Guided Type Debugging

Sheng Chen and Martin Erwig

Oregon State University
{chensh,erwig}@eecs.oregonstate.edu

Abstract. We present guided type debugging as a new approach to quickly and reliably remove type errors from functional programs. The method works by generating type-change suggestions that satisfy type specifications that are elicited from programmers during the debugging process. A key innovation is the incorporation of target types into the type error debugging process. Whereas previous approaches have aimed exclusively at the removal of type errors and disregarded the resulting types, guided type debugging exploits user feedback about result types to achieve better type-change suggestions. Our method can also identify and remove errors in type annotations, which has been a problem for previous approaches. To efficiently implement our approach, we systematically generate all potential type changes and arrange them in a lattice structure that can be efficiently traversed when guided by target types that are provided by programmers.

Keywords: Type debugging, type inference, error localization, type error messages, choice types, change suggestions

1 Introduction

One beauty of the Hindley-Milner type system is the type inference mechanism that computes principal types for expressions without any type annotation. However, when type inference fails, it is often difficult to locate the origin of type errors and deliver precise feedback to the programmer. Despite numerous efforts devoted to improve type error diagnosis in past three decades [22, 13, 12, 19, 25], every proposed approach behaves poorly in certain situations.

A major problem for the localization and removal of type errors is the inherent ambiguity in this problem. For example, the type error in the expression `not 1` can be fixed by either replacing the function or the argument. Without any additional information it is not clear what the correct solution is. In such a situation type checkers that produce suggestions for how to fix a type error have to fall back on some form of heuristics [13, 12, 14, 25, 4] to select or rank their recommendations. These heuristics are often based on some complexity measure for suggestions (for example, prefer simple changes over complex ones), or they try to assess the likelihood of any particular suggestion. A problem with most of these approaches is that while they may work reasonably well in some cases, they can also go wrong and be misleading. This presents a problem since lack of precision in tools leads to distrust by users. Moreover, for novices it can add to confusion and frustration [12].

```
rev [] = []
rev (x:xs) = rev xs ++ x

last xs = head (rev xs)
init = rev . tail . rev

rR xs = last xs : init xs
```

```
last :: [[a]]->a
(1) Is intended type an instance? (y/n) n

... interactions (2) through (8) omitted

(++) (rev xs) :: [b]->[b]
rev :: a->[b]
xs :: a
(9) Are intended types an instance? (y/n) y
Error located. Wrong expression:
(rev xs) ++ x
```

Fig. 1: An ill-typed program (left) together with an interaction session between a user and an algorithmic debugging tool (right) [7] .

As a partial solution to this problem programmers are advised to add type annotations to their program.[1] Type annotations can assist type checkers in producing better error messages, and they also enhance the readability of programs.[2]

However, type annotations are not without problems. In particular, too much trust in type annotations can have a negative impact on the precision of error localization. This happens when type annotations themselves are erroneous. A type checker that assumes that type annotations are always correct will not only miss the error in the annotation, but will also produce wrong error messages and misleading suggestions.

Incorrect type annotations are not a fringe phenomenon. Precisely because type annotations represent a useful form of redundancy in programs, they are widely used, and thus it may very well happen that, over time, they get out of sync with the rest of the program. This can happen, for example, when an annotation is not updated after its corresponding expression is changed or when another part of the program is changed so that it relies on a more generic form of an expression than expressed by the annotation. We have investigated a set of over 10,000 Haskell programs, which were written by students learning Haskell [10]. Of those, 1505 contained type errors (the remaining programs were well typed or contained parsing errors or unbound variables). We found that over 20% of the ill-typed programs contained wrong type annotations. We will discuss the impact of incorrect type annotations on type error messages in Section 5 in more depth.

We seem to face a dilemma now. To produce better feedback about type errors we have to rely on some form of user input, and information about the intended type of expressions is extremely helpful for this. At the same time, type annotations are not always reliable and sometimes even cause more problems. A solution is to elicit user input in a systematic and targeted manner. Specifically, we should ask for type information at the fewest number of possible places and where this information is most beneficial to our type debugger. This strategy ensures the availability of the latest up-to-date type information and avoids the potential problems with type annotations discussed earlier.

[1] http://en.wikibooks.org/wiki/Haskell/Type_basics#Type_signatures_in_code

[2] Type annotations can also improve the performance of type checkers. They also help make type inference decidable in richer type systems. A discussion of these aspects is beyond the scope of this paper.

```
What is the expected type of rR?
[a] -> [a]

Potential fixes:
1  change x from type a to type [a].
2  change ++
     from type [a] -> [a] -> [a]
     to type [a] -> a -> [a]
There are no other one-change fixes.
Show two-change fixes? (y/n)
```

```
What is the expected type of rR?
[[a]] -> [a]

Potential fixes:
1  change rev (in tail. rev)
     from type [[a]] -> [a] to type [a] -> [a]
2  change tail (in tail . rev)
     from type [a] -> [a] to type [a] -> [[a]]
Show more one-change fixes? (y/n)
```

Fig. 2: Guided Type Debugging. The target type for `rR` is `[a] -> [a]` (left) and `[[a]] -> [a]` (right). User inputs are shown in *italics*.

The idea of systematically eliciting user input for debugging (type) errors is not new [7, 20]. To illustrate our approach and compare it with these previous systems we present in Figure 1 an ill-typed program, taken from [7] (with some of the names changed). The error is attributed to the expression `rev xs ++ x`, and `x` should be replaced by `[x]`. We also show part of an interaction session between a user and the algorithmic debugger [7]. (Of the omitted seven questions, five are similar to `(1)`, asking questions about variables, whereas two are similar to `(9)`, asking questions about subexpressions and the relationships between their types.) The general strategy of this method is to systematically inquire about the correctness of each subexpression once an expression has been identified as ill typed (`rR` in the example). The first question is thus about the function `last`. If users respond "yes" to the question about a subexpression, then no more questions about that subexpression will be asked. Otherwise, the algorithmic debugger will switch the focus to that subexpression and ask questions about its subexpressions. From the debugging session, we can observe the following.

- To find the source of an error, the debugger has to work through chains of function calls and fragments of function definitions.
- The debugger interacts with users in typing jargon. Moreover, users have to track and connect information when the same type variable appears at different places (as, for example, the type variable `a` in the types for `rev` and `xs`).
- The length of a debugging trace depends on the distance between the origin of a type error and where it manifests itself. This distance can be large.
- The type debugger works in a linear fashion, and it is unclear how to support cases in which there is more than one type error.
- The final change suggestion delivered by the debugger still leaves some work for users to figure out what exactly the cause of the type error is and how to fix it.

The interactive type debugger [20] follows the same strategy and thus suffers from similar problems. We argue that even with the assistance of such debuggers, locating and removing type errors is still a nontrivial and arduous task.

In contrast, the guided type debugging (GTD) approach developed in this paper asks programmers to provide simple type signatures only. Moreover, most of the time, only one signature is needed to lead to a suggestion for how to fix the type error. More specifically, for a single expression, exactly one signature is needed. For a program with multiple function definitions and expressions containing type errors, we have to

distinguish between several cases. First, if only one expression is ill typed, GTD solicits a type annotation for that expression. In case there are more expressions that are ill typed, GTD asks for a type annotation for the first ill-typed expression. In case the program still contains ill-typed expressions after the user has fixed the first one, GTD again asks for a type annotation for the next ill-typed expression. This process repeats until the whole program becomes well typed. Note that the expression for which GTD requests a type annotation is not necessarily the cause of the type error, and GTD will point to the most likely cause of the type error in the program.

Figure 2 shows two examples. Here the debugger first asks the programmer for the intended type of the ill-typed expression `rR`. If the target type is `[a] -> [a]`, the debugger infers that there are exactly two potential suggestions with only one change to the program. The first suggestions is to change `x`, whose inferred type is `a`, to something of type `[a]`. The second suggestion is to change `++` of type `[a] -> [a] -> [a]` to something of type `[a] -> a -> [a]`. There are, of course, other changes that can lead the expression `rR` to the target type, but each such suggestion requires changes in at least two places. (The right half of Figure 2 shows the suggestions in case the target type is `[[a]] -> [a]`.)

With a user interface, we can envision a more flexible way of how GTD may be used. First, GTD type checks the program and marks all expressions that are ill typed. The user can go to any expression and specify the intended result type. GTD will then suggest a most likely change that satisfies the user's intention. Note that the user may even specify an intended type for a well-typed expression. For example, if the user specifies `[[a]] -> [[a]]` as an expected type for `foldr (:) []`, which is well typed, GTD will suggest to change `(:)` of type `a -> [a] -> [a]` to something of type `[a] -> [a] -> [a]`.

In summary, GTD will ask programmers significantly fewer questions than algorithmic type debugging. GDT will then work out the details and show exact change locations and suggestions. Moreover, GTD supports changes involving multiple locations.

To work as indicated, the GTD method has to find all potential changes that can in principle fix a particular type error. Moreover, it must be able to select among all changes those that satisfy the user-provided intended types of expressions. At the same time, all these tasks have to be done efficiently.

We employ counter-factual typing [4] (CF typing for short) to realize the first task. CF typing computes all potential type changes for fixing type errors by typing expressions once. For each change, it returns the information about change locations, the expected type for each location, and the result type of applying the change. We describe the concept of variational types as the underlying representation in Section 2 and the method of CF typing in Section 3.

To implement the second task efficiently, we exploit the instance-of relationship among result types of change suggestions. Specifically, we can arrange all changes in a lattice because changes involving more locations always produce more general result types than changes involving fewer locations. Given a user-provided target type, we can search through this lattice efficiently and narrow down the set of changes to a

manageable size. We describe the idea of type-change lattices and how they can help to find good type-change suggestions in Section 4.

The question of how GTD can deal with erroneous type annotations is discussed in Section 5. Section 6 discusses related work, and Section 7 concludes the paper.

2 Representing Type Errors by Variational Types

A type error results when the rules of a type system require an expression to have two conflicting (that is, non-unifiable) types. This happens, for example, when a function is applied to an argument of the wrong type or when branches of a conditional have different types. One of the simplest examples of a type error is a conflicting type annotation, as in the expression $e = \texttt{3::Bool}$. Now the problem for a type checker is to decide whether to consider the annotation or the type of the value to be correct. Without any further information this is impossible to know. Therefore, we should defer this decision until more context information is available that indicates which one of the two is more compatible with the context.

To represent conflicting type for expressions we employ the concept of *choice types* [6]. A choice type $D\langle\phi_1,\phi_2\rangle$ has a *dimension* D and contains two alternative types ϕ_1 and ϕ_2. For example, we can express the uncertainty about the type of the expression e with the choice type $D\langle\texttt{Int},\texttt{Bool}\rangle$. Choice types may be combined with other type constructors to build *variational types*, in which choices may be nested within type expressions, as in $D\langle\texttt{Int},\texttt{Bool}\rangle \to \texttt{Int}$.

We can eliminate choices using a selection operation, which is written as $\lfloor\phi\rfloor_s$. Here ϕ is a variational type, and s is a *selector*, which is of form $D.1$ or $D.2$. Selection replaces all occurrences of $D\langle\phi_1,\phi_2\rangle$ with its ith alternative. Choice types with different dimensions are independent of one another and require separate selections to be eliminated, whereas those with the same dimension are synchronized and are eliminated by the same selection. For example, $A\langle\texttt{Int},\texttt{Bool}\rangle \to A\langle\texttt{Bool},\texttt{Int}\rangle$ encodes two function types, but $A\langle\texttt{Int},\texttt{Bool}\rangle \to B\langle\texttt{Bool},\texttt{Int}\rangle$ encodes four function types. We use δ to range over decisions, which are sets of selectors (usually represented as lists). The selection operation extends naturally to decisions through $\lfloor\phi\rfloor_{s:\delta} = \lfloor\lfloor\phi\rfloor_s\rfloor_\delta$. In this paper, when we select δ from a type ϕ, we assume that selection eliminates all choices in ϕ.

The idea of CF typing is to systematically generate all changes (for variables and constants) that could fix a type inconsistency. Each such change is represented as a choice between the current and the new type. In the example, the choice type $A\langle\texttt{Int},\alpha_1\rangle$ is created for `3`.[3] The first alternative denotes the current type of `3`, and the second alternative denotes a type that can make `3` well typed within its context. We can think of the first alternative as the type `3` should have when it is not the cause of type errors and the second alternative as the type `3` ought to have when it is. Similarly, the annotation `Bool` may also be the cause of the type error. We thus create the choice type $B\langle\texttt{Bool},\alpha_2\rangle$, which says that if the type annotation is correct, it has the type `Bool`, otherwise it has the type α_2, an arbitrary type that makes the context well typed.

[3] One might wonder why a type variable is chosen and not just the type `Bool`. The reason is that when we are typing `3`, we have no knowledge about its context yet. We thus use α_1 to allow it to acquire any type that its context dictates.

But what should be the type of e? Usually, when we have two sources of type information for one expression, we unify the two types. Thus we would expect the unification result of $A\langle \texttt{Int}, \alpha_1 \rangle$ and $B\langle \texttt{Bool}, \alpha_2 \rangle$ to be the result type of e. However, the two choice types are not unifiable because `Bool` and `Int` fail to unify. (This is not surprising since the expression contains a type error.)

We address this problem through the introduction of *error types* [5], written as $\bot$, to represent non-unifiable parts of choice types. Specifically, we have developed a unification algorithm that computes a substitution for two variational types that is (a) most general and (b) introduces as few error types as possible. Because of the possibility of error types we call such substitutions *partial unifiers*. For the two types $A\langle \texttt{Int}, \alpha_1 \rangle$ and $B\langle \texttt{Bool}, \alpha_2 \rangle$, the algorithm computes the following partial unifier.

$$\theta = \{\alpha_1 \mapsto A\langle \alpha_4, B\langle \texttt{Bool}, \alpha_3 \rangle \rangle, \alpha_2 \mapsto B\langle \alpha_5, A\langle \texttt{Int}, \alpha_3 \rangle \rangle\}$$

The algorithm also computes a *typing pattern* that captures the choice structure of the result type and represents, using error types, those variants that would lead to a type error. In our example, the typing pattern is $\pi = A\langle B\langle \bot, \top \rangle, \top \rangle$. It indicates that the types at $[A.1, B.1]$ fail to unify ($\bot$) and all other variants unify successfully ($\top$). The typing pattern is used to mask the result type that can be obtained from the partial unifier. From θ and π we obtain the type $\phi = A\langle B\langle \bot, \texttt{Int} \rangle, B\langle \texttt{Bool}, \alpha_3 \rangle \rangle$ for e. Finally, from θ and ϕ we can derive following changes to potentially eliminate the type error in e.

- If we don't change e, that is, if we select $[A.1, B.1]$ from ϕ, there is a type error.
- If we change `3` but don't change the annotation `Bool`, that is, if we select $\delta = [A.2, B.1]$ from ϕ, we get the type `Bool`. Moreover, by selecting δ from $\theta(\alpha_1)$, we get `Bool`, which is the type that `3` should to be changed to.
- If we change the annotation `Bool` but don't change `3`, that is, if we select $\delta = [A.1, B.2]$ from ϕ, we get the result type `Int`. Moreover, by selecting δ from $\theta(\alpha_2)$ we get `Int` as the type the annotation `Bool` ought to be changed to.
- If we change both `3` and `Bool`, that is, if we select $[A.2, B.2]$ from ϕ, we get a more general type α_3. This means that e can be changed to some arbitrary value of any type. Note that α_3 will be very likely refined to a more concrete type if e occurs as a subexpression within some other context.

3 Counter-Factual Typing

In this section we describe the CF typing method [4], extended to handle type annotations. We work with the following syntax for expressions and types.

Expressions	e, f	::=	$c \mid x \mid \lambda x.e \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid e :: \tau$
Monotypes	τ	::=	$\gamma \mid \alpha \mid \tau \to \tau$
Variational types	ϕ	::=	$\tau \mid \bot \mid D\langle \phi, \phi \rangle \mid \phi \to \phi$
Type schemas	σ	::=	$\phi \mid \forall \overline{\alpha}.\phi$
Choice environments	Δ	::=	$\varnothing \mid \Delta, (l, D\langle \phi, \phi \rangle)$

$$\textsc{Con}\ \frac{c \text{ is of type } \gamma \qquad D \text{ fresh}}{\Gamma \vdash c : D\langle\gamma,\phi\rangle | \{(\ell(c), D\langle\gamma,\phi\rangle)\}} \qquad \textsc{Var}\ \frac{\Gamma(x) = \forall\overline{\alpha}.\phi_1 \qquad D \text{ fresh} \qquad \phi = \{\overline{\alpha \mapsto \phi'}\}(\phi_1)}{\Gamma \vdash x : D\langle\phi,\phi_2\rangle | \{(\ell(x), D\langle\phi,\phi_2\rangle)\}}$$

$$\textsc{Unbound}\ \frac{x \notin dom(\Gamma) \qquad D \text{ fresh}}{\Gamma \vdash x : D\langle\bot,\phi\rangle | \{(\ell(x), D\langle\bot,\phi\rangle)\}} \qquad \textsc{Ant}\ \frac{\Gamma \vdash e : \phi_1|\Delta \qquad D \text{ fresh} \qquad \phi_2 = D\langle\tau,\phi'\rangle \qquad \pi_1 = \phi_1 \bowtie \phi_2 \qquad \phi_3 = \pi_1 \lhd \phi_1}{\Gamma \vdash (e::\tau) : \phi_3 | \Delta \cup \{(\ell(\tau), \phi_2)\}}$$

$$\textsc{Abs}\ \frac{\Gamma, x \mapsto \phi \vdash e : \phi'|\Delta}{\Gamma \vdash \lambda x.e : \phi \to \phi'|\Delta} \qquad \textsc{Let}\ \frac{\Gamma, x \mapsto \phi \vdash e : \phi|\Delta \qquad \overline{\alpha} = FV(\phi) - FV(\Gamma) \qquad \Gamma, x \mapsto \forall\overline{\alpha}.\phi \vdash e' : \phi'|\Delta'}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \phi'|\Delta \cup \Delta'}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \phi_1|\Delta_1 \qquad \Gamma \vdash e_2 : \phi_2|\Delta_2 \qquad \phi_2' \to \phi' = \uparrow(\phi_1) \qquad \pi = \phi_2' \bowtie \phi_2 \qquad \phi = \pi \lhd \phi'}{\Gamma \vdash e_1\ e_2 : \phi|\Delta_1 \cup \Delta_2}$$

Fig. 3: Rules for counter-factual typing

We use c, γ, and α to range over value constants, type constants, and type variables, respectively. We have seen error types $\bot$ and choice types $D\langle\phi,\phi\rangle$ in Section 2. To simplify the discussion, we assume that type annotations are monotypes. However, this will not limit the expressiveness of the type system. We use η to denote substitutions mapping from type variables to variational types. We use the special symbol θ to denote substitutions that are partial unifiers. We use Γ to store the type assumptions about variables and treat Γ as a stack.

We use FV to denote free type variables in types, type schemas, and typing environments. The application of a substitution to a type schema, written as $\eta(\sigma)$, replaces free type variables in σ with the bindings in η. For presentation purposes, we assume we can determine the location of any given f in e with the function $\ell_e(f)$; the exact definition doesn't matter here. We may drop the subscript e when the context is clear.

We show the typing rules in Figure 3. The typing relation $\Gamma \vdash e : \phi|\Delta$ expresses that under the assumptions in Γ the expression e has the result type ϕ. All choice types that were generated during the typing process are stored (together with their locations) in Δ. Note that, due to the presence of choice types, the result type ϕ represents a whole set of possible result types that may be obtained by changing the types of certain parts of the expression. The information about what change leads to what type can be recovered from ϕ and Δ. For example, in the case of `3::Bool` we obtain the typing

$$\varnothing \vdash (\texttt{3::Bool}) : A\langle B\langle\bot,\texttt{Int}\rangle, B\langle\texttt{Bool},\alpha_3\rangle\rangle|\Delta$$
$$\text{where } \Delta = \{\ell(\texttt{3}) \mapsto A\langle\texttt{Int}, B\langle\texttt{Bool},\alpha_3\rangle\rangle, \ell(\texttt{Bool}) \mapsto B\langle\texttt{Bool}, A\langle\texttt{Int},\alpha_3\rangle\rangle\}$$

We create choices in rules Con, Var, Unbound, and Ant. The first alternative of each choice contains the type under normal typing, and the second alternative contains any

type to enable a change that is as general as the context allows. In rules ABS, LET, and APP, we collect generated choices from the typing of its subexpressions.

Most of the typing rules are self-explanatory. As one example let us consider the typing rule ANT for type annotations in more detail since it is new and introduces two operations that are crucial for typing applications. To type an expression $e::\tau$ we have to reconcile the inferred type ϕ_1 and the choice type ϕ_2 created for the annotation τ into one result type for e, which is achieved by using a common type ϕ_3. For the variants where ϕ_1 and ϕ_2 agree, ϕ_3 has the same type as ϕ_1. For other variants, ϕ_3 contains the error types $\bot$s. We use operations $\bowtie$ and $\lhd$ to realize this process.

The operation $\bowtie$ computes how well two types match each other. We use typing patterns introduced in Section 2 to formalize this notion.

$$\begin{array}{rl@{\qquad}rl}
\phi \bowtie \phi &= \top & D\langle\phi_1,\phi_2\rangle \bowtie D\langle\phi_3,\phi_4\rangle &= D\langle\phi_1 \bowtie \phi_3, \phi_2 \bowtie \phi_4\rangle \\
\bot \bowtie \phi &= \bot & D\langle\phi_1,\phi_2\rangle \bowtie \phi &= D\langle\phi_1 \bowtie \phi, \phi_2 \bowtie \phi\rangle \\
\phi \bowtie \bot &= \bot & \phi \bowtie D\langle\phi_1,\phi_2\rangle &= D\langle\phi_1 \bowtie \phi, \phi_2 \bowtie \phi\rangle \\
\phi \bowtie \phi' &= \bot & \phi_1 \to \phi_2 \bowtie \phi_1' \to \phi_2' &= (\phi_1 \bowtie \phi_1') \otimes (\phi_2 \bowtie \phi_2')
\end{array}$$

Note that the definition contains overlapping cases and assumes that more specific cases are applied before more general ones. The matching of two plain types either succeeds with $\top$ or fails with $\bot$, depending on whether they have the same syntactic representation. Matching two choice types with the same choice name reduces to a matching of corresponding alternatives. Matching a type with some choice type reduces to the matching of that type with both alternatives in the choice type.

The matching of two arrow types is more involved. For a variant to be matched successfully, both the corresponding argument types and result types of that variant have to be matched successfully. The $\otimes$ operation captures this idea. The definition can be interpreted as defining a logical "and" operation by viewing $\top$ as "true" and $\bot$ as "false". For example, when computing $\texttt{Int} \to A\langle\texttt{Bool},\texttt{Int}\rangle \bowtie B\langle\texttt{Int},\bot\rangle \to \texttt{Bool}$, we first obtain $B\langle\top,\bot\rangle$ and $A\langle\top,\bot\rangle$ for matching the argument and return types, respectively. Next, we use $\otimes$ to derive the final result as $A\langle B\langle\top,\bot\rangle,\bot\rangle$. From the result, we know that only matching the first alternative of A and first alternative of B will succeed.

$$\top \otimes \pi = \pi \qquad \bot \otimes \pi = \bot \qquad D\langle\phi_1,\phi_2\rangle \otimes \pi = D\langle\phi_1 \otimes \pi, \phi_2 \otimes \pi\rangle$$

The masking operation $\pi \lhd \phi$ replaces all occurrences of $\top$ in π with ϕ and leaves all occurrences of $\bot$ unchanged. It is defined as follows [5].

$$\bot \lhd \phi = \bot \qquad \top \lhd \phi = \phi \qquad D\langle\pi_1,\pi_2\rangle \lhd \phi = D\langle\pi_1 \lhd \phi, \pi_2 \lhd \phi\rangle$$

To type function applications in APP, we need a further operation $\uparrow(\phi)$ to turn ϕ into an arrow type when possible and introduce error types when necessary. We need this operation because the type of an expression may be a choice between two function types, in which case we have to factor arrows out of choices. For example, given $(\texttt{succ},\texttt{Int} \to \texttt{Int}) \in \Gamma$, we can derive $\Gamma \vdash \texttt{succ} : \phi|\Delta$ with $\phi = D\langle\texttt{Int} \to \texttt{Int}, \texttt{Int} \to \texttt{Bool}\rangle$ and $\Delta = \{(\ell(\texttt{succ}),\phi)\}$. Thus, we have to turn ϕ into $D\langle\texttt{Int},\texttt{Int}\rangle \to D\langle\texttt{Int},\texttt{Bool}\rangle$ if we apply `succ` to some argument.

We can observe that the rules CON, VAR, UNBOUND, and ANT introduce arbitrary types in the second alternative of result types. Thus, given e and Γ we have an infinite number of typings for e.

The following theorem expresses that there is a best typing (in the sense that it produces most general types with the fewest number of type errors) and that this is the only typing we need to care about. To compare the relation between different typings, we assume the same location at different typings generate the same fresh choice, and we write $\phi_1 \sqsubseteq \phi_2$ if there is some η such that $\eta(\phi_1) = \phi_2$.

Theorem 1 (Most general and most defined typing). *Given e and* Γ*, there is a unique best typing* $\Gamma \vdash e : \phi_1 | \Delta_1$ *such that for any other typing* $\Gamma \vdash e : \phi_2 | \Delta_2$, $\forall \delta$: $(\lfloor \phi_1 \rfloor_\delta = \bot \Rightarrow \lfloor \phi_2 \rfloor_\delta = \bot) \vee \lfloor \phi_1 \rfloor_\delta \sqsubseteq \lfloor \phi_2 \rfloor_\delta$.

In [4] we have presented a type inference algorithm that is sound, complete, and computes the most general result type (that is, at least as general as the result type of the best typing).

4 Climbing the Type-Change Lattice

The idea of guided type debugging is to narrow down the number of suggestions for how to remove a type error based on targeted user input. More specifically, given an ill-typed expression e for which CF typing has produced the variational type ϕ and the location information about changes Δ, we elicit from the programmer a target type τ and then want to identify the changes from ϕ and Δ that cause e to have type τ.

However, if the inference process produces a set of n choices with dimensions $\mathcal{D}$, which means that n potential changes have been identified, it seems that we have to check all 2^n combinations to find the right combination of changes that has the desired property. Fortunately, the structure of ϕ reveals some properties that we can exploit to significantly reduce the complexity of this process.

First, some of the created choices may not be needed at all. As the rules in Figure 3 show, the second alternative of created choice types can be any type. In many cases the best typing requires the second alternative to be identical to the first alternative, which means that no change is required. For example, when typing the expression `succ 1 + True`, the choice created for `1` is always $B\langle \texttt{Int}, \texttt{Int} \rangle$. Thus, we can remove choice B.

However, even after removing such non-relevant choices, the search space can still be exponential in the number of remaining choices. We can address this problem by searching, in a systematic way, only through some of the changes. To do this, we conceptually arrange all sets of changes in a *type-change lattice* (TCL). Note that we don't ever actually construct this lattice; it is a conceptual entity that helps to explain the algorithm for identifying type-change suggestions guided by a user-provided target type.

Each node in the lattice is identified by a subset of dimensions $\mathcal{C} \subseteq \mathcal{D}$ that indicates which changes to apply, and each $\mathcal{C}$ determines a decision $\delta_\mathcal{C}$, defined as follows.

$$\delta_\mathcal{C} = \{D.2 \mid D \in \mathcal{C}\} \cup \{D.1 \mid D \in \mathcal{D} - \mathcal{C}\}$$

With $\delta_\mathcal{C}$ we can determine the result type for e by $\lfloor \phi \rfloor_{\delta_\mathcal{C}}$ for the case when the changes indicated by $\mathcal{C}$ are to be applied. Usually, we attach $\lfloor \phi \rfloor_{\delta_\mathcal{C}}$ to the node $\mathcal{C}$ in TCLs.

A TCL comprises $n+1$ levels, where level k contains an entry for each combination of k individual changes. The bottom of this lattice (level 0) consists of a single node $\varnothing$,

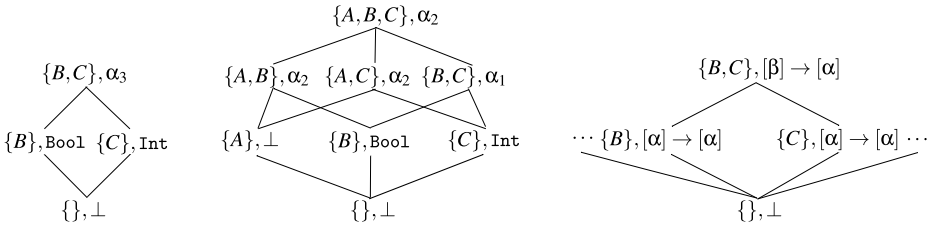

Fig. 4: Type-change lattices for the expressions `3::Bool` (left), `id (3::Bool)` (middle), and `rR` (right). The function `id` is assumed to have the type $\alpha_1 \to \alpha_1$, which could be a cause of the type error and thus a choice type is created for `id`. For the second example, the dimensions A, B, and C represent the changes for `id`, `3`, and `Bool`, respectively. For the `rR` example, B and C are created for locations `x` and `(++)`, respectively.

which produces the result type $\lfloor\phi\rfloor_{\delta_\varnothing} = \bot$. Level 1 consists of n entries, one for each single change $D \in \mathcal{D}$. The next level consists of all two-element subsets of $\mathcal{D}$, and so on The top-most level has one entry $\mathcal{D}$, which represents the decision to apply all changes. We show three example TCLs in Figure 4.

To find a change suggestion we traverse the TCL in a breadth-first manner from the bottom up.[4] For each entry $\mathcal{C}$ under consideration we check whether the type produced by it covers the target type as a generic instance, that is, $\lfloor\phi\rfloor_{\delta_\mathcal{C}} \sqsubseteq \tau$. Once we have found such an entry, we can present the programmer with a corresponding suggestion. In case there are several suggestions with k changes that satisfy the condition, we employ the heuristics developed in CF typing [4] to order them and present them to programmers in this order. If the programmer selects a suggested change, we're done. Otherwise, we continue the search by including the next level in the lattice and offer suggestions with one more change.

We illustrate how the algorithm works with the two earlier examples $e =$ `3::Bool` and `rR` (recall Figures 1 and 2). The TCLs are shown in Figure 4. Expression e is trivial since the algorithm will ask the user for the type of this expression and correspondingly suggest to either change the value or the annotation. To make this example more interesting, consider the slightly more general expression `id (3::Bool)`. For this expression, CF typing produces the following result, and the TCL is in the middle of Figure 4.

$$\begin{aligned}\phi &= A\langle B\langle C\langle \bot, \texttt{Int}\rangle, C\langle \texttt{Bool}, \alpha_1\rangle\rangle, B\langle C\langle \bot, \alpha_2\rangle, \alpha_2\rangle\rangle \\ \Delta &= \{(\ell(\texttt{id}), A\langle \alpha_1 \to \alpha_1, B\langle \texttt{Int} \to \alpha_2, C\langle \texttt{Bool} \to \alpha_2, \alpha_3 \to \alpha_2\rangle\rangle\rangle) \\ &\quad\ (\ell(\texttt{3}), B\langle \texttt{Int}, C\langle \texttt{Bool}, A\langle \alpha_1, \alpha_3\rangle\rangle\rangle) \\ &\quad\ (\ell(\texttt{Bool}), C\langle \texttt{Bool}, B\langle \texttt{Int}, A\langle \alpha_1, \alpha_3\rangle\rangle\rangle)\}\end{aligned}$$

Suppose the user-provided target type is `Bool`. The first of the changes on level 1 ($\{A\}, \bot$) will be dismissed since the application of change A cannot remove the type error. Since the result type of the next entry ($\{B\}$, `Bool`) matches exactly the target type,

[4] Again, the algorithm for finding change suggestions constructs part of this lattice on the fly as needed. The lattice is not represented explicitly.

this change will be suggested. We will look at the third entry as well, but it is dismissed since the target type `Bool` is not an instance of the result type `Int` for that entry. Thus, we will present the suggestion of change B (that is, location $\ell(3)$) to the programmer. From Δ we infer that the place should be changed to something of type `Bool`. Thus, the generated suggestion is to change `3` from type `Int` to type `Bool`.

If this suggestion is accepted, the debugger terminates. If, however, the programmer asks for more suggestions, we check change suggestions in two steps. First, we remove all the nodes in the lattice that are above the node that produced the previous suggestion. In this example, we remove all the nodes in the lattice above $\{B\}$ because the programmer doesn't want to apply any change that includes the B change, yielding a smaller lattice with only four nodes. Then we continue the search on the higher level. In this case the only node that remains on the second level is $(\{A,C\},\alpha_2)$. To get the result `Bool`, we derive the substitution $\eta = \{\alpha_2 \mapsto \texttt{Bool}\}$. By selecting $[A.2,B.1,C.2]$ from the variational type $\Delta(\ell(\texttt{id}))$, we derive that `id` should be changed from type $\alpha_1 \to \alpha_1$ to something of type $\eta(\texttt{Int} \to \alpha_2)$, which is $\texttt{Int} \to \texttt{Bool}$. By making the same selection $[A.2,B.1,C.2]$ from $\Delta(\ell(\texttt{Bool}))$, we derive that `Bool` should be changed to `Int`.

For the example `rR` we apply the same strategy. If the programmer provides the target type `[a] -> [a]`, only two of the 13 entries on level 1 qualify since their result types can be instantiated to the target type.

Despite the exponential size that TCLs can have in the worst-case and the corresponding worst-case time complexity to explore TCLs exhaustively, GTD search turns out to be very efficient in almost all cases for the following reasons.

- The change suggestions to be presented first will be encountered and found first during the search process.
- If a presented suggestion is rejected by the programmer, the lattice can be trimmed down by removing all the nodes higher in the lattice that are reachable from the node representing the rejected suggestion.
- The lattice narrows quickly toward the top since after a few layers the result types tend to become free type variables, which can be unified with the user-provided target type successfully, ending the search.

The first point is substantiated by following theorem, which states that generality of result types increases with the number of changes. We want to find suggestions that consist of as few as possible changes and whose result type is closest to the target type. This theorem ensures that when we traverse a TCL from the bottom up, we will encounter changes that have fewest locations first.

Theorem 2 (More change locations lead to more general types). *Given the best typing* $\Gamma \vdash e : \phi|\Delta$*, if* $C_1 \subseteq C_2$*, then* $\lfloor\phi\rfloor_{\delta_{C_2}} \sqsubseteq \lfloor\phi\rfloor_{\delta_{C_1}}$.

Proof. The proof is shown in Appendix A.

Guided type debugging can improve the precision of suggesting type changes for ill-typed programs at a low cost. We have tested the method and compared it with CF typing on 86 programs, which were collected from 22 publications (see [4] for details). Since many programs we collected were written in ML [9] and in OCaml [14], we

have translated them into the programs written in the calculus presented in the paper plus operations stored in the initial type environment. The following table shows the percentage of the programs a correct change suggestion could be provided for after n attempts. In all cases in which GTD helped to remove the type error, only one target type had to be supplied.

Method	No. of Attempts				
	1	2	3	≥ 4	never
CF Typing	67%	80%	88%	92%	8%
GTD	83%	90%	92%	92%	8%

With GTD we can now find the correct suggestions with the first attempt in 83% of the cases. We can fix 90% of the cases with at most two attempts. At the same time GTD adds never more than 0.5 seconds to the computing time.

5 Reporting Type Errors in Type Annotations

We use following example, which was written by a student learning Haskell [10], to compare the behaviors of different tools on reporting type errors in type annotations. We copied the code literally except for removing the type definition of `Table`, which is `[[String]]` and the definition of the function `collength`, whose type is `Table->Int`. Both are irrelevant to the type error.

```
buildcol :: Table->[String]
buildcol [] = [""]
buildcol (x:xs) = [" " ++ (replicate n '-')," " ++ (spaceout n (head x))]
                  where n = collength (x:xs)

spaceout :: String->String
spaceout n str = str ++ replicate (n-(length str)) ' '
```

The type annotation of `spaceout` contains one argument, but the function definition has two arguments. Based on the same student's follow-up programs we know that the annotation is incorrect. Note that this is also the only type error in the program because removal of the type annotation of `spaceout` restores type correctness of the program.

For this program, the Glasgow Haskell Compiler (GHC) 7.6.3 reports the following four type errors. The first two point to the use of `spaceout` in `buildcol`, and the other two point to the definition of `spaceout`.

- The first blames that `spaceout` is applied to two arguments while it takes only one.
- The second complains that the first argument type of `spaceout` should be `String`, but something of `Int` is given.
- The third reports that the definition of `spaceout` has two arguments while its type has only one.
- The fourth complains that `n` is of type `String`, but it should have type `Int` because it is used as the first argument to the operation `-`.

GHC reports these errors because it always trusts type annotations, and it pushes down type information from type annotations to expressions. In the definition `spaceout`, the parameters `n` and `str` both get the type `String`. When type annotation is correct, this scheme makes type checking more efficient and helps make type inference decidable [18]. However, when type annotation is incorrect, this leads to poor error messages.

Helium[5], a research tool with high quality error messages developed to assist students in learning Haskell, reports two type errors. The first reported error is similar to GHC's first error message. Moreover, Helium suggests to remove the first argument. Helium's second error message is almost the same as GHC's third message, blaming the definition of `spaceout`, but not the annotation. Type annotations were not supported in CF typing [4].

In contrast to previous tools, our guided type debugger is the first approach that can find errors in type annotations. In this example it directly suggests to change the type annotation `String -> String` to `Int -> String -> String`, which fixes all type errors in the program.

6 Related Work

The challenge of accurately reporting type errors and producing helpful error messages has received considerable attention in the research community. Improvements for type inference algorithms have been proposed that are based on changing the order of unification, suggesting program fixes, interactive debugging, and using program slicing techniques to find all program locations involved in type errors. We will focus our discussion on debugging and change-suggesting approaches. Since this problem has been extensively studied, summaries of the work in this area are also available elsewhere [12, 23, 24, 15, 4].

The idea of debugging type errors was first proposed by Bernstein and Stark [2]. Their work was based on the observation that type inference is able to infer types for unbound variables, which allows programmers to replace suspicious program fragments with unbound variables. If a replacement leads to a type correct program, the type errors have been located. The original work requires programmers to manually locate suspicious fragments and replace them with unbound variables. Braßel [3] has later automated this process.

By employing the idea of algorithmic debugging developed in debugging Prolog errors, Chitil [7] proposed an approach for debugging type errors. Chitil developed principal typing, where type inference is fully compositional by inferring also type assumptions, for building explanation graphs. Each node is a combination of the typings of its children. The idea of algorithmic debugging is to navigate through the graphs and ask questions about the correctness of each node. Each question is of form "Is the intended type of a specific function an instance of the type inferred?", and programmers will respond "yes" or "no". In Chameleon, Stuckey at al. [20, 21] presented a debugging approach that is similar to algorithmic debugging. Chameleon also allows programmers to ask why an expression has a certain type. There are other tools that don't allow user

[5] `http://www.cs.uu.nl/wiki/bin/view/Helium/WebHome`

inputs but allow programmers to navigate through the programs and view their types, such as Typeview [8] and the Haskell Type Browser [17].

While previous debugging approaches are operational in the sense that programmers have to be involved in the details of the debugging process, our approach is more declarative in the sense programmers only have to specify the intended result type. Moreover, we provide more precise change suggestions, such as the change location and the types expressions should be changed to. Moreover, in some cases we can be even more specific and suggest specific program edit actions, such as swapping function arguments (see [4] for some examples). In contrast, previous approaches can only locate a program fragment or a set of possible places as the cause of type errors, and thus often leave much work for programmers after the debugging is finished.

Researchers have paid considerable attention to the problem of making change suggestions when type inference fails. For such methods to work, however, the most likely error location has to be determined. Since there is seldom enough information to make this decision, approaches have resorted to various kinds of heuristics. For example, in the earliest work along this line, Johnson and Walz [13] used a heuristic of "usage voting", that is, when a variable has to be unified with many different types, the variable is chosen to have the type that is unified most often. Locations that require that variable to have a different type are then reported as problematic.

Seminal [14] uses the difference between the original (ill-typed) program and the changed (well-typed) programs as a heuristic. Top [12] uses more sophisticated heuristics [11], such as a participation-ratio heuristic, a trust-factor heuristic, and others. CF typing [4] uses heuristics, such as preferring expression changes to other places, favoring changes at lower places in the tree representations, and preferring simpler type changes over more complex ones. The most recent work by Zhang and Myers [25] employs Bayesian principles to locate type errors, but they don't make suggestions.

While previous approaches involve programmers only in a very limited way and allow them to accept or reject a suggestion, guided type debugging gives programmers the opportunity to provide more meaningful input and explicitly specify some of their goals. This is not complicating matters much since it requires only the formulation of type annotations. On the other hand, the input can be effectively exploited to shorten the debugging process considerably. The strategy of steering the derivation of changes by target types elicited from users is inspired by a technique to guide the debugging of spreadsheets by user-provided target values [1].

The idea of choice types seems to be similar to the concept of discriminative sum types [16, 17], in which two types are combined into a sum type when an attempt to unify them fails. However, there are several important differences. Choice types are named and thus provide more fine-grained control over the grouping of types, unification, and unification failures. Sum types are *always* unified component-wise, whereas we do this only for choice types under the same dimension. For choice types with different dimensions, each alternative of a choice type is unified with all alternatives of the other choice type. Other differences between guided type debugging and the error-locating method developed in [16] are as follows. First, their method extracts all locations involved in type errors and is thus essentially a type-error slicing approach, whereas our method always blames the most likely error location. Second, guided type

debugging provides change suggestions in all cases, whereas their method, like all error-slicing approaches, does not. Finally, error locations reported by their method may contain program fragments that have nothing to do with type errors. For example, a variable used for passing type information will be reported as a source of type errors only if it is unified once with some sum types during the type inference process. In our method, on the other hand, only locations that contribute to type errors are reported.

An additional contribution of guided type debugging is a better treatment of type annotations. We have investigated the reliability of type annotations and studied the problem of locating type errors in annotations, a problem that hasn't received much attention from the research community so far.

7 Conclusions

We have developed guided type debugging as an approach to produce better change suggestions faster in response to type errors in functional programs. Our approach differs from previous tools by incorporating programmer intentions more directly by asking targeted questions about types. This strategy is efficient and can effectively increase the precision of type-change suggestions. A further contribution our method is the effective identification and removal of inconsistent type annotations.

In future work, we plan to investigate the possibility of minimizing programmer input by exploiting the information about the evolution of programs. For example, the knowledge about which part of the program was changed last may in many cases allow the automatic derivation of the target types. Another question we will investigate is how to locate type errors in type annotations when omitting them will lead to undecidable type inference.

Acknowledgements

We thank Jurriaan Hage for sharing his collection of student Haskell programs with us. This work is supported by the the National Science Foundation under the grants CCF-1219165 and IIS-1314384.

References

1. R. Abraham and M. Erwig. GoalDebug: A Spreadsheet Debugger for End Users. In *29th IEEE Int. Conf. on Software Engineering*, pages 251–260, 2007.
2. K. L. Bernstein and E. W. Stark. Debugging type errors. Technical report, State University of New York at Stony Brook, 1995.
3. B. Braßel. Typehope: There is hope for your type errors. In *Int. Workshop on Implementation of Functional Languages*, 2004.
4. S. Chen and M. Erwig. Counter-Factual Typing for Debugging Type Errors. In *ACM Symp. on Principles of Programming Languages*, pages 583–594, 2014.
5. S. Chen, M. Erwig, and E. Walkingshaw. An Error-Tolerant Type System for Variational Lambda Calculus. In *ACM Int. Conf. on Functional Programming*, pages 29–40, 2012.
6. S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems*, 36(1):1–54, 2014.

7. O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ACM Int. Conf. on Functional Programming*, pages 193–204, September 2001.
8. O. Chitil, F. Huch, and A. Simon. Typeview: A tool for understanding type errors. In *International Workshop on Implementation of Functional Languages*, pages 63–69, 2000.
9. C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *European Symposium on Programming*, pages 284–301, 2003.
10. J. Hage. Helium benchmark programs, (2002-2005). Private communication.
11. J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In *Implementation and Application of Functional Languages*, pages 199–216. 2007.
12. B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, Sept. 2005.
13. G. F. Johnson and J. A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *ACM Symp. on Principles of Programming Languages*, pages 44–57, 1986.
14. B. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *ACM Int. Conf. on Programming Language Design and Implementation*, pages 425–434, 2007.
15. B. J. McAdam. *Repairing type errors in functional programs*. PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics., 2002.
16. M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *ACM Int. Conf. on Functional Programming*, pages 15–26, 2003.
17. M. Neubauer and P. Thiemann. Haskell type browser. In *ACM SIGPLAN Workshop on Haskell*, pages 92–93, 2004.
18. M. Odersky and K. Läufer. Putting type annotations to work. In *ACM Symp. on Principles of Programming Languages*, pages 54–67, 1996.
19. T. Schilling. Constraint-free type error slicing. In *Trends in Functional Programming*, pages 1–16. Springer, 2012.
20. P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive Type Debugging in Haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 72–83, 2003.
21. P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *ACM SIGPLAN Workshop on Haskell*, pages 80–91, 2004.
22. M. Wand. Finding the source of type errors. In *ACM Symp. on Principles of Programming Languages*, pages 38–43, 1986.
23. J. R. Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions*. PhD thesis, The University of Melbourne, January 2006.
24. J. Yang, G. Michaelson, P. Trinder, and J. B. Wells. Improved type error reporting. In *Int. Workshop on Implementation of Functional Languages*, pages 71–86, 2000.
25. D. Zhang and A. C. Myers. Toward General Diagnosis of Static Errors. In *ACM Symp. on Principles of Programming Languages*, pages 569–581, 2014.

A Proof of Theorem 2

To prove the theorem, we need a relation between the application of a change and the corresponding result type. This idea is formally captured in the typing relation in Figure 5. Note that we omit the rules for abstractions and let expressions because they can be obtained by simply adding χ to the left of turnstile, as we did for the rule for applications.

The rule system defines the judgment $\Gamma;\chi \vdash e : \tau$, where χ is a mapping that maps the location to the type that location will be changed to. In the rules, we use the notation

$$\textsc{Var-C}\quad \Gamma;\chi \vdash x : \chi(x)||\{\overline{\alpha \mapsto \tau}\}(\Gamma(x)) \qquad \textsc{Ant-C}\quad \Gamma;\chi \vdash (e::\tau) : \chi(\tau)||\tau$$

$$\textsc{Con-C}\quad \frac{c \text{ is of type } \gamma}{\Gamma;\chi \vdash c : \chi(c)||\gamma} \qquad \textsc{App-C}\quad \frac{\Gamma;\chi \vdash e_1 : \tau_1 \to \tau \qquad \Gamma;\chi \vdash e_2 : \tau_1}{\Gamma;\chi \vdash e_1\ e_2 : \tau}$$

Fig. 5: Rules for the type-update system

$\chi(e)||\tau$ to decide whether we should use the information in χ to override the type τ for the atomic expression e. More precisely, if $(\ell(e),\tau') \in \chi$, then $\chi(e)||\tau$ yields τ', and otherwise τ.

Given a decision δ and a change environment Δ, we can obtain the corresponding χ through the operation $\downarrow_\delta\Delta$, defined as follows.

$$\downarrow_\delta\Delta = \{l \mapsto \lfloor\phi_2\rfloor_\delta \mid (l, D\langle\phi_1,\phi_2\rangle) \in \Delta \wedge D.2 \in \delta\}$$

We have proved that if $\Gamma \vdash e : \phi|\Delta$, then for any decision δ, we have $\Gamma;\downarrow_\delta\Delta \vdash e : \lfloor\phi\rfloor_\delta$ [4]. Thus, the proof of Theorem 2 reduces to a proof of the following lemma.

Lemma 1 (More change locations lead to more general types). *Given* $\Gamma \vdash e : \phi|\Delta$ *and two decisions* δ_1 *and* δ_2*, let* $\chi_1 = \downarrow_{\delta_1}\Delta$ *and* $\chi_2 = \downarrow_{\delta_2}\Delta$*. If* $dom(\chi_1) \subseteq dom(\chi_2)$*, then* $\Gamma;\chi_1 \vdash e : \tau_1$ *and* $\Gamma;\chi_2 \vdash e : \tau_2$ *with* $\tau_2 \sqsubseteq \tau_1$*.*

Proof. The proof is by induction over the typing derivations. Since $\Gamma;\chi_1 \vdash e : \tau_1$ and $\Gamma;\chi_2 \vdash e : \tau_2$ are typing the same expression e, and since we are using the same set of rules, the derivation trees for them have the same structure. We show the proof for the cases of variable reference and application. The proof for other cases is similar.

- Case Var. There are several subcases to consider.
  (1) $\ell(x) \notin dom(\chi_1)$ and $\ell(x) \notin dom(\chi_2)$. In this case, $\chi_1(x)$ and $\chi_2(x)$ are both given by $\Gamma(x)$. Therefore, $\tau_1 = \tau_2 = \Gamma(x)$, and $\tau_2 \sqsubseteq \tau_1$ trivially holds.
  (2) $\ell(x) \notin dom(\chi_1)$ and $\ell(x) \in dom(\chi_2)$. We can formally prove that $\tau_2 \sqsubseteq \tau_1$ by an induction over the structure of expressions. An intuitive argument is that when we can change the original type (τ_1) to a new arbitrary type (τ_2) that makes its context well typed, the definition of the typing relation in Figure 3 maintains generality and doesn't make τ_2 more specific than τ_1.
  (3) $\ell(x) \in dom(\chi_1)$ but $\ell(x) \notin dom(\chi_2)$. This case is not possible.
  (4) $\ell(x) \in dom(\chi_1)$ and $\ell(x) \in dom(\chi_2)$. The proof for this case is similar to the one for case (2).
- Case App. The induction hypotheses are $\Gamma;\chi_1 \vdash e_1 : \tau_3 \to \tau_1$, $\Gamma;\chi_1 \vdash e_2 : \tau_3$, $\Gamma;\chi_2 \vdash e_1 : \tau_4 \to \tau_2$, $\Gamma;\chi_2 \vdash e_2 : \tau_4$, with $\tau_4 \to \tau_2 \sqsubseteq \tau_3 \to \tau_1$ and $\tau_4 \sqsubseteq \tau_3$. From $\tau_4 \to \tau_2 \sqsubseteq \tau_3 \to \tau_1$, we derive $\tau_4 \sqsubseteq \tau_3$ and $\tau_2 \sqsubseteq \tau_1$, which completing the proof for this case.