Doctoral Dissertation, 2013

Practicable Type Debugging for Functional Languages

OCHANOMIZU UNIVERSITY

Comparative Studies of Societies and Caltures,
Graduate Scholl of Humanities and Sciences
The Division of Advanced Sciences

KANAE Tsushima

September, 2013

Abstract

This thesis presents how to build a practicable type debugger. From the time the Hindley-Milner type system was first proposed, programmers have received benefits from types. At the same time, they have to struggle with type errors. Many approaches have been developed to help programmers locate the source of type errors. Although their implementations help programmers a lot, existing compilers often lack such support. We feel this situation puts too much of a burden on programmers: debugging an ill-typed program takes up a lot of their time, and compiler's error messages are too difficult for many new learners to understand. This situation is a shame for many statically typed languages.

To address this situation, we believe practicable type debugging is needed. First, we establish a manifesto of practicable type debugging. The properties of the manifesto can be grouped into two categories. One category is the producer side, where the properties focus on the implementation of type debugging systems. By satisfying these properties, a type debugging system can be applied to many languages. The other category is the consumer side, where the properties focus on the usability. If a type debugging system is not user-friendly and forces the programmers to deal with too big a burden, the programmers feel that debugging by hand would be better. Therefore, the usability of a type debugger is a crucial factor.

The main part of this thesis consists of two parts according to our manifesto of practicable type debugging.

First, we focus on the producer side of type debugging. To this end, we propose a type debugger *without* implementing any dedicated type inferencer. Conventional type debuggers require their own type inferencers separate from the compiler's type inferencer. The advantage of our approach is threefold. First, by *not* implementing a type inferencer, it is guaranteed that the debugger's type inference never disagrees with the compiler's type inference. Second, we can avoid the pointless reproduction of a type inferencer that should work precisely as the compiler's type inferencer. Third, our approach can withstand updates of the underlying language. The key element of our approach is that the interactive type debugging, as proposed by Chitil, does not require a type inference tree but only a tree with a certain simple property. We identify the property and present how to construct a tree that satisfies this property using the compiler's type inferencer. The property shows us how to build a type debugger for various language constructs. In this topic, we describe our idea and first apply it to the simply-typed lambda calculus. After that, we extend it with let-polymorphism and objects to see how our technique scales.

Second, we focus on the customer side of type debugging. To this end, we propose a *weighted* type error slicer. The problem of our type debugger is that it often requires many answers by programmers. This problem is solved partially by type error slices. Conventional type error slicers enable programmers to narrow the area for type debugging. However, type error slices become large when the original ill-typed programs are large. To search the source of the type error in the large slice is a burden on programmers. To ease this problem, we extend type error slices with the weights that means

the likelihood of each expression being the source of the type error. When a programmer write a program, he has some intentions of types. Therefore some programmer's intentions are sprinkled into the program even if it is ill-typed. The aim of this work is to detect programmer's intentions from an ill-typed program. The main idea is to abstract an ill-typed program and judge the likelihood by majority vote. In this topic, we describe our idea and introduce a type error slice using compiler's type inferencer. After that, we extend it with the simple weights and improve it to have better weights.

We propose these two approaches for *practicable type debugging*. Using our approaches, it is possible to implement a type debugger for many existing functional languages. Easy type debugging contributes easy learning of programmings and decreases the burden of programmers. This thesis is a step to achieve easy type debugging.

概要

本論文では、関数型言語のための実用可能な型デバッガ作成手法について述べる。Hindley-Milner の型推論が導入されて以来、プログラマは型による恩恵を受けて来た。それは同時にプログラマと型エラーとの戦いの幕開けでもあった。プログラマによる型エラーの原因特定を手助けするために多くの手法が提案されており、それらの実装は実際にプログラマを助けてきたが、未だ多くの言語が型デバッグに関する実装を欠いている。実際、型デバッグのサポートがない言語では、型エラーの原因を探る際、プログラマはコンパイラのエラーメッセージを元に、自身で型を推論するなど時間を割いているという現状がある。このような状況は初級者にとって学習の阻害になる上に、上級者にとってもデバッグを行う時間を必要とし、プログラマに負担を強いる。

この状況を打開するためには、実用的な型デバッグ手法が必要となる。しかし、「実用的な型デバッグ手法」とはかなり曖昧な表現であるため、まず我々はそれが満たすべき性質を提案した。この性質は大きく二つの種類に分けられる。ひとつ目は、型デバッガを実装する側に関する性質である。その性質は、型デバッガを多くの言語に適用することを目的に定められている。ふたつ目は、型デバッガを使用するユーザ側に関する性質である。その性質は、実際にユーザにとって使いやすい型デバッガになることを目的に定められている。これらの性質が合わさることで、実装面・利用面の両方から実用的な型デバッグが実現される。

本論文は大きく分けて二つの内容を扱う。

まずひとつ目の内容では、型デバッガを実装する側に関する性質に注目し、それらの性質を満たす型デバッガを作成する手法を提案する。これまでの型デバッガは、それぞれ特殊な型推論器を必要としており、それが障害となって多くの言語への適用が困難であった。我々はコンパイラの型推論器を使用することによって、型推論器の実装を必要としない型デバッガを提案した。我々の手法の利点は三点存在する。型推論器を実装せずコンパイラの型推論器を使用することによって、コンパイラの型推論器との齟齬が発生し得ない点、型推論器を再び作成する手間がかからない点、コンパイラの変更に強い点である。提案する型デバッガの主軸となる方法は、Chitil の対話的な型デバッガに基づいている。説明の流れとしては、まず提案の概要を説明し、その提案を単純型付きラムダ計算に適用する。その後、それを let 多相やオブジェクトに拡張し、この提案がどこまで拡張可能かをみる。

ふたつ目の内容では、型デバッガを使用するユーザ側に関する性質に注目し、それらの性質のうちひとつを満たす、重み付き型エラースライスという手法を提案する。これまでの型エラースライス作成手法では、型デバッグの際に見る範囲を小さくすることが可能であった。これは実際にデバッグを行う際、ユーザの負担を軽減するために有効である。我々はこの方法を拡張し、スライスの各所に重みを付けた。ここでの重みとは、それぞれの箇所がどれくらいエラーの原因である可能性が高いかということを表す。よって、重み付き型エラースライスを使うことで、デバッグの際にエラーの可能性が高い箇所から質問をすることが可能となり、ユーザにとって使いやすい型デバッグを行うことが出来る。

このように実装面・利用面からそれぞれ型デバッガ作成手法を改良することによって、関数型言語のための実用的な型デバッグが実現された。これによって、型デバッグ機能を欠いた多くのプログラミング言語を対象とした型デバッガの実装が可能になる。型エラーの修正が容易になることは、型付きプログラミングの学習への貢献や、プログラム作成時の障害が減ることに

よる信頼性の高いソフトウェア開発などに繋がっており、そのための一歩が本研究によって為されたと考えている。

Acknowledgement

I'd like to thank my supervisor, Prof. Kenichi Asai. When I was an undergraduate student, I took Kenichi's course "Functional languages". I found type systems are beautiful in the class and felt type errors should be fixed easier. Therefore, his course motivates me to choose the topic of this thesis, type debugging. After I joined his laboratory, I learned a lot of things from him, for example, reading papers, writing papers, presenting talks and so on. I really feel this work is supported by him.

I'd like to thank Dr. Olaf Chitil. Because my type debugger in this thesis is based on his work [2], I asked him to give me some advices after writing the paper [27]. He gave me a lot of advices from a different perspective. The discussions with him made my work good.

In spring of 2012, I visited Prof. Olivier Danvy at Aarhus university for two months. He gave me advices of my researches, and advices for living abroad. The weekly talks held at the university were very interesting for me since there are many programming language labs. When I visited Aarhus again at September of 2012, I talked in the seminar and the participants gave me a lot of comments. Thanks to a Ph.D. student of Olivier, Ian Zerny, I could have contact with the students at Aarhus university.

From June of 2012, I joined IPL seminar at NII. The seminar is held by a group of Prof. Zhenjiang Hu. Although their research area (bidirectional

transformation) is not near my research area, the talks in the seminar were very interesting and the discussions about my work in the seminar helped me a lot. I presented three talks about my research in IPL seminar.

After I joined Kenichi's lab, I attended PPL (Programming and Programming Language workshop) every year. I had three presentations at PPL. I gained inspiration from the participants of PPL, especially their attitude to researches.

I'd like to thank Moe Masuko, my colleague of Kenichi's lab. We often discussed many things about programming languages, and I sometimes got ideas for research from the discussion. Because we often went to many conferences together, I could try many things.

I'd like to thank the colleagues of Kenichi's lab and the members of Prof. Daisuke Bekki's lab. Thanks to them, I enjoyed good life in the lab.

The last acknowledgment is for my family. My life as a Ph.D. student is supported by them.

謝辞

学部時代からの指導教官である、浅井健一先生に厚くお礼を申し上げます。そもそも本研究の動機は、先生の授業「関数型言語」や「コンパイラ構成論」を受講した際に感じたことがきっかけでした。関数型言語の授業では型システムによって保たれる性質の美しさを、コンパイラ構成論の授業では型推論の実装を通じて型システムの美しさを学びました。それと同時に、なぜ型エラーメッセージはもっと適切な場所を指摘してくれないのだろうか？なぜ推論した型を見せてくれないのだろうか？という疑問を感じ、それらの疑問が本研究の動機となっています。まず研究動機を与えてくださったことに大変感謝をしております。また研究室に配属されてからは、論文の読み方、書き方、発表の仕方等、多くのことを丁寧に教えて頂きました。これまでの研究や本研究の多くの部分を先生に支えて頂きました。厚くお礼を申し上げます。

Kent 大学の Olaf Chitil 先生の既存研究が本研究の型デバッガの基礎となっています。そのため Chitil 先生には、論文・内容に関してたくさんのアドバイスを頂きました。多くの違った視点からのアドバイスや先生とのメールでの議論によって本研究の型デバッガを改良することが出来たことに、大変感謝をしております。

Aarhus 大学の Olivier Danvy 先生のご好意により、2012 年の春には Aarhus 大学に 2 ヶ月滞在し、研究を行いました。Danvy 先生には研究についてご意見を頂いたことはもちろん、初めて滞在する海外の大学について多くのことを教えて頂きました。毎週 Aarhus 大学のプログラミング言語グ

ループで行われるセミナーは大変有意義で面白く、2012 年の 9 月に再び訪問した際には、型デバッガについて発表を行い、参加者の皆様からは多くのご意見を頂きました。また、Danvy 先生のもとで博士課程在学中の Ian Zerny 氏の助力によって、多くの学生の皆さんと交流出来たことを大変有り難く思っております。

2012 年の 6 月からは、NII で行われている胡振江先生のグループの IPL セミナーに参加させて頂きました。胡先生のグループとは私の研究分野は異なりますが、異なる分野の発表も大変面白く、本研究についても三度発表する機会を頂き、多くのご意見を頂きました。

浅井先生の研究室に所属してからは毎年 PPL (Programming and Programming Language workshop) に参加し、参加者のプログラミング言語の研究者の方々には大変お世話になりました。PPL では本研究の内容について二度、修士時代の研究について一度発表する機会を頂きました。PPL の参加者の方々との交流を通じて、研究に対する姿勢などを学べたことを大変有り難く思っております。

浅井研究室の増子萌氏には多くのことでご助力を頂きました。彼女とのプログラミング言語に関する議論によって、多くのことに気づかされ、研究のアイデアになることもありました。海外の学会へ共に参加することが多く、また Aarhus 大学にも一緒に訪問しました。思い返すと彼女が居たからこそ、挑戦出来たことが多くあったことに気づかされます。彼女の助力によって充実した研究生活を送れたことに厚くお礼を申し上げます。

浅井研究室の皆様、また戸次研究室の皆様には、大変お世話になりました。楽しく充実した研究室生活を過ごせたことに、大変感謝しております。

最後に、研究生活を支えてくれた家族に感謝を捧げます。

Contents

Chapter 1

Introduction

To ensure the reliability of programs, types are introduced to many languages. The benefit of types is that they guarantee various properties of well-typed programs. One popular property is "If a program is well-typed, the evaluation of the program will go well[1]." Thanks to these kinds of properties, we can receive many benefits. For example, a previous property ensured that programmers can run a well-typed program in safety, and another popular property states that "If a program is well-typed, its evaluation will surely halt." This property is accomplished with a strict type system (e.g, simply typed lambda-calculus).

The role of types is to sort programs according to our need. Here, this "need" has two sides: we need strong properties and expressive programs at the same time. However, to give an example of the difficulty of this, to have a halting property of programs, the expressiveness of the programs is weaker than languages which do not have halting properties. Put simply, types mean placing restrictions on programs. Properties ensured by types have a trade-off in the form expressiveness of programs. Research on advanced types show the history of searching for a good balance of these properties. Here, let us

[1] "go well" means the program does not cause any runtime type errors.

take a look at some of the advanced types.

Advanced types. Many advanced type systems based on the Hindley-Milner type system [15] have been proposed. Most of them were designed to increase the expressiveness of types (increasing acceptive programs) or the strength of properties.

One feature of these advanced types is "polymorphicness". The main idea of a polymorphic type is that it can be used to express several types. One polymorphic type, called a let-polymorphism, allows polymorphic types to be used for let-bounded functions. Because let-polymorphism allow programmers to reuse functions, it is used in ML (e.g. OCaml[3, 9], Standard ML [16] et al.). There are many polymorphic types, including rank-2 polymorphism [10, 11], parametric polymorphism [31], and so on. Because this thesis does not touch on these other polymorphic types, we do not present their details here.

Another feature of the advanced types is the dependent type, which allows types to dependent value or the other types. This type ensures the property that "if a program is well-typed, there are no errors caused by an out-of-bounds." Because dependent types ensure stronger properties than standard types, they are used in Coq [1], Agda [32] and Epigram [33].

There are many other types that have been proposed for properties relating to programs and expressiveness. Some of these are used in many languages, while most of them are just beginning to be used.

The balance of properties and expressiveness is one aspect of types. Another aspect is how to obtain the types of each expression to check the consistency of the programs. There are two main approaches to this. One approach requires programmers to write annotations (types in programs)

and cast types as much as possible. Because this approach often lack good properties, it is called "weakly typed." The other approach does not require annotations and instead, a compiler infers the types of each expression automatically. Because this approach often have some good properties, it is called "strongly typed." The benefit of this approach is strong properties and the unnecessity of writing types. However, this unnecessity often causes complicated type errors, so our target language in this thesis is the latter approach.

Difficulty of writing well-typed programs. Writing a well-typed program is not always easy, even in the Hindley-Milner type system. Furthermore, it is very difficult problem in advanced type systems. Although a compiler gives us an error message when a type error occurs, there is no straightforward explanation to why the type error occurred. Compounding the problem, the source of a type error can be far from the place reported by the compiler.

We believe that type debugging systems are needed for advanced type systems to become widespread. Although current compilers often lack a debugging system, this is compensated for by programmers' efforts, such as inferring types themselves. However, such efforts become impossible as type systems become more complicated. Therefore, we present a practicable type debugging system in this thesis.

Note on this thesis. In this thesis, we focus on strongly typed functional languages, especially the OCaml language [3, 9]. We use the syntax of OCaml version 3.12.1 as example programs.

1.1 Type checking and type errors

In this section, we give a basic overview of type errors. First, we describe how type checking processes proceed, and then we discuss why type errors are caused and where they come from.

1.1.1 Type checking

The checks that a compiler performs for the consistency of the types are called type checking. In statically typed language, type checking is performed during compiling. Type checking is often called *type inference* in strongly typed functional languages because compilers infer types in the checking phase. Because the expressions are checked before execution, the safety of the types is maintained.

To see the flow of type checking, let us consider an example program of `1 + 2`. When we see this program, we think we can evaluate it successfully, for three reasons:

- "`+`" is an operator that receives two numbers.
- `1` and `2` are numbers.
- Because the upper two have no conflicts, the expression can be evaluated successfully.

The type checkers of compilers also judge this program as well-typed with the same reasoning:

- Because `+` receives two numbers and returns one number, the type of `+` is `int -> int -> int`.
- Because 1 and 2 are numbers, their types are `int`.

- Because there is no conflict, we can evaluate the program.

Because the types of a function "+" and its two arguments match, that is, this program is well-typed, this program never causes type errors in the evaluation phase.

To see an example of how type checking fails, let us consider an example program of `1 + true`. This program has a type conflict because `+` requires two numbers and the second argument is not a number. In this case, the program is ill-typed. This is how the type checking process proceeds. If the final result of an expression can be typed, the programmers receive benefits immediately. Otherwise, the programmers have to struggle with type errors.

1.1.2 Two conflicting expressions

There are many cases in which written programs are not well-typed. In this subsection, we take a look at why type errors are caused.

A type error occurs when types of two expressions conflict with each other. Let us consider the following example:

```
let rec f g lst = match lst with
  | [] -> []
  | fst :: rest -> (g fst) :: (f g rest) in
(f 1 [2;3;4]) @ [5;6;7]
```

In this program, the two boxed expressions have a type conflict causing a type error. The first argument `g` of the function `f` is used as a function in `(g fst)`, but an integer `1` is passed as `g` in `(f 1 [2;3;4])`. Because a function type `'a -> 'b` cannot be unified with `int`, a type error occurs. To locate these two conflicting expressions is useful when one of them is the source of a type error. Unfortunately, that is not always the case.

1.1.3 The source of a type error

The source of a type error cannot be determined solely from the conflict of types. For example, suppose that a call to `f` in the previous example is wrapped by a call to `h`.

```
let rec f g lst = match lst with
  | [] -> []
  | fst :: rest -> (g fst) :: (f g rest) in
let h n lst = f n lst in
(h 1 [2;3;4]) @ [5;6;7]
```

In this program, although `(g fst)` and `(h 1 [2;3;4])` are the conflicting expressions, the source of the type error may be in the definition of `h`: if we replace the boxed expression with `(f (fun x -> x + n) lst)`, the program is well-typed. Because which of these expressions is the source of the type error depends on the programmer's intention, we cannot locate the source of the type error automatically.

1.2 Thesis outline

When programmers encounter type errors, they usually wish to locate the sources by type debugging systems automatically. Unfortunately, this is impossible. However, type debugging systems have the potential to support programmers by removing some of their struggles. To overcome the problem of many compilers lacking type debugging systems, our aim here is to provide practicable type debugging systems.

In the next chapter, we present an overview of approaches to type errors. We classify them into three categories: typing algorithms, type error slicing,

and type debugging. All of them are used in this thesis, and after this overview they are described in details in Chapter 12.

In Chapter 3, we discuss our manifesto for practicable type debugging. We consider "*practicable*" from two sides. One is the implementation side. To implement type debuggers for many existing compilers, their implementation must be both easy and accurate. The other side is for use. If we implement a type debugger but it is not useful, the type debugger is meaningless.

Because "practicable type debugging" has two meanings, the main topic of this thesis consists of two parts.

The first topic relates to the implementation side and is covered in Chapters 4 to 7. The main idea of this topic is to use a compiler's type inferencer for constructing a tree for debugging. Thanks to the type inferencer, we can implement the debugger easily. In Chapter 4, we describe why type debugging is needed and the problems inherent in previous systems. After that, we present our idea of how to solve the problems. In Chapter 5, we apply the idea to simply-typed lambda calculus and extend the language with let-polymorphism. To keep our type debugger accurate, we introduce a property. In Chapter 6, we extend the language to other features, such as objects, modules, and weak polymorphism to see how our technique scales. In Chapter 7, we describe our implementation of a type debugger for OCaml and conclude this topic.

The second topic relates to the user side and is covered in Chapters 8 to 11. The main idea of this topic is to reduce the burden on the programmers for type debugging by weighted type error slicing. Although this approach also uses a compiler's type inferencer, its most unique point is that it focuses on which part is likely to be the source of the type error. When we write programs, we have some intentions of types even if the programs are ill-typed.

The programmer's intentions obtained by ill-typed programs are useful for type debugging. In Chapter 8, we describe the problem of our type debugger and ways of solving it. In Chapter 9, we introduce a type error slicer using the compiler's type inferencer. In Chapter 10, we extend it to have weights that mean the likelihood of each expression being the source of the type error. by simple majority vote. In Chapter 11, we improve the weighted type error slice to have better weights.

Chapter 2

Background

In this chapter, we present an overview of the background of this work. To analyse the necessary property of practicable type debugging, we overview several techniques for type errors, such as typing algorithms, type debugging, and type error slicing.

2.1 Typing algorithms

The popular approach to support programmers for type errors is constructing a new type inference for improving type error messages. In this section, we take a look at three algorithms.

The standard purpose of type inference algorithms is to infer the type of the expressions. When the expressions are well typed, their final results are generally the same. However, when the expressions are ill-typed, their behaviors are quite different.

To see the difference between typing algorithms, let us consider the following three ill-typed examples:

```
(fun x -> (x + 1, x 3)) 4                                    (1)
```

In this program, because we use `x` as `int` in `x + 1` and as a function in `x 3`, type error occurs. Here, we assume that the source of this type error is in `x 3`. Although the programmer who wrote it thinks that the type of `x` is `int`, he forgot to write `+` in `x 3`.

```
(fun x -> (x ^ x) *. 3.)                                    (2)
```

In this program, because `^` is a string concatenation in OCaml, `x ^ x` returns `string`. However it is passed to float-point multiplication `*.`. In this way, this program is ill-typed. We assume that the source of the type error in this program is `^`. Although the programmer misunderstood that `^` is the exponential operator, it is string concatenation in OCaml.

```
List.map (fun (fst :: snd) -> fst + snd) [(1, 2); (2, 3)]    (3)
```

In this program, because we use `snd` as `'a list` in `(fst :: snd)` and as `int` in `fst + snd`, type error occurs. We assume that the source of the type error in this program is `(fst :: snd)`. If we replace `::` with `,`, this program becomes well-typed. Using these examples, we see how different the three typing algorithms are.

2.1.1 Algorithm *W*

Algorithm W [4] is the de facto standard typing algorithm. It receives an expression and an environment for inferring types. The received environment includes the expected types for variables. The point of algorithm W is these environments. The environments are carried during the type checking process and updated by the uses of the variables.

If the types of an environment are all programmer's intended types, algorithm W often locates the source of the type error correctly. Otherwise

algorithm W misidentifies the source of the type error by the wrong environment. For example, in `(1)`, when it infers the type of `x + 1`, algorithm W has an environment {`x:int`} . This environment is correct for the programmer. After that, algorithm W infer the type of `x 3`. Because the program `x 3` has a type conflict with the environment {`x:int`}, algorithm W locates the source of the type error is in `x 3`. In this example, algorithm W works well. However, in `(2)`, algorithm W does not locate the source of the type error. Because the type of `(x ^ x)` is `string` along of the type of `^ string -> string -> string` and the type of `*.` is `float -> float -> float`, there is a type conflict. Algorithm W misidentifies the source of the type error is in `(x ^ x) *. 3.` As just described, if the obtained environment and inferred type are the correct, algorithm W works well. Otherwise, it misidentifies the source of the type error.

2.1.2 Algorithm *M*

Algorithm M [12] receives an expression and an environment the same way as W, and it receives the expected type of the expression additionally. If the types of an environment and the expected type are correct for programmer, this algorithm works well. Otherwise, it misidentifies the source of the type error by wrong information. For example, in `(2)`, because the type of `*.` is `float -> float -> float`, it requires `float` to its arguments. Algorithm M uses this information when it infers the type of `(x ^ x)`. Because this expected type `float` requires `^` to be `'a -> float` and the type of `^` is `string -> string -> string`, algorithm M locate the source of the type error is `^`. However, in `(3)`, algorithm M does not locate the source of the type error. When it infers the type of `(fst :: rest)`, it has an environment {`fst:'a, snd:'a list`}. Although this environment is not correct for programmers,

the typing algorithm has no way to know it. It continues typing and infers the type of `fst + snd`. Because this expression has a type conflict with the environments, it misidentifies `fst + snd` is the source of the type error.

2.1.3 Compositional typing

Compositional typing has been proposed by Chitil [2] for type debugging. We will take a look of its debugging side later (Section 2.3). Because this typing does not require an environment or an expected types of an expression, the identified parts are larger than the previous two algorithms. The previous two type inferences assume inferred types are correct during type inference. Because compositional type inference removes this assumption, it can infer as many types as possible.

2.1.4 Essentials of typing algorithms

Because algorithm M carried more information than algorithm W and compositional typing, Algorithm M stops earlier than they do. If the information is correct for programmers, it produces better result than them. Otherwise, it misidentifies the source of the type error. This problem is also in algorithm W. Although type inference algorithms sometimes identify the source of a type error correctly, it is impossible for them to always identify the source of a type error correctly by just one error message. Let us consider the following example:

```
let rec f lst n = match lst with
  | [] -> []
  | fst :: rest -> (fst ^ n) :: (f rest n)
in f [1;2;3]
```

In OCaml, the operator `^` is a function of the concatenation of two strings. Although `^` requires that `f` has type `string list -> string -> string`, we apply `f` to `[1;2;3]`. This is why this program is ill-typed.

In this case, we can consider several potential error sources. For example, the source of the type error may be in `^`. If we replace `^` with `**` (a function for an exponentiation), this program becomes well-typed. On the other hand, the source of the type error may be in `[1;2;3]`. If we replace `[1;2;3]` with `["1";"2";"3"]`, this program becomes well-typed, as well.

In other words, if we consider exactly the same ill-typed program, the source of the type error is going to be different depending on the programmer's intention.

2.2 Type debugging

To locate the sources of type errors, we have to debug programs somehow with our intended types. Without a special type debugging system, we often use our intended types to debug an ill-typed program by annotations. Because annotations are the same as writing programmer's intended types, a type inferencer could potentially produce better type error messages.

The other approach is type debugging systems. The type debugger that Chitil proposed [2] asks programmers several questions about their intended types. It then uses the types to locate the source of the type error.

These debugging systems have two main advantages over annotations. First, they enable us to omit writing types in programs. Because this is a benefit of type inference, we can say this point preserves the benefit of type inference. Second, it is quite frankly unrealistic to annotate every part of a program. Although annotations can help programmers, they do not always locate the source of the type error.

The advantage of type debugging system as the technique for type errors is that they can locate the source of the type error. The disadvantage is that the need of programmer's intention, sometimes become a burden on programmers.

2.3 Type error slicing

Type error slicing is often used for type debugging. It is a popular technique to narrows the area that relates to type errors. A type error slice consists of some parts of an ill-typed program. Each part of a type error slice relates to the type error. A type error slice of the previous example in Section 2.1.4 is

```
let rec f ... n = ...
  | ... -> ...
  | ... -> (... ^ n) ...
in f [1;2;3]
```

This slice includes the parts we consider as potential sources of the type error. The advantage here is that we can obtain type error slices without any special programmer's input (e.g., annotations), unlike type debugging. Although it narrow the area, it can not locate the source of the type error. Moreover, a type error slice for a large program may also be large. To locate the source of the type error, programmers must search for the source of the type error from a slice, which can be time-consuming.

Chapter 3

A manifesto of practicable type debugging

In this chapter, we discuss the required properties for practicable type debugging. In this thesis, "practicable" has two meanings. One is the producer side of type debuggers, and the other is the consumer side.

3.1 Producer side

First, let us consider the producer side of type debuggers. The producer side relates to the implementation of type debuggers. One problem here is that many languages lack type debugging systems. There have been several implementation proposed for improving the type errors, but because new languages are proposed and changed so quickly, many of them still lack type debugging systems. To overcome this situation, we need a type debugging system that can be applied to many languages easily.

Producer (implementer) side To implement type debuggers for existing languages, the type debugging systems should satisfy following properties:

(1) Have type debugger behavior and compiler behavior that is consistent

Usually, programmers use particular language compilers. When they debug programs by a type debugger, it must produce the same result as the compilers. This means the type debugger should infer exactly the same types for expressions as the compiler's type inferencer. If the object language is small, this is not so difficult. However, if we want an implementation for a wide variety of languages, it becomes very hard and we have to simply deal with it as best we can.

(2) Be easy to implement

(3) Be easy to adapt to updated compiler

These two properties are important to ensure that type debuggers will work with existing compilers. They include not only easing the effort of implementing the debugger itself but also the conceptual understanding of how to design and extend the debugger for larger language constructs. Because a type debugging system must be expanded to cope with more languages, the expansion must be easy to understand. Additionally, to preserve property (1), we have to catch and deal with compiler updates.

(4) Be accurate

The part located by the type debugger must be the source of the type error the programmers made. If this property is lacking, the type debugger is meaningless.

(5) Be applicable to many languages

To implement type debuggers for languages that lack type debuggers, the type debugging system should be applicable to many languages.

Therefore, they need low restrictions. To compare this measure, type debugging systems need to establish their restrictions.

3.2 Consumer side

The other side is for the uses of type debuggers. If we can make a type debugger easily but it is not useful for programmers, it is meaningless. Essentially, a practicable type debugger must be practical for programmers.

Consumer (user) side

(1) Be easy to debug

The debugging process should require minimal effort on the part of the programmers. If type debugging is not easy, the programmers will feel that hand debugging is better. Practicable type debugging cannot be realized without this property.

(2) Produce good messages

This property is important for programmers to understand why type errors occur. When a type debugger locates the source of a type error, it should explain why it is the source. This property is mostly accomplished by a manifesto of type error messages [30].

(3) Have a good user interface

Type debuggers often have to show a focused part of the ill-typed program. To show it exactly and clearly, graphical user interfaces are required.

(4) Be quick enough to use

Type debuggers are often used interactively by programmers. To use them without stress, they have to be quick. Because type debuggers are interactive, they can use the waiting time for user inputs.

Chapter 4

An embedded type debugger

From this Chapter 4 to Chapter 7, we focus on the producer side of type debuggers. In this chapter, we describe how we can locate the source of a type error by previous systems and what is the problems of them. After that, we present our idea to solve the problems.

As we saw in Chapters 1 and 2, the main purpose of type debuggers is to locate the source of a type error. First, we see how we can locate the source of a type error using programmer's intention.

4.1 Locating the source of a type error

A standard type inference tree. To locate the source of a type error, we basically detect the difference between an inferred type and a programmer's intended type. Let us consider a small example:

```
(fun x -> x + x) true
```

This program is ill-typed, because `true` is passed to `x`, but `x` is consumed by an integer addition `+`. Let us assume that the programmer wrote this program, because he mistakenly thought that `+` was the logical *or* operator.[1]

[1]This is an example of the source of this type error. If the programmer has a different

Since the logical *or* operator in OCaml is `||`, the programmer's intended program is `(fun x -> x || x) true`.

We show a standard type inference tree for this example constructed by the compiler in Figure 4.1 and programmer's intended type tree in Figure 4.2. By detecting the difference between these two type inference trees, we can locate an expression that includes the source of a type error. For example, since types of expressions in the boxed part differ in Figures 4.1 and 4.2, the source of the type error resides in the expression `(fun x -> x + x)`. However, we cannot further identify which subexpression of this expression is the root cause of the type error, as long as we use a compiler's type inference tree.

The standard type inference tree is not suited for type debugging, because a type of an expression can depend on the types of other expressions. In the above example, the type of `x` does not have to be `int` if it appears independently. It becomes `int`, because it is used as an argument of `+`. Such information is lost in the standard type inference tree, because the type of `x` becomes `int` throughout, once it is unified with the argument type of `+`.

The most general type tree. To break the dependency between expressions, we introduce the most general type tree. We show the most general type tree for our example in Figure 4.3. The most general type tree holds the most general type for each subexpression. For example, `x` has a typing {`x:'a`} $\vdash$ `x:'a` for any type `'a`, because `x` alone does not require any constraints on its type. The type of `x` is constrained only when it is used in a context. For example, `x + x` has a typing {`x:int`} $\vdash$ `x + x:int`, because `+` requires that `x` has type `int`. Using this most general type tree, we can exactly locate the source of a type error by detecting difference between inferred types and

intention, other fixes are possible, such as replacing `true` with `1`.

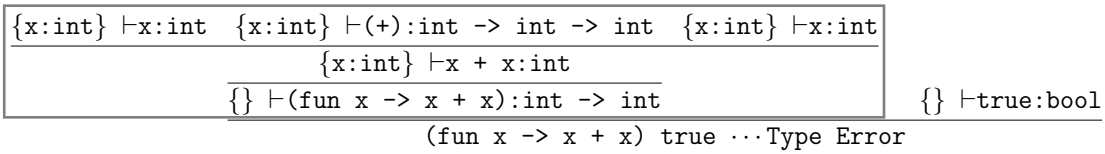


Figure 4.1: A standard type inference tree

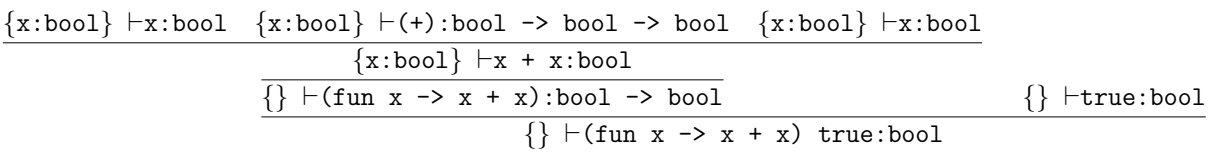


Figure 4.2: Programmer's intended type tree

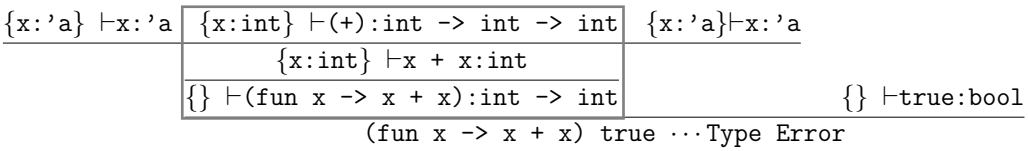


Figure 4.3: The most general type tree

intended types. By comparing Figures 4.3 and 4.2, we find that the type conflict occurs in the boxed part of Figure 4.3. We can then locate the source of the type error to be `+`. Note that the type of `x` (at the two leaves of the tree) does not contradict with programmer's intended type, because `bool` is an instance of `'a`.

Algorithmic debugging. Of course, a tree with programmer's intended types exists only in programmer's mind. To extract programmer's intention, we use algorithmic debugging proposed by Shapiro [20]. Algorithmic debugging is used to identify the location of an error in a tree by traversing over the tree according to oracles. For oracles, questions for the programmer are often used. It is originally used for Prolog, but algorithmic debugging can be used for any tree structures and is applied to various areas, to locate run-time errors [18], semantic errors [21], etc. To debug Figure 4.3 using algorithmic

debugging, we start from the root of the tree where a type error occurs. The type debugger first asks if the two child nodes are correctly typed according to programmer's intention. Since the programmer's intended type for `(fun x -> x + x)` is not `int -> int` but `bool -> bool`, the programmer answers no to the first question. From this answer, the type debugger determines that the source of the type error resides within this expression. Next, the type debugger asks whether the intended type of `x + x` is `int`. Again, the answer is no, and the type debugger moves into the subexpression. By repeating this process, the type debugger locates the source of the type error as `+`.

4.2 Problems

Chitil [2] constructed the most general type tree by inferring types compositionally, and located the source of a type error interactively using algorithmic debugging. Using his type debugger, one can locate the source of a type error by simply answering questions.

Following Chitil's work, we implemented a type debugger for a subset of OCaml together with some improvements [24] and used it in a course in our university. However, due to the need to implement a tailor-made type inferencer, we encountered at least three problems.

Implementation of a type inferencer. First, to implement a type inferencer that returns exactly the same type as the compiler's type inferencer is tedious and error-prone. Even for a small language, we had to fully understand the behavior of the compiler's type inferencer. For example, a compiler has an initial environment for typing. If a tailor-made type inferencer lacks a part of the initial environment, it cannot infer the same type as the compiler's type inferencer. Furthermore, the discrepancy between the two type inferencers becomes apparent only when we find unexpected debugging behavior. It

makes it hard to detect errors in the tailor-made type inferencer.

Support for advanced features. Secondly, to implement a type inferencer for advanced features, such as objects and modules, is difficult and takes time. In our previous type debugger [24], we could implement the main subset of OCaml, including functions, lists, and pattern matching, but not the advanced features, such as user-defined data structures, objects, and modules. This is unfortunate: a type debugger would be particularly useful in the presence of such advanced features.

Compiler's updates. Thirdly, to reimplement the type inferencer every time the compiler is updated is costly. In the last three years, the OCaml compiler had two major updates and two minor updates. It is not realistic to follow all these updates and reimplement the type inferencer.

To solve these problems, we propose *not* to implement a tailor-made type inferencer but to use the compiler's own type inferencer as is to construct the most general type tree.

4.3 Our approach

Rather than implementing our own type inferencer, we use a compiler's type inferencer to construct the most general type tree. Construction consists of two stages. First, the erroneous program to be debugged is decomposed into subprograms. This decomposition determines the overall shape of the tree. Then, the type of each subprogram is inferred by passing the subprogram to the compiler's type inferencer. For example, if a program M is decomposed into subprograms, $M_1, \ldots, M_n$, we first construct the left tree below.

$$\frac{M_1 \quad \ldots \quad M_n}{M} \quad \Rightarrow \quad \frac{M_1 : \tau_1 \quad \ldots \quad M_n : \tau_n}{M : \tau}$$

We then infer their types (possibly an error) by passing each of M_i (and M) to the compiler's type inferencer to obtain its type τ_i (and τ). Note that unlike the standard type inference, types of subexpressions are *not* determined by applying typing rules to the parent expression. Rather, they are determined by executing the compiler's type inferencer for each subexpression independently.

The above explanation is somewhat simplistic, because we did not consider bindings. To cope with bindings properly, we actually maintain a context C of an expression M, treating $C[M]$ as a complete closed program (where $C[M]$ is the expression C whose hole is filled with M, possibly capturing free variables of M). We call M in $C[M]$ the *focused* expression.

Overview from Chapter 5 to Chapter 7. In the rest of this topic, we first show a type debugger for the simply-typed lambda calculus in Section 5.1 and a necessary property for decomposition in Section 5.2. To expand the type debugger to Hindley-Milner type system, we extend it with let polymorphism in Section 5.3.

In Chapter 6, we extend the type debugger with several extension to see how our technique scales. The extension includes objects in Section 6.1, weak polymorphism in Section 6.2 and modules in Section 6.3. Although these extensions is a bit complicated, the basic idea is the same.

We describe our implementation of a type debugger for OCaml that uses OCaml's own type inferencer in Section 7. We explain how to find a minimum part of an ill-typed program and debug it in Section 7.1. After that, we describe the details of our implementation in Section 7.2, and concludes in Section **??**.

Chapter 5

A type debugger for Hindley-Milner type system

In this chapter, we propose a type debugger using the idea which we introduced in the previous chapter. First, we present a type debugger for the simply-typed lambda calculus. After that, to extend it to Hindley-Milner type system, we expand its syntax with let-polymorphism. Because the type debugger for Hindley-Milner type system is the basics of our approach, it explains most of our type debugger.

5.1 The simply-typed lambda calculus

In this section, we introduce a type debugger for the simply-typed lambda calculus. Although simple, it is enough to explain the basic behavior of our type debugger.

The language. We show the syntax of lambda calculus $\lambda_{\rightarrow}$ in Figure 5.1. It includes constants, variables, abstractions, and applications. We assume that basic primitive operations (such as + that we will use in examples) are predefined as constants. Types include type variables, type constants, and

function types.

Tree structure determined by decomposition. Let us consider a type inference tree for $\lambda x.x + 1$. Since the only subprogram of $\lambda x.x + 1$ is $x + 1$ and it is further decomposed into three subprograms, x, $(+)$, and 1, the overall structure of the tree should look like:

$$\dfrac{\dfrac{\Gamma_0 \vdash x \quad \Gamma_0 \vdash (+) \quad \Gamma_0 \vdash 1}{\Gamma_0 \vdash x + 1}}{\Gamma_0 \vdash \lambda x.x + 1}$$

where Γ_0 is the initial environment used by the type inferencer of the underlying compiler and contains all the bindings for the supported constants. However, the above subprograms are not directly typable using the compiler's type inferencer, because they include free variables (such as x).

Decomposition with contexts. To make a subprogram typable, we enclose it with a context that supplies necessary bindings for free variables. In this language, a context is defined as either an empty context $\Box$ or a lambda binding $\lambda x.C$ (Figure 9.1). The most general type tree of $\lambda x.x + 1$ becomes as follows:

$$\dfrac{\dfrac{\Gamma_0 \vdash \lambda x.[x] : \texttt{'a -> ['a]} \quad \Gamma_0 \vdash \lambda x.[(+)] : \texttt{'a -> [int -> int -> int]} \quad \Gamma_0 \vdash \lambda x.[1] : \texttt{'a -> [int]}}{\Gamma_0 \vdash \lambda x.[x + 1] : \texttt{int -> [int]}}}{\Gamma_0 \vdash [\lambda x.x + 1] : \texttt{[int -> int]}}$$

Looking at the focused expressions filled in the context, we see that it has the same structure as the previous tree. Thanks to the contexts, all the subprograms are now typable under Γ_0. The types enclosed by `[...]` correspond to the types of focused expressions.

Although the above tree is similar to the standard type inference tree for $\lambda_{\rightarrow}$:

$$
\begin{array}{rcll}
(M : term) & ::= & c & \text{(constant)} \\
 & | & x & \text{(variable)} \\
 & | & \lambda x.M & \text{(abstraction)} \\
 & | & M_1\ M_2 & \text{(application)} \\
(\tau : typ) & ::= & b & \text{(type variable)} \\
 & | & \text{int}, \text{bool}, ... & \text{(type constants)} \\
 & | & \tau_1 \rightarrow \tau_2 & \text{(function type)} \\
(C : context) & ::= & \Box & \text{(empty context)} \\
 & | & \lambda x.C & \text{(lambda context)}
\end{array}
$$

Figure 5.1: The syntax of simply-typed lambda calculus $\lambda_{\rightarrow}$

$$
\begin{array}{rcl}
Dec : context * term & \rightarrow & (context * term)\ list \\
Dec[\![(C, c)]\!] & = & [] \\
Dec[\![(C, x)]\!] & = & [] \\
Dec[\![(C, \lambda x.M)]\!] & = & [(C[\lambda x.\Box], M)] \\
Dec[\![(C, M_1\ M_2)]\!] & = & [(C, M_1); (C, M_2)]
\end{array}
$$

Figure 5.2: The decomposition function Dec for $\lambda_{\rightarrow}$

$$
\begin{array}{rcl}
env = (var * typ)\ list & & \\
Collect : context \rightarrow typ \rightarrow env & \rightarrow & (env * typ) \\
Collect_{\Box}[\![\tau]\!]\mu & = & (\mu, \tau) \\
Collect_{\lambda x.C}[\![\tau_1 \rightarrow \tau_2]\!]\mu & = & Collect_C[\![\tau_2]\!]\mu[x \mapsto \tau_1]
\end{array}
$$

Figure 5.3: The function $Collect$ to obtain types of free variables for $\lambda_{\rightarrow}$

$$
\begin{array}{rcl}
Judge[\![(C, M)]\!] & = & \text{let } M' = C[M] \text{ in} \\
 & & \text{let } \tau = \texttt{typing}\ M' \text{ in} \\
 & & \text{let } (\gamma, \tau') = Collect_C[\![\tau]\!][] \text{ in} \\
 & & (\gamma, \tau')
\end{array}
$$

Figure 5.4: The function $Judge$ to obtain typing for $\lambda_{\rightarrow}$

$$\dfrac{\dfrac{\Gamma_0, x : \texttt{int} \vdash x : \texttt{int} \quad \Gamma_0, x : \texttt{int} \vdash (+) : \texttt{int -> int -> int} \quad \Gamma_0, x : \texttt{int} \vdash 1 : \texttt{int}}{\Gamma_0, x : \texttt{int} \vdash x + 1 : \texttt{int}}}{\Gamma_0 \vdash \lambda x.x + 1 : \texttt{int -> int}}$$

they have two important differences. First, the type of x is *not* constrained to `int` at the leaf nodes. Since we treat all the subderivations independently, each judgement depends only on its subexpressions. It enables us to locate where the type of x is first forced to `int`. Secondly, the type environment contains only the predefined constants. It enables us to use the compiler's type inferencer to infer the type of each expression. We simply pass it to the compiler's type inferencer and obtain its type. This is in contrast to the standard type inference tree where the environment contains free variables.

Other Approach. A compiler's type inferencer is usually designed to accept an open expression and an environment for its free variables. Although we could use this extra flexibility for the type debugger, it does not lead to a simpler type debugger. In this thesis, we chose to use contexts, to avoid going into the underlying compiler implementation together with the representation of environments. If we want to implement type debuggers for various languages, it would require substantial investigation of the underlying compiler. The method proposed here has an advantage that we can treat the compiler's type inferencer completely as a black box that accepts an expression and returns its type.

Construction of the most general type tree. The most general type tree is built as follows. A program to be debugged $C[M]$ is first decomposed into subprograms using the decomposition function *Dec* defined in Figure 5.2. It basically decomposes M and returns a list of its subprograms, but it main-

tains its contexts properly so that the resulting subprograms (pairs of a context and a decomposed term) are always closed. When the decomposition of $C[M]$ is $[C_1[M_1]; \ldots; C_n[M_n]]$, all the subprograms become the children of $C[M]$ in the tree.

The type of each subprogram $C[M_i]$ is determined using the compiler's type inferencer by passing $C[M_i]$ to it. When the context C is empty $\Box$, the returned type is the type of the expression. When the context is not empty, we split the obtained type into two: types for free variables and the type for the focused expression. If we obtain the type of $\lambda x.[x+1]$ as `int -> int`, for example, we associate the type of x to be `int` (the argument part of `int -> int`) and the type of $x+1$ to be `int` (the body part of `int -> int`). This is done by the function *Collect* in Figure 5.3.

Using *Dec* and *Collect*, we construct a judgement for $C[M]$ in the tree as shown in Figure 5.4. First, we construct a closed term M' by plugging M into C. It is then passed to the compiler's type inferencer written as `typing` here. When we obtain a type τ of M', we split it into an environment γ holding types of variables in the context and a type τ' for M. Using them, we can construct a judgement for (possibly open) M (in the context C) as $\Gamma_0, \gamma \vdash M : \tau'$. For $\lambda x.[x+1]$, for example, we have $\Gamma_0, x : \texttt{int} \vdash x+1 : \texttt{int}$.[1]

5.2 The decomposition property

In our type debugger, the most general type tree is constructed by first decomposing an expression into subexpressions and then inferring their types using the compiler's type inferencer. The shape of the tree is determined by

[1]Before, we wrote $\Gamma_0 \vdash \lambda x.[x+1] : \texttt{int -> [int]}$ to emphasize that we are using the compiler's type inferencer to infer the type of M in C. Since we are interested in the type of M itself together with the types of its free variables, we also write it using the standard notation $\Gamma_0, x : \texttt{int} \vdash x+1 : \texttt{int}$.

how we decompose an expression. However, it does not mean that we can use any arbitrary decomposition. We require that the decomposition satisfies the following necessary property:

Definition 1 (The decomposition property) *The decomposition function Dec should satisfy the following property for any context C and term M:*

$$T(C[M]) \Rightarrow \forall (C', M') \in Dec[\![(C, M)]\!], T(C'[M'])$$

where T is a predicate stating that a given expression is well typed (under the compiler's type inferencer).

The decomposition property states that if a program $C[M]$ is well typed, all of its decomposed subprograms are also well typed. Although this property looks trivial, it does preclude $x + 1$ as a decomposition of $\lambda x.x + 1$, because the latter is well typed, but the former is not typable with unbound x. In the next section, we will see how this property guides us to define decomposition that is suitable for type debugging.

This property is essential for our type debugger. Since the source of a type error is detected by tracking conflicts between inferred types and intended types, we can no longer continue type debugging into subexpressions if their inferred types are not available from the compiler's type inferencer. Therefore, we design decomposition carefully so that it satisfies the property and thus keeps the typability of expressions. In the following sections, we sketch why the presented decomposition satisfies this decomposition property. For the simply-typed lambda calculus, we reason as follows.

Decomposition for $\lambda_{\rightarrow}$ satisfies the decomposition property. We need to show that for each case of the definition of *Dec* in Figure 5.2, all the subexpressions in the right hand side are well typed if the left hand side is well typed. For

constants and variables, it is satisfied vacuously. For abstraction, because the expression in the left hand side $C[\lambda x.M]$ is identical to the expression in the right hand side $C[\lambda x.[M]]$, the decomposition property is satisfied. For application, we notice that if $C[M_1 M_2]$ is well typed, $M_1 M_2$ is also well typed in a type environment consistent with C (formally proven by induction on C). Hence, both M_1 and M_2 are well typed in the same environment. Since C has all the necessary bindings for M_1 and M_2 and C simply adds binding to them, both $C[M_1]$ and $C[M_2]$ are well typed as required.

5.3 Let polymorphism

In this section, we extend our idea to let polymorphism.

The language. We show the syntax of λ_{let} in Figure 5.5. It extends the simply-typed lambda calculus with pairs, fixed points, and let expressions. Types are also extended accordingly. Unlike the standard let-polymorphic calculus, we do not introduce type schemes. Type schemes are required only for inferring types. Once the type inference is done (in the compiler), all the expressions in the most general type tree are given mono types (possibly containing type variables).

Naive decomposition. To support a let expression in the type debugger, we first need to define its decomposition. Because a let expression contains two subexpressions, the let-bound expression and the main body, we are tempted to define its decomposition as these two subexpressions. However, straightforward decomposition leads to violation of the decomposition property (Section 5.2). Let us consider the following program:

$$1 + (let\ id = \lambda x.x\ in\ (id\ id)\ 2.0)$$

Since *id* in the second subexpression (*id id*) 2.0 is free, we need to supply its context. If we naively follow the previous section, however, we end up with the following tree:

$$\dfrac{\vdash [1] : \texttt{int} \quad [+] : \texttt{int}\rightarrow\texttt{int}\rightarrow\texttt{int} \quad \boxed{\dfrac{\vdash [\lambda x.x] : \texttt{'a}\rightarrow\texttt{'a} \quad \vdash (\lambda id.[(id\ id)\ 2.0]) \cdots \texttt{Type Error}}{\vdash [let\ id = \lambda x.x\ in\ (id\ id)\ 2.0] : \texttt{float}}}}{\vdash [1 + (let\ id = \lambda x.x\ in\ (id\ id)\ 2.0)] \cdots Type\ Error}$$

Although the bottom expression in the boxed part is well typed, one of its subexpressions is not well typed. Thus, this decomposition does not satisfy the decomposition property.

The reason why ($\lambda id.[(id\ id)\ 2.0]$) is not typable is clear. In the original expression, *id* is used polymorphically, while in the decomposed subexpression, *id* is bound by λ and thus monomorphic. From this example, we observe that we need to preserve the polymorphic types of let-bound variables, when decomposing expressions.

Decomposition with let context. To preserve polymorphic types of let-bound variables, we extend the context with a let context (Figure 5.5). We also extend it with a fix context since it is a (monomorphic) binder. Using the let context, the above tree changes as follows, satisfying the decomposition property:

$$\dfrac{\vdash [1] : \texttt{int} \quad [+] : \texttt{int}\rightarrow\texttt{int}\rightarrow\texttt{int} \quad \boxed{\dfrac{\vdash [\lambda x.x] : \texttt{'a}\rightarrow\texttt{'a} \quad \vdash (let\ id = \lambda x.x\ in\ [(id\ id)\ 2.0]) : \texttt{float}}{\vdash [let\ id = \lambda x.x\ in\ (id\ id)\ 2.0] : \texttt{float}}}}{\vdash [1 + (let\ id = \lambda x.x\ in\ (id\ id)\ 2.0)] \cdots \texttt{Type Error}}$$

Construction of the most general type tree. To enable inspection of the definition of let-bound variables, we change the decomposition function as shown in Figure 5.6. The definition is the straightforward extension of the previous definition except for the variable case. When we decompose a variable, we

$$
\begin{array}{rcll}
(M : term) & ::= & ... \mid (M_1, ..., M_n) & \text{(tuple)} \\
& \mid & fix\ f\ x \to M & \text{(fixed point)} \\
& \mid & let\ x = M_1\ in\ M_2 & \text{(let expression)} \\
(\tau : typ) & ::= & ... \mid \tau_1 * ... * \tau_n & \text{(product type)} \\
(C : context) & ::= & ... \mid fix\ f\ x \to C & \text{(fix context)} \\
& \mid & let\ x = M\ in\ C & \text{(let context)}
\end{array}
$$

Figure 5.5: The syntax of the let-polymorphic language λ_{let} (new cases only)

$$
\begin{array}{rcl}
Dec : context * term & \to & (context * term)\ list \\
Dec[\![(C, x)]\!] & = & Get(C, x, \Box, \text{None}) \\
Dec[\![(C, (M_1, ..., M_n))]\!] & = & [(C, M_1); ...; (C, M_n)] \\
Dec[\![(C, fix\ f\ x \to M)]\!] & = & [(C[fix\ f\ x \to \Box], M)] \\
Dec[\![(C, let\ x = M_1\ in\ M_2)]\!] & = & [(C, M_1); (C[let\ x = M_1\ in\ \Box], M_2)]
\end{array}
$$

Figure 5.6: Dec for λ_{let} (new cases only)

$$
\begin{array}{rcl}
Get : context * var * context * & & \\
\quad (context * term)\ option & \to & (context * term)\ list \\
Get(\Box, v, C, p) & = & \begin{cases} [] & \text{if } p = \text{None} \\ [(C', M)] & \text{if } p = \text{Some}(C', M) \end{cases} \\
Get(\lambda x.C', v, C, p) & = & \begin{cases} Get(C', v, C[\lambda x.\Box], \text{None}) & \text{if } x = v \\ Get(C', v, C[\lambda x.\Box], p) & \text{if } x \neq v \end{cases} \\
Get(fix\ f\ x \to C', v, C, p) & = & \begin{cases} Get(C', v, C[fix\ f\ x \to \Box], \text{None}) & \\ \qquad \text{if } v \in \{f, x\} & \\ Get(C', v, C[fix\ f\ x \to \Box], p) & \\ \qquad \text{if } v \notin \{f, x\} & \end{cases} \\
Get(let\, x = M\ in\ C', v, C, p) & = & \begin{cases} Get(C', v, C[let\, x = M\ in\ \Box], & \\ \qquad \text{Some(C, M)}) \quad \text{if } x = v & \\ Get(C', v, C[let\, x = M\ in\ \Box], p) & \\ \qquad \text{if } x \neq v & \end{cases}
\end{array}
$$

Figure 5.7: The function Get to search definition of variables for λ_{let}

$$
\begin{array}{rcl}
env = (var * typ)\ list & & \\
Collect : context \to typ \to env & \to & (env * typ) \\
Collect_{fix\, f\, x \to C}[\![\tau_1 \to \tau_2]\!]\mu & = & Collect_C[\![\tau_2]\!]\mu[f \mapsto (\tau_1 \to \tau_2); x \mapsto \tau_1] \\
Collect_{let\, x = M\, in\, C}[\![\tau]\!]\mu & = & Collect_C[\![\tau]\!]\mu
\end{array}
$$

Figure 5.8: $Collect$ for λ_{let} (new cases only)

search for its definition using *Get* defined in Figure 5.7. When the variable is bound by a let expression, *Get* returns its (inner-most) definition as the decomposition of the variable. Otherwise, the variable is bound by lambda or fix, so *Get* returns no decomposition. Using this decomposition function, we can further debug into the definition of let-expressions to identify the source of a type error.

Since the context is extended with a let context and a fix context, the definition of *Collect* is extended accordingly as shown in Figure 5.8. It collects types for lambda- and fix-bound variables and discards let-bound variables since they do not appear in the type returned by the compiler. (We assume that the compiler's type inferencer returns $\tau_1 \rightarrow \tau_2$ as the type of $fix\ f\ x \rightarrow M$ (and hence of f) where τ_1 is the type of x and τ_2 is the type of M.)

As the program to be debugged becomes larger, the number of let-bound variables increases. Since we can debug into the definition of let-bound variables when their types conflict with the programmer's intention, we can skip asking for the type of let-bound variables as an oracle each time. (For example, in the previous tree, the type debugger can skip the node $\vdash [\lambda x.x]$:'a→'a). Rather, we only ask for variables in a context that are bound by lambda or fix. This is consistent with Chitil's approach that maintains an environment for polymorphic variables separately.

Decomposition for λ_{let} satisfies the decomposition property. We can confirm that the decomposition property is still satisfied. The interesting case is for variables. (Other cases are similar to the reasoning shown for $\lambda_{\rightarrow}$.) Assume that $C[x]$ is well typed. We first observe that $Get(C_1, x, C_2, p)$ maintains an invariant that $C_2[C_1]$ is always the same across the recursive call, because at each recursive call, the topmost frame of C_1 is simply moved to the hole

of C_2. This ensures that all the contexts appearing in the definition of *Get* are well typed (as contexts), because the initial context $[C[x]]$ with which *Get* is called from *Dec* is well typed. Next, the returned expression $C[M]$ is collected only from the let case. Because $C[let\ x = M\ in\ C']$ is well typed, we hence have that $C[M]$ is also well typed as required.

Observe how the decomposition property serves as a guideline for what we have to do and what we can do to incorporate let expressions. We have to define the decomposition function so that the let polymorphism is preserved. On the other hand, as long as the decomposition property is satisfied, we have the liberty of defining the decomposition in a way the debugging process becomes easier for programmers to understand. By defining the decomposition of let-bound variables as their definition, the debugger's focus moves from the use of variables to their definition.

Chapter 6

A type debugger for extensions

In this chapter, we extend the previous type debugger with several extensions, objects, weak polymorphism and modules.

6.1 Objects

So far, we have seen that interactive debugging is possible for various language constructs by suitably defining a *Dec* function that satisfies the required property. This idea extends to advanced language constructs. In this section, we introduce objects and see how they can be supported in a similar way.

The language. We show the syntax of the object language λ_{obj} in Figure 6.1. It models OCaml-style objects where an object is defined using a class (in which single inheritance is allowed) and is created by the *new* construct. Besides the inheritance declaration, an object can contain method and value declarations. In OCaml, class names (to be more precise, the object structures denoted by the class names) are used as types. We use them as is in our type debugger, abbreviated as *obj* in Figure 6.1.

$$
\begin{array}{rcll}
(L : classobj) & ::= & inherit\ x & \text{(inheritance declaration)} \\
& | & method\ x = M & \text{(method declaration)} \\
& | & val\ x = M & \text{(value declaration)} \\
(M : term) & ::= & ... \mid x_1 \# x_2 & \text{(method invocation)} \\
& | & new\ x & \text{(object creation)} \\
& | & class\ x\ v_1 ... v_n = & \text{(class definition)} \\
& & \quad object(v') & \\
& & \quad\ L_1 ... L_n & \\
& & \quad end\ in\ M & \\
(\tau : typ) & ::= & ... \mid obj & \text{(object type)} \\
(C : context) & ::= & ... \mid class\ x\ v_1 ... v_n = & \text{(class context)} \\
& & \quad object(v') & \\
& & \quad\ L_1 ... L_n & \\
& & \quad end\ in\ C &
\end{array}
$$

Figure 6.1: The syntax of the object language λ_{obj} (new cases only)

$$
\begin{array}{rcl}
Dec : context * term & \rightarrow & (context * term)\ list \\
Dec[\![(C, x_1 \# x_2)]\!] & = & SearchObj(C, x_1, \Box, []) \\
Dec[\![(C, new\ x)]\!] & = & SearchObj(C, x, \Box, []) \\
Dec[\![(C, class\ x\ v_1 ... v_n = & = & [(C[class\ x\ v_1 ... v_n = \\
\quad object(v') & & \quad\ object(v') \\
\quad\ L_1 ... L_n & & \quad\ L_1 ... L_n \\
\quad end\ in\ M)]\!] & & \quad end\ in\ \Box], M)]
\end{array}
$$

Figure 6.2: *Dec* for λ_{obj} (new cases only)

$$
\begin{array}{rcl}
Get : context * var * context * & & \\
\quad (context * term)\ option & \rightarrow & (context * term)\ list \\
Get(class\ x\ v_1 ... v_n = & = & Get(C', v, C[class\ x\ v_1 ... v_n = \\
\quad object(v') & & \quad object(v') \\
\quad L_1 ... L_n & & \quad L_1 ... L_n \\
\quad end\ in\ C', v, C, p) & & \quad end\ in\ \Box], p)
\end{array}
$$

Figure 6.3: *Get* for λ_{obj} (new cases only)

$$
\begin{array}{rcl}
SearchObj' : classobj\ list * context & \rightarrow & (context * term)\ list \\
SearchObj'([], C) & = & [] \\
SearchObj'((inherit\ x) :: r, C) & = & SearchObj(C, x, \Box, [])@ \\
 & & \quad SearchObj'(r, C) \\
SearchObj'((method\ x = M) :: r, C) & = & (C, M) :: SearchObj'(r, C) \\
SearchObj'((val\ x = M) :: r, C) & = & SearchObj'(r, C[let\ x = M\ in\ \Box])
\end{array}
$$

$$
\begin{array}{rcl}
SearchObj : context * var * context * & & \\
\quad (context * term)\ list & \rightarrow & (context * term)\ list \\
SearchObj(\Box, v, C, p) & = & p \\
SearchObj(\lambda x.C', v, C, p) & = & SearchObj(C', v, C[\lambda x.\Box], p) \\
SearchObj(fix\ f\ x \rightarrow C', v, C, p) & = & \\
\quad SearchObj(C', v, C[fix\ f\ x \rightarrow \Box], p) & &
\end{array}
$$

$SearchObj(class\ x\ v_1 ... v_n = object(v')\ L_1 ... L_n\ end\ in\ C', v, C, p) =$
  if $x = v$ then
    $SearchObj(C', v, C[class\ x\ v_1 ... v_n = object(v')\ L_1 ... L_n\ end\ in\ \Box],$
      $SearchObj'(L_1 ... L_n, C[\lambda v_1 ... \lambda v_n.\lambda v'_v.\Box]))$
  else $SearchObj(C', v, C[class\ x\ v_1 ... v_n = object(v')\ L_1 ... L_n\ end\ in\ \Box], p)$

Figure 6.4: The function *SearchObj* to search for the definition of objects for λ_{obj}

Construction of the most general type tree. The decomposition function *Dec* is extended with the new constructs in Figure 6.2 and the *Get* function used in the variable case is extended with the class context in Figure 6.3. The interesting cases are for *new* and method invocation of *Dec*. In both cases, we need to identify the object mentioned in the expressions (in case their types contradict with intended types, so that we can debug into the object). For this purpose, the function *SearchObj* in Figure 6.4 is used. Its behavior is similar to that of *Get*, but differs in that *SearchObj* collects *all* the method declarations in the designated object. In particular, if the object contains inheritance declaration, those method declarations are collected, too (see *SearchObj'*).

We collect all the declarations in an object because types of declared methods in an object are mutually dependent. Thus, we need to ask for the types of all these method declarations to locate the source of type errors. For example, consider the following program:

```
class counter = object (self)
                  val mutable n = 0
                  method incr = n <- n+1
                  method get = n
                end
let t = (new counter) in
t#incr; ("now, the conter is" ^ t#get)
```

The last line results in a type error, because `t#get` returns an integer, which is in conflict with the intended type (i.e., `string`). To find the source of this type error, we first look up `t`'s class definition `counter` and search for the definition of the `get` method. However, we find here that the `get` method itself does not force the type of `n` as an integer. It simply returns a

value of `n`. Instead, `n` is an integer because it is assigned `0` and `n+1` elsewhere in the class. Thus, we need to examine all the declarations in an object to find the source of type errors.

Since any method declarations can be the source of type errors, we collect all the method declarations in a class definition, and return them as decomposition of the object reference. Although this strategy is necessary in general, it could lead to a large number of questions. Its practical implementation is future work.

Decomposition for λ_{obj} satisfies the decomposition property. We can confirm that *Dec* satisfies the decomposition property as follows. First, *Get* will return a list of well-typed subexpressions only, using the similar argument we described in Section 5. For *new* and method invocation, we have to show that *SearchObj* returns a list of well-typed subexpressions. It can be proved by observing that *SearchObj* simply collects subexpressions in an object in a suitable context. The only interesting case is for a class declaration, where we have to properly insert bindings for the arguments to the class and the self variable v'. Note that declared values are put into let contexts in *SearchObj'*.

6.2 Weak polymorphism

In this section, we introduce references to see the interaction of weak polymorphism in our type debugging. We show the syntax in Figure 6.5. It includes references, dereferences, and assignments to a reference. Types are extended with a reference type. Let us consider a typical example where the weak polymorphism arises:

```
let id_ref = ref (fun x -> x) in
(!id_ref 0, !id_ref true)
```

$$
\begin{array}{rcll}
(M : term) & ::= & c \mid x \mid \lambda x.M \mid M_1\ M_2 \mid (M_1, ..., M_n) & \\
& \mid & ref\ M & \text{(reference)} \\
& \mid & !M & \text{(dereference)} \\
& \mid & v := M & \text{(assignment)} \\
& \mid & fix\ f\ v_1\ ...\ v_n \rightarrow M \mid let\ v\ = M_1\ in\ M_2 & \\
(\tau : typ) & ::= & b \mid \text{int} \mid \text{bool} \mid ... \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * ... * \tau_n & \\
& \mid & \tau\ ref & \text{(reference type)} \\
(C : context) & ::= & \Box \mid \lambda x.C \mid fix\ f\ v_1 ... v_n \rightarrow C \mid let\ x = M\ in\ C &
\end{array}
$$

Figure 6.5: The syntax and types for the language with references

$$
\begin{array}{rcl}
Dec : context * term & \rightarrow & (context * term)\ list \\
Dec[\![(C, ref\ M)]\!] & \rightarrow & [(C, M)] \\
Dec[\![(C, !M)]\!] & \rightarrow & [(C, M)] \\
Dec[\![(C, v := M)]\!] & \rightarrow & [(C, M)]
\end{array}
$$

Figure 6.6: Dec for the language with references (for new constructs only)

$$
\begin{array}{rcl}
ExpVar : context\ *\ var\ list & \rightarrow & var\ list \\
ExpVar[\![(\Box, vs)]\!] & = & vs \\
ExpVar[\![(\lambda x.C, vs)]\!] & = & ExpVar[\![(C, vs \setminus \{x\})]\!] \\
ExpVar[\![(fix\ f\ v_1 ... v_n \rightarrow C, vs)]\!] & = & ExpVar[\![(C, vs \setminus \{f, v_1, ..., v_n\})]\!] \\
ExpVar[\![(let\ x = M'\ in\ C, vs)]\!] & = & \text{if } (\texttt{is\_expansive}\ M') \\
 & & \text{then } ExpVar[\![(C, (x :: vs))]\!] \\
 & & \text{else } ExpVar[\![(C, vs)]\!]
\end{array}
$$

Figure 6.7: The function $ExpVar$ (ExpansiveVar) to collect expansive variables

Since the identity function `(fun x -> x)` is put into a cell, `id_ref` is given a weak polymorphic type `('_a -> '_a) ref`. The weak type variable `'_a` can be instantiated only once. Since it becomes int at the first application, a type error occurs at the second application where `'_a` needs to become bool.

It is not difficult to support references in our type debugger. We simply need to extend *Dec* to handle new constructs (Figure 6.6). We could then identify the source of the above type error as the second line, because whole

$$
\begin{array}{rcl}
env = (var * typ)\ list & & \\
AttachVar : var\ list * term & \rightarrow & term \\
AttachVar[\![([], M)]\!] & = & M \\
AttachVar[\![(v :: vs, M)]\!] & = & (v, AttachVar[\![(vs, M)]\!])
\end{array}
$$

Figure 6.8: $AttachVar$ to pair expansive variables with a focused expression

$$
\begin{array}{rcl}
env = (var * typ)\ list & & \\
CollectVar : var\ list \rightarrow typ & \rightarrow & env \\
CollectVar_{[]}[\![\tau]\!] & = & ([], \tau) \\
CollectVar_{v::\ vs}[\![\tau_1 * \tau_2]\!] & = & \text{let } (\mu, \tau) = CollectVar_{vs}[\![\tau_2]\!] \text{ in} \\
 & & (\mu[v \rightarrow \tau_1], \tau)
\end{array}
$$

Figure 6.9: The function $CollectVar$ to obtain types of expansive variables

$$
\begin{array}{rcl}
Judge[\![(C, M)]\!] & = & \text{let } vs = ExpVar[\![(C, [])]\!] \text{ in} \\
 & & \text{let } M' = C[AttachVar[\![(vs, M)]\!]] \text{ in} \\
 & & \text{let } \tau = \texttt{typing}\ M' \text{ in} \\
 & & \text{let } (\gamma, \tau') = Collect_C[\![\tau]\!] \text{ in} \\
 & & \text{let } (\gamma', \tau'') = CollectVar_{vs}[\![\tau']\!] \text{ in} \\
 & & (\gamma @ \gamma', \tau'')
\end{array}
$$

Figure 6.10: $Judge$ for the language with references

the expression is not typable but the two subexpressions together with their context, namely

```
let id_ref = ref (fun x -> x) in [!id_ref 0]
```

and

```
let id_ref = ref (fun x -> x) in [!id_ref true]
```

are both typable. The most general type tree becomes as follows, where C contains a binding for `id_ref`:

$$
\frac{\vdash C\texttt{[!id\_ref 0]:int} \quad \vdash C\texttt{[!id\_ref true]:bool}}{\vdash C\texttt{[(!id\_ref 0, !id\_ref true)]} \cdots \texttt{Type Error}}
$$

However, the above behavior is sometimes not very informative. Consider the following example:

```
let pair x y = fun f -> f x y in
let fst x y = x in
let snd x y = y in
let p = pair 1 true in
(p snd, p fst)
```

In this program, a pair is Church-encoded using a function. Then, a pair `p` of `1` and `true` is constructed, and its swapped tuple is returned. Because `p` is bound to a non-value, however, it has a weak type `(int -> bool -> '_a) -> '_a`. When p is applied to `snd` of type `'a -> 'b -> 'b`, the weak type variable `'_a` is instantiated to `bool`, and a type error occurs when `p` is applied to fst of type `'a -> 'b -> 'a`, where `'_a` needs to be instantiated to int.

For this program, our type debugger again reports that the expression `(p snd, p fst)` is the source of the type error, because both `p snd` and `p fst` are typable in the current context C (containing four let bindings):

$$\frac{\vdash C\texttt{[p snd]:bool} \quad \vdash C\texttt{[p fst]:int}}{\vdash C\texttt{[(p snd, p fst)]} \cdots \texttt{Type Error}}$$

if both the types are consistent with programmers intention.

However, if the programmer intends that `p` be fully polymorphic, he would be puzzled why the conclusion is not typed as `bool * int`. In fact, although the type of `p` is constrained to `(int -> bool -> bool) -> bool` at `p snd`, that information is discarded in the most general type tree and a fresh `p` is used to infer the type of `p fst`. Remember that all the types are inferred by passing each expression to the compilers type inferencer independently.

Also, note that our type debugger asks the programmer only for the type of focused expression and the types of its free variables that are not bound by let. Since `p` is bound by let in this case, the type debugger asks only the types of `p snd` and `p fst` (both of which have intended types). Thus, the programmer has no opportunity to say that the type of `p` is too restrictive.

To handle weak polymorphism more properly, we examine the type of weak variables and ask if their instantiation is in conflict with the programmers intention. In the above case, we construct the following tree:

$$\frac{\vdash C\texttt{[(p, p snd)]}:\tau_1\ \texttt{*}\ \texttt{bool} \quad \vdash C\texttt{[(p, p fst)]}:\tau_2\ \texttt{*}\ \texttt{int}}{\vdash C\texttt{[(p, (p snd, p fst))]}\cdots\texttt{Type Error}}$$

where $\tau_1 =$`(int -> bool -> bool) -> bool`

$\tau_2 =$`(int -> bool -> int) -> int`

Since the definition of `p` is expansive, we pair it with the focused expression and obtain its type from the compiler. We then ask the programmer if the type of `p` is as intended. In our case, since τ_1 (and τ_2) is not polymorphic enough, the programmer can reply no, and the debugger will move to the definition of p to find why it is not polymorphic.

To enable this behavior, $ExpVar$ in Figure 6.7 collects a list of expansive variables, $AttachVar$ in Figure 6.8 pairs them with the focused expression, and $CollectVar$ in Figure 6.9 extracts the types of expansive variables. When collecting expansive variables, care must be taken for variables with the same name. For example, in the following context,

`fun x -> let x =` *expansive_expression* `in` $\Box$

`x` has to be treated as expansive (because `x` in $\Box$ refers to the inner `x`), but not when `fun x ->` appears inside the let expression. We can obtain a judgement for an expression using Figure 6.10.

We can easily confirm that the required property holds for this language, because the decomposition function for the new constructs takes simply the subexpression of the original expression and the pairing of expansive variables does not affect the typability of expressions. By modifying the expression to be typed without violating the property, we can design a type debugger that shows more useful information for the programmer.

6.3 Modules

Similarly to objects, we can introduce modules, too. Figure 6.11 shows the syntax. We introduce accesses to a value in a module, open, and module declarations. A module declaration contains variable and type declarations.

The decomposition function Dec is extended to cope with new constructs straightforwardly (Figure 6.12). $SearchMod$ (in Figure 6.14) is defined similarly to $SearchObj$ in the previous section. Since the declarations in a module is ordered (in contrast to method declarations in objects which are mutually recursive), we do not collect all the declarations but maintain the order of declarations in a context and returns a designated definition. The treatment of open in Get (in Figure 6.13) is interesting. When we search for the definition of a variable v and the current context is $open\ X$, we search for the definition of v in the module X. It enables us to search for the definition of variable v that is defined in the module X, when the type error was located at the variable v.

$$
\begin{array}{rcll}
(M : term) & ::= & c \mid x \mid \lambda x.M \mid M_1\ M_2 \mid (M_1, ..., M_n) & \\
& \mid & X.x & \text{(module access)} \\
& \mid & fix\ f\ x \rightarrow M \mid let\ v = M_1\ in\ M_2 & \\
& \mid & open\ X\ in\ M & \text{(open)} \\
& \mid & type\ x = \tau\ in\ M & \text{(type definition)} \\
& \mid & module\ X = struct\ D_1...D_n\ end\ in\ M & \text{(module definition)} \\
\\
(D : definition) & ::= & let\ x = M & \text{(value declaration)} \\
& \mid & type\ x = \tau & \text{(type declaration)} \\
\\
(\tau : typ) & ::= & b \mid \text{int} \mid \text{bool} \mid ... \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * ... * \tau_n & \\
(C : context) & ::= & \Box \mid \lambda x.C \mid fix\ f\ v_1...v_n \rightarrow C \mid let\ x = M\ in\ C & \\
& \mid & open\ X\ in\ M & \text{(open context)} \\
& \mid & type\ x = \tau\ in\ C & \text{(type context)} \\
& \mid & module\ X = struct\ D_1...D_n\ end\ in\ C & \text{(module context)}
\end{array}
$$

Figure 6.11: The syntax and types of the module language

$$
\begin{array}{rcl}
Dec : context * term & \rightarrow & (context * term)\ list \\
Dec[\![(C, c)]\!] & = & [] \\
Dec[\![(C, X.x)]\!] & = & SearchMod[\![(C, X, x, \Box, None)]\!] \\
Dec[\![(C, open\ X\ in\ M)]\!] & = & [(C[open\ X\ in\ \Box], M)] \\
Dec[\![(C, type\ x = \tau)]\!] & = & [(C[type\ x = \tau\ in\ \Box], M)] \\
Dec[\![(C, module\ X = & = & [(C[module\ X = \\
\quad struct\ D_1...D_n\ end\ in\ M)]\!] & & \quad\quad struct\ D_1...D_n\ end\ in\ \Box], M)]
\end{array}
$$

Figure 6.12: *Dec* for the module language

$$
\begin{array}{rcl}
Get : context * var * context* & & \\
\quad (context * term)\ option & \rightarrow & (context * term)\ list \\
Get[\![open\ X\ in\ C', v, C, p]\!] & = & let\ t = SearchMod[\![C, X, v, \Box, None]\!]\ in \\
 & & if\ t = None \\
 & & then\ Get[\![(C', v, C[open\ X\ in\ \Box], p)]\!] \\
 & & else\ Get[\![(C', v, C[open\ X\ in\ \Box], t)]\!] \\
Get[\![module\ X = struct & = & Get[\![(C', v, C[module\ X = struct \\
\quad D_1 ... D_n\ end\ in\ C', v, C, p]\!] & & \quad\quad D_1 ... D_n\ end\ in\ \Box)], p]\!] \\
Get[\![(type\ x = \tau\ in\ C', v, C, p)]\!] & = & Get[\![(C', v, C[type\ x = \tau\ in\ \Box), p]]\!]
\end{array}
$$

Figure 6.13: *Get* for the module language

$$
\begin{array}{rcl}
SearchMod' : definition\ list * var* & \rightarrow & (context * term)\ option \\
SearchMod'[\![([], v, C, p)]\!] & = & p \\
SearchMod'[\![((let\ x = M) :: r, v, C, p)]\!] & = & if\ v = v' \\
 & & then\ SearchMod'[\![(r, v, C[let\ x = M\ in\ \Box], Some(C, M))]\!] \\
 & & else\ SearchMod'[\![(r, v, C[let\ x = M\ in\ \Box], p)]\!] \\
SearchMod'[\![((type\ x = \tau) :: r, v, C, p)]\!] & = & SearchMod'[\![(r, v, C[type\ x = \tau\ in\ \Box], p)]\!]
\end{array}
$$

$$
\begin{array}{rcl}
SearchMod : context * var * var* & \rightarrow & (context * term)\ option \\
context * (context * term) option & & \\
SearchMod[\![(\Box, V, C, p)]\!] & = & p \\
SearchMod[\![(\lambda x.C', V, v, V, p)]\!] & = & SearchMod[\![(C', V, v, C[\lambda x.\Box], p)]\!] \\
SearchMod[\![(fix\ fv_1 ... v_n \rightarrow C', V, v, C, p)]\!] & = & SearchMod[\![(C', V, v, C[fix\ fv_1 ... v_n \rightarrow \Box], p)]\!] \\
SearchMod[\![(let\ x = M\ in\ C', V, v, C, p)]\!] & = & SearchMod[\![(C', V, v, C[let\ x = M\ in\ \Box], p)\\
SearchMod[\![(open\ X\ in\ C', V, v, C, p)]\!] & = & SearchMod[\![(C', V, v, C[open\ X\ in\ \Box], p)\\
SearchMod[\![(module\ X = struct\ D_1 ... D_n\ end\ in\ C', V, v, C, p)]\!] & = & \\
\quad if\ X = V & & \\
\quad then\ let\ t = SearchMod'[\![(D_1 ... D_n, v, C, None)]\!]\ in & & \\
\quad\quad (if\ t = None\ then & & \\
\quad\quad SearchMod[\![(C', V, v, C[module\ X = struct\ D_1 ... D_n\ end\ in\ P], None)]\!] & & \\
\quad\quad else\ SearchMod[\![(C', V, v, C[module\ X = struct\ D_1 ... D_n\ end\ in\ P], t)]\!]) & & \\
\quad else\ SearchMod[\![(C, V, v, C[module\ X = struct\ D_1 ... D_n\ end\ in\ P], p)]\!] & & \\
SearchMod[\![(type\ x = \tau\ in\ C', V, v, C, p) & = & SearchMod[\![(C', V, v, C[type\ x = \tau\ in\ P], p)]\!]
\end{array}
$$

Figure 6.14: The function *SearchMod* to collect definition of modules

Chapter 7

Implementation of a type debugger

In this chapter, we explain our implementation of a type debugger. First, we consider about the structure of type debugger. After that, we explain the details of our implementation.

7.1 The structure of a type debugger

In the previous chapters, we assume that a program to debug is ill-typed and all its subprograms are well-typed. This assumption is not always hold in ill-typed programs. We have to search such points from ill-typed program before debugging.

Therefore there are two phases when we debug an ill-typed program. The first phase is searching the starting point for debugging. The second phase is debugging an ill-typed program using user's intentions.

7.1.1 The searching phase

The starting point is often different from the root of the ill-typed program. For example, let us consider the following ill-typed program:

```
(2 - true) + 4
```

This program is ill-typed, because one of its sub-programs is ill-typed. The most general type tree of this program is as follows:

```
                          -:int -> int -> int  2:int  true:bool
+:int -> int -> int              (2 - true) ··· ill-typed          4:int
-------------------------------------------------------------------------
                       (2 - true) + 4 ··· ill-typed
```

In this case, the starting point of debugging is `(2 - true)`. Because all its sub-programs are well-typed, we can debug this program using programmer's intention and locate the source of the type error. To find the starting point, we only decompose the ill-typed sub-expression. In this case the root node is ill-typed, however it has an ill-typed sub-expression. We choose the ill-typed sub-expression `(2 - true)` and check whether all its sub-expressions are well-typed. Because all its subexpressions (`-`, `2` and `true`) are well-typed, the starting point is `(2 - true)`.

As just described, we can search the start point automatically. Because we do not have the waiting time for programmer's inputs in this process, the computational complexity is important for practice. We consider the computational complexity of this process is how many times this algorithm calls compiler's type inference. The computational complexity is linear to the height of the of the tree. In our implementation, this process takes 1 second for a 373 line's program.

We can consider this phase is a part of algorithmic debugging. Because programmers intend that the whole program is well-typed, all its sub-programs are well-typed in the user's intention. Therefore we can detect that the ill-typed original program and its ill-typed sub-programs are not programmer's intended types.

7.1.2 The debugging phase

In this section, we consider how to construct the most general tree for the debugging phase.

The naive construction The naive idea to construct the most general type tree is constructing the whole tree before debugging. Because this is very simple, we can implement easily. To construct the whole of the most general tree, we decompose the program to sub-programs and decompose them repeatedly. Although this naive construction works fast after the construction of the tree, to construct the whole tree takes time. This computational complexity is linear to the size of the program.

Additionally, we do not use the whole of the tree in the debugging phase. Although it depends on the programmer's answers, some inferences are needlessness.

Our construction To avoid the needless inference and reduce the construction time, we do not construct the whole of the tree. To ask programmers about the tree, we only need one judgement at the time. After receiving the programmer's answer, we choose the next expression depending on programmer's answer and infer its type. By repeating this process we can debug ill-typed programs without the most general type tree. This is the reason that our *Dec* is the one-step decomposition.

7.2 Our implementation for OCaml

We have implemented a type debugger for OCaml 3.12.1. To minimize the implementation efforts, we utilize the following components from OCaml as is:

- the abstract syntax tree for structures, expressions, and types (together with the lexer, the parser, and the pretty printer)
- the type inferencer `typing` (that accepts an expression and returns its type, both expressed using the above abstract syntax tree)
- the `is_expansive` function (that accepts an expression and returns a boolean to judge whether the given expression needs to be kept monomorphic or not)

By using exactly the same abstract syntax as OCaml, we can not only avoid reproducing the same abstract syntax but also utilize OCaml's own lexer, parser, and pretty printer. In addition to the type inferencer, we utilize the `is_expansive` function. Although OCaml has its own criteria for weak polymorphism [5], we can stay away from it by using OCaml's `is_expansive` function as is. Furthermore, this approach is robust to updates of OCaml: if the syntax and the interface of the two functions are the same, we can use the same debugger.

A slight complication is that OCaml treats a let expression without `in` differently from the one with `in`: the former is a structure, while the latter is an expression. We support both styles by splitting the context into two: the structure part and the expression part.

Another complication is the use of patterns in place of a variable declaration. For example, instead of `fun lst ->`, one can write `fun (first :: rest) ->`. Because patterns have type constraints, they may be the source of a type error. To make such an error detectable, we included patterns as the decomposition of the expression.

The rest of the language constructs are supported without requiring any special treatment. For each new construct, we define its decomposition and

show that it satisfies the decomposition property. Our type debugger supports all features of OCaml including weak polymorphism and modules.

To construct the most general type tree, we use the compiler's type inferencer many times. Although it appears that our type debugger incurs significant overhead, this is not the case, because we do not have to construct the whole tree beforehand. Instead, the most general type tree is constructed as we debug: after the root node is constructed, the rest of the tree can be constructed during the interaction with the programmer.

Chapter 8

Weighted type error slices

In Chapters 4 to 7, we described a practical debugger on the producer side. However, there are still problems on the user side. In Chapters 8 to 11, we describe an approach to satisfy a property "easy to debug" for the consumer side. In this chapter, we take a look at the problem of the previous type debugger and propose the idea of weighted type error slices to reduce the burden of type debugging on programmers.

8.1 A problem with our type debugger

Since the source of a type error depends on the programmer's intentions, our type debugger needs programmer's inputs. The type debugger developed in Chapters 4 to 7 has a problem that in some cases it requires programmers to make too many inputs. To illustrate this problem more clearly, let us consider the following example:

```
let f n lst = List.map (fun x -> x ^ n) lst in
 (f 2.0 [3.0; 4.0])
```

Because this program is ill-typed, OCaml returns the following error message.

```
Error: This expression has type float but an expression was
```

```
expected of type string
```

This error message indicates that the underlined part causes a type conflict. It says that the compiler expects `2.0` to be `string`, but it is `float`. There are many ways to resolve this type conflict. The correctness of the fix depends on the programmer's intention. If a programmer intends `f` to be a function that receives `string` as its first argument, s/he can understand this error message immediately and fix the program. Otherwise, s/he can understand little information that compiler expects `2.0` to be `string` for some reason. Moreover, s/he has to look for the source of the type error by hand. Of course, our type debugger can locate the source of type error by posing questions to the programmer. The following sequence is a set of questions and answers regarding one programmer's intentions.

```
Is your intended type of f string -> string list -> string list?
> no
Do you use n as string in the definition of f?
> no
Do you use lst as string list in the definition of f?
> no
Is your intended type of List.map 'a -> 'a list -> 'a list?
> yes
Is your intended type of (fun x -> x ^ n) string -> string ?
> no
Do you use x as string in x ^ n?
> no
Is your intended type of ^, string -> string -> string?
> no
The source of the type error is located at : ^
```

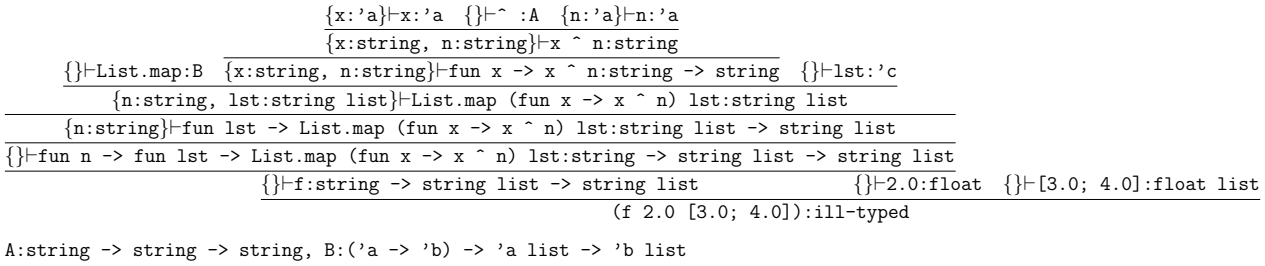


Figure 8.1: The most general type tree

By answering questions, we found that the source of this type error is in "`^`". These questions are made from the most general type tree of this program (see Figure 8.1).

When we debug a program with the most general type tree, the type debugger may ask questions about all areas of the tree. In this example program, because the tree is not so big, it is not so tedious to answer all these questions. However, if the original ill-typed program is large, its tree will also be large. In such a case, programmers would have to answer too many questions and may feel that debugging by hand is better. Thus, in order to implement a practical type debugger, we have to solve this problem.

8.2 Type error slices and their problem

To see what we can do to solve this problem, let us review the previous example. Why do type errors occur in programs? Put simply, type errors are caused by two conflicting types. In the previous example, we have two conflicting types in `2.0` and `^`. However, two conflicting types do not cause a type error only by themselves. Some parts of the program require that two conflicting types are of the same type. In the previous example, we passed `2.0` (its type is `float`) to the function `f` as the first argument. However, the type of `f` is forced to be `string -> string list -> string list` through

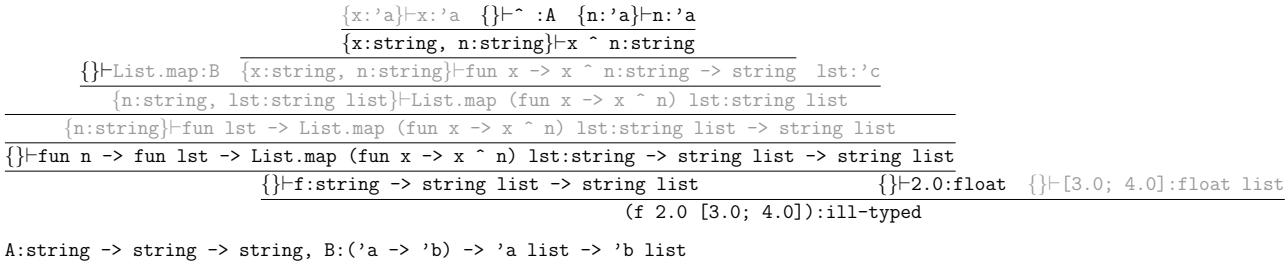


Figure 8.2: The reduced most general type tree

application of `^` (in `x ^ n`). This flow of unifying types contributes the type conflict between `2.0` and `^`. In this way, type errors are caused by two things: two conflicting types and a flow of unifying types that forces the two conflicting types to be the same. To put it the other way around, if these two things appear in the program, surely there is a type error. Furthermore, this means that the other parts did not cause the type error.

The following is the abstracted program of the previous example. It contains only two conflicting types and a flow that forces the two conflicting types to be the same.

```
let f n ... = ... (fun ... -> ... ^ n) ... in
(f 2.0 ...)
```

The parts abstracted by "`...`" do not contribute to this type conflict. In general, these abstracted programs are called type error slices. Since they include all parts related to a type error conflict, programmers can locate the source of the type error by debugging only the type error slices.

By using type error slices, we can narrow the area of the most general tree to search for the source of the type error. For the previous example, we can obtain a reduced most general tree by using type error slices. Figure 8.2 shows the image of the reduced tree. In this tree, the gray parts are not

contained in a type error slice and the type debugger avoid asking questions about them.

As described above, type error slices can reduce the burden on programmers. Additionally the advantage of type error slices is that they do not need user's inputs to obtain them. However, the problem still remains: if the original program is huge, its slice could be large.

8.3 The solution

To overcome this problem, we want to know which part is likely the source of the type error. When we look at a type error slice of an ill-typed program, we might conclude that each part has an equal chance of being the source of the type error. For example, in the previous slice, `2.0` and `^` look to be at the same level as far as the source of the type error goes. This conclusion, however, is proved false by the following observation. In the previous example, we can consider another slice:

```
let f ... lst = List.map (fun x -> x ^ ...) lst in
(f ... [3.0; 4.0])
```

The point here is that the slice includes "`^`" but not "`2.0`." To see this clearly, let us consider the following slice:

```
let f ... ... = ... (fun ... -> ... ^ ...) ... in
(f ... ...)
```

This slice is the intersection of the previous two slices. Because it is well-typed, it is not a type error slice. Although this slice may not include the source of the type error, it does include suspicious parts of the source. This observation about the intersection of several type error slices suggests that

each part of a type error slice has a different chance of being the source of the type error. If we know which expression should we see, we can reduce the burden on programmers.

8.4 Our approach

The intersections of type error slices are very intuitive and produce good results. However, to obtain such intersections, we have to obtain the type error slices first. The computational complexity needed to obtain a type error slice of an ill-typed program is $O(n^2)$, where n is the size of the program. Furthermore, to obtain all slices, we have to repeat this calculation $n!$ times. Because this cost is heavy for large programs, we need another way to obtain the likelihood of each expression being the source of a type error.

8.4.1 Brief overview

Let us consider the program `[1;2;true]`. Because two elements of this list are numbers and one element is a boolean, we think the minority `true` looks wrong. This is the key point of our approach. The problem is how we can obtain such information. The main ideas that we will exploit are abstraction of programs and majority vote.

First, we abstract one part of the program and infer its type. The result of doing this for the above example is shown in the table below.

one abstracted program	well-typed?
`[1; 2; ..]`	○
`[1; ..; true]`	×
`[..; 2; true]`	×

If an abstracted program is ill-typed, its sub-programs may contribute to the type error. For example, because `[1; ..; true]` is ill-typed, its sub-programs `1` and `true` contribute to the type error. Therefore, we count the

number of contributions of each abstracted subprogram. The following table is the result of doing so.

one abstracted program	contributions
`1`	1
`2`	1
`true`	2

This table show us the likelihood of each expression being the source of a type error. From this, we know that `true` has a higher probability of being the source of the type error than `1` or `2` has. This result is what we anticipated.

8.4.2 The points and contributions

Our approach has two main points. One is that we use the compiler's type inferencer. Most type error slicers use a tailor-made type inferencer. Although a tailor-made type inferencer has a certain flexibility, its results may not correspond to those of the compiler's type inferencer, and it has low scalability. In contrast, by using the compiler's inferencer we can make a type error slicer that has maintainability and high scalability. To obtain type error slices using a compiler's type inferencer, we abstract an ill-typed program and infer its type. Although this main idea of this approach is proposed by Schilling [19], we introduce some restriction to the obtained type error slices.

The other point is that we obtain *weighted* type error slices. The weights are the likelihoods of the expressions being the source of a type error and they help programmers during debugging.

Overview of Chapters 9, 10 and 11. In these chapters, we propose an approach to create weighted type error slices. In Chapter 9, we describe a type error slicer using a compiler's type inferencer. In Chapter 10, we modify this slicer by adding weights to it and make it work in Chapter 11.

Chapter 9

An embedded type error slicer

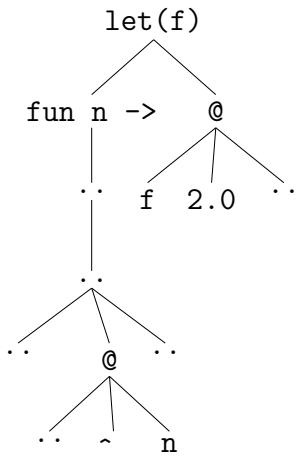
In this chapter, we introduce a type error slicer using a compiler's type inferencer. We show the syntax of the target language λ_{let} in Figure 9.1. It extends the simply typed lambda-calculus with let expressions and tuples. In Figure 9.1, l denotes the labels of each expression and each label is unique in the program. We also show the types of the target language. Because we use the compiler's type inferencer to obtain the type error slices, we do not treat the types directly. We only use the information whether the program is well-typed or ill-typed.

To obtain type error slices using the compiler's type inferencer, we infer the types of slices in order to know whether the slices are well-typed or ill-typed. If a slice is ill-typed, the slice includes the minimal type error slice that we want to obtain. However, slices are incomplete programs. Therefore, we can not infer their types directly. To infer the types of slices using compiler's type inferencer, we define the correspondence between slices and complete programs.

To set an idea of what type error slices are like, let us consider the type error slice in the previous chapter again:

```
let f n ... = ... (fun ... -> ... ^ n) ... in
(f 2.0 ...)
```

By this slice, we can obtain the following tree.



This tree shows that there are two patterns of the abstracted parts. One pattern abstracts a whole of sub-expression. (In the upper tree, they appear as `..` in the leaves.) The other abstracts the constructor itself containing subprograms. (In the upper tree, they appear as `..` in the nodes.) We show the syntax of the slices in Figure 9.2. We insert two pieces of syntax into the target syntax. One is a hole $\Box$ that abstracts whole sub-expressions. The other is a skeleton $S \mathbin{@\!\!\!/} S$ that abstracts the topmost expression itself but contains abstracted or unabstracted sub-expressions. These new syntaxes do not have labels.

By defining the syntax of the slices, we define the following properties of slices.

$(l : label)$	:=	(location)	
$(M : term)$	:=	c^l	(constant)
	|	x^l	(variable)
	|	$\lambda^l x.M$	(lambda abstraction)
	|	$@^l M\ M$	(application)
	|	$let^l\ x = M\ in\ M$	(let expression)
	|	$(M, .., M)^l$	(tuple)
$(\tau : type)$	:=	α	(type variable)
	|	$int, float, ...$	(type constant)
	|	$\tau \rightarrow \tau$	(function type)
	|	$\tau * .... * \tau$	(tuple type)

Figure 9.1: The syntax and types of let-polymorphic language λ_{let}

$(S : slice)$	:=	c^l	(constant)
	|	x^l	(variable)
	|	$\lambda^l x.S$	(lambda abstraction)
	|	$@^l S\ S$	(application)
	|	$let^l\ x = S\ in\ S$	(let expression)
	|	$(S, .., S)^l$	(tuple)
	|	$\Box$	(hole)
	|	$@\!\!\!/\ S\ ..\ S$	(skeleton)

Figure 9.2: The syntax of slices

Definition 2 (Type error slices) *We call a slice S as a type error slice, iff the inferred result of S by the compiler's type inferencer is a type error.*

Definition 3 (Inclusion relation of slices) *A slice S contains S', written $S \supset S'$, iff $L \supset L'$, i.e. the set of all labels L of S contains the set of L' of S'.*

Because there are fewer elements in a slice, it has fewer restrictions about types. Therefore, the following lemma holds [1].

Lemma 1 *If a slice S is well-typed, a slice S' such that $S \supset S'$ is well-typed as well.*

Definition 4 (Minimality of type error slices) *A type error slice S is minimal, iff all slices S' such that $S \supset S'$ are well-typed.*

The minimal type error slices do not contain parts that are superfluous to a type conflict. To obtain minimal type error slices, we abstract an ill-typed program as much as possible while maintaining its ill-typedness. In the following parts, we call the minimal type error slices the type error slices simply.

9.1 The algorithm

Let us consider $@\,(\lambda x.\lambda y.(@\,(@\,+\,x)y))\,true$ [2] as an example. First, let us focus on the topmost function application and abstract each subprogram or function application itself. The abstracted programs are $@\,\Box\,true$, $@\,(\lambda x.\lambda y.(@\,(@\,+\,x)\,y))\,\Box$ and $@\,(\lambda x.\lambda y.(@\,(@\,+\,x)\,y))\,true$. Because all these abstracted programs are well-typed, we know that we need this topmost application maintain ill-typedness. Next, let us focus on its sub-program $(\lambda x.\lambda y.(@\,(@\,+\,x)\,y))$. Although, this sub-program is well-typed itself, the problem is the type of the sub-program with its context. In this case, its context is `fun s ->` $@$ `s` $true$. We keep the whole program (the focused expression and its context) ill-typed and abstract the focused expression as

[1] This lemma does not hold in the language which has polymorphic recursions. To type polymorphic recursions, we need proper type annotations (restrictions).

[2] This program is a curried program of $(\lambda x.\lambda y.(x+y))true$

much as possible in its context. Therefore we abstract the focused program $(\lambda x.\lambda y.(@\,(@\,+\,x)\,y))$ with the context `fun s ->` $@$ `s` $true$. Because the lambda abstractions $\lambda x.\lambda y.M$ are binders, we insert them in the context and focus on the sub-program. In this case, by inserting $\lambda x.\lambda y.$ in the context, we obtain a new context `fun s ->` $@\,(\lambda x.\lambda y.\mathtt{s})\,true$.

One of the abstracted programs of $(@\,(@\,+\,x)\,y)$ is $(\cancel{@}\;(@\,+\,x)\,y)$. It is ill-typed under the context `fun s ->` $@\,(\lambda x.\lambda y.\mathtt{s})\,true$. Therefore, we make the obtained slice $(\cancel{@}\;(@\,+\,x)\,y)$ more abstract. In the second abstraction, we obtain $(\cancel{@}\;(@\,+\,x)\,\Box)$. In the third abstraction, because all abstracted slices of $(\cancel{@}\;(@\,+\,x)\,\Box)$ are well-typed, we know this function application itself is minimal. After that we focus on the subprograms of $(\cancel{@}\;(@\,+\,x)\,\Box)$. However, it turns out that its sub-programs are minimal. Consequently, we obtain a type error slice $@(\lambda x.\lambda y.(\cancel{@}\;(@\,+\,x)\,\Box))\,true$ as the final result.

In this way, we can obtain a type error slice by abstracting sub-programs from the root of the abstraction tree and inferring their types by using the compiler's type inferencer.

9.2 Program

In Figures 9.3 to 9.5, we show the program for obtaining type error slices. In this program, $infer$ is the compiler's type inferencer and $Type\_Error$ is an exception that the compiler's type inferencer raises when it finds a type conflict.

The function *abst_one* receives a slice and returns a list of slices in which each sub-program or the topmost constructor is abstracted. For example, in the function application case $@s_1s_2$, *abst_one* returns $\cancel{@}\;s_1s_2$, $@\Box s_2$ and $@s_1\Box$. If the abstracted slice is the same as the original (in case when a sub-program was already $\Box$), it does not return the same slice to avoid infinite

loops.

The function *check* receives a slice s and its context cxt and returns a type error slice s' such that $s \supset s'$. First, it calls the function *abst_one* and obtains a list of slices of s in which each sub-program or the topmost constructor is abstracted. It searches for an ill-typed slice in the list by using the compiler's type inferencer $infer$. If one or more ill-typed slices appear in the list, *check* returns the one that it found first. Otherwise, the slice s is minimal about the topmost constructor. In this case, it raises an exception Not_found and the exception will be caught by try expressions in *get_slice*.

The function *get_slice* is the main function for obtaining type error slices. It receives a slice s and its context cxt and returns a type error slice under the context. The following invariants hold in *get_slice*.
(1) A slice s is ill-typed under its context cxt, and
(2) the context cxt is well-typed itself.
These invariants are needed for the proof of completeness and minimality of type error slices.

The details of the function *get_slice* are as follows. In the case of a variable v and a constant c, v and c must be ill-typed under the context. (Otherwise, they are abstracted by the abstraction of the upper constructor.) In the case of a lambda abstraction $\lambda x.s$, we add $\lambda x.$ to the context and focus on its subprogram s. In the other syntax, *get_slice* abstracts them by using *check*. If the received slice is not minimal for the focused constructor, it calls *check* and abstracts the slice until the slice becomes minimal. Then, it focuses on the subprograms. For example, in the case of the function application $@\, s_1\, s_2$, if we assume that this function application itself is minimal, *get_slice* abstracts s_1 first and then abstracts s_2 under the new s_1'. It returns $@\, s_1'\, s_2'$ in which each subprogram is minimal for itself.

$$
\begin{array}{lcl}
abst\_one & : & slice \rightarrow slice\ list \\
abst\_one[\![s]\!] & = & ERROR\ when\ s = c^l, v^l, \lambda^l x.s \\
 & & (*\ \text{This is never called}\ *) \\
abst\_one[\![@^l s_1 s_2]\!] & = & [\circledcirc\ s_1 s_2; @\Box\ s_2; @^l s_1\ \Box] \setminus (@^l s_1\ s_2) \\
abst\_one[\![\circledcirc\ s_1..s_n]\!] & = & [\circledcirc\ s_1..\Box; ..; \circledcirc\ \Box..s_n] \setminus (\circledcirc\ s_1\ s_2) \\
abst\_one[\![let^l\ x = s_1\ in\ s_2]\!] & = & [let^l\ x = \Box\ in\ s_2; let^l\ x = s_1\ in\ \Box] \\
 & & \setminus (let^l\ x = s_1\ in\ s_2) \\
abst\_one[\![(s_1, .., s_n)^l]\!] & = & [(\Box, .., s_n)^l; ..; (s_1, .., \Box)^l; \circledcirc\ s_1..s_n] \\
 & & \setminus (s_1, .., s_n)^l
\end{array}
$$

Figure 9.3: The function *abst_one* to abstract the focused constructor or its sub-expression

$$
\begin{array}{lcl}
check & : & (slice * (slice \rightarrow slice)) \rightarrow slice \\
check(s, cxt) & = & let\ abst\_list\ = abst\_one[\![s]\!]\ in \\
 & & let\ rec\ loop\ lst = match\ lst\ with \\
 & & \mid [] \rightarrow raise\ Not\_found \\
 & & \mid fst :: rest \rightarrow try(infer\ (cxt\ fst); loop\ rest) \\
 & & \qquad with\ Type\_Error \rightarrow \\
 & & \qquad\qquad get\_slice[\![(fst, cxt)]\!]\ in \\
 & & loop\ abst\_list
\end{array}
$$

Figure 9.4: The function *check* to obtain a type error slice

When we call *get_slice*, the previous invariants have to hold. However, if the slice is $\Box$, the invariant does not hold because $\Box$ is well-typed. Therefore we wrap *get_slice* with *get_slice*$'$ to check whether the received slice is $\Box$. If the received slice is $\Box$, because it can not be abstracted any further, *get_slice*$'$ returns $\Box$.

$$
\begin{array}{lcl}
get\_slice & : & (slice * (slice \to slice)) \to slice \\
get\_slice[\![(\Box, cxt)]\!] & = & ERROR \\
 & & (* \text{ This case never happens } *) \\
get\_slice[\![(v^l, cxt)]\!] & = & v^l \\
get\_slice[\![(c^l, cxt)]\!] & = & c^l \\
get\_slice[\![(\lambda^l x.s, cxt)]\!] & = & \\
 & & \lambda^l x.(get\_slice[\![s, (fun\ y \to cxt(\lambda^l x.y))]\!]) \\
get\_slice[\![(@^l s_1\ s_2, cxt)]\!] & = & \\
 & & try(check(@^l s_1\ s_2, cxt))\ with\ Not\_found \to \\
 & & \quad let\ s'_1 = get\_slice'[\![(s_1, (fun\ x \to cxt(@^l x\ s_2)))]\!]\ in \\
 & & \quad let\ s'_2 = get\_slice'[\![(s_2, (fun\ x \to cxt(@^l s'_1\ x)))]\!]\ in \\
 & & \quad (@^l s'_1\ s'_2) \\
get\_slice[\![(\circledcirc s_1..s_n, cxt)]\!] & = & \\
 & & try(check(\circledcirc\ s_1..s_n, cxt))\ with\ Not\_found \to \\
 & & \quad let\ s'_1 = get\_slice'[\![(s_1, (fun\ x \to cxt(\circledcirc\ x..s_n)))]\!]\ in\ .. \\
 & & \quad let\ s'_n = get\_slice'[\![(s_n, (fun\ x \to cxt(\circledcirc\ s'_1..x)))]\!]\ in \\
 & & \quad (\circledcirc\ s'_1\ s'_2) \\
get\_slice[\![(let^l\ x = s_1\ in\ s_2, cxt)]\!] & = & \\
 & & try(check(let^l\ x = s_1\ in\ s_2, cxt))\ with\ Not\_found \to \\
 & & \quad let\ s'_1 = get\_slice'[\![(s_1, (fun\ y \to cxt(let^l\ x = y\ in\ s_2)))]\!]\ in \\
 & & \quad let\ s'_2 = get\_slice'[\![(s_2, (fun\ y \to cxt(let^l\ x = s_1\ in\ y)))]\!]\ in \\
 & & \quad (let^l\ x = s'_1\ in\ s'_2) \\
get\_slice[\![((s_1, .., s_n)^l, cxt)]\!] & = & \\
 & & try(check((s_1, .., s_n)^l, cxt))\ with\ Not\_found \to \\
 & & \quad let\ s'_1 = get\_slice'[\![(s_1, (fun\ y \to cxt(y, ..., s_n)^l))]\!]\ in\ .. \\
 & & \quad let\ s'_n = get\_slice'[\![(s_n, cxt(fun\ y \to cxt(s'_1, ..., y)^l))]\!]\ in \\
 & & \quad (s'_1, .., s'_n)^l \\
 & & \\
get\_slice' & : & (slice * (slice \to slice)) \to slice \\
get\_slice'[\![(s, cxt)]\!] & = & if\ s = \Box then\ \Box \\
 & & else\ get\_slice[\![(s, cxt)]\!]
\end{array}
$$

Figure 9.5: Type error slicer *get_slice*

Chapter 10

A weighted type error slicer

In the previous chapter, we introduced a type error slicer using compiler's type inferencer. In this chapter, we extend it so that it has weights. The weights are the probabilities of each subprograms to be the source of the type error. The target language is the same as in the previous chapter (see Figure 9.1). The syntax for slices is also the same (see Figure 9.2).

10.1 The flow of algorithm

To see how we can add weights to the slicer, let us consider an example program $(\lambda^1 f.((@^3(@^4 *\ f)\ 1), (@^5 f\ 2), (@^6(@^7 +\ f)\ 3))^2)$. [1] (We omit the labels for variables and constants.) The initial context is the empty context $\lambda x.x$. First, let us focus on the topmost lambda abstraction. We add $\lambda^1 f.$ to the context and move to its sub-program $((@^3(@^4 *\ f)\ 1), (@^5 f\ 2), (@^6(@^7 + f)\ 3))^2$. Here, we abstract each subprogram or the constructor itself. The abstracted slices of $((@^3(@^4 *\ f)\ 1), (@^5 f\ 2), (@^6(@^7 +\ f)\ 3))^2$ are the followings:

- $(\Box, (@^5 f\ 2), (@^6(@^7 +\ f)\ 3))^2$

[1] This program is a curried program of $(\lambda f.(f * 1, f\ 2, f + 3))$.

- $((@^3(@^4 * \ f)\ 1), \square, (@^6(@^7 + \ f)\ 3))^2$
- $((@^3(@^4 * \ f)\ 1), (@^5 f\ 2), \square)^2$

In these slices, the first and third slices are ill-typed. In an ill-typed program, its sub-programs contribute to the type error. Therefore, we count the number of times that each sub-program is contained within ill-typed program. The counts for this example are in the table below.

a subprogram	numbers of contributions
$(@^3(@^4 * \ f)\ 1)$	1
$(@^5 f\ 2)$	2
$(@^6(@^7 + \ f)\ 3)$	1

From this table, we know that the sub-program labeled 5 contributes more than the other subprograms do.

To obtain the type error slices, we choose one type error slice from the abstracted slices and make it more abstract. If we select the first type error slice, we obtain a minimal type error slice $(\square, (@^5 f\ 2), (@^6(@^7 + \ f)\ \square))$ about this constructor. Although this slice is the same as the one in the previous chapter, it has information about the weight. The node labeled 5 is heavier than the node labeled 6.

By majority vote between sub-programs, we can find a sub-program that is most likely to be the source of the type error. Thanks to the weighted type error slice, the type debugger can ask about the suspicious expressions that it finds. In the previous example, the type error slicer produces a weighted slice $\lambda^1 f.(\square, (@^5 f\ 2), (@^6(@^7 + \ f)\ \square))$. Using this result, the type debugger know that it should ask about $(@^5 f\ 2)$ first, because it has the heaviest weight among the sub-programs. If the programmer answers that the judgement of $(@^5 f\ 2)$ is correct, the type debugger will then ask about $(@^6(@^7 + \ f)\ \square)$.

$$
\begin{array}{lcl}
inc\_one & : & slice \rightarrow unit \\
inc\_one(\square) & = & () \\
inc\_one(s^l) & = & inc\_weight\ l \\
inc\_slice & : & slice \rightarrow unit \\
inc\_slice(@^l s_1\ s_2) & = & inc\_weight\ l; inc\_one\ s_1; inc\_one\ s_2 \\
inc\_slice(@\ s_1..s_2) & = & inc\_one\ s_1; ..; inc\_one\ s_2 \\
inc\_slice(let^l\ x = s_1\ in\ s_2) & = & inc\_weight\ l; inc\_one\ s_1; inc\_one\ s_2 \\
inc\_slice((s_1, .., s_n)^l) & = & inc\_weight\ l; inc\_one\ s_1; ..; inc\_one\ s_n \\
inc\_slice(\_) & = & ERROR
\end{array}
$$

$$
\begin{array}{lcl}
check & : & (slice * (slice \rightarrow slice)) \rightarrow slice \\
check((s, cxt), f) & = & let\ abst\_list\ = abst\_one[\![s]\!]\ in \\
 & & let\ rec\ loop\ lst = match\ lst\ with \\
 & & \mid [] \rightarrow [] \\
 & & \mid fst :: rest \rightarrow try(infer\ (cxt\ fst); loop\ rest) \\
 & & \qquad\qquad with\ Type\_Error \rightarrow \\
 & & \qquad\qquad\quad inc\_slice\ fst; \\
 & & \qquad\qquad\quad fst :: (loop\ rest)\ in \\
 & & let\ illtyped\_slices = loop\ abst\_list\ in \\
 & & match\ illtyped\_slices\ with \\
 & & \mid [] \rightarrow raise\ Not\_found \\
 & & \mid \_ \rightarrow get\_slice[\![(choice\ illtyped\_slices, cxt)]\!]
\end{array}
$$

Figure 10.1: Weighted type error slicer

Because this weighted slice is also a type error slice, it never asks about $(@^3(@^4 *\ f)\ 1)$.

10.2 Program

Figure 10.1 lists the program to produce weighted type error slices. We use a hash table to save the weights of each sub-program. The key of the array is the labels of each sub-program and its value is the weights of each sub-

program. The function *inc_weight* receives a label l and increase its weight. The function *choice* selects an element from the received list.

We can extend the previous chapter's program to a weighted type error slicer simply by changing the function *check*. We show *check* in Figure 10.1. The other function *get_slice* is the same as in the previous chapter's program.

The function *inc_one* receives a slice (whose label is l) and increases the weight of l. When the received slice is $\Box$, *inc_one* does nothing because $\Box$ does not contribute to the type error. This is why $\Box$ has no labels. The function *inc_slice* receives a slice and increases the weights of its sub-programs. For example, if *inc_slice* receives a function application $@^l s_1\ s_2$, it increases the weight of l and its sub-programs. It calls *inc_one* to increase the weights of the sub-programs. If it receives a skeleton expression ($\circledcirc\ S_1\, S_2$), it only increases the weights of its sub-programs because skeletons do not contribute to a type error.

The function *check* is similar to *check* in the previous chapter, but it works harder than the previous one. In the previous *check*, *loop* terminates immediately when it finds an ill-typed slice and calls the function *get_slice*. In this program, *loop* works on all elements of an abstracted list *abst_list* obtained by *abst_one*. During the loop, the program collects all type error slices from all elements. The reason for this is that programmers may have intuitions about *abst_list*. Therefore, after the loop, we increase the weights according to the collected type error slices. Finally, we choose one of the slices and make it more abstract by calling *get_slice*. If there are no type error slices in the abstracted list, the behavior is the same as described in the previous chapter.

Chapter 11

An improved weighted type error slicer

The previous chapter described weighted type error slicer that compares the sub-programs of an expression and the constructor itself. Although it produces a bit better type error slices than the standard type error slicers, it does not likely match a programmer's intuition. Because it depends on the structure of programs, it sometimes does not produce the expected result. To get a handle on this problem, let us consider the previous example $(\lambda^1 f.((@^3(@^4 *\ f)\ 1), (@^5 f\ 2), (@^6(@^7 +\ \ f)\ 3))^2)$. Here, the sub-program labeled 5 is more suspicious than the other sub-programs. This can be inferred by comparing the three sub-programs; two of them indicate that$f$ is a number. However, the majority vote sometimes does not work well. If we change the structure of this program to $(\lambda^1 f.((@^3(@^4 *\ f)\ 1), ((@^5 f\ 2), (@^6(@^7 +\ \ f)\ 3))^{2'})^2)^1$, the previous type error slicer does not work well. In this chapter, we improve the type error slicer so that it will work well on this sort of example.

[1]The original program has a tuple of three elements, and the modified program has a tuple of two elements in which one element is itself a tuple of two elements.

11.1 The flow of the algorithm

Let us consider an example program $(\lambda^1 f.((@^3(@^4 * f)\ 1), ((@^5 f\ 2), (@^6(@^7 + f)\ 3))^{2'})^2)$. The outer most expression is a function abstraction; we focus on its subprogram $((@^3(@^4 * f)\ 1), ((@^5 f\ 2), (@^6(@^7 +\ f)\ 3))^{2'})^2$ and add $\lambda^1 f.$ to the context. We abstract this focused expression and obtain the following slices.

- $(\square, ((@^5 f\ 2), (@^6(@^7 +\ f)\ 3))^{2'})^2$
- $((@^3(@^4 *\ f)\ 1), \square)^2$
- $\cancel{@}\ (@^3(@^4 *\ f)\ 1)\ ((@^5 f\ 2), (@^6(@^7 +\ f)\ 3))^{2'}$

Among these slices, the first and third slices are ill-typed. If we select the first slice or third slice and abstract it, we obtain $\cancel{@}\ \square((@^5 f\ 2), (@^6(@^7 +\ f)\ 3))^{2'}$. To abstract its sub-programs, we focus on $((@^5 f\ 2), (@^6(@^7 +\ f)\ 3))^{2'}$ with the context $(\texttt{fun s ->}\lambda^1 f.(\square, \texttt{s}))$. However, this context does not have the information that f is a number in the node labeled 3 of the original program. In the next abstraction, we obtain the following abstracted programs:

- $(\square, (@^6(@^7 +\ f)\ 3))^{2'}$
- $((@^5 f\ 2), \square)^{2'}$
- $\cancel{@}\ (@^5 f\ 2)(@^6(@^7 +\ f)\ 3)$

with the context $(\texttt{fun s ->}\lambda^1 f.(\square, \texttt{s}))$. In the upper programs, because the third program is ill-typed and the others are well-typed, we can obtain the following weights of each sub-program from these slices:

a sub-program	numbers of contributions
$(@^5 f\ 2)$	1
$(@^6(@^7 +\ f)\ 3)$	1

This table shows that their weights are the same. This behavior is different from our intuition. Because f is a number in two places of the original program (the part labeled 4 and the part labeled 7) and a function in one place (the part labeled 5), the part labeled 5 looks to be the source of the type error.

The problem is the lost information in the context. To preserve these information, we use two contexts. One context is to obtain type error slices, the same as in the previous chapter's context. The other context is to obtain the weights, that is, the non-abstracted context. To explain these two contexts, let us consider the previous example. When we focus on $((@^5 f\ 2), (@^6(@^7 +\ f)\ 3))^{2'}$, the context for obtaining the type error slice is (`fun s ->`$\lambda^1 f.$~~`@`~~ $\Box$`s`). Another context for obtaining the weights is (`fun s ->`$\lambda^1 f.((@^3(@^4 * f)\ 1), s)^{2'}$). The latter context includes more information than the former context. Therefore, we can detect the mismatch between the focused program and the outer context. These two contexts make it possible to obtain the following weights:

a sub-program	numbers of contributions
$(@^5 f\ 2)$	2
$(@^6(@^7 +\ f)\ 3)$	1

These weights correspond to our intuition. By using the unnecessary parts for the type error slices, we become aware of the programmer's intention. The previous weighted type error slicer compares only the sub-programs of one node; on the other hand, this improved weighted type error slicer compares sub-programs with the outer context. The previous one can perform horizontal comparisons in the tree, whereas the current one can perform horizontal and vertical comparisons.

```
check : (slice * (slice → slice) * (slice * (slice → slice))) → slice
check(s, cxt, (p, wrap)) = let abst_list = abst_one⟦s⟧ in
                           let rec loop lst = match lst with
                            | [] → []
                            | fst :: rest →
                              try(infer (wrap fst); loop rest)
                              with Type_Error →
                                inc_slice fst;
                                try(infer(cxt fst); loop rest)
                                with Type_Error → fst :: (loop rest) in
                           let illtyped_slices = loop abst_list in
                           match illtyped_slices with
                            | [] → raise Not_found
                            | _ → get_slice⟦(choice illtyped_slices,
                                                cxt, (p, wrap))⟧
```

Figure 11.1: *check* for an improved weighted type error slicer

11.2 The program

Figure 11.1 and 11.2 show the program of an improved weighted type error slicer. In the program, *cxt* is the context for obtaining the type error slices and *wrap* is the context for obtaining the weights of the sub-programs.

In the Figure 11.1, the function *check* increases the weights if the slices with *wrap* are ill-typed. This is the main change from the previous type error slicer. The other changes are in *wrap*. During abstraction of a focused program, we use the same *wrap*. Once the focused program becomes minimal, we focus on its sub-programs and change *wrap*. The value of *wrap* is updated in the function *get_slice*. In the case of lambda abstraction, we add $\lambda^l x.$ to *wrap* the same way as *cxt*. In the case of function application, *wrap* needs the parts that are not included in the type error slices. To set

$$
\begin{array}{l}
make\_wrap : ((slice \to slice) * slice * (slice \to slice)) \\
\qquad\qquad \to (slice * (slice \to slice)) \\
make\_wrap(wrap, p, cxt) \;=\; try(infer\ (wrap\ \Box); (p, wrap)) \\
\qquad\qquad\qquad\qquad\quad with\ Type\_Error \to (p, cxt)
\end{array}
$$

$$
\begin{array}{l}
get\_slice : (slice * (slice \to slice) * (slice \to slice)) \to slice \\
get\_slice[\![(v^l, cxt, (p, wrap))]\!] \;=\; v^l \\
get\_slice[\![(c^l, cxt, (p, wrap))]\!] \;=\; c^l \\
get\_slice[\![(\lambda^l x.s, cxt, (p, wrap))]\!] \;= \\
\quad \lambda^l x.(get\_slice[\![(s, (fun\ y \to cxt(\lambda^l x.y)), (s, (fun\ y \to wrap(\lambda^l x.y))))]\!]) \\
get\_slice[\![(@^l s_1\ s_2, cxt, (p, wrap))]\!] \;= \\
\quad let\ (@^l p_1\ p_2) = p\ in \\
\quad try(check(@^l s_1\ s_2, cxt, (p, wrap))) \\
\quad with\ Not\_found \to \\
\quad\ let\ cxt_1 = (fun\ x \to cxt(@^l x\ s_2))\ in \\
\quad\ let\ s_1' = get\_slice'[\![(s_1, cxt_1, \\
\qquad\qquad\qquad make\_wrap((fun\ x \to wrap(@^l\ x\ p_2)), s_1, cxt_1))]\!]\ in \\
\quad\ let\ cxt_2 = (fun\ x \to cxt(@^l s_1'\ x))\ in \\
\quad\ let\ s_2' = get\_slice'[\![(s_2, cxt_2 \\
\qquad\qquad\qquad make\_wrap((fun\ x \to wrap(@^l\ p_1\ x)), s_1, cxt_2))]\!]\ in \\
\quad (@^l s_1'\ s_2') \\
get\_slice[\![(s_1, .., s_n)^l]\!] \;=\; \text{Omitted}
\end{array}
$$

$$
\begin{array}{l}
get\_slice' : (slice * (slice \to slice) * (slice \to slice)) \to slice \\
get\_slice'[\![(s, cxt, (p, wrap))]\!] \;=\; if\ s = \Box\ then\ \Box \\
\qquad\qquad\qquad\qquad\qquad\quad else\ get\_slice[\![(s, cxt, (p, wrap))]\!]
\end{array}
$$

Figure 11.2: An improved weighted type error slicer

such parts, we have to use non-abstracted expression for *wrap*. However, the focused program $@^l s_1\ s_2$ may already be abstracted by the function calls of *get_slice*. Therefore, we introduce a new argument p to *get_slice* in order to preserve the original expression (non-abstracted one) and use it to update *wrap*. Because the new *wrap* has to be well-typed itself, we check

whether it is well-typed or not by using *make_wrap*. If it is ill-typed, we use *cxt*, which is always well-typed. In this case, *wrap* lost information; however, new information will be pushed onto non-abstracted subprograms in the next recursions.

Chapter 12

Related work

In this chapter, we compare our work with related work. We classify related work according to type, such as typing algorithms, type debugging systems, and type error slicing, etc.

12.1 Typing algorithms

The typical approach to improving type error messages is to design a new type inference algorithm. Wand [29] keeps track of the history how type variables are instantiated and shows the conflicting history when a type error arises. Lee and Yi [12] present the algorithm M that finds conflict of types earlier than the algorithm W and thus reports a narrower expression as an error. Heeren and Hage [8] use a constraint-based type inference for improving type error messages.

Neubauer and Thiemann [17] introduce a type system using sum types. The sum type allows the multiple types during an inference of types. The correct type of the sum type is chosen by the types of the surrounding programs. Our idea for weighted type error slice is inspired by their approach.

The type error messages by existing compilers are improved by these

approaches. Thanks to the improvements, if the source of the type error is near the point located by compilers, we can often find the source of the type error by hand. Although these improved type error messages are useful for programmers, it is in general not possible to identify the source of type errors by a single error message.

12.2 Type debugging systems

To locate the source of type errors, Chitil [2] uses compositional type inference and constructs an interactive type debugger for a subset of Haskell. Based on his work, we designed a type debugger for OCaml using the compiler's own type inferencer rather than a tailor-made type inferencer. The use of the compiler's type inferencer enables us to build a type debugger for a larger language easily. Stuckey, Sulzmann, and Wazny [23] find the source of type errors using type inference via CHR solving. They implement a type debugger called Chameleon, which can explain why an inferenced type is derived by searching. Tailor-made type inference is used for this purpose.

12.3 Type error slicing

Haack and Wells [6] use slicing with respect to types to narrow the possibly erroneous parts of programs. By extracting the slice related to type errors, they help the programmer to identify the source of type errors. The advantage of this approach is that the process is automatic and the programmer does not have to answer questions.

Schilling [19] obtains slices using the compiler's type inferencer. To obtain type error slice, he abstracts an ill-typed program and infers its type by compiler's inferencer. Although we choose the same approach there are

some difference between them. Schilling starts from the empty program and instantiate it by parts of ill-typed programs. On the other hand, we start from the ill-typed programs and abstract them. Thanks to the decremental approach, we extend type error slicing with the weights.

Our work is an extension of these works for type error slicing by the weights. The likelihood of each expression being the source of the type error makes type debugging easier.

12.4 Type error correction

Lerner et al. [13] propose automatic type-error correction. They replace the erroneous part with various syntactically correct similar expressions, and see if they type check. If they do, they are displayed as the candidates for fixing the type error. Since the system automatically shows us possible fixes without intervention, the system is useful if the programmer's intended fix is shown. Unfortunately, it does not always produce the intended program.

12.5 Visualization of types

As visualizing tools of types, Simon, Chitil, and Huch [22] show TypeView that allows programmers to browse through the source code and to query the types of each expression. McAdam [14] displays types as graphs and extracts various facts from them that are useful for debugging. Our previous Emacs interface [24] is inspired by these works, and we will continue to build such interface.

Chapter 13

Conclusion

In this thesis, our aim is to establish the approaches to achieve *practicable type debugging*. First, we proposed a manifesto of practicable type debugging. The manifesto consists of two categories, the producer side and the consumer side.

By satisfying the properties of the producer side, we can implement the accurate type debuggers easily. To satisfy these properties, we propose a type debugger using compiler's type inferencers. The key observation is that we only need the most general type tree with the decomposition property; such a tree can be constructed using the compiler's type inferencer. The decomposition property guided the design of our type debugger: we maintained contexts so that the property is satisfied all the time. We have fleshed out our thesis that it is possible and also practical to write a type debugger by piggy-backing on the built-in type inferencer of an existing compiler.

By satisfying the properties of the consumer side, type debuggers become useful and user-friendly tools for programmers. To satisfy one of the properties, we propose a weighted type error slicer. We obtained a weighted type error slices by observing each node to reduce the burden on programmers. Although the conventional type error slicer may produce large type error

slices, our approach enables to see some parts of the type error slices. The piggy-backing on the built-in type inferencer of an existing compiler is good idea in this topic too.

In this thesis, our main observation is the following:

- To fit existing implementation, using their features is a better idea than reproducing new implementation.

Of course, there are some situation where this observation does not fit. For example, it is impossible to reuse the existing implementation. However, we believe that the meaning of "the good programming language" often includes their good tools for programming. To implement good tools for the languages, their compiler should be easy to reuse their functions. Additionaly, the existing programming languages are becoming complex. Therefore the reuse of their functions will become essential to implement good tools for them.

The another observation is the following:

- For debugging, it is important to extract the information from the ill-typed programs.

This looks natural, however most compilers extract the minimum information from the ill-typed programs. In our weighted type error slicer, we could extract more information from the ill-typed program than standard compilers.

We plan to continue the present line of work as follows. First, we want to explore how far the idea presented in this thesis scales. In particular, we are interested in supporting type classes [7] in Haskell and GADTs introduced in OCaml 4.0. We will investigate how we can define decomposition of a

program with type classes or GADTs and see if it satisfies the property (Section 5.2). Second, we want to implement a weighted type error slicer for our type debugger. Although almost all of the syntax is simple extension of our proposed weighted type error slicer, we have to treat patterns as special syntax. Third, we want to perform thorough user tests. We have built an Emacs interface based on our previous work [24] and the type debugger is in use in several courses in our university. From the user tests, we plan to obtain various feedback including usefulness and how to effectively show the type information to novices. We also plan to obtain feedback from the user tests with skilled programmers.

Acknowledgement About our type debugger, we present papers and talks at PPL2012 [25], computer software (a Japanese journal) [26] and the post-proceedings of IFL2012 [27]. About weighted type error slicer, we present a paper and a talk at PPL2013[28].

Bibliography

[1] Barras, Bruno, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1997.

[2] Chitil, O. "Compositional Explanation of Types and Algorithmic Debugging of Type Errors," *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming (ICFP'01)*, pp. 193–204 (2001).

[3] Cousineau, Guy and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.

[4] Damas, Luis and Robin Milner. "Principal type schemes for functional programs". *In ACM Symposium on Principles of Programming Languages (POPL)*, Albuquerque, New Mexico, pp. 207–212, 1982.

[5] Garrigue, J. "Relaxing the value restriction," In Y. Kameyama and P. J. Stuckey, editors, *Functional and Logic Programming (LNCS 2998)*, pp. 196–213 (April 2004).

[6] Haack, C., J. B. Wells. "Type Error Slicing in Implicitly Typed Higher-Order Languages," *Science of Computer Programming - Special issue on 12th European symposium on programming (ESOP'03)*, Volume 50 Issue 1-3 (2004).

[7] Hall, C., K. Hammond, S. P. Jones and P. Wadler. "Type classes in Haskell," *ACM Transactions on Programming Languages and Systems (TOPLAS)* Volume 18 Issue 2 (1996).

[8] Heeren, B., J. Hage. "Parametric Type Inferencing for Helium," Technical Report UU-CS-2002-035, Utrecht University, 2002.

[9] INRIA, http://caml.inria.fr, http://www.ocaml.org

[10] Jim, Trevor. "Rank-2 type systems and recursive definitions," Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, Laboratory for Computer Science, November 1995.

[11] Jim, Trevor. "What are principal typings and what are they good for?," In ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida, pp. 42–53. ACM, 1996.

[12] Lee, O., K. Yi. "Proofs about a Folklore let-polymorphic Type Inference Algorithm," *ACM Transactions on Programming Languages and Systems*, pp. 707-723 (1998).

[13] Lerner, B. S., M. Flower, D. Grossman, C. Chambers. "Searching for Type-Error Messages," *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*, pp. 425–434 (2007).

[14] McAdam, B. J. "Generalising techniques for type debugging," In Trends in Functional Programming, chapter 6. Intellect, (2000).

[15] Milner, H. "A Theory of Type Polymorphism in Programming," Journal of Computer and System Science (JCSS) 17, pp. 348–374

[16] Milner, R., M. Tofte, R. Harper and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[17] Neubauer, M.m and P. Thiemann. "Discriminative Sum Types Locate the Source of Type Errors," *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming (ICFP'03)* pp. 15–26, 2003.

[18] Nilsson, H. *Declarative Debugging for Lazy Functional Languages*, PhD thesis, Linköping, Sweden (1998).

[19] Schilling, T. "Constraint Free Type Error Slicing," *Proceedings of the 12th international conference on Trends in Functional Programming (TFP'11)*, pp. 1–16 (2012).

[20] Shapiro, E. Y. *Algorithmic Program Debugging*, MIT Press (1983).

[21] Silva, J., and O. Chitil. "Combining Algorithmic Debugging and Program Slicing," *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming (PPDP'06)*, pp. 157–166 (2006).

[22] Simon, A., O. Chitil and F. Huch. "Typeview: A tool for understanding type errors," *Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages,* PP. 63–69 (2000).

[23] Stuckey, P. J., M. Sulzmann, J. Wazny, "Interactive type debugging in Haskell," *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell (Haskell'03)*, pp. 72–83 (2003).

[24] Tsushima, K., and K. Asai. "Report on an OCaml type debugger," *ACM SIGPLAN Workshop on ML*, 3 pages, (2011).

[25] Tsushima, K., and K. Asai. "A Type Debugging Approach using Compiler's Type inferencer," *14th Programming and Programming Language workshop (PPL2012)*

[26] Tsushima, K. and K. Asai. A Type Debugging Approach using Compiler's Type Inferencer (In Japanese) *Computer Software* (Japanese Journal) vol.30, no.1, pp.180–186, 2013.

[27] Tsushima, K., and Asai, K.. "An Embedded Type Debugger," *Proceedings of the 24th International Workshop on Implementation of Functional Languages (IFL'12)*, to appear in LNCS, Springer (2013).

[28] Tsushima, K., and K. Asai. "A Weighted Type Error Slicing," (in Japanese) *15th Programming and Programming Language workshop (PPL2013)*

[29] Wand, M. "Finding the Source of Type Errors," *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'86)*, pp. 38–43 (1986).

[30] Yang, J., G. Michaelson., P. Trinder., and J. B. Wells. "Improved Type Error Reporting," *International Workshop on Implementation of Functional Languages*, pp. 71–86 (2000).

[31] Pierce, Benjamin C., *Types and Programming Languages*, The MIT Press

[32] http://wiki.portal.chalmers.se/agda/pmwiki.php

[33] https://code.google.com/p/epigram/