ChameleonIDE：通过交互式可视化与探索梳理类型错误

Shuai Fu莫纳什大学信息技术学院克莱顿，澳大利亚
yshuai.fu@monash.edu

Tim Dwyer莫纳什大学信息技术学院克莱顿，澳大利亚
ytim.dwyer@monash.edu

Peter J.Stuckey莫纳什大学信息技术学院克莱顿，澳大利亚
peter.stuckey@monash.edu

Jackson Wain莫纳什大学信息技术学院克莱顿，澳大利亚
ORCID：0000-0003-2006-3538

Jesse Linossier莫纳什大学信息技术学院克莱顿，澳大利亚
ORCID：0000-0001-6782-7019

摘要——**动态类型语言在教育和软件行业中广受欢迎。虽然入门门槛较低，但这类语言容易出现运行时类型错误，并且长期来看会影响代码质量和可维护性。静态类型语言在这些方面表现出优势，但可学习性和易用性不足。尤其是，修复类型错误对新手用户和专家都构成挑战。此外，编译器类型错误消息以静态方式呈现，且偏向于程序代码中错误首次出现的位置。为帮助用户解决此类类型错误，我们推出了 ChameleonIDE，这是一款类型调试工具，能够以不偏不倚的方式向用户呈现类型错误，让用户探索错误可能出现位置的完整上下文。程序员可以通过交互方式，根据自己的意图验证推理步骤。通过涉及实际程序员的三项研究，我们发现，与传统的基于文本的错误消息相比，ChameleonIDE 更能有效地帮助用户修复类型错误。在难度较高的任务中，这种差异尤为显著。此外，积极使用 ChameleonIDE 交互功能的程序员，比被动阅读类型错误输出的程序员更高效地修复类型错误。**

索引词—类型、类型错误、调试、可视化、探索

近几十年来，JavaScript 和 Python 等动态类型编程语言日益流行 [1]。这些语言的入门门槛较低，对编程初学者尤其如此：它们无需类型声明，变量类型或对象结构可以动态修改，函数还可以借助特设多态和运行时反射处理动态输入。然而，研究表明，动态类型语言会对开发生产力 [2]、代码可用性 [3]，和代码质量 [4]–[6] 产生负面影响。研究还发现，这类语言往往会产生易出错代码 [7]–[9]，并且需要程序员严格自律，才能避免陷阱 [7]。基于这些原因，近年来许多现代动态类型语言已将静态类型注解引入核心语言特性中（例如*TypeScript* [10] 和 *mypy* [11]）。

函数式编程语言长期以来都以严谨的类型系统和富有表现力的类型级特性著称。类型推断和代数类型等技术，几十年来一直是 ML 和 Haskell 等函数式语言中的标准实践，近来还广泛应用于 Rust 和 TypeScript 等多范式语言。Haskell 中引入的各种类型系统创新，在数年乃至数十年后进入主流语言，因此许多人将 Haskell 视为“类型系统实验室” [12]。作为泛型编程的一种实现，类型类于 1988 [12], 年引入 Haskell，如今已见于 C# [13], 、Java [14], 和 TypeScript [15] 等大多数流行语言中。

使用静态类型语言编程的一项关键挑战是，有时很难解决类型错误 [16], [17]。具体而言，类型错误可能指向并非其根本原因的位置，可能用晦涩的语言描述错误，也可能给出具有误导性的修复建议 [18]。

本文介绍了 ChameleonIDE，这是一款面向 Haskell 的交互式类型调试工具。它能够可视化类型错误的相关上下文：错误发生或可能发生的位置，以及导致错误的代码部分。此外，ChameleonIDE 允许程序员交互式地探索代码中所有可能推断出多种类型的部分，并消除歧义。其最显著的功能包括类型比较工具（第 II-A0a 节）、候选表达式卡片（第 II-A0b 节）和推导步骤（第 II-A0c 节）。这些功能集成在调试环境中，程序员可根据自己的偏好和调试需求，分别启用或禁用这些功能。ChameleonIDE 是开源软件，可通过 [19] 获取。

本文的贡献如下：

- 我们介绍了 ChameleonIDE 的设计与实现，用于可视化类型错误的相关上下文，并让程序员能够以小块为单位交互式地探索和验证错误位置。

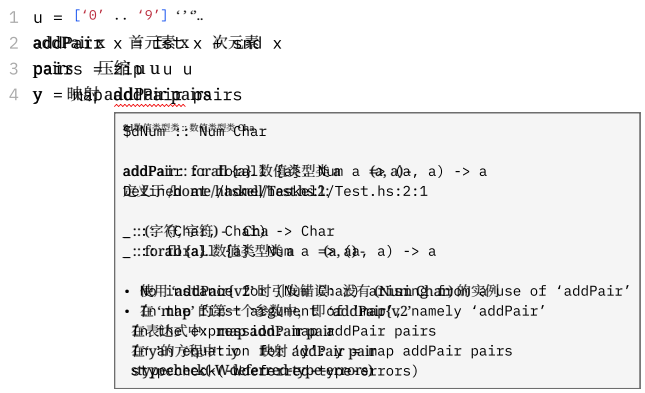


图 1. 在 Visual Studio Code [21] 和 Haskell Vscode 扩展 [22]中显示的类型错误。addPair 表达式被指认为导致类型错误的原因。这可能与程序员的意图不符。

- 我们报告了三项旨在评估 ChameleonIDE 的实验结果。

我们的实验表明，与传统的基于文本的错误消息相比，使用 ChameleonIDE 的程序员能更快地修复类型错误。在解决较难任务时，这种差异更为显著。此外，积极使用 ChameleonIDE 交互功能的程序员，比仅阅读类型错误输出的程序员修复类型错误更快。尽管 ChameleonIDE 是为 Haskell 语言设计的，我们计划将其底层理念扩展到其他强类型语言，例如 Rust 或 TypeScript。

I. 动机

ChameleonIDE 的设计需求源于传统类型错误的局限性，这些局限性已在多项研究中有所记载（例如 [17], [20]），而我们将在此通过几个示例加以说明。

*1)*传统类型错误只显示有限的位置信息：Haack 和 Wells [23] 指出：“仅将程序中的一个节点或子树标记为错误位置，会让程序员难以理解类型错误。要选出修复类型错误的正确位置，程序员必须找出所有其他参与该错误的程序位置。”图 1 中的类型错误可以在多个位置修复。例如，将第 1 行的 ['0'..'9'] 替换为 [0..9]，，或者将第 2 行的 fst x 和 snd x 替换为 read (fst x) 和 read (snd x)。在类型错误消息中，只有第 4 行的 addPair 表达式被归咎于错误。在这个小例子中，整个上下文都一目了然；但在大型程序中，导致类型错误的代码行可能在源代码中相距甚远，这就会成为问题。

2) 传统类型错误存在偏向：一种常见的偏向是，类型错误虽然在某个表达式中报告，却也可能出现在其他多个表达式中。如图 1 所示，错误消息武断地只聚焦于 addPair，却忽略了 u 的定义中的字面量也可能有误。另一种偏向是，传统类型错误常被描述为期望类型与实际类型之间的冲突。这种描述方式是大多数静态类型语言中的标准做法。然而，何为期望、何为实际，是不同合一顺序所导致的副作用，并非程序员的本意。在这两种情况下，错误消息都可能使程序员错误地认为部分代码有效，并错误地指责其他部分。

3) 传统类型错误的解释效果不佳： 当编译器拒绝一个程序时，类型检查的内部状态是复杂计算的结果。然而，这一过程的细节难以向用户解释，编译器通常也不会报告这些细节。对于图 1 所示的典型类型错误，判断类型错误的依据来自此前的声明。程序员必须用不够严谨的方法重新发现这些依据。

A. *ChameleonIDE* 的设计目标

基于传统类型错误的局限性，我们为 ChameleonIDE 提出了以下设计要求：**显示** 类型错误发生或可能发生的所有位置。**解释** 类型错误时避免使用行话和类型检查器的内部构造。

不要根据 计算顺序来预设应归咎于哪个表达式，也不要预设某个表达式可能的类型中哪一个是“实际”类型或“预期”类型。

II. CHAMELEON IDE

ChameleonIDE 由两部分组成：一个类型推断引擎和一个新颖的交互式调试界面。调试界面从头开始设计；类型推断引擎则是对原始 Chameleon 的重新实现，并包含若干新改进，详见第 II-B 节。

A. 调试界面

ChameleonIDE 调试界面提供了三项主要功能，用于可视化并解释类型错误。

a) 类型比较工具：类型比较工具会用不同颜色显示冲突类型，并将每种类型与一个或多个以对应颜色高亮显示的错误位置相关联（图 3）。如果程序员知道表达式预期的类型（他们通常知道），就能排除一半可能的位置。将鼠标悬停在某个可能的类型上，可以通过二分查找来辅助排查，此时只有导致该类型的相关位置会被高亮显示。

b) 候选表达式卡片：候选表达式是指可以推断出具有两种冲突类型的表达式。检测到类型错误时，ChameleonIDE 会提供所有候选表达式的列表，程序员可以点击某张候选表达式卡片，选择要解决的问题。在图 4 所示的示例中，x 和 y 都是候选表达式。修复任一类型错误都可以使这两个表达式都成为类型正确的表达式。

程序员点击某张候选表达式卡片即可选中它。选中卡片后，冲突类型块中的信息会随候选表达式的变化而更新。

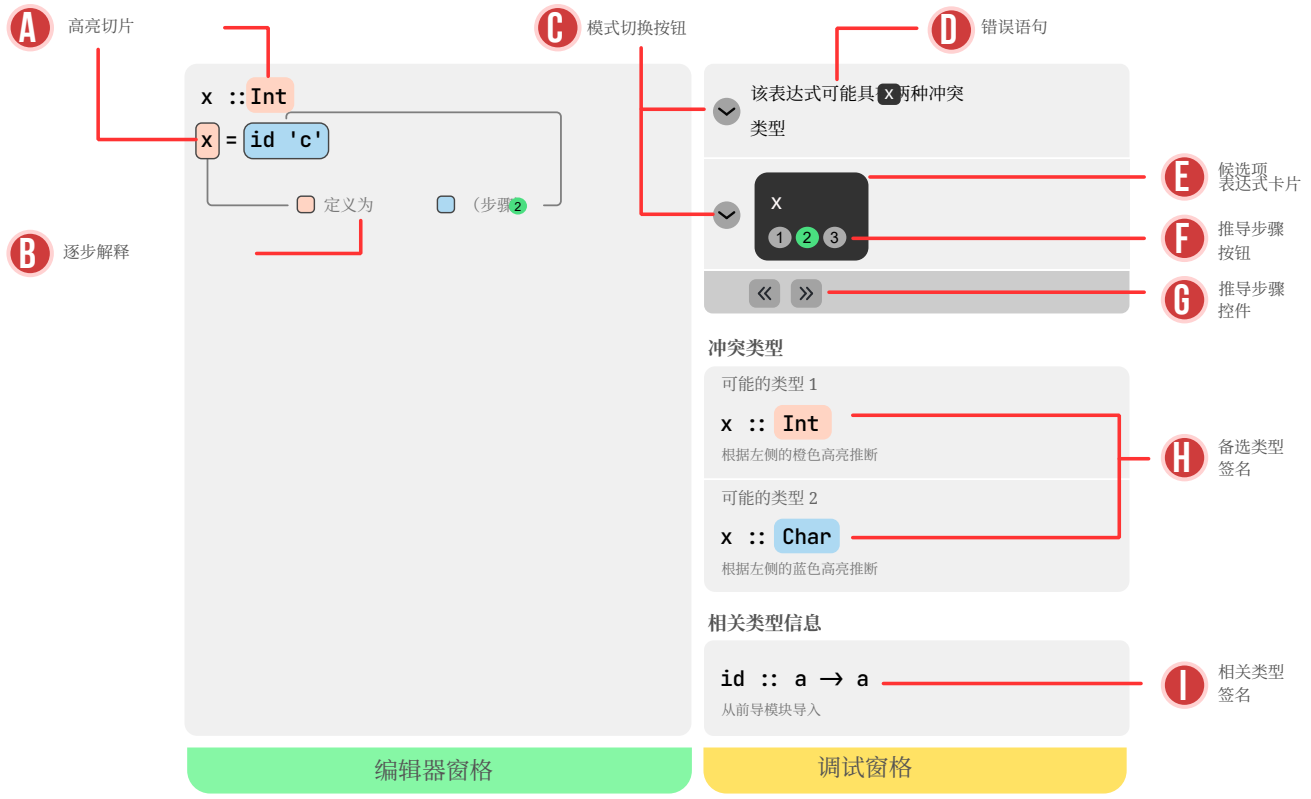


图 2。**ChameleonIDE 的组成**。编辑器窗格（左侧）类似于传统的代码编辑器。源代码片段可能会以某种颜色高亮显示（A）。此外，启用推导步骤后，解释层（B）也会显示。调试窗格包含三个区块。首先，错误语句块包含一条错误语句（D）、一个可选的候选表达式卡片列表（E）、一个推导步骤列表（F），以及一个用于增加/减少推导步骤的控制栏（G）。其次，冲突类型块显示两种备选类型（H）。第三，相关类型信息块会显示其他信息（I），这些信息可能有助于理解类型错误。



图 3。**启用类型比较工具后的 ChameleonIDE**。ChameleonIDE 为表达式 u 识别出冲突类型，并将相关位置与每种类型关联起来。请将此输出与图 1 中传统的类型错误消息进行比较。



图 4. **启用候选表达式卡片的 ChameleonIDE**。 表示类型错误可能出现在 x 或 y 的定义中。

在编辑器窗格中，某些错误位置的高亮颜色会根据更新后的候选表达式而变化。或者，程序员也可以将鼠标悬停在某张卡片上，预览候选表达式的变化。光标移开后，悬停效果即会恢复。

c) 推导步骤：推导步骤让程序员可以逐一探索所有错误位置（图 5）。步骤以按顺序编号的圆形按钮（步骤按钮）列表形式显示，编辑器窗口中还会显示一个解释层。在

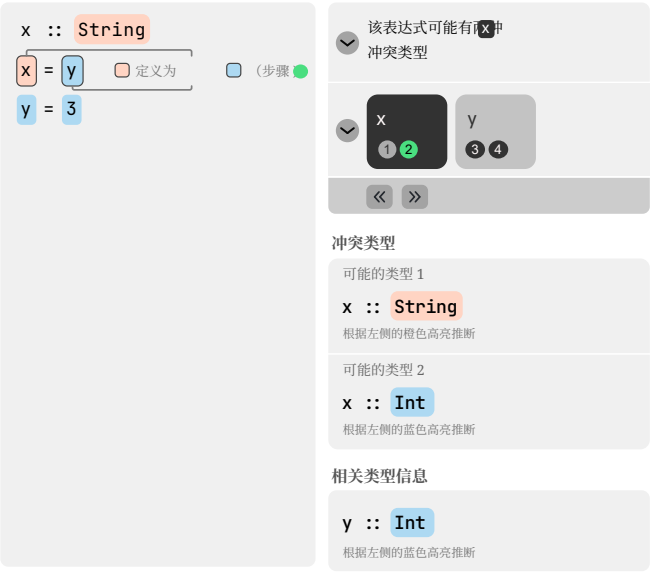


图 5. **启用推导步骤的 ChameleonIDE**。ChameleonIDE 分四个步骤解释类型错误。截图中当前处于第 2 步，ChameleonIDE 显示表达式 x 和 y 应具有相同的类型。

解释层中，正在检查的两个位置会以轮廓标出，并用一条线连接起来。这条线旁附有一段易于理解的文本，用于说明这两个位置之间的语义联系。程序员可以自由激活任意步骤。当前激活的步骤以绿色显示。激活某个步骤时，部分高亮会切换颜色，解释层中的消息也会相应变化。图 5 中的程序会生成一个步骤列表，如图 6 左侧所示。

程序员可以使用鼠标和键盘快捷键来递增或递减步骤编号，或跳转到任意步骤。程序员通过浏览所有推导步骤，并核实每条说明是否符合自己的意图，来解决类型错误。最终，他们会找到不匹配的步骤，并可通过修改两个标出的错误位置之一来修复类型错误。

在内部，推导步骤代表了将错误位置划分为两组的不同方式，这两组分别用两种颜色表示。每种颜色都会推断出候选表达式的不同类型。每次递增或递减步骤，都会改变两种颜色的分界点（图 6 中的虚线）。

d) 多种模式：Nielson 指出，在设计可用性时，最重要的两个问题是了解用户的任务以及用户之间的差异 [24]。通过分析用户如何使用 ChameleonIDE，我们意识到，理想的调试界面应适应具体的程序员和编程任务。有时，程序员希望调试器只需“给出答案”；有时则希望深入问题领域，寻找最优解。为了满足用户对信息密度和控制粒度进行自定义的需求，ChameleonIDE 提供了三种模式：基础、均衡和高级。程序员可以在不同模式之间切换

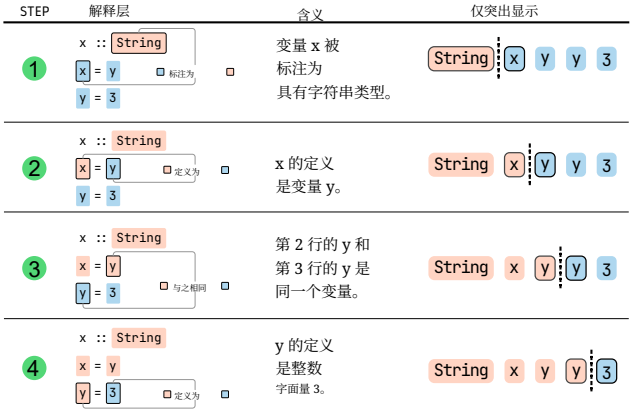


图 6. 一次性显示所有推导步骤的情况。实际上，步骤会逐个显示。程序员可以使用步骤控制栏（图 2-G）递增或递减步骤编号，也可以直接点击某个步骤按钮（图 2-F）。递增或递减推导步骤，可以直观地理解为移动分界点（虚线）的位置，也就是蓝色和橙色高亮的分界处。

Mode	功能
基础模式	类型比较工具
平衡模式	类型比较工具 候选表达式卡片
高级模式	类型比较工具 候选表达式卡片 推导步骤

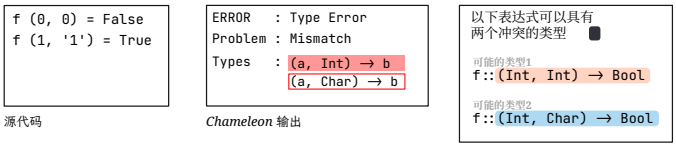
表 I
CHAMELEONIDE 模式与功能

点击模式切换开关即可切换模式（图 2-C）。不同模式下可用的功能汇总于表 I。

B. 类型推断引擎

Chameleon 最初是 2000 年代初开发的一款命令行工具，旨在改进 Haskell 编程语言的类型错误报告。与格拉斯哥 Haskell 编译器（GHC） [25]，产生的传统类型错误不同，后者使用 Hindley–Milner 类型推断系统，而 Chameleon 则通过约束求解来推断类型。在 Chameleon 中，约束依据类型规则从源代码中生成。此外，每个约束都标记有其生成位置。如果程序类型正确，这组约束就是一致的，否则就是不一致的。发生类型错误时，系统会使用高效算法推导出一个仍包含矛盾的最小约束子集。这个子集称为最小不可满足子集（MUS）。据此，Chameleon 可以利用 MUS 中约束的标签，报告一组位置。Stuckey 等人 [26]证明，与 MUS 中约束相关联的程序位置都与类型错误有关，并且必然包含错误原因。

尽管成功借鉴了其底层理念，但由于项目语言标准和所用库都已过时，我们无法复用 Chameleon 的原始实现。



源代码 *Chameleon* 输出 *ChameleonIDE* 输出

图 7. 报告相同的类型错误时，Chameleon 使用更抽象的类型 Int -> a 和 Char -> a，而 ChameleonIDE 使用具体类型（不含类型变量的类型）Int -> Bool 和 Char -> Bool。

我们的 ChameleonIDE 实现从多个方面扩展了原有的 Chameleon 方法。

a) 从类型错误中恢复具体类型：仅使用来自 MUS 的约束，足以定位类型错误；但要从类型错误中恢复类型，我们还需要程序中与该类型错误无关部分的约束。例如，考虑一个类型错误的二元组，它可以被赋予两种可能的类型：(Int, Int) 和 (Int, String)。从 Chameleon 重建出的类型可能是 (a, Int) 和 (a, String)。尽管理论上恢复出的类型是正确的，但它们引入了记号 a，表示一个可以是任意类型的泛型类型变量，这使错误消息更难理解。为了解决这个问题，在 ChameleonIDE 中，对于 MUS 中的每个约束 c，我们从所有约束中找出一个极大可满足子集（MSS），其中包含 MUS 的其他所有元素，但不包含 c。这些极大可满足子集虽然无助于定位错误，却能产生最具体的类型，见图 7。像 Int 和 String 这样的具体类型，通常能为程序员提供额外信息。例如，使用 (Float, Float) 这样的类型，程序员可能想表达二维空间中的一个点。然而，(a, Float) 这样的类型无法保留这类信息。

b) 类型错误解释：此外，ChameleonIDE 支持类型解释。与 [27]，中的类型解释系统类似，ChameleonIDE 能够生成类型错误的人类可读解释。为此，我们会为抽象语法树中的节点添加注释，标明约束及所使用的类型推断规则。我们根据约束及其附带的注释生成推断历史。

```
如果a那么b否则c
a = "True"
```

清单 1. 一个类型错误的简单程序。它从第 1 行生成两个约束，从第 2 行生成一个约束。

例如，对于清单 1 中的程序，ChameleonIDE 会生成以下约束和标签（括号内）$T_a = Bool$（if 条件）、$T_b = T_c$（if 分支）、$T_a = String$（定义）。显然，由于 T_a 无法同时与布尔值和字符串统一，因此该程序的类型不正确。ChameleonIDE 可以根据 MUS 构造人类可读的解释。清单 1 的示例输出如下：

a 的类型是布尔值，因为 a 是 if 语句的条件；然而，a 的类型是字符串，因为 a 被定义为字符串字面量 "True"。这一解释有助于推导步骤（第 II-A0c 节）。

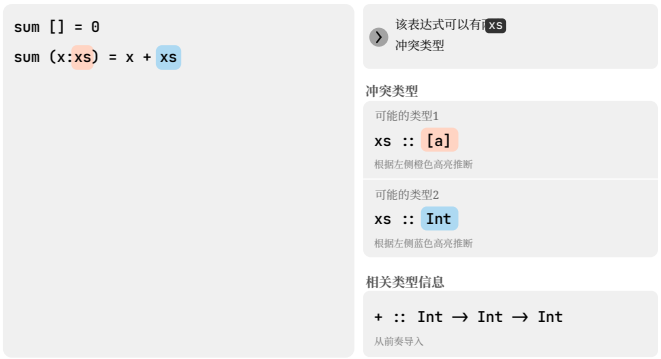


图8. Maxine 用于计算整数列表总和的代码；ChameleonIDE 在表达式 xs 上报告了错误。

III. 演示

本节将通过演示 ChameleonIDE 的使用示例来介绍它。这些示例从一位假想的 Haskell 程序员 Maxine 的视角展开。

A. 基础模式

Maxine 编写了一个函数，用于计算数字列表的总和，但 ChameleonIDE 显示存在类型错误（图 8）。阅读错误报告后，Maxine 意识到错误围绕着表达式 xs 展开。也就是说：xs 既可以是 [a]，也可以是整数。通过对照冲突类型块中的颜色（图 2-H）和高亮的错误位置，Maxine 知道 [a] 是由 : 运算符的模式匹配产生的，而整数则是通过 + 对两个表达式进行加法产生的。

此时，Maxine 知道可能的类型 1 符合她的意图，因此带有蓝色高亮的错误位置一定有误。检查程序后，她发现 Maxine 忘了在加法右侧递归调用 sum 函数。

B. 平衡模式

Maxine 编写了更多代码，以便在整数列表中只对偶数求和，并重用之前编写的 sum 函数。保存文件后，ChameleonIDE 显示表达式 sum 中存在类型错误（图 9）。然而，这并没有帮助，因为 Maxine 刚刚验证过 sum 的实现。切换到平衡模式后，ChameleonIDE 显示了两张卡片：sum 和 evens。

因此，Maxine 点击 evens 卡片，ChameleonIDE 报告表达式[Int] 和 [Int] -> [Int] 有两种可能的类型（图 10）。Maxine 知道表达式 evens 保存着一个临时的偶数列表（因此属于 [Int] 类型），便判断可能的类型 2 并非预期结果。蓝色高亮的位置必定包含问题根源。Maxine 很快就发现，列表 l 并未传递给 filter 函数。

C. 高级模式

为了说明第 III-B 节中所示任务的推导步骤，首先，Maxine 点击步骤 5（图 11），并



图 9. Maxine 计算偶数之和的代码。ChameleonIDE 报告了一个错误，并给出两个候选表达式。

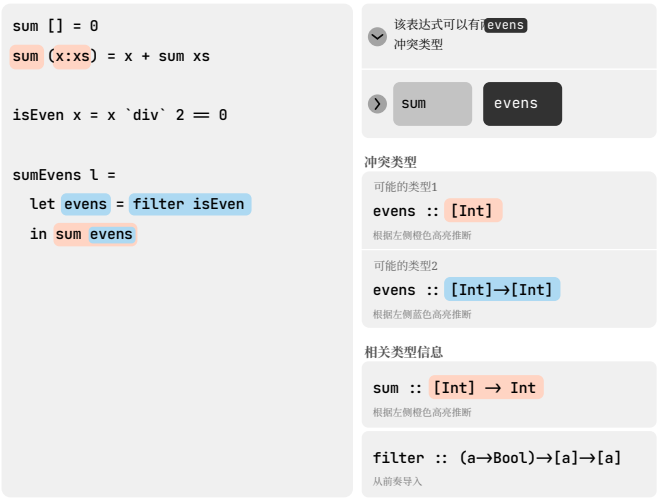


图 10. 点击 evens 卡片（5）后，冲突类型面板会发生变化，显示 evens 的可能类型；变化高亮颜色则反映出这样一种假设：evens 的定义是错误的原因。

确认 evens 的两处出现应当相同，且第二次使用意味着 evens 是一个整数列表。其次，她点击步骤 6（图 12），并确认 evens 应与右侧的声明具有相同类型。

最后，Maxine 点击步骤7（图 13），看到 filter 函数应用于一个参数 isEven。通过查阅相关类型信息，Maxine 发现 filter 需要两个参数，而这里只提供了一个。

IV. 评估

我们进行了三项用户研究，迭代改进 ChameleonIDE 用户界面，并根据图 14 评估了若干研究问题。

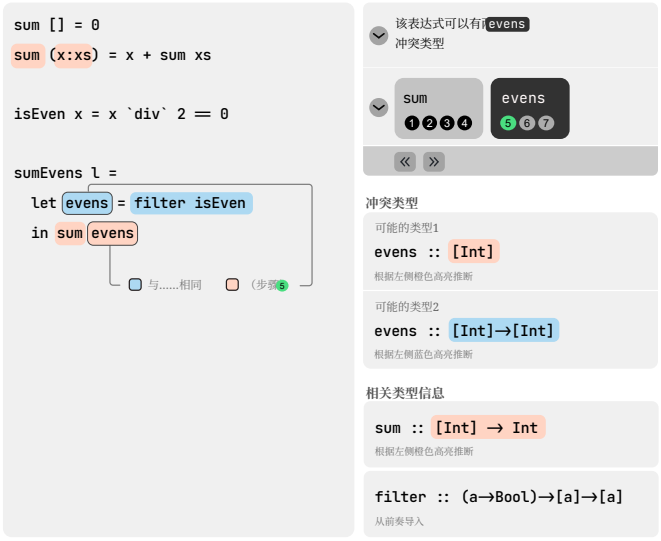


图 11. Maxine 在高级模式下计算偶数之和的代码。当前步骤为步骤5，ChameleonIDE 说明，表达式 evens 的两处出现应具有相同的类型。

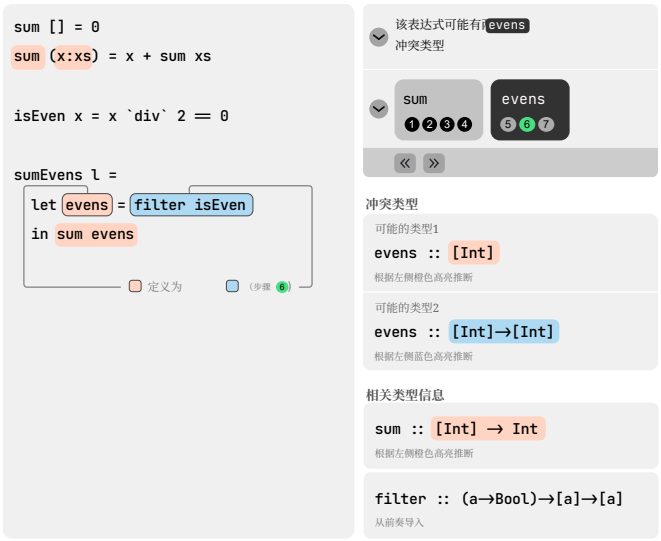


图 12. 在步骤 6 中，ChameleonIDE 解释说，evens 被定义为表达式 filter函数 isEven。左侧和右侧应具有相同的类型。

A. 实验设计

招募： 参与者通过 Reddit*r/haskell* 和 *r/programminglanguages* 社区招募。参与过程完全匿名；这些实验的详细伦理影响已由作者所在机构的机构审查委员会审查并批准。

实验设置： 实验在线进行，且无人监督。所有用户研究均使用作者开发的基于 Web 的调试环境。

培训与分组： 征得同意后，参与者接受了关于工具界面和交互功能的交互式培训。参与者还拿到了一份速查表，

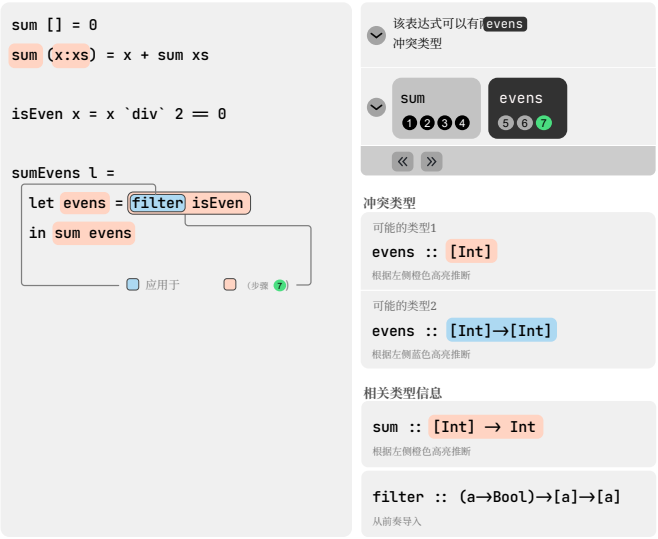


图 13。在步骤7中，ChameleonIDE 解释了 filter函数应用于函数 isEven。借助右下方相关类型信息面板中 filter函数的类型信息，Maxine 可以发现该类型错误：filter函数需要两个参数，但实际只接收了一个。

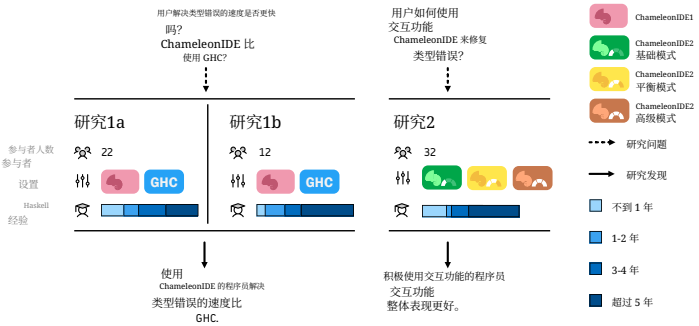


图 14。ChameleonIDE 评估的时间线。

其中总结了界面的主要功能，并且在整个研究期间都可以随时查阅。数据收集开始前，参与者进行了4次试运行（每种设置各2次）。所有研究均采用被试内设计，以评估不同工具或功能集的效果，同时对参与者编程熟练度的差异进行平衡。每项研究中，参与者都需要完成一系列编程任务（研究1a和1b各8项，研究2有9项）。每项任务中，参与者会收到一个包含一个或多个类型错误的 Haskell 文件，然后需要借助提供的工具修正代码。

数据收集： 时间从每项任务开始时计起，直到程序首次成功通过类型检查且通过所有功能测试。参与者若遇到困难，可以跳过任务。完成所有任务后，参与者会被要求填写一份事后问卷。问卷问题包括他们使用 Haskell 的经验以及对这些工具的反馈。

我们使用浏览器会话录制工具 [28] 来录制

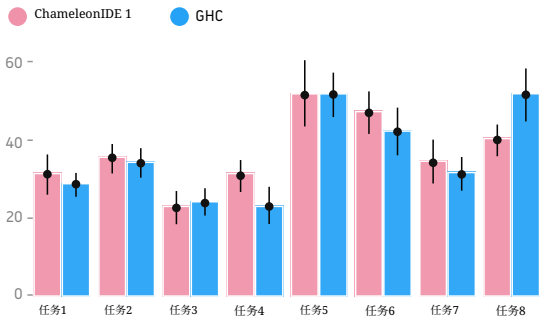


图 15. 研究1a 任务完成时间（秒），以及 95%置信区间。

研究过程。这使我们能够发现研究中的可用性问题，并识别一般模式。

B. ChameleonIDE 人类研究

*1）**ChameleonIDE 1***： 这是一个早期版本的用户界面，与图（2-13）所示不同。它采用了类型推断引擎，该引擎能在发生类型错误后恢复大多数具体类型，并提供一组精简的调试功能。ChameleonIDE 1 的主要功能包括显示两个（或更多）备选类型、显示所有可能的错误位置、根据备选类型将可能的错误位置分组，以及恢复具体类型。简而言之，ChameleonIDE 1 等同于设为基础模式的 ChameleonIDE 2。

研究人员开展了两项研究（1a 和 1b），比较使用 ChameleonIDE 1 和 GHC 编译器错误消息解决类型错误的效果。我们选择 GHC 编译器错误消息作为基线，因为它是 Haskell 中处理类型错误的标准工具。

两项研究都设置了八项任务。研究 1a 的任务取自作者所在机构 Haskell 编程课程的练习。第二项研究的任务则选自 GitHub [29] 上排名前 20 的 Haskell 主题。随后，作者手动在程序中加入类型错误。两项研究中的类型错误包括简单不匹配、易混淆的语法、缺失实例、优先级和结合性、无限类型，以及容易混淆的列表与元素。这些类别依据 Tirronen 的研究 [16]中常见的类型错误而定。

研究（1a 和 1b）探讨以下研究问题：**研究问题1**。 与 *GHC* 编译器错误消息相比，程序员使用 *ChameleonIDE* 解决类型错误的速度是否更快?

结果： 研究1a中收集的数据（图 15）显示，任务 1-7之间没有显著差异。事后看来，这些任务对大多数用户而言都是简单的挑战，而且参与者之间的个体差异通常比不同处理条件之间的差异更显著。不过，有一个值得注意的现象：在任务8中，ChameleonIDE组的表现优于GHC组。我们将这一显著差异归因于任务8的难度。该源文件更长，涉及更多语言特性（抽象数据类型和高级函数）。对于这类错误，GHC难以生成相关的错误消息。根据这一结果，我们假设，如果使用篇幅更长、更贴近实际的源代码任务，我们可能会观察到更显著的

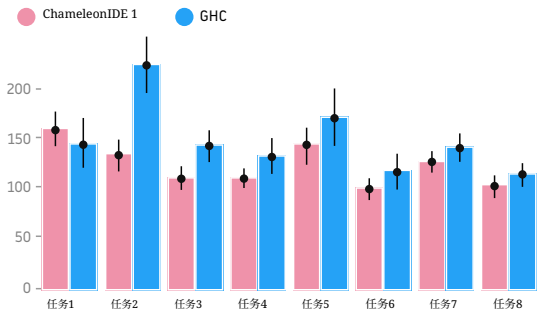


图16。研究1b任务完成时间（秒），含95%置信区间。

交互级别	描述
最少	用户通过进行以下操作完成了任务：修改源代码、进行类型检查，以及阅读错误消息。
Low	用户在所有模式下都只主动使用通用功能，例如将鼠标悬停在“可能的类型 1”和“可能的类型 2”以缩小错误空间。
High	用户完成了低交互组的所有任务，但使用了平衡模式和高级模式专有的功能，例如激活步骤和表达式卡片。

表II 程序员交互级别及其描述

差异。这一假设也得到了最常见反馈的支持：受访者普遍认为任务过于简单，不足以进行有意义的评估。一位参与者说：“看起来比GHC更美观，但如果不在更复杂的任务上试用，我就无法判断它在实践中是否能帮到我。”

因此，在研究1b中，我们引入了更具挑战性的任务。结果确实发现，除了任务1之外，ChameleonIDE 组在几乎所有任务中的表现都快于 GHC 组（图16）。我们采用双样本配对 t 检验比较 ChameleonIDE 组和 GHC 组的完成时间。两组之间存在显著差异：$t(23) = -3.86$，$p = 0007$。对于任务1，我们推测，由于对 ChameleonIDE 的界面不熟悉，一些参与者花了更多时间进行探索。对于其他所有任务，从视频录像中可以看到，许多 ChameleonIDE 用户会自信地跳过阅读无关的代码片段，而 GHC 用户通常会通读整个程序。在问题更难、代码更杂乱时，我们注意到程序员开始体会到 ChameleonIDE 的优势。一位参与者表示：“我注意到它最有用的功能是指出发生冲突的两处用法的位置；GHC 经常让人难以弄清楚它是如何得出某个类型结论的。”另一位参与者表示：“我认为 ChameleonIDE 在处理错误消息方面比 GHC 做得好得多。我喜欢它能显示类型判断的依据。这让人很容易弄清楚如何修正错误。”

2）***ChameleonIDE 2***：根据研究1的观察结果，我们为 ChameleonIDE 引入了几项新功能，最终形成了图（2-13）所示的用户界面。在这一版本中提供了交互功能，例如推导步骤、候选表达式和模式切换。在 ChameleonIDE 1 与 ChameleonIDE 2 的开发过程中，还设计并制作了另外几个用户界面的原型 [30] 。研究2探讨了以下研究问题：

研究问题2：程序员如何使用 *ChameleonIDE 2* 中的交互功能？。

更具体地说：

- **研究问题2.1** 程序员如何使用 ChameleonIDE 2 提供的高级功能？
- **研究问题2.2** 程序员在调试类型错误时是否更喜欢切换模式？
- **研究问题2.3** 程序员对 ChameleonIDE 2 提供的三种模式有何偏好？

每次运行中，每项任务的初始模式都会在三种不同模式之间轮换，并在九项任务中重复三个周期。每个周期中三种模式的顺序在所有参与者之间均衡安排。不过，参与者可以随时切换到其他模式。

结果：研究2的方法比研究1更具探索性。我们鼓励程序员自行探索使用该工具的方式。通过对收集到的日志数据进行事后分析，我们得以归纳出一些有关该工具使用方式的有趣模式。

研究问题2.1。数据中最显著的特点是，用户使用该工具的方式差异很大。有些用户广泛使用各项功能，而另一些用户则没有主动探索所提供的信息便完成了任务。基于这种差异，我们在表II中将用户分为三组。

如图17所示，完成各项任务所需的时间大致与参与者的交互程度相关。交互程度较高的参与者通常表现更好，而交互程度最低的参与者表现较差。针对多重比较的 Tukey诚实显著差异检验发现，最低交互组与高交互组之间（$p \leq 0.001$，95%置信区间 = [18.26，31.41]），以及最低交互组与低交互组之间（$p \leq 0.001$，95%置信区间 = [11.96，26.67]）的完成时间存在显著差异。三项任务的结果偏离了总体趋势：在任务4和任务6中，交互程度较高的用户表现更差；而在任务9中，总体趋势则更加明显。与研究1a和1b一样，这种差异可能与任务难度有关。任务4和任务6比其他任务短。这两个任务的理想修复方案在源代码中相对靠前（都在源代码的前两行）。只需从上到下阅读的用户，就能迅速发现错误，无须借助 ChameleonIDE 提供的信息跳过无关代码段。这降低了 ChameleonIDE 在这些任务中所表现出的效益。另一方面，任务9是所有任务中最长的，而且还涉及嵌套层次很深的类型定义，较难在脑中跟踪。

另一个观察结果是，在使用 ChameleonIDE 的模式切换功能时，我们通过呈现

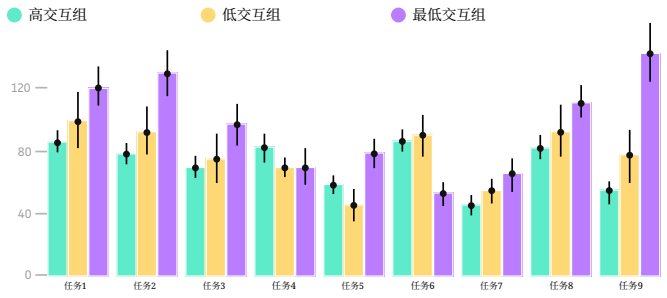


图17. 研究2的任务完成时间（秒），含95%置信区间。

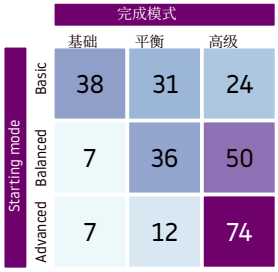


图18. 研究2按起始模式划分的模式切换情况。用户绝大多数切换到了更复杂的界面模式。

每项任务和每位参与者的起始模式与结束模式，并将其显示在相关矩阵中（图18）。这一观察结果揭示了使用多模式调试工具的两个特点。首先，对于**研究问题2.2**，程序员在这一点上大致分为两类：53% 会切换模式，47% 会留在同一模式。其次，对于**研究问题2.3**，程序员切换模式时，通常会选择信息更丰富的模式，而非更精简的模式。

C. 局限性

评估有效性的一项威胁在于参与者人数。尽管每项研究都收到了数百名线上参与者的报名，但研究的放弃率很高（尤其是研究1b）。这在意料之中：编程挑战难度较高，而自愿参与者没有报酬。由于我们在线招募参与者，并对所有参与者的信息进行了匿名化处理，因此先前研究的参与者可能会参加后续研究。这造成了参与者熟悉程度上的差异。为此，我们在每项研究中使用新的代码挑战，并在数据收集前进行试运行，以帮助新参与者熟悉情况。远程、无人监督地开展研究，使我们无法在用户遇到可用性问题时介入。为缓解这一问题，我们在开展每项研究前进行了认知走查和沙盒试点。

未来的评估若能采用更贴近现实的任务，将会受益匪浅。我们的人体研究中使用的任务并不像专业 Haskell 程序员在典型的生产代码库中可能遇到的那样复杂。值得探究 ChameleonIDE 如何用于处理涉及多个文件和软件包，并包含更多令人困惑的抽象概念（如单子、单子变换器和透镜）的类型错误。

V. 讨论

本文介绍了交互式类型调试工具 ChameleonIDE，并阐述了该工具在多轮迭代中如何根据用户评估和反馈不断演进其设计，同时考察了这一通用方法相较于传统静态类型错误消息的有效性。我们发现，使用 ChameleonIDE 的程序员比使用传统的基于文本的错误消息的程序员更快地调试错误。当任务并非微不足道时，这种效果更加明显。我们还发现，主动使用 ChameleonIDE 交互功能的程序员，比被动阅读类型错误输出的程序员更高效地修复类型错误。本节将讨论对这些结果的几种解读。

A. 对阅读源代码的影响

从研究1a的结果中，我们观察到，调试工具的选择对程序员解决简单类型错误的速度影响不大。相反，在研究1b中面对更贴近实际的问题（源代码更长、错误位置分布更分散）时，使用 ChameleonIDE 的程序员表现得更高效。一种解释是，ChameleonIDE 能更直接地将程序员引向问题所在，从而减少阅读时间。早期研究 [31], [32] 表明，阅读源代码通常是解决编程问题的第一步，而且需要分多轮进行。尽管传统编译器错误消息工具最初显示的错误位置较少，但这些位置可能并不完整，这意味着程序员不得不扩大阅读范围，却得不到明确指引。相比之下，ChameleonIDE 最初会显示更多错误位置。不过，错误位置的完整性可以让程序员确信，源代码中哪些部分可以放心跳过。

B. 制定调试计划

根据研究2的结果，我们发现，使用交互式工具的程序员比被动阅读错误输出的程序员更快地修复类型错误。在难度较高的任务中，这种效果更为明显。我们推测，造成这一结果的因素之一是 ChameleonIDE 有助于制定调试计划。我们观察到，使用 ChameleonIDE 时，程序员会制定不同的调试计划来解决问题。在用户研究2中，高交互性组的参与者中，有些程序员会逐步浏览演绎步骤，以此指导源代码阅读；有些则会跳转到演绎链的两端，因为类型通常会在那里得到明确、具体的确定。相比之下，低 交互性组的参与者通常会制定相似的计划，包括仔细阅读程序文本，并根据自己对程序的理解手动标注表达式。

C. 外显中间类型信息

我们推测，ChameleonIDE 交互式调试工具之所以有效，另一个原因是它们能帮助程序员有效地组织中间信息。

```
f z
    | z   == 3 =假真
    | z == '4'
```

清单 2. ChameleonIDE 在表达式 f 和 z 中报告了一个错误

对于清单 2 中所示的程序，ChameleonIDE 提供了两个候选表达式：f 的类型可以是 Int -> Bool 或 Char -> Bool；z 的类型可以是 Int 或 Char。尽管理论上这两种表述是等价的，但程序员通常需要根据前者推导出后者，反之亦然。而这项推导可能涉及多个层次。程序员必须在这一系列复杂的心智运算中记住所有中间类型及其推理过程。借助候选表达式卡片和推导步骤，这些中间信息会显示在屏幕上，随时可供查阅。一项近期关于工作记忆的研究[33]指出，这种方法可能有助于程序员管理认知负荷，并释放工作记忆空间用于高层次思考。

VI. 相关工作

A. 查找所有类型错误位置

许多研究都探讨过如何找出所有会导致类型错误的位置 [23], [26], [34], [35]。类型错误切片 [23] 是一种用于找出针对该类型错误的完备且最小的位置集合的技术。该技术通过内部标注的约束和最小不可满足子集（MUS）生成来生成这些切片。Haack 的工作支持的语言是 Standard ML 的一个子集。最初的 Chameleon [26] 使用约束处理规则（CHR），支持在 Haskell 中计算类型错误切片。Chameleon 还支持高级类型级特性（类型类和函数依赖类型）。该项目还引入了通过命令行界面查询类型信息的功能。尽管 Chameleon 的基础牢固地建立在类型论的研究成果之上，但其设计从未通过用户研究进行评估。虽然找出所有错误位置有助于理解类型错误，但这只是良好类型错误报告提案宣言中列出的 7 项属性之一 [20]。据我们所知，我们的研究是首个针对涉及类型错误切片的交互式类型调试系统开展的以用户为中心的评估。

B. 生成高质量的错误解释

一般而言，编译器错误消息的一个缺陷是，它们无法用人类语言解释错误。正如 [36],“错误消息看起来采用了自然语言的形式，却像源代码一样难以阅读。”所言。生成更好的错误解释的一种经过充分研究的方法是使用 ECEM（增强型编译器错误消息）。通过一系列混合方法研究，Prather 表明 [37] ECEM 有助于人们理解编译器错误。Decaf [38]是一种工具，可以将 Java 编译器错误消息改写为增强版本。在一项针对 200 多名 CS1 学生的研究中，结果表明，Decaf 能减少他们在编程实践中出现的整体错误数量。Berik 提出了一种基于论证理论构建编译器错误消息的框架 [39]，并指出，采用简单论证布局或扩展论证布局的错误消息更贴近人类表达。这些研究表明，改进编译器错误消息的语言具有重要意义。ChameleonIDE 在构建错误语句时遵循了大多数原则和建议。然而，早期研究针对的并非单独的类型错误，而是一般的编译器错误（有些甚至包括运行时错误）。这些研究没有考虑类型错误的细微之处，例如替代类型推断。此外，这些解释系统是专为新手用户设计的。

C. 交互式调试

现代编程工具可以提供替代性的代码编写方式、显示实时反馈，并通过可视化呈现复杂的编程上下文。许多工具都旨在利用这些能力改善调试体验。下面列举两个例子。Hazel Tutor [40]是一个面向 OCaml 语言的交互式、由类型驱动的环境。它可以通过弹出窗口建议模板表达式（作者称之为“策略”），自动填充类型空洞。它还提供基于光标的类型检查器，使程序员能够查询程序不同部分的类型。Whyline [41] 是一个 Java 调试系统，允许用户提出诸如“变量 X 为什么具有值 Y”之类的问题。它还允许用户以交互方式提出后续问题，以进一步了解错误的性质。这些调试工具是开发 ChameleonIDE 的重要动机。不过，它们关注调试过程中的不同方面。Java Whyline 主要解决意外运行时行为的问题，而 Hazel Tutor 则专注于利用类型空洞提供开发辅助。

VII. 结论

我们介绍了 ChameleonIDE，这是一款面向 Haskell 编程语言的类型调试工具。它的基于约束的类型推断引擎能够提供客观、全面的错误位置报告。我们通过程序员评估了该工具的设计。研究发现，尤其是在处理更复杂的任务时，与传统的基于文本的错误消息相比，ChameleonIDE 能帮助程序员更快地修复类型错误。此外，研究还表明，主动使用 ChameleonIDE 交互功能的程序员，比仅仅阅读类型错误输出的程序员修复类型错误更快。目前，ChameleonIDE 支持 Haskell；未来，我们计划扩展其类型检查系统，使其能够支持 Rust 或 TypeScript 等其他强类型语言。

致谢

Peter Stuckey 的工作部分得到了 OPTIMA ARCITTC、项目编号 IC200100009 的支持。

参考文献

[1] R. Chatley、A. Donaldson 和 A. Mycroft，《接下来的 7000 种编程语言》，载于《计算与软件科学：现状与展望》，丛书：计算机科学讲义，B. Steffen 和 G. Woeginger 编。Springer International Publishing，2019年，第 250– 282 [ ] 10 1007 978 3 319 91908 9 15 –页。在线。可查阅：https://doi.org/ 。 / - - - - S. Kleinschmager、R. Robbes、A. Stefik、S. Hanenberg 和 E. Tanter，《静态类型系统能否提升软件系统的可维护性？一项实证研究》，载于《2012年第20届IEEE国际程序理解会议（ICPC）》[2] 2012年，第 153–162 页，ISSN：1092-8138。[3] C. Mayer、S. Hanenberg、R. Robbes、E. Tanter 和 A. Stefik，《静态类型系统对无文档软件可用性的影响：一项实证研究》，载于《ACM面向对象编程系统、语言与应用国际会议论文集》，丛书：OOPSLA '12。计算机协会，2012年，第 683–702 页。 [在线]。可查阅：https://doi.org/ 10 1145 2384616 2384666 。 / 。Z. Gao、C. Bird 和 E. T. Barr，[4] 《使用类型还是不使用类型：量化 JavaScript 中可检测的程序错误》，载于《2017年第39届IEEE/ACM国际软件工程会议（ICSE）》，2017年，第 758–769 页，ISSN：1558-1225。[5] B. Ray、D. Posnett、P. Devanbu 和 V. Filkov，《GitHub 上编程语言与代码质量的大规模研究》，第 60 卷，第 10 期，第 91 100, 2017 [ ] 10 1145 3126905 – 页。在线。可查阅：https://doi.org/ 。 / L. A. Meyerovich 和 A. S. Rabkin，[6] 《编程语言采用情况的实证分析》，载于《2013年ACM SIGPLAN面向对象编程系统、语言与应用国际会议论文集》，丛书：OOPSLA '13。计算机协会，2013年，第 1–18 页。 [在线]。可查阅：https://doi.org/ 10 1145 2509136 2509515 。 / 。Z. Chen、Y. Li、B. Chen、W. Ma、L. Chen 和 B. Xu，[7] 《Python 系统中与动态类型相关实践的实证研究》，载于《第28届国际程序理解会议论文集》，丛书：ICPC '20。计算机协会，2020年，第 83–93 页。 [ ] 10 1145 3387904 3389253 在线。可查阅：https://doi.org/ 。 / 。B. Wang、L. Chen、W. Ma、Z. Chen 和 B. Xu，[8] 《Python 动态特性对变更倾向性影响的实证研究》，2015年，第 134–139 页。[9] Z. Xu、P. Liu、X. Zhang 和 B. Xu，《用于程序错误检测的 Python 预测分析》，载于《2016年第24届ACM SIGSOFT软件工程基础国际研讨会论文集》，丛书：FSE 2016。计算机协会，2016年，第 121–132 页。 [ ] 10 1145 2950290 2950357 在线。可查阅：https://doi.org/ 。 / 。微软。带类型语法的 JavaScript[10] [ ]。在线。可查阅：https://www.typescriptlang.org/ [11] mypy。mypy——适用于 Python 的可选静态类型。 [在线]。可查阅：http://mypy-lang.org/ [12] P. Hudak、J. Hughes、S. Peyton Jones 和 P. Wadler，《Haskell 的历史：懒惰而有类》，载于《第三届ACM SIGPLAN编程语言史会议论文集》，丛书：HOPL III。计算机协会，2007年，第 12–1–12–55 页。 [ ] 10 1145 1238844 1238856 在线。可查阅：https://doi.org/ 。 / 。Bill Wagner。[13] 2022（）类型参数上的约束——C#编程指南。 [在线]。可查阅：https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters [14] 甲骨文。（2022年）泛型方法和有界类型参数（JavaTM 教程： ＞学习 Java 语言 ＞泛型（已更新））。[在线]。可查阅：https://docs.oracle.com/javase/tutorial/java/generics/boundedTypeParams.html [15] 微软。（2022年）文档——泛型。 [在线]。可查阅：https://www.typescriptlang.org/docs/handbook/2/generics.html [16] V. Tirronen、S. Uusi-M¨akel¨a 和 V. Isom¨ott¨onen，《理解初学者在 Haskell 中的错误》，第 25 卷，e11 页，2015年，出版方：剑桥大学出版社。[17] J. Hage，《类型错误诊断中的已解决问题与开放问题》，第 13 页，2020年。[18] B. Wu 和 S. Chen，《类型错误是如何修复的，学生又做了什么？》，第 1 卷，第 105:1–105:27 页，2017年。 [在线]。可查阅：https://doi.org/ 10 1145 3133929 。 / Fu, Shuai。[19] 2022（）Chameleon类型调试器。[在线]。可查阅：https://chameleon.typecheck.me/ [20] J. Yang, G. Michaelson, P. Trinder, 和 J. B. Wells，“改进类型错误报告”，载于《第12届函数式语言实现国际研讨会论文集》。Citeseer，2000年。[21] 微软。Visual Studio Code——代码编辑，重新定义。 [在线]。网址：https://code.visualstudio.com/ [22] Haskell。Visual Studio Code 的 Haskell 扩展。 [在线]。网址：https://marketplace.visualstudio.com/items?itemName=haskell.haskell [23] C. Haack 和 J. B. Wells，“隐式类型高阶语言中的类型错误切片”，第50卷，第1期，第189–224页，2004年。 [在线]。网址：https://www.sciencedirect.com/science/article/pii/S016764230400005X [24] Jakob Nielsen，《可用性工程》。Morgan Kaufmann，1993年。[25] Ben Gamari。（2022年）主页——格拉斯哥 Haskell 编译器。 [在线]。网址：https://www.haskell.org/ghc/ [26] P. J. Stuckey、M. Sulzmann 和 J. Wazny，“Haskell 中的交互式类型调试”，载于《ACM SIGPLAN Haskell '03 研讨会论文集》。ACM出版社，2003年，第72–83页。 [在线]。网址：http://portal.acm.org/citation.cfm?doid =871895 871903 。Y. Jun、G. Michaelson 和 P. Trinder，[27] “解释多态类型”，第45卷，第436–452页，2002年。[28] OpenReplay。（2022年）OpenReplay：开源会话回放。 [在线]。网址：https://openreplay.com/ [29] GitHub。（2022年）GitHub 主题：Haskell。 [在线]。网址：https: //github.com/topics/haskell[30] Fu, Shuai 和 Dwyer, Tim 和 Stuckey, Peter，“交互式 Haskell 类型推断探索（扩展摘要）”，2021年。[在线]。网址：https://icfp21.sigplan.org/details/TyDe-2021/6/Interactive-Haskell-Type-Inference-Exploration-Extended-Abstract- [31] A. Jbara 和 D. G. Feitelson，“程序员如何阅读常规代码：一项使用眼动追踪的对照实验”，载于《2015年 IEEE 第23届程序理解国际会议》，2015年，第244–254页，ISSN：1092-8138。[32] N. Peitek、J. Siegmund 和 S. Apel，“什么决定了程序员的阅读顺序？一项眼动追踪研究”，载于《第28届国际程序理解会议论文集》。ACM，2020年，第342–353页。 [在线]。网址：https://dl.acm.org/doi/10.1145/ 3387904.3389279[33] W. Crichton、M. Agrawala 和 P. Hanrahan，“工作记忆在程序跟踪中的作用”，2021年。 [在线]。网址：http://arxiv.org/abs/ 2101 06305 。Z. Pavlinovic、T. King 和 T. Wies，[34] “实用的基于SMT的类型错误定位”，载于《第20届 ACM SIGPLAN 函数式编程国际会议论文集》，丛书：ICFP 2015。计算机协会，2015年，第412–423页。 [在线]。网址：https://doi.org/ 10 1145 2784731 2784765 。/ 。T. Schilling，“无约束类型错误切片”，[35] 载于《函数式编程趋势》，丛书：计算机科学讲义，R. Pe˜na 和 R. Page 编。施普林格，2012年，第1–16页。[36] T. Barik、J. Smith、K. Lubick、E. Holmes、J. Feng、E. Murphy-Hill 和 C. Parnin，“开发者会阅读编译器错误消息吗？”，载于《第39届软件工程国际会议论文集》，丛书：ICSE '17。IEEE 出版社，2017年，第575–585页。 [在线]。网址：https://doi.org/ 10 1109 2017 59 。/ICSE。 。J. Prather、R. Pettit、K. H. McMurry、A. Peters、J. Homer、N. Simone 和 M. Cohen，[37] “新手与编译器错误消息的交互：一种人因方法”，载于《2017年 ACM 国际计算教育研究会议论文集》，丛书：ICER '17。计算机协会，2017年，第74–82页。 [在线]。网址：https://doi.org/ 10 1145 3105726 3106169 。/ 。B. A. Becker，“一种有效改进编译器错误消息的方法”，[38] 载于《第47届 ACM 计算机科学教育技术研讨会论文集》。ACM，2016年，第126–131页。 [在线]。网址：https://dl.acm.org/doi/ 10 1145 2839509 2844584 。/ 。T. Barik、D. Ford、E. Murphy-Hill 和 C. Parnin，[39] “编译器应如何向开发者解释问题？”，载于《2018年第26届欧洲软件工程会议与软件工程基础研讨会联合会议论文集》，2018年，第633–643页。[40] H. Potter 和 C. Omar，“Hazel Tutor：引导新手掌握类型驱动开发策略”，2020年，第10页。[41] A. J. Ko 和 B. A. Myers，“利用 Java Whyline 查找程序输出的原因”，载于《SIGCHI计算系统中的人因会议论文集》，丛书：CHI '09。计算机协会，2009年，第1569–1578页。 [在线]。网址：https://doi.org/10.1145/1518701.1518942