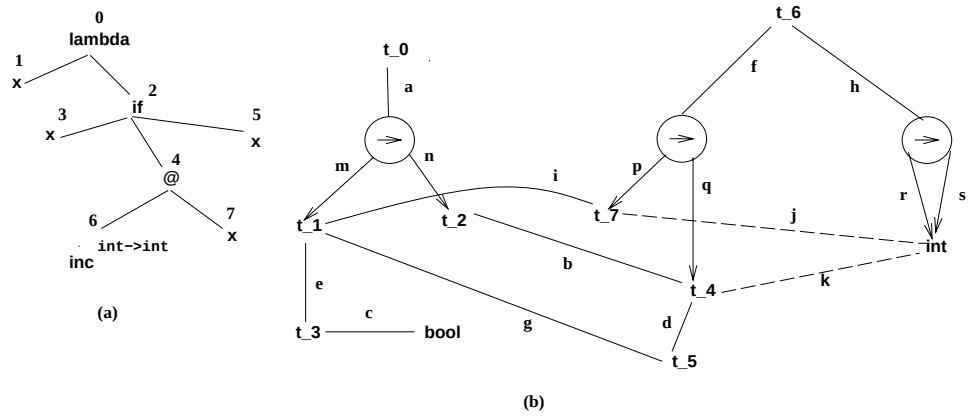

Figure 1: Parse tree and type constraint graph for example program

$$
\begin{aligned}
\text{0: } e_0 \text{ is a } \lambda \text{ with formal at 1 and body at 2} &\implies t_0 \doteq t_1 \to t_2 : a \\
\text{2: } e_2 \text{ is an if with the 'then' part at 4} &\implies t_2 \doteq t_4 : b \\
\text{2: } e_3 \text{ is the test part of an if} &\implies t_3 \doteq \texttt{bool} : c \\
\text{2: } e_4 \text{ and } e_5 \text{ are then and else parts of an if} &\implies t_4 \doteq t_5 : d \\
\text{3: } e_3 \text{ is a variable bound at 1} &\implies t_3 \doteq t_1 : e \\
\text{4: } e_4 \text{ is an application of the function at 6 to the argument at 7} &\implies t_6 \doteq t_4 \to t_7 : f \\
\text{5: } e_5 \text{ is a variable bound at 1} &\implies t_5 \doteq t_1 : g \\
\text{6: } e_6 \text{ is a constant of type } \texttt{int}\to\texttt{int} &\implies t_6 \doteq \texttt{int}\to\texttt{int} : h \\
\text{7: } e_7 \text{ is a variable bound at 1} &\implies t_7 \doteq t_1 : i
\end{aligned}
$$

Figure 2: Type constraints and reasons generated for the example program

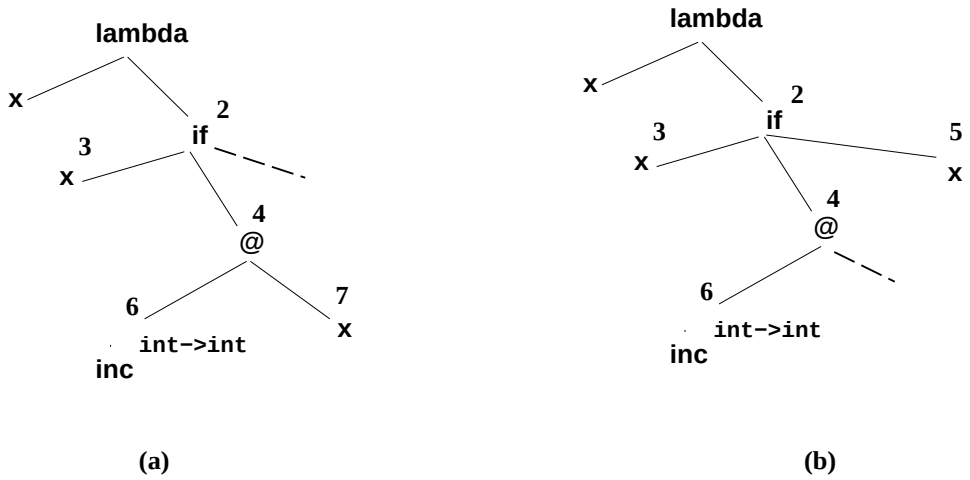

Figure 3: Slices in terms of location sets for example program

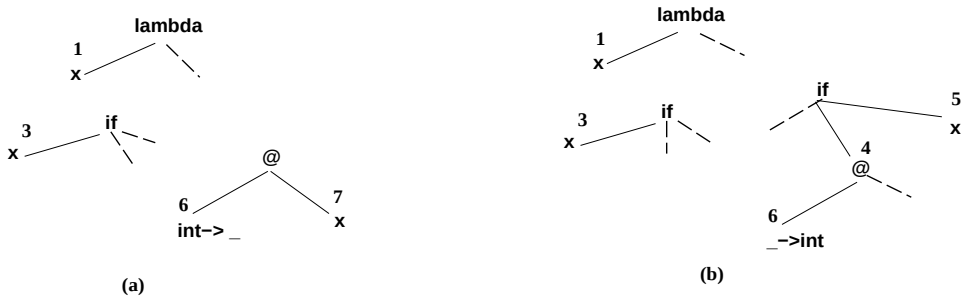

Figure 4: Slices in terms of syntactic constraints for example program

syntactic constraints that implies sufficient type constraints to generate a given type error. In the remainder of this paper we formalize these notions of type constraint, syntactic constraint, implication, derivation, and minimality.

4 Diagnostic type inference for a simple language

Let the set of *expressions* $e \in Exp$ be a free term algebra generated over a ranked alphabet $\mathcal{A}$ and a set V of *variables*. Let the set of *types* $\tau \in T$ be another free term algebra over a ranked alphabet $\mathcal{T}$ and a set TV of *type variables*.

A *type environment* is a finite function from V to types. $A[x:\tau]$ denotes the extended type environment $\lambda v.\mathit{if}\ v \equiv x\ \mathit{then}\ \tau\ \mathit{else}\ A(x)$. A *typing judgement* $A \rhd e : \tau$ is an ordered triple consisting of a type environment A, expression e, and type τ. A subset of the set of typing judgements are termed *well typings*. Elements of this subset are defined by derivability in the logic of a *type inference system*. Figure 5 presents a type inference system for the language λ_c^{if} introduced in the previous section.

The typability problem ("is e typable?") is defined as: Given a type system X and an expression $e \in Exp$, do there exist A and τ such that $A \rhd e : \tau$ is a theorem of the type system X?

Most strategies for solving typability problems for a large class of type systems involve reduction to the unification problem and its variants. Unification solves systems of *constraints* over terms generated over a free algebra. The reduction of the typability problem for an expression e consists of generating a system of constraints from e.

It is well-known that the typability problem for λ_c^{if} is reducible to first-order unification [2, 14, 22]. We use λ_c^{if}-typability as the running example in this paper, but our techniques are applicable to a broad range of type inference systems and unification variants, including the ML type system and polymorphic recursion (where typability is reducible to semiunification [8]).

A *type equation*, η, is a tuple $\tau \doteq \tau'$, where τ and τ' are types. A *system of type equations* E is a multiset of type equations.

We often assume expressions are *decorated*: every subexpression is subscripted with a distinct index, called its *location*. The decorated expression syntax for $Exp_{\lambda_c^{if}}$ is

$$e_i ::= x_i \mid \lambda_i x_j.e_k \mid @_i\, e_j\, e_k \mid \mathsf{if}_i\, e_j\, e_k\, e_l \mid c_i^\tau \qquad (4)$$

Reducibility of λ_c^{if}-typability to first-order unification is achieved by the reduction function, $\mathcal{R}_{\lambda_c^{if}}$, given in Figure 6. Each type equation is annotated with the location of the current subexpression that generated the equation. This information plays no role in the unification of the type equations, but is used for building slices that derive a particular type constraint.

Standard presentations of rewrite rules transform the initial sysmtem of equations into "solved form" or fail [11, 12, 16]. Since our focus is on the sources of non-unifiability, we instead adopt a presentation due to Port[17] in which there is no failure. Given a set of type constraints E, let E^* be the closure of E under the rewrite rules of Figure 7.

Definition 4.1: A *type constraint* δ is any pair of the form $\tau \doteq \tau'$ or $\tau \prec \tau'$.

Definition 4.2: δ is an *error symptom* if it has one of the

$$\text{(VAR)}\quad \frac{}{A[x:\tau] \rhd x : \tau}$$

$$\text{(CONST)}\quad \frac{}{A \rhd c_\tau : \tau}$$

$$\text{(ABS)}\quad \frac{A[x:\tau] \rhd e : \tau'}{A \rhd \lambda x.e : \tau \to \tau'}$$

$$\text{(APP)}\quad \frac{A \rhd e : \tau \to \tau' \qquad A \rhd e' : \tau}{A \rhd @\ e\ e' : \tau'}$$

$$\text{(IF)}\quad \frac{A \rhd e : \mathtt{bool} \qquad A \rhd e' : \tau \qquad A \rhd e'' : \tau}{A \rhd \mathsf{if}\ e\ e'\ e'' : \tau}$$

Figure 5: The λ_c^{if} type system

$\mathcal{R}_{\lambda_c^{if}}(e) = \mathcal{R}(A', e)$ **where**

$A' = \{x \mapsto t_j \mid x \text{ free in } e \text{ and } t_j \text{ is fresh}\}$

and

$\mathcal{R}(A, e_i) =$ **case** e_i of

x_i	$\{i \stackrel{e}{\Longrightarrow} t_{A(x)} \doteq ti\}$
c_i^τ	$\{i \stackrel{e}{\Longrightarrow} t_i \doteq \tau\}$
$\mathsf{if}_i\ \ e_j\ e_k\ e_m$	$\mathcal{R}(A, e_j) \cup \mathcal{R}(A, e_k) \cup \mathcal{R}(A, e_m) \cup$ $\{i \stackrel{e}{\Longrightarrow} t_j \doteq \mathtt{bool},$ $i \stackrel{e}{\Longrightarrow} t_k \doteq t_m,$ $i \stackrel{e}{\Longrightarrow} t_i \doteq t_k\}$
$@_i\ e_j\ e_k$	$\mathcal{R}(A, e_j) \cup \mathcal{R}(A, e_k) \cup \{i \stackrel{e}{\Longrightarrow} t_j \doteq t_k \rightarrow t_i\}$
$\lambda_i\ x_j.e_k$	$\mathcal{R}(A, \{x : j\}, e_k) \cup\ \{i \stackrel{e}{\Longrightarrow} t_i \doteq t_j \rightarrow t_k\}$

Figure 6: Definition of $\mathcal{R}_{\lambda_c^{if}}$

Initially, $E^* = E$.

1. ($\doteq$ *propagation*) If $f(\tau_1, \ldots, \tau_n) \doteq f(\tau_1', \ldots, \tau_n') \in E^*$, then for each $1 \leq i \leq n$, add the constraint $\tau_i \doteq \tau_i'$ to E^*.

2. ($\doteq$ *transitivity via variables*) If $\tau \doteq t$ and $t \doteq \tau' \in E^*$, then add $\tau \doteq \tau'$ to E^*.

3. ($\prec$ *identification*) If $t \doteq \tau \in E^*$ and τ is not a type variable, then add $t' \prec t$ to E^* if t' is a variable subterm of τ.

4. ($\prec$ *transitivity*) If $t \doteq t' \in E^*$, and $t'' \prec t \in E^*$ or $t' \prec t'' \in E^*$, then add $t'' \prec t'$ or $t \prec t''$ (respectively) to E^*.

Figure 7: Rewrite rules for diagnostic unification

following forms:

1. $f(\tau_1, \ldots, \tau_n) \doteq g(\tau_1', \ldots, \tau_m')$ where $f \not\equiv g$ (*functor clash error*)

2. $t \prec t$ (*occurs check error*)

Definition 4.3: A *type constraint set*, E, is any finite set of type constraints. Given type constraint sets E and E', $E \sqsubseteq E'$, read "E is *weaker than* E', if $E \subseteq E'$.

Notation: E_e denotes $\mathcal{R}(e)$. $E \vdash \delta$ (read E *derives* δ) denotes $\delta \in E^*$. $e \rightsquigarrow \delta$ denotes $E_e \vdash \delta$

Theorem 1 (Port[17]) *A system of type constraints E is non-unifiable if and only if $E \vdash \delta$ for some error symptom δ.*

The rules for diagnostic unification are implemented in the standard manner using a term graph [16]. Constraints in E^* correspond to connectivity in the term graph. A functor clash corresponds to an undirected path between two nodes with different functor symbols. An occurs check error corresponds to a path consisting of undirected and directed edges (pointing either way) from a variable vertex to itself.

The operation mapping E to its closure E^* under the rewrite rules in Figure 7 is monotonic. This ensures the following important property:

Proposition 1 *For finite E, if $E \vdash \delta$, then*

$$\mathcal{E}(e, \delta) = \{E \sqsubseteq E_e \mid min(E, \delta)\} \tag{5}$$

is non-empty, where

$$min(E, \delta) \stackrel{\text{def}}{=} E \vdash \delta \ \wedge\ \forall E' \sqsubseteq E, E \vdash \delta \text{ implies } E = E'$$

If $e \rightsquigarrow \delta$, Port's algorithm computes $\mathcal{E}(e, \delta)$ by finding a regular expression encoding all the minimal subsets of the initial type constraints (edges) in the term graph connecting the nodes representing the type terms of the constraint. For example, the expression corresponding to the minimal subsets of type equations for the example in Section 3 is $hf(i + dg)ec$. The "words" in the language of this regular expression are the minimal subsets $\{h, f, i, e, c\}$ and $\{h, f, d, g, e, c\}$.

4.1 Locations sets as slices

In general, a "slice" is any partially specified syntactic description of a program. The partial description of the syntax may be a set of subexpressions, a subgraph of the expression's syntax tree, or some other syntactic information about the expression.

First we define a slice to be a set of (locations of) subexpressions in a given expression: A *location set* is any subset of the set of locations of all subexpressions of a decorated expression. We assume that the typability-to-unification reduction function generates type constraints in a *syntax-directed* manner: for each type constraint, there is a location associated with its generation. It is easy to verify syntax-directedness for the function $\mathcal{R}_{\lambda_c^{if}}$ defined in Figure 6. This defines an *implication* relation $\stackrel{e}{\Longrightarrow}$ between locations and type constraints for each expression. (This was the reason for associating location names with type constraints in $\mathcal{R}_{\lambda_c^{if}}$.)

Were the relation between locations and the type constraints generated a one-to-one function, the minimal location sets contributing to a symptom could be directly computed from the set of minimal subsets of type constraints deriving symptom type constraint. Unfortunately, this is not so even in λ_c^{if}. For example, for $\mathsf{if}_1\ \mathtt{true}_2^{bool}\ 1_3^{\mathtt{int}}\ 0_4^{\mathtt{int}}$, $1 \stackrel{e}{\Longrightarrow} t_1 \doteq t_3$, $1 \stackrel{e}{\Longrightarrow} t_2 \doteq \mathtt{bool}$, and $2 \stackrel{e}{\Longrightarrow} t_2 \doteq \mathtt{bool}$.

Let $\mathbf{S}$ denote the set of all location indices. Let S_e be the set of location indices in e. We use the meta symbol s to refer to a location and the meta symbol S to refer to sets of locations.

Definition 4.4: A *location set* is a set of locations of program subexpressions. Location sets are ordered by the subset ordering: we write $S \sqsubseteq S'$ (read "S weaker than S'") if $S \subseteq S'$.

Any syntax-directed definition of the reduction function defines a relation $\stackrel{e}{\Longrightarrow} \subseteq \mathbf{S} \times \mathbf{E}$ such that $s \stackrel{e}{\Longrightarrow} \eta$ if and only if η was generated "at" location s in e.

We extend $\stackrel{e}{\Longrightarrow}$ to sets of locations and type constraints: ($\mathcal{P}(A)$ refers to the set of finite subsets of A.)

Definition 4.5: Given expression e, the relation $\stackrel{e}{\Longrightarrow} \subseteq \mathcal{P}(S_e) \times \mathcal{P}(E_e)$ is given by

$$S \stackrel{e}{\Longrightarrow} E \stackrel{\text{def}}{=} \forall \eta \in E, \exists s \in S. s \stackrel{e}{\Longrightarrow} \eta$$

The order on $\stackrel{e}{\Longrightarrow}$ is pairwise: $S \stackrel{e}{\Longrightarrow} E \sqsubseteq S' \stackrel{e}{\Longrightarrow} E' \stackrel{\text{def}}{=} S \sqsubseteq S' \wedge E \sqsubseteq E'$.

Notation: Let $S \sqsubseteq S_e$ and $E \sqsubseteq E_e$.
$S \stackrel{e}{\Longrightarrow} E \vdash \delta$ denotes $S \stackrel{e}{\Longrightarrow} E \wedge E \vdash \delta$.
$S \stackrel{e}{\rightsquigarrow} \delta$ denotes $\exists E. S \stackrel{e}{\Longrightarrow} E \wedge E \vdash \delta$.

The following is a straightforward consequence of the definition of $\stackrel{e}{\Longrightarrow}$:

Proposition 2 *Suppose $S \stackrel{e}{\Longrightarrow} E$ and $S \sqsubseteq S' \sqsubseteq S_e$ and $E' \sqsubseteq E$. Then $S' \stackrel{e}{\Longrightarrow} E$ and $S \stackrel{e}{\Longrightarrow} E'$.*

That is, $\stackrel{e}{\Longrightarrow}$ is upward closed on its first argument and downward closed on its second argument.

Proposition 3 *1. If $S \stackrel{e}{\Longrightarrow} E \vdash \delta$, then there is a minimal $E' \sqsubseteq E$ such that $S \stackrel{e}{\Longrightarrow} E' \vdash \delta$.*

2. If $S \stackrel{e}{\Longrightarrow} E$ then there is a minimal $S' \sqsubseteq S$ such that $S' \stackrel{e}{\Longrightarrow} E$.

Proof: By Propositions 1 and 2 □

Given that $e \rightsquigarrow \delta$, our goal is to specify all slices of e that generate just enough type constraints to imply δ. In other words, given $e \rightsquigarrow \delta$, we want the minimal elements of the ordered set $\{S \sqsubseteq S_e \mid S \stackrel{e}{\rightsquigarrow} \delta\}$, where the ordering $\sqsubseteq$ is given by Definition 4.4. These minimal elements, by definition, satisfy the following conditions.

Definition 4.6: A set of slices $\mathcal{S} \subseteq \mathcal{P}(S_e)$ satisfies the soundness, minimality, or completeness properties with respect to derivation of δ, if it satisfies, respectively

Soundness: $\forall S \in \mathcal{S}, S \stackrel{e}{\rightsquigarrow} \delta$.

Minimality: $\forall S \in \mathcal{S}, \mathit{min}(S, e, \delta)$, where

$$\mathit{min}(S, e, \delta) \stackrel{\text{def}}{=} \forall S' \sqsubseteq S, S' \stackrel{e}{\rightsquigarrow} \delta \text{ implies } S = S'$$

Completeness: $\forall S' \sqsubseteq S_e, S' \stackrel{e}{\rightsquigarrow} \delta$ implies $\exists S \in \mathcal{S}. S \sqsubseteq S'$.

The following proposition gives us a way to compute the minimal slices:

Proposition 4 *The set of minimal slices of e leading to δ can be computed as*
$\mathit{min} \bigcup_{E \in \mathcal{E}(e,\delta)} \mathit{min}\{S \sqsubseteq S_e \mid S \stackrel{e}{\Longrightarrow} E\}$

Proof:

$$\begin{aligned} & \mathit{min}\{S \sqsubseteq S_e \mid S \stackrel{e}{\rightsquigarrow} \delta\} \\ = \; & \mathit{min}\{S \sqsubseteq S_e \mid S \stackrel{e}{\Longrightarrow} E \wedge E \vdash \delta \text{ for some } E \sqsubseteq E_e\} \\ = \; & \mathit{min}\{S \sqsubseteq S_e \mid S \stackrel{e}{\Longrightarrow} E \wedge E \in \mathcal{E}(e,\delta)\} \text{ by Prop 3(1)} \\ = \; & \mathit{min} \bigcup_{E \in \mathcal{E}(e,\delta)} \mathit{min}\{S \sqsubseteq S_e \mid S \stackrel{e}{\Longrightarrow} E\} \text{ by Prop 3(2)} \end{aligned}$$

□

4.2 Abstract algorithm to compute minimal slices

The computation of $\mathcal{S}(e, \delta)$ follows the proof of Propositions 4:

1. Compute $\mathcal{E}(e, \delta)$.
2. Compute $\mathcal{S}_1 = \bigcup\{\mathcal{S}(e, E) \mid E \in \mathcal{E}(e, \delta)\}$, where $\mathcal{S}(e, E) = \{S \mid S \stackrel{e}{\Longrightarrow} E \wedge \mathit{min}(S, e, E)\}$ and
$$\mathit{min}(S, e, E) \stackrel{\text{def}}{=} \forall S' \sqsubseteq S. S' \stackrel{e}{\Longrightarrow} E \text{ implies } S' = S$$
3. Compute the set $\mathcal{S}_2$ of minimal elements (under $\sqsubseteq$) of $\mathcal{S}_1$.

Our definition of completeness is with respect to program slices rather than type constraint sets. In other words, it is not true that for any $E \in \mathcal{E}(e, \delta)$ there is an $S \in \mathcal{S}_2$ such that $S \stackrel{e}{\Longrightarrow} E$ and $E \vdash \delta$. This is because it is possible to have an $E \in \mathcal{E}(e, \delta)$ such that for every $S \in \mathcal{S}(e, E)$, there is an $S' \sqsubset S$ such that $S' \stackrel{e}{\Longrightarrow} E'$ and $E' \neq E$.

4.3 Lazy computation of slices

Port's algorithm computes in $O(n^3)$ time a (dag representation of) a regular expression whose language is $\mathcal{E}(e, \delta)$, where n is the size of the type constraint graph.

The lazy algorithm consists of picking a "term" w from the regular expression denoting $\mathcal{E}(e, \delta)$. Since the $\stackrel{e}{\Longrightarrow}$ relation is representable as a bipartite graph with vertices partitioned into S_e and E_e, finding $\mathcal{S}(e, w)$ corresponds to finding minimal dominator sets that dominate the vertices in w in the graph of $\stackrel{e}{\Longrightarrow}$. However, the minimality condition in step 3 of the abstract algorithm requires minimality with respect to *all* the elements of $\mathcal{S}_1$. In principle this is impractical, since the number of minimal subsets may be exponential. At present we do not know an efficient way to generate an

expression encoding all the minimal elements of $\mathcal{S}_2$. In practical situations, however, the non-minimality of the slices as a result of omitting step 3 of the abstract algorithm is unlikely to be a serious problem. This is because, in the context of locating obscure type bugs, which is where slices will be most useful, we expect the total number of slices in $\mathcal{S}_1$ to be quite small even for large programs.

5 Syntactic constraints

Location-set slices indicate which subexpressions of an ill-typed program generated type constraints that resulted in the type error. They do not specify *how* these locations interacted to cause the error. This section introduces a program-slicing method that reveals the structural relationships between program fragments that are instrumental in causing a type error.

We introduce predicates over location indices of subexpressions and other values (type expressions). Terms involving such predicates we call *syntatic constraints*. A *partial syntactic description* of a program is a set of syntactic constraints.

To better isolate locations "relevant" to the generation of a type constraint, we add a "bottom" element, $\_$, to our domain of syntactic objects, which is a placeholder for irrelevant information.

Definition 5.7:

Let $N = \{0, 1, \ldots\}$ be a set of *indices* and $N_\_$ be the flat domain of *locations* obtained by lifting N with $\_$.

Let $C = \{c \mid c^\tau \in \mathcal{C}\}$ be the set of *constant names* and $C_\_$ be the flat domain obtained by lifting C with $\_$.

Let $\langle T^\Sigma_\_, \sqsubseteq\rangle$ denote the partially ordered free-algebra of terms obtained from any signature Σ that consists of at least the elements of $N_\_$ treated as nullary constants. Call the elements of $T^\Sigma_\_$ *t-terms*. T-terms of the form i, where $i \in N$ are usually written as t_i.

The syntax of types $\tau \in T_\_$ for λ^{if}_c changes to accommodate $\_$ in the following way:

$$\tau ::= t_i \mid \tau{\rightarrow}\tau' \mid \texttt{bool} \mid \texttt{int} \mid \_ \qquad (6)$$

where $i \in N$.

What relations are used as syntactic constraints depends on what information about the syntax tree is deemed interesting. For the λ^{if}_c, we define them as follows:

Definition 5.8:

A *syntactic constraint* (or s-term) $s \in \mathbf{S}$ is any term of the following form:

1. $\kappa(i_0, \ldots, i_n)$ where $i_0, \ldots, i_n \in N_\_$ and (κ, n) is an n-ary constructor of a compound expression of λ^{if}_c: $(\kappa, n) \in \{(\mathsf{lambda}, 2), (@, 2), (\mathsf{if}, 3)\}$.
2. $\lambda\mathsf{bind}(i, j)$, where $i, j \in N_\_$.
3. $\mathsf{const}(i, c, \tau)$, where $i \in N_\_$, $c \in C_\_$, and $\tau \in T_\_$.

The partial order on $N_\_$, $C_\_$, and $T_\_$ induces a partial order $(\mathbf{S}, \sqsubseteq)$:

- $\kappa(i_0, \ldots, i_n) \sqsubseteq \kappa(i'_0, \ldots, i'_n)$ where $i_j \sqsubseteq_{N_\_} i'_j$, $0 \le j \le n$, and (κ, n) is an n-ary constructor of a compound expression of λ^{if}_c.
- $\lambda\mathsf{bind}(i_0, i_1) \sqsubseteq \lambda\mathsf{bind}(i'_0, i'_1)$, where $i_j \sqsubseteq_{N_\_} i'_j$, $0 \le j \le 1$.
- $\mathsf{const}(i, c, \tau) \sqsubseteq \mathsf{const}(i', c', \tau')$, where $i \sqsubseteq_{N_\_} i'$,$c \sqsubseteq_{C_\_} c'$, and $\tau \sqsubseteq_{T_\_} \tau'$.

Informally, $s \sqsubseteq s'$ if s can be obtained from s' by erasure of (zero or more) subterms of s'.

Definition 5.9: A *partial syntactic description* S is any finite set of syntactic constraints. The partial order on partial syntactic descriptions is the inclusion order on $\mathcal{O}(\mathbf{S})$, the set of all downward closed subsets of $\mathbf{S}$. $S_1 \sqsubseteq S_2$ if and only if $\downarrow S_1 \subseteq \downarrow S_2$, where $\downarrow S = \{s \in \mathbf{S} \mid \exists s' \in S. s \sqsubseteq s'\}$.

Definition 5.10: A type constraint $\eta \in \mathbf{E}$ is an unordered tuple $\tau \doteq \tau'$, where $\tau, \tau' \in T_\_$.

The partial order on t-terms extends to type constraints and sets of type constraints:

$$\tau_1 \doteq \tau_2 \sqsubseteq \tau'_1 \doteq \tau'_2 \stackrel{\text{def}}{=} \tau_1 \sqsubseteq \tau'_1 \wedge \tau_2 \sqsubseteq \tau'_2 \vee \tau_1 \sqsubseteq \tau'_2 \wedge \tau_2 \sqsubseteq \tau'_1$$

The partial order on sets of type constraints is the inclusion order on $\mathcal{O}(\mathbf{E})$. i.e., $E_1 \sqsubseteq E_2 \stackrel{\text{def}}{=} \downarrow E_1 \subseteq \downarrow E_2$ where $\downarrow E = \{\eta \in \mathbf{E} \mid \exists \eta' \in E. \eta \sqsubseteq \eta'\}$.

5.1 Relating syntactic and type constraints

Just as the type system specifies how individual subexpressions of a (well-typed) expression contribute to the type of the overall expression, we would like to specify how an individual syntactic constraint engenders a type constraint. To this end the relation $\Longrightarrow \subseteq \mathbf{S} \times \mathbf{E}$ is defined by the schema given in Figure 8. The order on *implies* is pairwise:

$$s \Longrightarrow \eta \;\sqsubseteq\; s' \Longrightarrow \eta' \stackrel{\text{def}}{=} s \sqsubseteq s' \text{ and } \eta \sqsubseteq \eta'$$

Once again our syntax directed reduction function for reducing typability of e to unification of type constraints defines an implication relation between syntactic and type constraints. This is accomplished with the following:

Definition 5.11: Given an expression e, $\overset{e}{\Longrightarrow}\subseteq\Longrightarrow$ is given by the down-set $\downarrow (\overset{e}{\underset{*}{\Longrightarrow}})$, where $\overset{e}{\underset{*}{\Longrightarrow}} \subseteq \Longrightarrow$ is given in Figure 9 $\overset{e}{\Longrightarrow}$ extends to sets of syntactic and type constraints:

$$S \overset{e}{\Longrightarrow} E \stackrel{\text{def}}{=} \forall \eta \in E, \exists s \in S. s \overset{e}{\Longrightarrow} \eta$$

$\lambda(i_0, i_1, i_2) \Longrightarrow i'_0 \doteq i'_1 \rightarrow i'_3,\ i'_j \sqsubseteq i_j, 0 \leq j \leq 2$

$@(i_0, i_1, i_2) \Longrightarrow i'_1 \doteq i'_2 \rightarrow i'_0,\ i'_j \sqsubseteq i_j, 0 \leq j \leq 2$

$\mathsf{if}(i_0, i_1, i_2, i_3) \Longrightarrow i'_1 \doteq \mathtt{bool},\ i'_1 \sqsubseteq i_1.$

$\mathsf{if}(i_0, i_1, i_2, i_3) \Longrightarrow i'_0 \doteq i'_2,\ i'_0 \sqsubseteq i_0, i'_2 \sqsubseteq i_2.$

$\mathsf{if}(i_0, i_1, i_2, i_3) \Longrightarrow i'_2 \doteq i'_3\ i'_2 \sqsubseteq i_2, i'_3 \sqsubseteq i_3.$

$\lambda\mathsf{bind}(i_0, i_1) \Longrightarrow i'_0 \doteq i'_1,\ i'_0 \sqsubseteq i_0, i'_1 \sqsubseteq i'_1.$

$\mathsf{const}(i, c, \tau) \Longrightarrow i' \doteq \tau',\ i' \sqsubseteq i, \tau' \sqsubseteq \tau$

Figure 8: The relation $\Longrightarrow$ between syntactic and type constraints

$\overset{e}{\Longrightarrow} = \mathcal{W}(\emptyset, e)$ **where**

$\mathcal{W}(A, e_i) =$ **case** e_i of

x_i	**case** $A(x)$ of
	$(\lambda : j)$ $\quad \{\lambda\mathsf{bind}(i,j) \overset{e}{\Longrightarrow} t_j \doteq t_i\}$
c_i^τ	$\{(i, c, \tau) \overset{e}{\Longrightarrow} t_i \doteq \tau\}$
$\mathsf{if}_i\ \ e_j\ e_k\ e_m$	$\mathcal{W}(A, e_j) \cup \mathcal{W}(A, e_k) \cup \mathcal{W}(A, e_m) \cup$
	$\{\mathsf{if}(i,j,k,l) \overset{e}{\Longrightarrow} t_j \doteq \mathtt{bool},$
	$\mathsf{if}(i,j,kl) \overset{e}{\Longrightarrow} t_k \doteq t_m,$
	$\mathsf{if}(i,j,k,l) \overset{e}{\Longrightarrow} t_i \doteq t_k\}$
$@_i\ e_j\ e_k$	$\mathcal{W}(A, e_j) \cup \mathcal{W}(A, e_k) \cup$
	$\{@(i,j,k) \overset{e}{\Longrightarrow} t_j \doteq t_k \rightarrow t_i\}$
$\lambda_i\ x_j.e_k$	$\mathcal{W}(A, \{x : j\}, e_k) \cup$
	$\{\mathsf{lambda}(i,j,k) \overset{e}{\Longrightarrow} t_i \doteq t_j \rightarrow t_k\}$

Figure 9: Definition of $\overset{e}{\underset{*}{\Longrightarrow}}$

Let S_e be $domain(\overset{e}{\underset{*}{\Longrightarrow}})$ and E_e be $range(\overset{e}{\underset{*}{\Longrightarrow}})$.

The set of minimal partial syntactic descriptions of an expression e that imply a type constraint set E is

$$\mathcal{S}(e, E) = \{S \sqsubseteq S_e \mid S \overset{e}{\Longrightarrow} E \ \wedge\ min(S, E)\}$$

where $min(S, E) \overset{\text{def}}{=} \forall S' \sqsubseteq S. S' \overset{e}{\Longrightarrow} E$ implies $S = S'$.

Definition 5.12: $E \vdash \delta \overset{\text{def}}{=} \delta \in E^*$, where E^* is the closure under the rules in Figure 7 augmented with the rule

- (Removal of $\_$) Remove any constraint of the form $\_ \doteq \tau$ where $\tau \in T_\_$ before applying any other rule.

5.2 Abstract algorithm

The abstract algorithm is the same as in Section 4, since the new definitions of type constraints, slices, $\vdash$, and $\sqsubseteq$ are such that Propositions 1, 2, 3, and 4 fall hold.

The minimal E's are obtained by a refinement to Port's algorithm that requires the inclusion of directed edges of the unfication graph in the specification of the path expressions. The refinement is straightforward and will be reported elsewhere [3]. Figure 10 shows the derivation of the minimal "proofs," along with the related syntactic and type constraints for the example of Section 3.

6 Polymorphism

In this section we sketch how our analysis may be extended to ML-style polymorphic type reconstruction. Rather than using generic type variables, we adopt a technique due to Henglein that reduces the type reconstruction problem of ML to solving a system of equations and inequations (SEI) by semiunification [8]. Type constraints are defined as type equations or inequations of the form $\tau \leq \tau'$, where τ and τ' are type terms. The crucial advantage of this approach is that the type system can be specified by a system of deduction rules in which the collection of side conditions implied by a program form an instance of the semiunification problem.

Traditional presentations, on the other hand, employ side conditions that do not have the form of a type equation or inequation, for in the LET rule the side condition for "closing" over a type is expressed as a set equation [4, 15]. The set equation is neither a type inequation or equation.

Henglein's framework enables the definition of $\overset{e}{\Longrightarrow}$ as a relation between syntactic constraints and type constraints; the traditional approach would require a third relational element not expressible as a type equation or inequation.

The reduction to semiunification does not introduce any new syntax for type terms; types are still specified by the grammar in Equation 2. However, the form of type constraints now includes type inequations of the form described above.

The graph rewrite rules implementing semiunification add not only new edges but also new vertices to the term graph. Port's technique can be extended to associate source information with each new edge (or vertex) in the type constraint graph. Details will be reported elsewhere [3].

7 Conclusions

Most systems of static program analysis provide too little or too much diagnostic information when the result expected by the user does not match the results of the analysis.

Rather than providing an ad hoc algorithm, the framework presented here makes it possible to precisely characterize what diagnostic information to provide so as to adequately account for a consequence of the type deduction system. As far as we know, this is the first comprehensive diagnostic analysis of type errors in terms of minimal proofs of untypability and minimal program fragments implying

path : $r^{-1}hfpiec$			path : $s^{-1}hfqdgec$		
$\mathsf{const}(6, \_, \mathtt{int}\to\_)$	$\stackrel{e}{\Longrightarrow}$	$\mathtt{int}\to\_ \doteq t_6$	$\mathsf{const}(6, \_, \_\to\mathtt{int})$	$\stackrel{e}{\Longrightarrow}$	$\_\to\mathtt{int} \doteq t_6$
$@(\_, 6, 7)$	$\stackrel{e}{\Longrightarrow}$	$t_6 \doteq t_7\to\_$	$@(4, 6, \_)$	$\stackrel{e}{\Longrightarrow}$	$t_6 \doteq \_\to t_4$
$\lambda\mathsf{bind}(7, 1)$	$\stackrel{e}{\Longrightarrow}$	$t_7 \doteq t_1$	$\mathsf{if}(\_, \_, 4, 5)$	$\stackrel{e}{\Longrightarrow}$	$t_4 \doteq t_5$
$\lambda\mathsf{bind}(3, 1)$	$\stackrel{e}{\Longrightarrow}$	$t_1 \doteq t_3$	$\lambda\mathsf{bind}(5, 1)$	$\stackrel{e}{\Longrightarrow}$	$t_5 \doteq t_1$
$\mathsf{if}(\_, 3, \_, \_)$	$\stackrel{e}{\Longrightarrow}$	$t_3 \doteq \mathtt{bool}$	$\lambda\mathsf{bind}(3, 1)$	$\stackrel{e}{\Longrightarrow}$	$t_1 \doteq t_3$
			$\mathsf{if}(\_, 3, \_, \_)$	$\stackrel{e}{\Longrightarrow}$	$t_3 \doteq \mathtt{bool}$

Figure 10: Minimal proofs and constraint sets for example program

type constraints that have as their consequence a give type constraint.

This paper introduces the notion of syntactic constraints that partially describe the syntax of a program. It is *relations* between program subexpressions that give rise to constraints between types or other attributes of the subexpressions. This leads to an implication relation between syntactic and attribute constraints.

The framework presented here assumes that analysis of program properties can be reduced to solving systems of constraints over attributes. If the reduction is syntax-directed then every attribute constraint generated by the reduction is associated with some syntactic constraint.

Our technique is directly applicable to the problem of diagnosing results of constraint-based binding-time analysis [7, 10]. Since well-annotatedness corresponds to well-typing, a program slice associated with an annotation of a particular subexpression will reveal those program elements that contribute to the subexpression's annotation.

We intend to implement these techniques in a "debugging" front end of an existing type inference system. Due to algorithmic complexity of the techniques proposed here, we do not expect them to be used in routine tracking of simple type errors. Rather, they may be invoked when a type error is especially difficult to understand given the feedback provided by a traditional type inference system.

It appears possible to define program slices with respect to the result of other static analyses such as set-based or data-flow analyses. For instance, if a set-based analysis includes a particular value v in the set approximating the value of a program subexpression, techniques similar to those introduced in this paper might be used to identify minimal slices that contribute to the presence of v in the approximation.

References

[1] Beaven, M., and Stansifer, R. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages* (1994).

[2] Cardelli, L. Basic polymorphic typechecking. *Science of Computer Programming 8* (1987), 147–172.

[3] Choppella, V. *Type-based diagnostic program analysis.* PhD thesis, Indiana University, 1995. Forthcoming.

[4] Damas, L., and Milner, R. Principal type-schemes for functional languages. In *Proc. 9th ACM Symp. on Principles of Programming Languages* (January 1982), pp. 207–212.

[5] Duggan, D., Ophel, J., and Bent, F. Explaining type reconstruction. In *ACM SIGPLAN Programming Language Design and Implementation* (June 1994), ACM.

[6] Gomard, C. K. Partial type inference for untyped functional programs. In *Proceedings of the 17th ACM Symposium on Programming Languages* (1990).

[7] Henglein, F. Efficient type inference for higher-order binding-time analysis. In *Functional Programming Languages and Computer Architecture* (1991), pp. 448–472.

[8] Henglein, F. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems 15*, 2 (April 1993), 253–289.

[9] Johnson, G. F., and Walz, J. A. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of the 13th ACM Symposium on Programming Languages* (1986), pp. 44–57.

[10] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1992.

[11] Lassez, J., Maher, M. J., and Marriot, K. Unification revisited. In *Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann, 1988, ch. 15, pp. 587–625.

[12] Martelli, A., and Montanari, U. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems 4*, 2 (April 1982), 258–282.

[13] Maruyama, H., Matsuyama, M., and Araki, K. Support tool and strategy for type error correction with

polymorphic types. In *Proceedings of the Sixteenth annual international computer software and applications conference, Chicago* (September 1992), IEEE, pp. 287–293.

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

[15] Mitchell, J. C. Type systems for programming languages. In *Handbook of Theoretical Computer Science*, van Leeuwen et al., Eds. North-Holland, 1991.

[16] Paterson, M., and Wegman, M. Linear unification. *Journal of Computer and System Sciences 16*, 2 (1978), 158–167.

[17] Port, G. S. A simple approach to finding the cause of non-unifiability. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium* (1988), R. A. Kowalski and K. A. Bowen, Eds., MIT Press, pp. 651–665.

[18] Reiter, R. A theory of diagnosis from first principles. *Artificial Intelligence 32* (1987), 57–95.

[19] Soosaipillai, H. An explanation based polymorphic type checker for Standard ML. Master's thesis, Heriot-Watt University, 1990.

[20] Walz, J. A. *Extending Attribute Grammars and Type Inference Algorithms.* PhD thesis, Cornell University, February 1989. TR 89-968.

[21] Wand, M. Finding the source of type errors. In *13th Annual ACM Symp. on Principles of Prog. Languages.* (January 1986), pp. 38–43.

[22] Wand, M. A simple algorithm and proof for type inference. *Fundamenta Informaticae 10* (1987), 115–122.

[23] Weiser, M. Program slicing. *IEEE Transactins on Software Engineering 10*, 4 (1984), 352–357.