arXiv:2304.04937v1 [cs.PL] 11 Apr 2023

OCaml 程序跟踪

Darius Foo 新加坡国立大学
dariusf@comp.nus.edu.sg

Wei-Ngan Chin 新加坡国立大学
chinwn@comp.nus.edu.sg

摘要

本报告将介绍一个用于 OCaml 程序应用级跟踪的框架。我们概述了针对主要技术难题的一种解决方案：与现有方法相比，该方案能够以更低的开销和维护负担记录带类型的值。随后，我们将演示围绕这一方案构建的工具，用于可视化和探索程序执行过程。

1 概述

缺乏良好的调试工具是 OCaml 用户年度调查中经常被提及的痛点 [10]。尽管生态系统中存在各种调试工具和方法（回溯、`#trace`、ocamldebug、ocamli），*printf* 调试仍然是一个重要的选择 [11, 15], ，很可能是因为它易于使用 [15]——在任何情况下都能完整发挥功能。

为改善这一状况，我们实现了一个框架，用于系统地捕获执行轨迹，并提供以不同方式筛选和查看这些执行轨迹的工具，以定位错误。具体而言，我们通过源代码插桩来实现这一点，类似于 Hat [14], 等经典跟踪器：程序会经过转换，在运行时记录其执行过程。跟踪非常适合在生产环境中使用，也能与监控很好地配合（监控可提供驱动执行的输入），并且可以作为*ppx* [1] 预处理器实现，无需修改编译器或运行时。

自动跟踪 OCaml 程序为何如此困难？人们或许会设想使用 ppx，在每次函数调用中插入*printf* 语句。首先，OCaml 尚未提供临时多态机制，因此无法简单地显示值，而必须在每个调用点提供一个打印器。为了自动生成此类打印器，*typpx* [3], *typedppxlib* [13],和 *genprintlib* [12] 等早期工作会在 ppx 编译阶段调用类型检查器（或读取其输出）。这样做的缺点是，生成的代码在预处理后必须再次由类型检查器验证，而且这些扩展严重依赖编译器的实现细节，通常还会将类型检查器的源代码副本作为自带依赖。

接下来人们可能会尝试以一种非类型化的方式记录值，例如使用`Marshal`，并尝试根据运行时可用的信息重构原始值。然而，由于OCaml 在编译时验证类型安全性，运行时仅保留了极少量的类型元数据，例如用于区分变体构造器的标签。这导致不同类型的值使用相同的内存表示。例如，`(Some 1)` 这一`option` 类型的值，与`(Ok 1)` 这一`result` 类型的值无法区分。

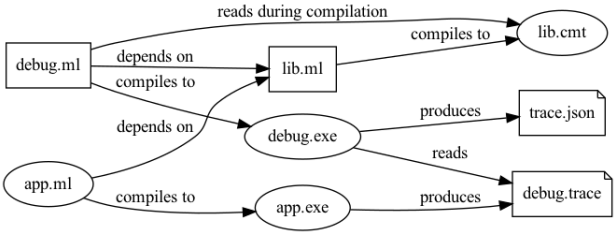


图 1. 修改后的编译流程

内存。任何试图“向程序员展示系统，而非机器”的方法 [8] 因此都必须包含某个编译时组件，以了解用户定义的类型。

我们框架的主要创新在于针对这一问题提出了一种解决方案，即采用替代编译流水线，从而避免多次对代码进行类型检查的成本。框架的其余部分涉及以各种方式分析执行轨迹的工具，其依据的直觉是，单一视图不足以识别每一种漏洞 [15]。我们的愿景是，用户可以借助我们的框架自动为其项目添加插桩，并轻松记录执行轨迹；随后，这些执行轨迹可以与源代码结合进行查询或重放，以协助调试、学习和程序理解。

在演示文稿的其余部分，我们将介绍框架的设计及其所作出的权衡。

2 设计

如前所述，跟踪 OCaml 程序的主要难点在于，如何在不多次运行类型检查器、不对其进行侵入式修改，也不自行维护其源代码的情况下，获取类型信息来构建打印器。

广泛使用的跟踪格式（例如 Com- mon TraceFormat）[2] 提供了部分解决方案：为了使执行轨迹尽可能紧凑，它们将跟踪内容与元数据分离，因此无需浪费空间来编码分隔符等结构元素。元数据会单独传递给解码器，使其能够读取跟踪。在 OCaml 中进行专门化处理时，输出值并不需要类型，只有解释这些值时才需要类型。

这个思路构成了编译流水线的基础，如图 1所示。`lib.ml` 和 `app.ml` 分别代表用户库（没有入口点的模块）和可执行程序 （有入口点的模块）；我们假设`app.ml` 没有缺陷，只有`lib.ml` 需要

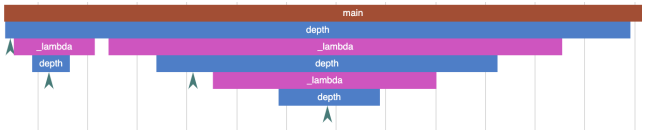


图 2. 火焰图展示 深度

跟踪。`debug.ml` 是用户为使用我们的框架而创建的独立可执行程序。矩形节点由 ppx 处理。

1. `lib.ml` 首先通过 ppx 编译，以便对函数进行插桩，并使用 `Marshal` 序列化值。2. `debug.ml` 通过第二个 ppx 编译，该 ppx 会读取`cmt` 文件`lib.ml`，生成能够理解如何解释已序列化值的代码。3. 运行`app.exe`以生成跟踪。4. 运行`debug.exe` 以读取跟踪，并将其转换为下游工具更易查询的形式。

这实质上为构建过程引入了一种显式的阶段化形式，使一个模块的编译可以依赖另一个模块的类型 。由于使用 `cmt` 文件，后者只需运行一次类型检查器；如果从未读取跟踪，前者也不会产生任何编译开销。这种方法还继承了紧凑跟踪格式的优点，运行时开销更低——即使 OCaml 中出现某种形式的临时多态，这也是继续采用这种方法的理由。维护负担很小，因为除编译器的公共 API 外，只依赖`Cmt_格式` ；无需将类型检查器 vendoring 进来。

这种做法的代价在于，现在变得不标准的是构建过程，而且其表达能力不及诸如 *typedppxlib* [13], 这样的方法，后者可以在编译时以任意方式使用类型。值得注意的是，当 `lib.ml` 的其他部分必须读取依赖于类型的值时，这种显式分阶段处理就行不通了。尽管如此，这条编译流水线或许能用于当前工作以外的场景，例如在 *protobuf* [4] 等 RPC 框架中进行类型安全的反序列化。

3 可视化

有了在 OCaml 程序中输出任意带类型的值的方法后，我们现在来考虑如何呈现这些数据。我们可以输出一个简单的事件序列，但由于我们目前会记录函数参数和返回值，以及 **`match`** 判别对象，因此我们改为将它们整理成调用树，并将其作为下游工具的通用格式。

我们已经原型化了多种用于分析执行轨迹的用户界面。最简单的是批处理命令行工具，可用于打印调用树，或查找特定调用的所有实例（例如`hat-observe` [7]）。其他工具则会生成不同的跟踪格式，例如 Chrome 的跟踪事件格式 [5],，可将其提供给

```
let rec depth t =match twith| Leaf _ -> 0| Node
sub ->List. fold_right (fun ct -> max ( depth c) t) sub
 0 + 1
```

图 3。 一个简单的递归函数

Magic Trace [9] 用于将其渲染为火焰图 [6]。Magic Trace 还提供使用 SQL 查询执行轨迹的功能。

用于计算玫瑰树深度的一个简单递归函数（图 3）渲染结果如图 2所示。在图中，时间向右推进，调用栈向下增长。人字形符号表示**`match`**，展现了调用`depth` 时首先执行**`match`**，随后调用高阶函数`fold_right` （该函数本身不会被跟踪，因为它不在`lib.ml`中）。接着会递归调用 depth，并在返回前执行**`match`**。之后，`fold_right`又会调用高阶函数，依此类推。点击每个条形图即可查看用类 OCaml 语法表示的参数和返回值。参数值随着调用栈向下增长而向下传递，返回值则向上传递。

我们使用VSCode的调试适配器协议实现了另一个可视化工具，允许用户像在交互式调试器中一样前后浏览执行轨迹，因为他们不受实际执行的限制。它结合了经典工具`hat-trail` 和`hat-explore` [7]。

这些工具各有所长，能够支持不同的调试工作流。例如，异常很难在火焰图视图中呈现，因为栈展开难以用纯累加的方式展示：处理程序所在的栈帧可能是火焰图中栈上方的任意一帧。在交互式调试器的用户界面中理解异常则相对简单，因为用户可以看到控制流发生跳转，并按需向前或向后导航，以确认行为。反向执行也有助于从后往前探索计算：从错误出发，回溯并查找其成因的上下文。火焰图视图和命令行工具可用于定位特定的错误调用，并以此作为交互式分析的起点。

4 未来工作

本项目的目标是让人们能够轻松理解 OCaml 代码的行为，并使这种能力广泛适用。除了调试之外，我们还希望执行轨迹的多种交互式视图能够助力探索新的代码库和开展教学。

主要的局限在于，我们的框架要求重新编译要跟踪的项目，以便利用类型信息，因此需要源代码。库代码

也不会被跟踪，因此一旦控制流离开用户代码，它在调试方面就不再有用。因此，它最适用于外部依赖不多的项目；一个很适合的应用领域是编程语言工具。

我们希望研究 Magic Trace（及其对 Intel PT 的使用），看看能否将它与我们为解除这些限制所做的工作结合起来。

引用

[1] OCamlverse 作者。2020。《元编程与 PPX》。检索于 2022 年 6 月 8 日，来源：https://ocamlverse.github.io/content/metaprogramming.html[2] Mathieu Desnoyers。2010 年 2 月 2 日。《通用跟踪格式》。检索于 2022 年 6 月，来源：https://diamon.org/ctf/ [3] Jun Furuse。2022。《typpx》。检索于 2022 年 6 月 2 日，来源：https://github.com/chetmurthy/typpx [4] Google。2001。《Protocol Buffers》。检索于 2022 年 6 月 8 日，来源：https://developers.google.com/protocol-buffers [5] Google。2016。《Trace Event Format》。检索于 2022 年 6 月 3 日，来源：https://docs.google.com/document/d/1CvACIvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview [6] Brendan Gregg。2011。《火焰图》。检索于 2022 年 6 月 7 日，来源：https://www.brendangregg.com/flamegraphs.html [7] York Functional Programming Group。2008。《Hat——Haskell 跟踪器》。检索于 2022 年 6 月 3 日，来源：https://www.cs.york.ac.uk/fp/hat/ [8] Mark Halpern。1965。《计算机编程：调试时代开启》。《计算机与自动化》14, 11 (1965), 28–31。[9] Tristan Hume。2022。《magic-trace》。检索于 2022 年 6 月 2 日，来源：https://magic-trace.org/ [10] Xavier Leroy。2020。《OCaml 用户调查 2020》。检索于 2022 年 6 月 2 日，来源：https://www.dropbox.com/s/omba1d8vhljnrcn/OCaml-user-survey-2020.pdf?dl=0 [11] ocaml.org。《调试》。检索于 2022 年 6 月 3 日，来源：https://ocaml.org/docs/debugging [12] progman1。2020。《genprintlib》。检索于 2022 年 6 月 8 日，来源：https://github.com/progman1/genprintlib [13] Eduardo Rafael。2022。《typedppxlib》。检索于 2022 年 6 月 2 日，来源：https://github.com/EduardoRFS/typedppxlib [14] Malcolm Wallace、Olaf Chitil、Thorsten Brehm 和 Colin Runciman。2001。《Haskell 的多视图跟踪：新 Hat》。（2001）。[15] John Whitington。2020。《通过解释调试函数式程序》。博士论文。莱斯特大学。