arXiv:2302.01265v3 [cs.LO] 7 Feb 2023

代数效应与处理器的自动验证框架（扩展版）*

Tiago Soares 和 Mário Pereira

NOVA LINCS，新里斯本大学科学技术学院，葡萄牙

摘要。 代数效应与处理器是一种强大的抽象，可用于构建非局部控制流机制，例如可恢复异常、轻量级线程、协程、生成器和异步输入/输出。所有这些特性都具有非常复杂的语义，因此给演绎验证技术带来了颇具挑战性的问题。事实上，目前提出的、用于演绎验证包含这些构造的程序的技术寥寥无几，而涉及自动证明的技术则更少。本文介绍了对 Cameleer 的扩展。Cameleer 是一款针对 OCaml 代码的演绎验证工具，该扩展支持对代数效应与处理器进行推理。所提出的方法使用异常来模拟效应与处理器的行为，并采用去函数化来处理效应处理器所暴露的延续。

关键词： OCaml· 代数效应 · 自动证明 · 演绎验证 · Cameleer· GOSPEL

1 引言

最新版本的 OCaml 编译器引入了多项新功能，使并行代码和并发代码运行效率更高，也更易于编写。其中包括代数效应与处理器 [22]。简而言之，代数效应与异常类似：当执行某个效应时，程序执行会暂停，控制权交给最近的外围处理器。异常处理器与效应处理器的主要区别在于，后者会公开一个（有界）延续函数，使我们能够返回到执行该效应的位置并恢复执行。

这一工业级语言的扩展意味着，验证工具有很大的动力增加对代数效应与处理器的支持。尽管已有一些研究 [5, 16] 提出了针对这类程序的推理规则，但都没有探讨自动证明。因此，我们提出的研究问题是：必须开发哪些工具和方法

*本研究部分由 HORIZON 2020 Cameleer 项目（玛丽·斯克沃多夫斯卡-居里资助协议编号：897873）和 NOVA LINCS（编号：UIDB/04516/2020）资助

才能将自动演绎证明应用于具有效应与处理器特性的 *OCaml* 程序？

为回答这一问题，我们提出一个框架，用于自动验证使用代数效应的程序。我们将该框架实现为 Cameleer [15], 的扩展。Cameleer 是一款用于验证 OCaml 代码的演绎验证工具，其规范以 GOSPEL [3] （通用 OCaml 规范语言）编写，随后会被转换为 WhyML。WhyML 是 Why3 验证框架所使用的规范与编程语言 [2]。据我们所知，Cameleer 也是唯一一个面向 OCaml 程序的自动化演绎验证平台，因此是这一项目的理想选择。

为实现这一目标，我们必须能够在证明中表示处理器所暴露的延续函数。为此，我们将采用一种把代数效应转换为 Why3 的方法，并在其中使用去函数化 [21], ——这是一种将高阶程序转换为一阶程序的程序变换技术。据我们所知，在证明场景中使用去函数化尚属首创。

本工作的贡献可概括如下：

1. 扩展 GOSPEL，加入描述代数效应与处理器行为所需的逻辑元素；2. 通过去函数化，在 WhyML 中嵌入 OCaml 的处理器与效应；3. 扩展 Cameleer，以自动将带有 GOSPEL 注解的 OCaml 程序转换为这种嵌入，从而能够在 Why3 框架内进行演绎验证；4. 构建一组案例研究，并使用我们提出的方法对其进行验证。

记法。 OCaml 和 GOSPEL 与 WhyML 共享许多语法特征。由于本文会持续展示这三种语言的示例，因此我们在每段代码清单的右侧标注其所用的语言。

2 速览代数效应

在介绍我们如何计划对代数效应进行规约与证明之前，先来看一个模拟可变引用的程序，它实际上并未使用可变状态。为此，我们引入以下两种不同的代数效应：

```
效应 Get：整数效应 Set：整数 -
> 单位类型
```
OCaml

Get 和 Set 效应的作用分别是实现通常的解引用和赋值运算。

上述定义可以解读如下：当执行 Get 效应时，最近的外层处理器会调用相应的延续，并

传入一个整数值作为参数，该值表示引用的当前值。相反，Set 效应会将一个整数（即引用的新值）发送给处理器。处理器将简单地返回单位类型。这意味着，任何后续的 Get 效应都将取回同一个值。

利用这些效应，我们定义以下函数，将引用的值加一：

```
let inc () : unit =                                          OCaml
  let current = perform Get in
  perform (Set (1 + current))
```

要调用此函数，我们必须定义一个处理器来强制执行预期行为。遵循不使用可变状态的规则，我们创建以下匿名函数来表示程序状态：

```
let create_env () : int -> int = match inc () with          OCaml
  | r ->                 fun _ -> r
  | effect Get k ->      fun x -> (continue k x) x
  | effect (Set n) k -> fun _ -> (continue k ()) n
```

这里，create_env 会返回另一个函数，该函数接收引用的初始值，并相应地处理每种效应。如果没有执行任何效应，处理器就会进入第一个分支，并构造一个始终返回结果 r 的函数。

处理 Get 效应时，我们会返回一个函数，该函数以参数 x（即引用的当前值）调用延续 Why3k。需要注意的是，当我们使用 continue 函数调用延续时，处理器仍处于安装状态，这意味着它会处理剩余的任何效应。这种处理器称为*深处理器*，而浅处理程序 最多只能处理一个效应 [7]。这也意味着调用延续会返回另一个函数，因为它就是处理器的返回类型。调用 continue 后，计算会恢复到执行效应的位置，也就是 inc 函数内部。因此，Set 效应会被执行（并由同一处理器的下一个分支处理）。这意味着 continue 会返回一个函数，我们将 x 传给该函数，以确保引用的值保持不变。

处理 Set 效应时，它会返回一个调用延续的函数，而该延续又会再次返回一个新函数。接着，这个函数会应用于 n，即通过该效应传递的值。

Get 和 Set 效应的用法类似于状态单子的用法，例如 Haskell 语言中的用法[1]。事实上，代数效应与单子有许多相似之处，因为这两种构造都为带效应计算提供了抽象。使用效应的优点是，执行效应时采用*直接风格*，这与调用函数十分相似。而使用单子时，则需要将每个操作都封装在单子类型中。此外，效应在执行效应的动作与其实际实现之间提供了清晰的*抽象屏障* [10]。

[1] https://wiki.haskell.org/State_单子

3 规范化效应与处理器并进行推理

证明处理器的正确性会带来一些规范和推理方面的挑战，其中最主要的挑战是证明每次调用 continue 都是可靠的。对于上一节的示例，这意味着要证明，对于每个 Get 效应，处理器都会返回正确的值。为此，我们必须定义一份逻辑契约，规定处理器调用延续时必须遵守的要求。此外，执行效应的函数也必须满足一组要求，才能执行该效应。

在本文中，我们将任何调用 continue 的函数称为服务器，而将任何调用 perform 的函数称为客户端。这两个参与方在触发效应或响应效应时必须遵守的规则，由协议 [5]描述。协议包含前置条件（客户端在触发效应时必须遵守的条件）、后置条件（服务器调用延续时必须遵守的条件），以及一个修改子句，其中列出了授予服务器写入权限的所有变量。

3.1 客户端规范

为了介绍我们如何在 Cameleer 中实现协议，我们使用一个简单的示例：创建一个程序，让客户端向服务器发送一个值。随后，服务器获取该值，将全局引用 p 设为该值，并返回 p 的旧值。为此，我们首先定义该引用，以及一个名为 XCHG 的效应，它接收一个整数值并返回另一个整数值。然后，我们按照前面概述的规则创建一个协议，如下所示：

```
令 p: 整数 引用 = 引用 0效应                                    GOSPEL +OCaml
XCHG: 整数 -> 整数

(*@ 协议XCHG x: 后置条件 !p =x &&
回复 = old !p修改集合 p *)
```

这里，我们定义了一个 GOSPEL 协议，它保证当服务器将控制权交还给客户端时，该引用保存着客户端传递的值 x，而服务器会回复 p 的旧值（因此使用了关键字 reply）。此外，我们还声明服务器可以修改 p 的内容。该协议没有前置条件，因为客户端可以发送的值不受限制。每个协议都以某个效应命名，这意味着它规定了两个参与方在使用该效应进行通信时必须遵守的规则。

现在，让我们继续指定一个使用此效应的客户端。这相当于以下内容：

```
令 xchg (n: 整数) : 整数 =perform                              GOSPEL +OCaml
(XCHGn)(*@ 后置条件!p = n && 结果 =
旧值 !pperforms XCHG*)
```

performs 子句列出了某个函数可能执行的效应。xchg 函数的后置条件与协议的后置条件完全相同，因为它只是调用 perform 来执行效应 XCHG。

3.2 服务器规约

考虑下面这个处理 XCHG 效应的简单 OCaml 处理器：

```
令 服务器 () =                                          GOSPEL +OCaml
tryxchg(xchg 42)with效应
(XCHG n)k->令 old_p = !p
inp := n;continue kold_p
```

至少从非正式的角度来看，很容易发现这个处理器遵守了先前定义的协议。然而，在一般情况下，只有在我们明确规定函数是通过异常 分支退出还是正常结束时，才能确立这一点。我们将这部分规范称为处理器不变式，这是因为此处涉及深处理器，它们以递归方式处理效应。对于 XGHG 服务器，我们给出以下规范：

```
尝试 xchg(xchg 42) 与 ...(*@ try_后置条件!                GOSPEL +OCaml
p  = old !p && 结果  = 42返回 整数 *)
```

上述不变量通过 try_后置条件 提供，这是我们在 GOSPEL 中引入的一个新子句，用于推理处理器。对于上面的示例，每当执行到达异常分支时，我们就知道处理器的最后一条指令是调用 continue。由于处理器仍保持安装状态（同样，这是深层处理器的缘故），它会持续调用 continue ，直到不再执行任何效应。在实践中，这意味着双重调用 xchg 产生的两个效应都会由此处理器处理。第二次调用 continue 后，函数 server 正常结束执行，因此处理器规范允许我们推导出该函数返回 42，且存储在 p 中的值保持不变（即使该引用被写入两次）。最后，returns 关键字指定了此函数的返回类型.....

4 将效应和处理器转换为 WhyML

在介绍了用于规范效应与处理器的基于协议的方法之后，我们现在将重点转向利用此类规范进行演绎验证。本节将介绍一种翻译方案：将带有 GOSPEL 注释、包含效应与处理器的 OCaml 程序，转换为 WhyML 程序。因此，Why3 生成的验证条件必须确保原始 OCaml 程序遵循其规范。这主要意味着要证明客户端和服务器遵守既定协议。为便于说明，我们重新使用第 3 节中的 xchg 示例。

4.1 关于客户端的推理

客户端通过效应发出的每个请求，都必须遵守相应协议的前置条件，并且无论服务器发送何种回复，客户端的后置条件都必须成立。为了证明这些性质，我们的翻译流水线引入了以下函数，用于模拟执行 XCHG 效应：

```
val perform_XCHG (x : int) : int                                    WhyML
  ensures { let reply = result in
            reply = old !p && !p = x }
  writes  { p }
```

perform_XCHG 的规范与 XCHG 协议完全相同。使用此函数，可以轻松地将客户端翻译为 WhyML，如下所示：

```
  let xchg (n : int): 整数后置条件 { !p = n                          WhyML
  && 结果 =旧值 !p }= perform_XCHGn
```

使用 perform_XCHG 函数，我们可以检查请求是否满足协议的前置条件，以及函数的后置条件是否对任何可能的回复都成立。前者很容易证明：由于没有前置条件，客户端可以随时发出请求。至于后者，协议规定，服务器只能在将 p 引用设为客户端发送的值之后才能回复，并且只能返回该引用的旧值。这正是我们在 perform_XCHG 后置条件中所写的内容，因此 xchg 的函数体也符合其规范。

4.2 关于服务器的推理

采用我们的翻译方法，证明客户端与证明调用其他函数的函数并无不同。另一方面，证明服务器则带来了一些有趣的挑战：

– 如何将 OCaml 的效应翻译为 WhyML? – Why3 的 xchg 函数没有任何形式的异常返回，它只是调用了一个模拟处理器行为的函数。我们如何将其嵌入 Why3? – 如何在 Why3 中表示延续，以及 OCaml 的 continue 函数? – 这类延续应当呈现什么样的规范?

为了解决第一个问题，我们只需选择将效应表示为 WhyML 异常。具体来说，我们创建一个异常，使其接收与原始效应相同的参数。对于 XCHG，它接收一个整数值，因此得到以下编码：

```
  异常 XCHG 整数                                                    WhyML
```

采用这种效应表示方式，OCaml 效应处理器会通过异常处理器转换为 WhyML。第 3.2 节中的处理器转换如下：

```
尝试 xchg(xchg 42) 使用                                    WhyML
| XCHG n->令 old_p  =!p
在p: = n; continue k
old_p
```

为了使这个处理器生效，xchg函数必须能够抛出异常。我们的翻译方案会为添加 以下 perform_XCHG 函数添加子句：

```
引发{XCHG }                                                WhyML
```

实际上，我们会将 GOSPEL 规范中的每个perform 子句翻译为 WhyML 引发子句，这意味着 xchg 也会具有相同的子句。

4.3 表示续延

细心的读者可能会注意到，在 WhyML 对处理器的翻译中，延续 k 并未出现。事实上，在 Why3 中表示和处理延续颇具挑战，因为该框架对高阶计算的支持非常有限。

延续本质上是只能通过专用 continue 函数调用的函数。此外，它们只能被调用一次（如果同一个延续被多次调用，OCaml 运行时会抛出异常）。我们在 WhyML 中通过以下记录类型对延续进行编码：

```
类型延续 'a 'b  = 抽象{可变 _有效: 布尔                    WhyML
值; }
```

此声明引入了一种多态延续类型，其参数类型为 'a，结果类型为 'b。布尔字段 _valid 用于检查延续是否已被*continue*调用，以确保一次性行为。从编程的角度来看，类型 continuation 是一种抽象 数据类型。只能在规范或幽灵代码中检查 _valid 的值。

为了将延续用作附带逻辑契约的值，我们采用一种技术 [11, 20]，以便表示和操作延续的规范。这通过以下一对谓词实现：

```
类型 状态                                                  WhyML

谓词 前置（延续 'a 'b）'a 状态
后置条件谓词 post (延续 'a 'b) 'a 状态 状态 'b
```

这些谓词没有特定的定义，而是在需要时于后文中公理化。若延续的前置条件成立，则前置条件谓词成立。该谓词作用于延续的参数，以及延续开始执行时状态的一种表示。另一方面，若延续的后置条件成立，则后置条件谓词成立。该谓词作用于延续的参数、初始状态、延续执行后的状态，以及延续返回的值。状态类型会被实例化为特定程序所操作的那部分可变状态。例如，对于 xchg 示例，状态将被细化为一个仅含单个可变字段 _p 的记录类型，该字段表示存储在引用 p 中的值。

采用这种方法，我们在 Why3 中定义 continue 函数，如下所示：

```
val continue (k : continuation 'a 'b) (arg : 'a) : 'b      WhyML
  requires { k._valid }
  requires { pre k arg {_p = !p} }
  ensures  { post k arg (old {_p = !p}) {_p = !p} result }
  writes   { k._valid, p }
```

调用此函数时，它将遵守延续 k 的契约。此外，它要求延续有效（即尚未被调用）。continue 返回后，延续将失效，因为我们在 写入子句中声明k._valid 字段会被写入，因此之后不能再假定它被设为 true。

这种延续数据类型的编码实际上是一种去函数化形式 [4], ，其中 continue 类似于去函数化过程中生成的 Why3apply 函数。我们的方法有一个重要区别：我们不将延续表示为代数数据类型。相反，我们在其规范中完整地描述这些值，接下来将对此进行说明。

4.4 规范化延续

利用所有这些不同的部分，我们现在可以通过创建一个 continuation 类型的值 k，并使用先前定义的前置条件谓词和后置条件谓词对其进行规范，从而完成 WhyML 证明。对于 XCHG 处理器示例，具体如下：

```
尝试 xchg(xchg 42) with| XCHG n->                          WhyML
val k : 延续 整数 整数 inassume { ...};
let old_p  =!p inp := n;continue k
old_p
```

WhyML val..in构造允许引入局部抽象值，即，而不必指定特定的实现。放在 assume 子句中的条件本质上是对 k 的规范。首先，我们说明该延续有效：

```
假设{k._有效 }                                                    WhyML
```

接下来，使用前置条件谓词，我们说明 k 的前置条件是协议的 后置条件。乍看之下，这似乎有些违背直觉；但协议的后置条件是服务器通过挂起的延续将控制权交给客户端时必须遵守的条件。因此，客户端调用该延续时必须确保满足协议的后置条件。总之，协议的后置条件如下：

```
(*@ protocol XCHG n :                                    GOSPEL +OCaml
    ensures reply = old !p && !p = n *)
```

现在，我们必须将后置条件翻译为 WhyML，并用前置条件谓词对其进行编码，如下所示：

```
假设 { 令效应_p  = !p inforall 回复 状态。                        WhyML
前置条件 k 回复状态  <->回复 = 效应_p &&
状态。_p  = n }
```

在这个子句中，我们说明 k 的前置条件是回复必须等于 eff_p，也就是创建延续时 p 所取的值（换句话说，就是执行效应时的值）。接下来，利用状态类型，我们说明调用 k 时，存储在 p 中的值等于 n，即客户端传入的值。

最后，借助后置条件谓词，我们说明 k 的后置条件就是处理器的后置条件。这是因为调用 k 会重新安装处理器；每次调用 k 都会在退出处理器时终止。再一次，处理器的后置条件为

```
try_ensures !p = old !p && result = 42                   GOSPEL + OCaml
```

要将其转换为 WhyML，我们需要访问调用 xchg 之前 p 所持有的值。为此，我们创建一个辅助变量 init_p，并将处理器的后置条件转换如下：

```
let init_p  =!p intry xchg(xchg 42) with| XCHG n->...            WhyML
assume { forall 回复 old_状态 状态结果. post k 回复
old_状态 状态 结果 <->状态._p  = 3&& 结果  = init_p }
```

值得注意的是，我们没有使用调用延续时的初始状态，也没有使用其后置条件。这是因为这些值仅与延续的前置条件相关。

生成的验证条件。 我们退一步来看，Why3 为生成的 WhyML 程序生成了哪些验证条件，以及从原始 OCaml 程序的角度来看，这些验证条件意味着什么

案例研究	# 验证条件	代码行数/ 规范/ 幽灵代码	证明时间
控制反转	15	20 / 25 / 2	0.74
除法解释器	19	18 / 16 / 2	0.74
Koda-Ruskey 算法	142	26 / 20 / 0	17.60
参考文献	36	37 / 22 / 3	0.91
Xchg	13	13 / 7 / 0	0.64
浅处理程序	13	21 / 7 / 0	0.54

图1：使用扩展版 Cameleer 验证的案例研究汇总。

。对于客户端，Why3 会生成一个验证条件，用于检查 xchg 函数的后置条件是否成立。由于该函数的后置条件与 perform_XCHG 函数的后置条件相同，因此这一点很容易证明。至于服务器，我们会生成四个验证条件。首先，必须证明函数通过非异常分支终止时，处理程序不变量仍然成立。由于处理程序不变量与 xchg 的后置条件相同，因此这一点也能证明。接下来，要证明延续尚未被调用（这显然成立，因为延续创建时即为有效状态），并且延续的前置条件得到满足。鉴于该处理器遵守协议，这个验证条件也能通过。最后，要证明通过异常情况退出时，处理程序不变量仍然成立。由于处理器的最后一条指令是调用延续，而延续的后置条件就是处理程序不变量，因此这一点也能证明。

我们在附录A中概述了将带有效应与处理器的 OCaml 程序翻译为 WhyML 的一般翻译方案。我们认为这种方法具有足够的通用性，因此可以将其应用于任何带有异常的证明感知语言。

5 评估

为了测试我们的方法，我们对若干具有代表性的案例研究进行了规约和验证。除了前文已详细介绍的 XCHG 示例外，我们还验证了以下案例研究：

1. 将高阶迭代器函数转换为生成器。这个过程通常称为控制反转。2. 一种在除零时执行效应的除法解释器。3. 如第2节所示，使用代数效应和深处理器实现可变状态。4. Koda-Ruskey 算法。5. 使用浅处理程序实现深处理器。我们对浅处理程序的支持仍处于实验阶段，因为目前我们还有大量繁琐的验证条件需要处理，以应对未处理效应。

报告中的案例研究均已通过 Alt-Ergo 2.4.2、CVC4 1.8 和 Z3 4.8.6 SMT求解器的组合实现自动验证。图1总结了我们已验证案例研究的重要指标：生成的验证

条件数；OCaml 代码、GOSPEL 规范和幽灵代码的总行数（后者也计入 OCaml 代码行数）；以及重放证明所需的时间（秒）。运行时间是在搭载 Linux Mint 20.1、OCaml 5.0.0 和 Why3 1.5.1 的 Lenovo Thinkpad X1 Carbon 第 8 代笔记本电脑上，运行十次后取平均值测得的。

所有示例，包括 OCaml 实现、GOSPEL 规范和 Why3 证明会话文件，均已公开提供[2]。值得注意的是，本文始终使用一种便于阅读的效应与处理器语法，而 OCaml 编译器并不原生支持该语法（事实上，这是编译器某个面向实验的分支中提出的语法）。不过，我们的所有案例研究均使用 OCaml 5.0.0 所支持的效应与处理器库编写。

一个更为复杂的案例研究。所列举的大多数证明虽然并非微不足道，但都属于规模相对较小的示例。例外是 Koda-Ruskey 算法 [12]，它枚举 n 叉树森林的有效着色。该算法的完整 OCaml 实现和 GOSPEL 规范如图 2 所示。简而言之，树的节点可以着色为白色或黑色。如果一个节点被着色为白色，则其所有子节点都必须为白色；如果被着色为黑色，则其子节点的颜色不受任何限制。每当算法得到一种新的着色时，就会执行一次效应。全局数组 bits 用于表示每一种新的着色。

这不仅是一项极具挑战性的证明，也是我们所知唯一一个使用代数效应实现 Koda-Ruskey 算法的实现。虽然我们不会逐一介绍完整的规范，但关键在于，这个函数的协议是在局部定义的。这是因为我们希望声明：每当执行效应时，森林都具有有效的着色。不过，由于森林是作为参数传入的，我们必须在函数作用域内定义协议，才能引用该参数。我们的证明，即关于森林和着色的辅助定义与引理，紧密遵循了此前针对 Koda-Ruskey 算法一阶栈式版本的 Why3 实现 [6]。

6 相关工作

程序员最常用的静态验证方式是类型检查。一种扩展类型系统能力的方法是添加效应系统 [1]。在普通类型系统中，我们通常会有如下类型签名：$f : \alpha \to \beta$，其中 f 是一个函数，它接收一个类型为 α 的参数，并返回一个类型为 β的值。使用效应系统时，类型签名会多出一个参数：行 ρ。行只是一个签名列表，用于指明该函数会执行哪些效应。借助这些类型信息，我们可以在编译时确定程序是否会执行任何未处理效应，从而消除另一种常见的错误来源。效应系统不仅可用于跟踪代数效应，也可用于跟踪任何类型的非纯行为

[2] https://github.com/mrjazzybread/cameleer/tree/effects/tests

效应 Yield： 单位类型 *OCaml*

```
let koda_ruskey (f : 森林) =let rec enum_eff (f : 森林) ((baton : 森林 list) [@幽灵代码])= match f with| E ->perform Yield| N (i, l, r) ->if bits.(i) = 白色 then begin enum_eff r baton; bits.(i) <- 黑色; try enum_ eff l (r :: baton) with| Yield ->enum_eff r baton; continue k () end else begin begin tryenum_eff l (r :: baton) with| effect Yield -> enum_eff r baton; continue k () end; bits.(i) <- 白色; enum_eff r baton end (*@ 前置条件 valid_nums_forest f (length bits) 前置条件 disjoint_stack f baton 前置条件 any_forest f bits.elts 后置条件 forall i. not mem_forest i f && not mem_stack i baton ->bits [i] =old bits [i]后置条件 inverse(Cons f baton) (oldbits.elts) bits.elts variant f 协议 Yield { 前置条件 valid_ coloring f bits.elts 前置条件 forall i. not mem_stack i baton -> bits [i] = old bits [i]前置条件 unchanged baton (old bits.elts) bits.elts | |inverse baton (old bits.elts) bits.elts 后置条件 forall i. not mem_stack i baton -> bits [i] = old bits [i]后置条件 inverse baton (old bits.elts) bits.elts 修改集合 bits } *) in enum_eff f [](*@ 前置条件 valid_nums_forest f (length bits) 前置条件 white_forest f bits.elts 执行 Yield *)
```

图 2：Koda-Ruskey 算法的实现与规范。

例如可变状态和发散 [13]。这意味着我们可以在类型信息中标明函数是否为纯函数。

尽管效应处理器目前仍相对小众，但已有一些研究使用等式理论对其行为进行建模。这种方法的先驱是 Plotkin 和 Power [17]，他们将效应视为带效应计算之间的等式来进行推理。也有研究将这种代数方法应用于效应与处理器。例如，Plotkin 和 Pretnar [18,19] 开发了一种逻辑，使用这种方法描述效应的语义，并通过判断处理器是否满足一组等式来验证处理器。此外，还有一些实现将这些推理规则嵌入 Coq 这一交互式定理证明器中 [14, 24]。

另一条重要的研究脉络，是将分离逻辑用于推理效应与处理器的并发行为。例如，Timany 和 Birkedal [23] 针对支持 $call/cc$ 的语言，开发了一个用于验证全栈续延的框架。此外，Hinrichsen 等人 [8] 开发了 Coq [9] 的扩展，用于推理使用协议 进行消息传递的程序；这些协议定义了两个参与方在交互过程中必须遵守的规则。受这项研究启发，de Vilhena 和 Pottier [5] 为带有代数效应的程序开发了一个简化版的协议。他们的框架将效应与效应处理器解释为两个参与方之间的通信：引发效应的一方和处理该效应的处理器。为了对这种通信建模，他们使用协议描述通信过程中双方可以相互发送的值。支配这些协议的逻辑建立在 Iris 这一用 Coq 编码的高阶分离逻辑之上。事实上，这些协议正是我们开发的 Cameleer 扩展的主要灵感来源。

7 结论与未来工作

我们介绍了对 Cameleer 工具的扩展，使其能够对代数效应与处理器进行规范描述和推理。这促使我们对 GOSPEL 本身进行了若干扩展，具体包括在规范层面引入协议，以及添加 performs 和 try_ensures 子句。为了探索这种规范方法，我们在 Cameleer 中实现了一个将 OCaml 翻译为 WhyML 嵌入的翻译方案，其中采用异常和延续去函数化的方式，对效应与处理器进行推理。据我们所知，这项对 Cameleer 的扩展是首个为代数效应提供自动化验证途径的框架。

作为未来的工作，我们计划扩展工具以处理隐藏状态，因为目前我们只能处理在顶层定义的可变状态。此外，我们的工具只能证明在调用生成的延续时仍处于安装状态的处理器；我们计划增加对这类处理器的支持：其延续可能产生未处理的效应。最后，我们的长期目标是借助 Cameleer，致力于验证那些大量使用效应的 OCaml 库。

与处理器。一个很好的候选对象是 eio[3], a 库，它为 OCaml 语言提供了直接风格并发原语（通过效应与处理器实现）。

参考文献

1. Bauer, A., Pretnar, M.：代数效应与处理器的效应系统。载于：Heckel, R., Milius, S.（编）《计算机科学中的代数与余代数》。第 1–16 页。Springer Berlin Heidelberg，柏林，海德堡（2013） 2. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.：Why3：管理你的证明器群。Boogie 2011：第一届中间验证语言国际研讨会（2012年5月） 3. Charguéraud, A., Filliâtre, J., Lourenço, C., Pereira, M.：GOSPEL——为 OCaml 提供一种形式化规范语言。载于：《形式化方法——未来三十年——第三届世界大会》。《计算机科学讲义》，第 11800 卷，第 484 501 2019 10 1007 978 3 030 30942 8_29 – 页。Springer（），。4 /- - - - . Danvy, O., Nielsen, L.R.：实践中的去函数化。载于：第三届 ACM SIGPLAN 声明式编程原理与实践国际会议论文集，2001年9月5–7日，意大利佛罗伦萨。第 162–174 页。ACM（2001）。https://doi.org/10.1145/773184.773202, https://doi.org/10.1145/ 773184.773202 5. de Vilhena, P.E., Pottier, F.：效应处理器的分离逻辑。《ACM程序设计语言会议论文集》**5**(POPL)（2021年1月）。https://doi.org/10.1145/3434314, https: //doi.org/10 3434314 6 1145. / . Filliâtre, J., Pereira, M.：形式化地生成森林的所有理想（验证珍珠）。载于：Blazy, S., Chechik, M.（编）《经验证的软件：理论、工具与实验——第八届国际会议，VSTTE 2016，多伦多，加拿大，2016年7月17–18日，修订精选论文》。《计算机科学讲义》，第 9971 卷，第 46–55 页（2016）。https://doi.org/10.1007/978-3-319-48869-1_4, https://doi.org/ 10 1007 978 3 319 48869 1_4 . / - - - - . Hillerström, D., 7 Lindley, S.：浅层效应处理器。载于：Ryu, S.（编）《编程语言与系统》。第 415–435 页。Springer International Publishing，Cham（2018） 8. Hinrichsen, J.K., Bengtson, J., Krebbers, R.：Actris：分离逻辑中基于会话类型的推理。《ACM程序设计语言会议论文集》**4**(POPL)（2019年12月）。https:// doi.org/10 3371074 9 1145 3371074, 10 1145 . / https://doi.org/ . / . Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.：Iris：从基础构建——高阶并发分离逻辑的模块化基础。《函数式编程杂志》**28**(e20)（2018）。https: // doi.org/10.1017/S0956796818000151, https://hal.archives-ouvertes.fr/ hal-01945446 10. Kammar, O., Lindley, S., Oury, N.：处理器实战。SIGPLAN Not. **48**(9), 145–158（2013年9月）。https://doi.org/10.1145/2544174.2500590, https://doi. org/10 2500590 11 1145 2544174 . / . . Kanig, J., Filliâtre, J.C.：Who：具有效应的高阶程序验证器。载于：ACM SIGPLAN ML 研讨会。爱丁堡，苏格兰，英国（2009年8月），http: //www.lri.fr/~filliatr/ftp/publis/wml09.pdf

[3] https://github.com/ocaml-multicore/eio

12. Koda, Y., Ruskey, F.：森林偏序集理想的格雷码。《算法杂志》**15**(2), 324–340（1993）。https://doi.org/https://doi.org/ 10.1006/jagm.1993.1044, https://www.sciencedirect.com/science/article/ pii/S0196677483710448 13. Leijen, D.：Koka：使用行多态效应类型编程。《理论计算机科学电子札记》**153**（2014年6月）。https://doi.org/10. 4204 8 14 153/EPTCS. . . Letan, T., Régis-Gianas, Y., Chifflier, P., Hiet, G.：在 Coq 中对含效应和效应处理器的程序进行模块化验证。载于：FM 2018——第 22 届国际形式化方法研讨会。《计算机科学讲义》，第 10951 卷，第 338–354 页。Springer，牛津，英国（2018年7月）。https://doi.org/10.1007/978-3-319-95582-7_20, https://hal.inria.fr/hal-01799712 15. Pereira, M., Ravara, A.：Cameleer：OCaml 演绎验证工具。载于：Silva, A., Leino, K.R.M.（编）《计算机辅助验证——第 33 届国际会议，CAV 2021，线上会议，2021年7月20–23日，论文集，第二部分》。《计算机科学讲义》，第 12760 卷，第 677–689 页。Springer（2021）。https://doi.org/10.1007/978-3-030-81688-9_31, https://doi.org/10.1007/978 3 030 81688 9_31 - - - - . Plotkin, G., Power, J.：代数运算与通用效应。《应用范畴结构》**11**, 69 1023064908962 17 94 02 2003 10 1023 – ( ). https://doi.org/ . /A: . Plotkin, G., Power, J.：计算效应与运算：概述。《理论计算机科学电子札记》**73**, 149–163（2004）。https://doi.org/https://doi.org/10.1016/j.entcs.2004.08.008, https://www.sciencedirect.com/science/article/pii/S1571066104050893，第六届域理论研讨会论文集 18. Plotkin, G., Pretnar, M.：代数效应逻辑。载于：2008年第23届年度 IEEE 计算机科学逻辑研讨会论文集。第 118 129 rd 页。LICS '08，IEEE 计算机学会，美国（2008）。https://doi.org/10.1109/LICS.2008. 45, 45 19 10 1109 2008 https://doi.org/ . /LICS. . . Plotkin, G., Pretnar, M.：代数效应处理器。载于：Castagna, G.（编）《编程语言与系统》。第 80–94 页。Springer Berlin Heidelberg，柏林，海德堡（2009）20. Régis-Gianas, Y., Pottier, F.：按值调用函数式程序的霍尔逻辑。载于：Audebaud, P., Paulin-Mohring, C.（编）《程序构造数学》，第九届国际会议，MPC 2008，法国马赛，2008年7月15–18日，论文集。《计算机科学讲义》，第 5133 卷，第 305–335 页。Springer（2008）。https://doi.org/10.1007/978-3-540-70594-9_17, https: //doi.org/10 1007 978 3 540 70594 9_17 . / - - - - . Reynolds, J.C.：高阶编程语言的定义解释器。载于：ACM 年度会议论文集——第 2 卷。第 717–740 页。ACM '72，美国纽约州纽约市（1972）。https: //doi.org/10 805852 22 1145 800194 805852, 10 1145 800194 . / . https://doi.org/ . / . . Sivaramakrishnan, K., Dolan, S., White, L., Kelly, T., Jaffer, S., Madhavapeddy, A.：为 OCaml 增补效应处理器。载于：第 42 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集。第 206–221 页。PLDI 2021，美国纽约州纽约市（2021）。https://doi.org/10.1145/3453483.3454039, https://doi.org/10 3454039 23 1145 3453483 . / . . Timany, A., Birkedal, L.：并发程序与延续的机械化关系验证。《ACM程序设计语言会议论文集》**3**(ICFP)（2019年7月）。https://doi. org/10.1145/3341709, https://doi.org/10.1145/3341709

24. Xia, L.y., Zakowski, Y., He, P., Hur, C.K., Malecha, G., Pierce, B.C., Zdancewic, S.：交互树：在 Coq 中表示递归程序和非纯程序。ACM程序设计语言会议论文集4(POPL)（2019年12月）。https://doi.org/10.1145/3371119, https://doi.org/10.1145/3371119

一种通用转换方案

$\mathcal{H}$(e$_{0t}$, $\overline{E\bar{x}}$, $\bar{e}$, $\overline{S_{hensures}}$,$term_{post}$, τ, Σ, $term_{state}$) $=$

let handler() =ensures{$S_{hensures_1}$} 后置条件{$S_{hensures_2}$} ... let init_状态 = $S_{\bar{x}}$ in try e_{0t} with|...| $E_n\bar{x}$ ->let eff_状态 = $S_{\bar{x}}$ in val gen_k 单位类型 : 延续 (π_2(Σ(E_n))) τ后置条件{valid result} 后置条件{ forall 参数 状态. pre result 参数 状态 <->E_post 参数 效应_状态 状态 result } 后置条件{ let f =result in forall 参数 irrelevant_old_状态 stateresult. post f 参数 irrelevant_old_状态 状态 result <->let 状态_old = init_状态 in$term_{post} \wedge term_{state}$} in letk = gen_k () in e_{nh}|... in handler ()

WhyML

图 3：效应处理器的转换规则。

$$\mathcal{O}(\epsilon) = \text{真}$$
$$\mathcal{O}(x :: t) = \text{状态}.x = \text{状态_旧}.x \wedge \mathcal{O}(t)$$

图 4：未修改状态

$$\mathcal{T}(\tau_1 \to \tau_2 \to \tau_3) =$$
$$\quad let\ s = \mathcal{T}(\tau_2 \to \tau_3)\ in$$
$$\quad \tau_1 :: \pi_1(s)\ \pi_2(s)$$

$$\mathcal{T}(\tau_1 \to \tau_2 \to \tau_3) =$$
$$\quad (\tau_1 \to \tau_2) = \tau_1,\ \tau_2$$

$$\mathcal{T}(\tau_1 \to \tau_2 \to \tau_3) =$$
$$\mathcal{T}(\tau) = unit,\ \tau$$

图 5：效应类型转换。

$\mathcal{D}\ (args = x_1 :\ \tau_1,\ \ x_2 :\ \tau_2,\ \ ...,\ \ x_n :\ \tau_n)\ \tau_r\ e_t\ \bar{S}\ term_{pre}\ term_{post} =$

```
设 f args  S = e_t 在                                          WhyML
值 生成_f () : λ (τ1, τ2,..., τn)  τr
后置条件{
  forall 参数 状态. pre result 参数 状态  <->
    设 x1,  x2 ...  = 参数在
    term_pre
}
后置条件{
  设 f =结果在
  对于所有参数旧_状态 状态 结果。
  后置函数 参数 旧值_状态 状态 结果  <->
  令 x1,  x2 ...  = 参数在
  term_post
} 在 生成_f()
```

图 6：去函数化函数。

$$\mathcal{S}(term :: xs) = h(term) \wedge \mathcal{S}(xs)$$
$$\mathcal{S}(\epsilon) = true$$

图 7：项的组合

$\mathcal{P}$(E, $\bar{x}$, Σ, $term_{pre}$, $term_{post}$, mod) =WhyML 谓词 pre_E (参数： π_1(Σ(E))) (状态 : state) =let $\bar{x}$ = 参数 in$term_{pre}$谓词 post_E (参数： π_1(Σ(E))) (old_状态 : state) (状态： S_τ) (回复： π_2(Σ(E))) =let $\bar{x}$ = 参数 in$term_{post}$val perform_E (参数: π_1(Σ(E))) : π_2(Σ(E)) 前置条件{pre_E 参数 $S_{\bar{x}}$} 后置条件{post_E 参数 (old $S_{\bar{x}}$) $S_{\bar{x}}$ result} 写入{mod} 图 8：协议转换函数。

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{effect } E \ : \ \tau]\!] \rightsquigarrow \texttt{exception } E \dashv \Sigma[E \to \mathcal{T}(\tau)]} \ (\textsc{TEffect})$$

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{perform E}\bar{\texttt{x}}]\!] \rightsquigarrow \texttt{perform_E } \bar{\texttt{x}} \dashv} \ (\textsc{TPerform})$$

$$\frac{\mathcal{S}(\overline{Sp}_{requires}) = term_{pre} \qquad \mathcal{S}(\overline{Sp}_{ensures}) = term_{post}}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{protocol } E\bar{x} : \overline{Sp}]\!] \rightsquigarrow \mathcal{P}(E,\ \bar{x},\ \Sigma,\ term_{pre},\ term_{post},\ Sp_{modifies}) \dashv} \ (\textsc{TProtocol})$$

$$\frac{\Sigma \circ \Delta \circ \nu \cup \overline{x} \circ \emptyset \vdash [\![e]\!] \rightsquigarrow e_t \dashv \mu' \qquad \mathcal{S}(\bar{S}_{requires}) = pre \qquad \mathcal{S}(\bar{S}_{ensures}) = post}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{fun } \bar{S}\ \overline{(x : \tau_{arg})} : \tau_{ret} \to e]\!] \rightsquigarrow \mathcal{D}\ \overline{(x : \tau_{arg})}\ \tau_{ret}\ e_t\ \bar{S}\ pre\ post \dashv} \ (\textsc{TFun})$$

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{performs } E]\!] \rightsquigarrow \texttt{raises}\{E\ arg \to E_pre\ arg\ S_{\bar{x}}\} \dashv} \ (\textsc{TPerformsClause})$$

$$\frac{\forall i\ \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \mu_{i+1} \qquad e_0 = x \implies x \in \nu}{\Sigma \circ \Delta \circ \nu \circ \mu_0 \vdash [\![e_0\ e_1\ ...\ e_n]\!] \rightsquigarrow \texttt{apply } e_0\ (e_2, e_3, ..., e_n) \dashv S_{vars}} \ (\textsc{TAppDefun})$$

$$\frac{\forall i\ \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \mu_{i+1} \qquad \neg x \in \nu}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![x\ e_0\ e_1\ ...\ e_n]\!] \rightsquigarrow x\ e_{0t}\ e_{1t}\ \ldots e_{nt} \dashv \Delta(x) \cup \mu_{n+1}} \ (\textsc{TApp})$$

$$\frac{\begin{array}{c}\forall i.0 < i \leqq n \implies \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \mu_{i+1} \qquad \mathcal{S}(\overline{Sh_{ensures}}) = term_{post} \\ Sh_{returns} = \tau \qquad e_{try} = try\ e_0\ with\ Sh\ \texttt{effect } E\bar{x}\ k \to e\end{array}}{\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![e_{try}]\!] \rightsquigarrow \mathcal{H}(\overline{E\bar{x}},\ \bar{e},\ \overline{Sh_{ensures}},\ term_{post},\ \tau,\ \Sigma,\ \mathcal{O}(S_{vars} \setminus \mu_{n+1})) \dashv \mu_n} \ (\textsc{TTry})$$

$$\frac{\begin{array}{c}\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![e]\!] \rightsquigarrow e_t \dashv \mu \\ e \neq fun\ ... \qquad \bar{x} \neq \epsilon \implies \Delta' = \Delta[f \to \mu] \qquad \bar{x} = \epsilon \implies \Delta' = \Delta\end{array}}{\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![\texttt{let } \bar{S}\ f\ \bar{x} = e]\!] \rightsquigarrow \texttt{let } \bar{S}\ f\ \bar{x} = e_t \dashv \Delta'} \ (\textsc{TLet})$$

$$\frac{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![e_1]\!] \rightsquigarrow e_{1t} \dashv \mu' \qquad \Sigma \circ \Delta \circ \nu \circ \mu' \vdash [\![e_2]\!] \rightsquigarrow e_{2t} \dashv \mu''}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{if } a \texttt{ then } e_1 \texttt{ else } e_2]\!] \rightsquigarrow \texttt{if } a \texttt{ then } e_{1t} \texttt{ else } e_{2t} \dashv \mu''} \ (\textsc{TIf})$$

$$\frac{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![e_1]\!] \rightsquigarrow e_{1t} \dashv \mu' \qquad \Sigma \circ \Delta \circ \nu \circ \mu' \vdash [\![e_2]\!] \rightsquigarrow e_{2t} \dashv \mu''}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![e_1\ ;\ e_2]\!] \rightsquigarrow e_1\ ;\ e_2 \dashv \mu''} \ (\textsc{TSeq})$$

$$\frac{\begin{array}{c}\bar{x} \neq \epsilon \implies \mu = \emptyset \\ \Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![e_1]\!] \rightsquigarrow e_{1t} \dashv \mu' \qquad e \neq fun\ ... \qquad \bar{x} \neq \epsilon \implies \Delta' = \Delta[f \to \mu'] \\ \bar{x} = \epsilon \implies \Delta' = \Delta \qquad \Sigma \circ \Delta' \circ \nu \circ \mu' \vdash [\![e_2]\!] \rightsquigarrow e_{2t} \dashv \mu''\end{array}}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{let } \bar{S}\ f\ \bar{x} = e_1 \texttt{ in } e_2]\!] \rightsquigarrow \texttt{let } \bar{S}\ f\ \bar{x} = e_{1t} \texttt{ in } e_{2t} \dashv \mu''} \ (\textsc{TLetIn})$$

$$\frac{\forall i.0 \leqq i < n \implies \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \Sigma \circ \Delta \circ \nu \circ \mu_{i+1}}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{match } a \texttt{ with } \overline{p \to e_i}]\!] \rightsquigarrow \texttt{match } a \texttt{ with } \overline{p \to e_{it}} \dashv \Sigma \circ \Delta \circ \nu \circ \mu_n} \ (\textsc{TMatch})$$

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{modifies } \bar{s}]\!] \rightsquigarrow \texttt{writes}\{s\} \dashv} \ (\textsc{TModifies}) \qquad \Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\epsilon]\!] \rightsquigarrow \epsilon \dashv \ (\textsc{TEmpty})$$

$$\frac{\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![d]\!] \rightsquigarrow d_t \dashv \Sigma' \circ \Delta' \qquad \Sigma' \circ \Delta' \circ \nu \circ \emptyset \vdash [\![\bar{d}]\!] \rightsquigarrow d_{tl} \dashv \Sigma'' \circ \Delta''}{\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![d :: \bar{d}]\!] \rightsquigarrow d_t :: d_{tl} \dashv \Sigma'' \circ \Delta''} \ (\textsc{TDecl})$$

图9：归纳翻译规则。