程序语言语义学：数数手指一二三

Graham Hutton
原作者
诺丁汉大学

Chuigda Whitegive
翻译
Doki Doki λ CLub!

CAIMEO
翻译提议、校对

gjz010
Typst 技术支持

Gemini
校对
Google Deepmind

Claude
校对
Anthropic

译者前言

本文是文章 Programming language semantics: It's easy as 1, 2, 3 的中文翻译，部分字句有所改动。术语 *(terminology)* 在正文中第一次出现的地方以仿宋体（中文）或 *Italic (English)* 呈现，如果某个术语难以辨认，则总是会以仿宋体呈现。如遇翻译或排版质量问题，请在 https://github.com/chuigda/CoreBlogPHP-NG/issues 向译者报告。

摘要

程序语言语义学 *(programming language semantics)* 是计算机科学理论领域的重要话题之一，但新手却常常在入门时面临挑战。本文正是一篇程序语言语义学的入门教程，将整数和加法语言作为一个最小化的框架，以简明的方式呈现一系列语义概念。在这个框架下，一切就像"数数手指一二三"一样简单。

1. 介绍

语义学 *(semantics)* 是对意义 *(meaning)* 进行研究的学科的总称。在计算机科学中，程序语言语义学旨在赋予程序精确的数学意义。在研究新的主题时，从简单的例子开始理解基本概念大有裨益，而本文正是要给出这样一个能被用来演示程序语言语义学中诸多主题的例子：由整数值和加法运算符构成的简单算术表达式语言。

多年以来，这门语言在我自己的工作中扮演着核心角色。起初，它被用来辅助解释语义概念，而随着时间的推移，它也逐渐成为了我发现新思想的机制，并出现在我的许多出版物中。本文的目的是巩固这些经验，并展示如何使用这个简单的算术表达式语言以简单的方式呈现一系列语义概念。

使用最简洁的语言来探索语义概念，正是奥卡姆剃刀原理的一个例证 (Duignan, 2018)。奥卡姆剃刀原理是一种哲学原则，它倾向于用最简单的解释来阐明某种现象。尽管由整数和加法组成的语言没有提供实际编程所需的特性，但它却提供了恰好足以解释许多语义学概念的结构。特别地，整数提供了一个简单的"值"的概念，而加法运算符则提供了一个简单的"计算"的概念。这一语言在过去曾被许多作者使用，例如 McCarthy & Painter (1967), Wand (1982) 和 Wadler (1998)，仅举几例。而本文首次将这一语言作为通用工具来探索不同语义学话题。

当然，也可以使用一个更复杂的最小语言，例如带有可变变量的简单命令式语言，或是基于 λ 演算的简单函数式语言。但这会引入许多其他概念，例如存储、环境、替换和变量捕获。学习这些当然也很重要，但我的经验一次又一次地证明，先将注意力集中在整数和加法的简单语言上大有裨益。只要这个框架下理解了基本概念，读者就可以自行扩展语言，加入其他感兴趣的特性。作者自己的许多工作已经证明了这一方法是行之有效的。

本文以教程形式撰写，不假定读者具备语义学方面的背景知识，面向高年级本科生和博士新生。不过，我希望经验丰富的读者也能从中获得对自己工作有用的灵感。初学者不妨先重点关注第 2 至 7 节，这些章节介绍并比较了一些广泛使用的语义学方法（指称 *denotational*、小步 *small-step*、

语境 *contextual* 和大步 *big-step*），并阐述了如何运用归纳法进行语义推理。而更有经验的读者可能希望直接跳到第 8 节，该节提供了一个扩展示例，展示如何使用续延 *(continuation)* 和去函数化 *(defunctionalisation)* 从语义中系统地推导出抽象机 *(abstract machine)*。

请注意，本文并非旨在全面或深入地阐述语义学，而是旨在总结极简方法的基本思想和优势，并提供进一步阅读的参考资料。本文通篇使用 Haskell 作为元语言来实现语义学思想，这有助于使这些思想更加具体并得以实现。所有代码均可在补充材料中在线获取。

2. 算术表达式

我们首先定义要研究的语言：使用加法运算符 $+$ 由整数集 $\mathbb{Z}$ 构建的简单算术表达式。形式化地说，这些表达式的语言 E 可由如下上下文无关文法定义：

$$E ::= \mathbb{Z} \mid E + E$$

也就是说，一个表达式要么是一个整数值，要么是两个子表达式相加。表达式以常规的文本形式书写，例如 $1 + (2 + 3)$ ——我们假设可以根据需要自由使用括号以消除歧义。表达式的文法也可被直接翻译成 Haskell 数据类型声明，这里我们使用内建的 `Integer` 类型来表示任意精度的整数：

```
data Expr = Val Integer | Add Expr Expr
```

例如，表达式 $1 + 2$ 可以被表示为 Haskell 项 `Add (Val 1) (Val 2)`。从现在起，我们主要考虑用 Haskell 表示的表达式。

3. 指称语义

在文章的第一部分，我们展示了如何使用我们简单的表达式语言，来解释和比较多种不同的为语言赋予语义的方法。在本节中，我们将探讨语义学的指称方法 (Scott 和 Strachey, 1971)——使用值化函数 *(valuation function)* 将语言中的词项 *(term)* 映射到适当的语义域 *(semantic domain)* 中的值 *(value)* 来定义词项的意义。

形式化地说，对于由语法项 *(syntactic term)* 构成的语言 T，其指称语义由两部分组成：一个语义值 *(semantic value)* 集合 V，以及一个类型为 $T \to V$ 的值化函数，该函数将词项映射到以值表示的意义。值化函数通常写作 $[\![t]\!]$——将词项用语义括号 *(semantic bracket)* 括起来，表示对词项 t 应用值化函数的结果。语义括号也被称作牛津括号或斯特雷奇括号，以纪念克里斯托弗·斯特雷奇在指称方法上的开创性工作。

值化函数必须是组合性的 *(compositional)*：复合词项的意义完全由其子项的意义决定。组合性确保了语义的模块化，因而有助于理解，同时也支持了使用简单的等式推理来证明语义的性质。当语义值集合明确时，指称语义常被视同于其值化函数。

类型 `Expr` 的算术表达式有一个非常简单的指称语义：设语义域 V 为 Haskell 的整数类型 `Integer`，并按以下两个等式，定义类型为 `Expr -> Integer` 的值化函数：

$$\begin{aligned} [\![\texttt{Val n}]\!] &= n && [V\text{-Val}] \\ [\![\texttt{Add x y}]\!] &= [\![\texttt{x}]\!] + [\![\texttt{y}]\!] && [V\text{-Add}] \end{aligned}$$

等式 $[V\text{-Val}]$ 声明整数的值就是该整数自身，而等式 $[V\text{-Add}]$ 声明一个加法表达式的值是其两个子表达式的值相加。这一定义显然满足组合性要求，因为复合表达式 `Add x y` 的意义完全是由将运算符 $+$ 应用于两个子表达式 `x` 和 `y` 各自的意义定义的。

组合性简化了推理过程，因为它允许等价代换 *(replace equals by equals)*。例如，我们的表达式语义满足以下特性：

$$\frac{[\![\texttt{x}]\!] = [\![\texttt{x'}]\!] \qquad [\![\texttt{y}]\!] = [\![\texttt{y'}]\!]}{[\![\texttt{Add x y}]\!] = [\![\texttt{Add x' y'}]\!]}$$

也就是说，我们可以随意将加法表达式的两个参数表达式替换成具有相同意义的其他表达式，而不会改变整个加法的意义。这一性质可用简单的等式推理，根据值化函数的定义和参数表达式的假设 $[\![\texttt{x}]\!] = [\![\texttt{x'}]\!]$ 和 $[\![\texttt{y}]\!] = [\![\texttt{y'}]\!]$ 证得：

$$\begin{aligned}
&[\![\texttt{Add x y}]\!] \\
&= \quad \{\text{规则 } [V\text{-Add}]\} \\
&[\![\texttt{x}]\!] + [\![\texttt{y}]\!] \\
&= \quad \{\text{两条假设}\} \\
&[\![\texttt{x'}]\!] + [\![\texttt{y'}]\!] \\
&= \quad \{\text{规则 } [V\text{-Add}]\} \\
&[\![\texttt{Add x' y'}]\!]
\end{aligned}$$

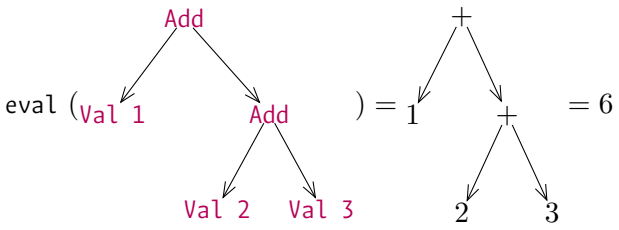
在实践中，由于词项的语义是归纳地构建的，指称语义的证明常用结构归纳法 *(structural induction)* 进行 (Burstall, 1969)。作为例子，让我们看看如何证明这个表达式语言的语义是全函数的 *(total)*：对于任何表达式 e，都存在整数 n，使得 $[\![e]\!] = n$。

全函数性 *(totality)* 的证明通过对表达式 e 的结构进行归纳来进行。对于基准情况 *(base case)* $e = \texttt{Val n}$，根据值化函数的定义，等式 $[\![\texttt{Val n}]\!] = n$ 显然成立。对于归纳情况 *(inductive case)* $e = \texttt{Add x y}$，由归纳假设可得，存在整数 n, m，使得 $[\![\texttt{x}]\!] = n$ 且 $[\![\texttt{y}]\!] = m$，接着应用值化函数，有 $[\![\texttt{Add x y}]\!] = [\![\texttt{x}]\!] + [\![\texttt{y}]\!] = n + m$，从而归纳情况也得证。

值化函数也可以直接翻译成 Haskell 函数定义，只须简单地将数学定义改写成 Haskell 代码：

```
eval :: Expr -> Integer
eval (Val n)   = n
eval (Add x y) = eval x + eval y
```

更一般地说，指称语义可以被视为一个由函数式语言编写的求值器 *(evaluator)* 或解释器。例如，使用上述定义，我们有 `eval (Add (Val 1) (Add (Val 2) (Val 3)))` $= 1 + (2 + 3) = 6$，或者可以这样画成图：



在这个例子中我们注意到表达式的求值方式：将每个 `Add` 构造子 *(constructor)* 替换为整数加法函数 `+`，并移除 `Val` 构造子——或者说，将每个 `Val` 替换成整数上的恒等函数 `id`。这也就是说，尽管函数 `eval` 是递归定义的，因为语义是组合性的，其行为可以被理解为简单地用其他函数替换表达式中的构造子。用这种方式，指称语义也可以被视为一个通过“折叠 *(fold)*”源语言的语法来定义的求值函数：

```
eval :: Expr -> Integer
eval = fold id (+)
```

`fold` 算子 (Meijer et al., 1991) 体现了用其他函数替换语言中构造子的思想。这里，构造子 `Val` 和 `Add` 分别被函数 f 和 g 替换：

```
fold :: (Integer -> a) -> (a -> a -> a) -> Expr -> a
fold f g (Val n) = f n
fold f g (Add x y) = g (fold f g x) (fold f g y)
```

注意由 `fold` 定义的语义在定义上就是组合性的，因为表达式 `Add x y` 的折叠结果完全是将给定的函数 `g` 应用于两个参数表达式 `x` 和 `y` 的折叠结果来定义的。

本节最后我们补充两点。首先，如果我们把文法 $E ::= \mathbb{Z} \mid E + E$ 而不是类型 `Expr` 定义为源语言，那么指称语义写出来就是这样：

$$\begin{aligned} [\![n]\!] &= n \\ [\![x + y]\!] &= [\![x]\!] + [\![y]\!] \end{aligned}$$

在这个版本中，同一个符号 $+$ 现在被用于两个不同的用途：在左边，它是一个语法性 *(syntactic)* 的构造子，用来构造词项；而在右边，它是一个语义算子 *(semantic operator)*，用来计算整数加法。我们选择类型 `Expr` 作为源语言，它提供了专用于构造表达式的构造子 `Val` 和 `Add`，使得语法和语义之间泾渭分明。

其次，注意到，上述语义并未指定求值顺序——也就是说，我们并未指定加法的两个参数应以何种顺序求值。在这个例子中，求值顺序对最后得到的值没有影响。若要显式指定求值顺序，就要向语义中引入额外的结构，我们将在第 8 节讲论抽象机时探讨这一点。

延伸阅读　关于指称语义的标准参考文献是 Schmidt (1986)，而 Winskel (1993) 的形式语义教科书则对该方法进行了简明扼要的介绍。为 λ 演算赋予指称语义的问题，特别是递归定义函数和类型所引出的技术问题，促成了域论的发展 (Abramsky & Jung, 1994)。

Hutton (1998) 进一步探讨了使用折叠算子定义指称语义的思想。简单的整数和加法语言也被用作研究一系列其他语言特性的基础，包括异常 (Hutton & Wright, 2004)、中断 (Hutton & Wright, 2007)、事务 (Hu & Hutton, 2009)、非确定性 (Hu & Hutton, 2010) 和状态 (Bahr & Hutton, 2015)。

4. 小步语义

另一种流行的语义学方法是操作性 *(operational)* 方法 (Plotkin, 1981)。在这一方法中，词项的意义是通过一个执行关系 *(execution relation)* 来定义的，该关系规定了词项如何在一个适当的机器模型上执行。操作语义有两种基本形式：小步 *(small-step)* 语义描述执行的每个步骤，大步 *(big-step)* 语义描述执行的总体结果。在本节中，我们将讲论小步语义——或称“结构化操作语义 *(structural operational semantics)*”，并在第 7 节回到大步语义。

形式化地说，对于由语法项构成的语言 T，其小步语义由两部分组成：一个执行状态 *(execution state)* 集合 S，以及 S 上的转移关系 *(transition relation)*，该关系将每个状态与其经一步执行可抵达的状态关联起来。若两个状态 s 和 s' 处于该关系中，则称存在 s 到 s' 的转移 *(transition)*，记作 $s \longrightarrow s'$。有时也会用到更一般的转移关系概念，但就本文而言，这种简单的概念已经足够了。当执行状态集合明确时，操作语义常被视同于其转移关系。

算术表达式有一个非常简单的小步操作语义：设状态集 S 为表达式的 Haskell 类型 `Expr`，并按以下三条推理规则定义 `Expr` 上的转移关系：

$$\frac{}{\texttt{Add (Val n) (Val m)} \longrightarrow \texttt{Val (n + m)}} \quad [E\text{-Add-Val}]$$

$$\frac{\texttt{x} \longrightarrow \texttt{x'}}{\texttt{Add x y} \longrightarrow \texttt{Add x' y}} \quad [E\text{-Add-L}] \qquad \frac{\texttt{y} \longrightarrow \texttt{y'}}{\texttt{Add x y} \longrightarrow \texttt{Add x y'}} \quad [E\text{-Add-R}]$$

规则 [E-Add-Val] 声明两个值可以相加得到一个值，形如这样的规则被称作*归约 (reduction)* 规则（又称*收缩 contraction* 规则），它规定了基本的操作是如何进行的。一个能与这种规则匹配的表达式称作*可归约表达式 (reducible expression)*，或简称*可归约项 (redex)*。与之相对地，规则 [E-Add-L] 和 [E-Add-R] 允许在加法运算的两侧进行转移，形如这样的规则被称作*结构 (structural)* 规则（又称*同余 congruence* 规则），它们规定了如何归约较大的词项。

注意这个语义是*非确定性 (non-deterministic)* 的，因为一个表达式可能有多种转移。例如，表达式 $(1+2)+(3+4)$（简短起见，这里使用通常的表达式语法）有两个可能的转移，因为在两条结构规则下，归约规则可以被应用于顶层加法的任意一侧：

$$(1+2)+(3+4) \longrightarrow 3+(3+4)$$
$$(1+2)+(3+4) \longrightarrow (1+2)+7$$

这样的转移会改变表达式的语法形式，但其所表示的值——在这个例子里是 10 ——却不会改变。形式化地说，我们现在可以捕捉到表达式的指称语义和小步语义间的一种简单关系：转移不会改变表达式的指称：

$$\frac{e \longrightarrow e'}{[\![e]\!] = [\![e']\!]} \quad [P\text{-Small-Step}]$$

这一性质可以通过对表达式 e 的结构进行归纳来证明：对于基准情况，$e = \texttt{Val n}$，上式显然成立，因为在我们的小步语义中，值没有转移规则，故前提 $e \longrightarrow e'$ 无法成立[1]。

对于归纳情况，我们将对三条推理规则分别进行进一步的分类讨论：

- 如果规则 [E-Add-Val] 适用，也就是前提 $e \longrightarrow e'$ 形如 `Add (Val n) (Val m)` $\longrightarrow$ `Val (n + m)`，箭头两边的求值结果都是 $n+m$，结论 $[\![\texttt{Add (Val n) (Val m)}]\!] = [\![\texttt{Val (n + m)}]\!]$ 成立。
- 如果规则 [E-Add-L] 适用，也就是前提 $e \longrightarrow e'$ 形如 `Add x y` $\longrightarrow$ `Add x' y`，其中存在某个转移使得 `x` $\longrightarrow$ `x'`，我们需要证明 $[\![\texttt{Add x y}]\!] = [\![\texttt{Add x' y}]\!]$。根据归纳假设并结合前提 `x` $\longrightarrow$ `x'`，可得 $[\![\texttt{x}]\!] = [\![\texttt{x'}]\!]$ 成立；由自反性可得，$[\![\texttt{y}]\!] = [\![\texttt{y}]\!]$ 成立；根据上节中加法的等价代换性质，等式 $[\![\texttt{Add x y}]\!] = [\![\texttt{Add x' y}]\!]$ 成立。
- 如果规则 [E-Add-R] 适用，则可以运用与 [E-Add-L] 类似的推理方法，只不过这次我们要转移的表达式是 `y` 而非 `x`。

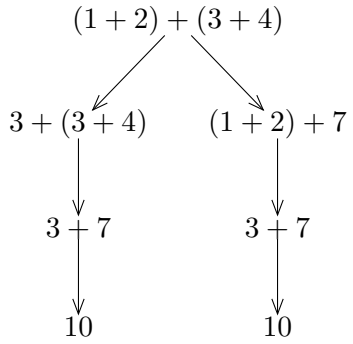
上述证明虽说正确，但却相当繁琐，因为其中涉及到许多分类讨论。下一节会展示如何运用*规则归纳法 (rule induction)* 原理，以一种更直接的方式证明语义之间的关系。

使用小步语义对表达式求值是通过一系列零个或多个转移步骤进行的。这通常通过对转移关系取*自反/传递闭包 (reflexive/transitive closure)* 来形式化地表示，记作 $\xrightarrow{*}$。例如，上述表达式求值为 10 的过程可以记作：

$$(1+2)+(3+4) \xrightarrow{*} 10$$

[1] 译注：在*实质蕴涵 (material implication)* $p \to q$ 中，若前提 p 为假，则蕴涵式 $p \to q$ 恒为真。

通过重复应用转移关系，我们还可以生成一个转移树，该树记录了表达式所有可能的执行路径。例如，上面的表达式会生成以下转移树，它记录了两种可能的执行路径：



转移关系也可以被翻译成 Haskell 函数定义：关系可以表示成一个非确定性函数，该函数返回与给定值相关联的所有值。使用列表推导式可以很容易地定义这样一个函数，该函数接受一个表达式，返回从该表达式经一次转移可以到达的所有表达式的列表：

```haskell
trans :: Expr -> [Expr]
trans (Val n)               = []
trans (Add (Val n) (Val m)) = [Val (n + m)]
trans (Add x y)             = [Add x' y | x' <- trans x] ++
                              [Add x y' | y' <- trans y]
```

与之相对地，我们也可以为转移树定义一个 Haskell 数据类型，并定义一个输入表达式、输出转移树的执行函数，该函数会反复对表达式应用转移函数：

```haskell
data Tree a = Node a [Tree a]

exec :: Expr -> Tree Expr
exec e = Node e [exec e' | e' <- trans e]
```

在这个定义中我们注意到，尽管 `exec` 是递归定义的，但每一层所做的事却非常简单，可以分解为以下两步：

- 生成当前节点：将当前表达式 `e` 直接作为节点的值（即对其应用恒等函数 `id`）；
- 生成子树的种子 *(seed)*：对 `e` 应用转移函数 `trans`，得到一组后继表达式 `[e']`，每个 `e'` 将作为种子，递归地展开为子树。

这两步恰好对应一个通用算子——`unfold` (Gibbons & Jones, 1998)：给定一个种子 `x`，一个提取节点值的函数 `f`，以及一个生成下一层种子的函数 `g`，`unfold` 会自动递归地将种子展开为一棵完整的树：

```haskell
unfold :: (a -> b) -> (a -> [a]) -> a -> Tree b
unfold f g x = Node (f x) [unfold f g x' | x' <- g x]
```

使用 `unfold` 算子，`exec` 可以写成：

```haskell
exec :: Expr -> Tree Expr
exec = unfold id trans
```

换句话说，小步语义 `trans` 只描述单步转移，而 `unfold` 将这些单步转移逐层递归地粘合起来，最终生成完整的转移树。这就是小步操作语义与树展开之间的对应关系。

总之，指称语义对应于“对语法树折叠”，而操作语义则对应于“向转移树展开”。从递归算子的角度审视语义，可以揭示一种容易被忽略的对偶性 *(duality)*，而这种对偶性至今仍未得到应有的广泛认知。

本节最后我们补充三点。首先，如果我们把表达式文法而不是类型 `Expr` 用作源语言，那么语义中的推理规则 [E-Add-Val] 看起来就会是这样：

$$\frac{}{n + m \longrightarrow n + m}$$

这规则着实令人费解——除非我们引入额外的记号，把 $\longrightarrow$ 左边的语法 $+$ 和右边的语义 $+$ 区分开，而这就是 `Expr` 类型在这里的妙用。

其次，注意到，上述语义并未指定求值顺序——或者更准确地说，它涵盖了所有可能的求值顺序。不过如果我们确实想指定一个求值顺序，所需的修改也直截了当。例如，将规则 [E-Add-R] 改成如下形式就能保证加法的第一个参数永远先于第二个参数被求值：

$$\frac{\texttt{y} \longrightarrow \texttt{y'}}{\texttt{Add (Val n) y} \longrightarrow \texttt{Add (Val n) y'}}$$

而指称语义则不同：如前文所述，要在指称语义中明确求值顺序需要额外的结构。能直截了当地指定求值顺序是小步语义的一大优势。

最后，我们使用 Haskell 作为元语言，因此转移关系被间接地实现成了一个非确定函数：`trans` 中等式的顺序很重要，因为其中的模式不是互斥的。而如果我们选择一个具有依值类型的元语言，如 Agda (Norell, 2007)，转移关系则可直接被实现为*归纳族 (inductive family)* (Dybjer, 1994)，定义的顺序就无关紧要了。但为了让语义学知识惠及普罗大众，我们没有选择更复杂的语言，尽管如此，我们也必须认识到这一选择的局限性。

延伸阅读 Plotkin (2004) 对操作语义方法的起源作了综述。当执行的精细结构至关重要时，例如在讨论并发语言 (Milner, 1999)、抽象机 (Hutton & Wright, 2006) 和效率 (Hope & Hutton, 2006) 时，小步方法非常有用。Hutton (1998) 探讨了使用展开运算符定义操作语义的思想，以及它与使用折叠运算符定义的指称语义之间的对偶性。

5. 规则归纳法

指称语义中的基本证明技巧是广为人知的结构归纳法：通过考察词项的语法结构来进行证明。而操作语义中的基本证明技巧是规则归纳法 (Winskel, 1993)，它不那么声名远扬但却同样实用，它通过考察语义规则的结构来进行证明。

我们先用一个简单的数值例子引入规则归纳法的概念，然后展示如何利用规则归纳法简化上一节中的语义证明。我们用以下两条规则归纳定义一个偶自然数集合 $\mathbb{E}$：

$$\frac{}{0 \in \mathbb{E}}\ [\mathbb{E}\text{-Base}] \qquad \frac{n \in \mathbb{E}}{n + 2 \in \mathbb{E}}\ [\mathbb{E}\text{-Ind}]$$

基准情况 [$\mathbb{E}$-Base] 声明 0 在集合 $\mathbb{E}$ 中；归纳情况 [$\mathbb{E}$-Ind] 声明对于 $\mathbb{E}$ 中的任何数 n，数字 $n + 2$ 也在 $\mathbb{E}$ 中。该定义的归纳性质意味着，集合 $\mathbb{E}$ 中除了通过有限次应用这两条规则所能得到的数字之外，没有任何其他东西，这有时被称为该定义的*最小性子句 (extremal clause)*。

对于归纳定义的集合 $\mathbb{E}$，规则归纳法的原则就是：要证明某项性质 P 对 $\mathbb{E}$ 中所有元素成立，只需要先证明基准情况，也就是 P 对于 0 成立；然后再证明归纳情况，也就是若 P 对任意元素 $n \in \mathbb{E}$ 成立，则 P 对 $n + 2$ 也成立。这样一来，我们就有了如下证明规则：

$$\frac{P(0) \qquad \forall n \in \mathbb{E}.\, P(n) \Rightarrow P(n + 2)}{\forall n \in \mathbb{E}.\, P(n)}\ [\mathbb{E}\text{-Rule-Ind}]$$

我们可以用规则归纳法来验证偶数集 $\mathbb{E}$ 对加法的封闭性，即两偶数之和仍是偶数：

$$\forall n \in \mathbb{E}.\, n + n \in \mathbb{E}$$

注意到，这一性质不能用自然数上的数学归纳法来证明，因为该性质仅对偶数成立，而非对任意自然数成立。要证明这一点，首先我们要定义性质 P，然后应用 $\mathbb{E}$-Rule-Ind，最后再展开 P 的定义，得到两个条件：

$$
\begin{aligned}
& \forall n \in \mathbb{E}.\, n + n \in \mathbb{E} \\
\Leftrightarrow \quad & \{\text{定义 } P(n) \Leftrightarrow n + n \in \mathbb{E}\} \\
& \forall n \in \mathbb{E}.\, P(n) \\
\Leftrightarrow \quad & \{\text{规则 } [\mathbb{E}\text{-Rule-Ind}]\} \\
& P(0) \quad \wedge \quad \forall n \in \mathbb{E}.\, P(n) \Rightarrow P(n+2) \\
\Leftrightarrow \quad & \{\text{展开 } P \text{ 的定义}\} \\
& 0 + 0 \in \mathbb{E} \quad \wedge \quad \forall n \in \mathbb{E}.\, n + n \in \mathbb{E} \Rightarrow (n+2) + (n+2) \in \mathbb{E}
\end{aligned}
$$

第一个条件可以被化简为 $0 \in \mathbb{E}$，由规则 [$\mathbb{E}$-Base]，它显然成立。第二个条件可以被重排成 $((n + n) + 2) + 2 \in \mathbb{E}$，这个条件可以从归纳假设 $n + n \in \mathbb{E}$ 出发，应用两次 [$\mathbb{E}$-Ind] 得到。

规则归纳法的思想可以很容易地泛化到存在多个基准情况和归纳情况的场景，也能处理有多个前提条件的规则。例如，对于表达式的小步语义，我们有一个基准情况 [E-Add-Val] 和两个归纳情况 [E-Add-L]、[E-Add-R]：

$$\frac{}{\texttt{Add (Val n) (Val m)} \longrightarrow \texttt{Val (n + m)}} \quad [E\text{-Add-Val}]$$

$$\frac{\texttt{x} \longrightarrow \texttt{x'}}{\texttt{Add x y} \longrightarrow \texttt{Add x' y}} \quad [E\text{-Add-L}] \qquad \frac{\texttt{y} \longrightarrow \texttt{y'}}{\texttt{Add x y} \longrightarrow \texttt{Add x y'}} \quad [E\text{-Add-R}]$$

若要证明性质 $P(e, e')$ 对任意转移 $e \longrightarrow e'$ 成立，就可以使用规则归纳法：

$$\frac{\begin{array}{l} P(\texttt{Add (Val n) (Val m)},\ \texttt{Val (n + m)}) \\ \forall\ \texttt{x} \longrightarrow \texttt{x'}.\, P(\texttt{x}, \texttt{x'}) \Rightarrow P(\texttt{Add x y},\ \texttt{Add x' y}) \\ \forall\ \texttt{y} \longrightarrow \texttt{y'}.\, P(\texttt{y}, \texttt{y'}) \Rightarrow P(\texttt{Add x y},\ \texttt{Add x y'}) \end{array}}{\forall e \longrightarrow e'.\, P(e, e')} \quad [E\text{-Rule-Ind}]$$

也就是说，要证明 $\forall e \longrightarrow e'.\, P(e, e')$，须证明：

- 性质 P 对语义的基准规则 [E-Add-Val] 定义的转移 $\texttt{Add (Val n) (Val m)} \longrightarrow \texttt{Val (n + m)}$ 成立；
- 如果 P 对于归纳规则 [E-Add-L] 前提中的转移 $\texttt{x} \longrightarrow \texttt{x'}$ 成立，则 P 对规则结论中的转移 $\texttt{Add x y} \longrightarrow \texttt{Add x' y}$ 也成立；
- 类似地，如果 P 对于归纳规则 [E-Add-R] 前提中的转移 $\texttt{y} \longrightarrow \texttt{y'}$ 成立，则 P 对规则结论中的转移 $\texttt{Add x y} \longrightarrow \texttt{Add x y'}$ 也成立。

这里我们将 [E-Rule-Ind] 的三个前提垂直排列，并且我们将 $\forall x, y.\, x \longrightarrow y \Rightarrow P(x, y)$ 简记作 $\forall x \longrightarrow y.\, P(x, y)$。

我们可以用以上规则归纳法来验证上一节中指称语义和小步语义的关系 [P-Small-Step]，这一关系可以简写作：

$$\forall e \longrightarrow e'.\, [\![e]\!] = [\![e']\!]$$

我们首先定义性质 P，然后应用 [E-Rule-Ind]，最后再展开 P 的定义，得到三个条件：

$$
\begin{array}{ll}
& \forall e \longrightarrow e'.\, [\![e]\!] = [\![e']\!] \\
\Leftrightarrow & \{\text{定义 } P(e, e') \Leftrightarrow [\![e]\!] = [\![e']\!]\} \\
& \forall e \longrightarrow e'.\, P(e, e') \\
\Leftrightarrow & \{\text{规则 } [E\text{-Rule-Ind}]\} \\
& P(\texttt{Add (Val n) (Val m)},\ \texttt{Val (n + m)}) \ \wedge \\
& \forall\ \texttt{x} \longrightarrow \texttt{x'}.\, P(\texttt{x}, \texttt{x'}) \Rightarrow P(\texttt{Add x y},\ \texttt{Add x' y}) \ \wedge \\
& \forall\ \texttt{y} \longrightarrow \texttt{y'}.\, P(\texttt{y}, \texttt{y'}) \Rightarrow P(\texttt{Add x y},\ \texttt{Add x y'}) \\
\Leftrightarrow & \{\text{展开 } P \text{ 的定义}\} \\
& [\![\texttt{Add (Val n) (Val m)}]\!] = [\![\texttt{Val (n + m)}]\!] \ \wedge \\
& \forall\ \texttt{x} \longrightarrow \texttt{x'}.\, [\![\texttt{x}]\!] = [\![\texttt{x'}]\!] \Rightarrow [\![\texttt{Add x y}]\!] = [\![\texttt{Add x' y}]\!] \ \wedge \\
& \forall\ \texttt{y} \longrightarrow \texttt{y'}.\, [\![\texttt{y}]\!] = [\![\texttt{y'}]\!] \Rightarrow [\![\texttt{Add x y}]\!] = [\![\texttt{Add x y'}]\!]
\end{array}
$$

这三个条件都可以通过在这表达式的指称语义上进行简单的计算来验证：

$$
\begin{array}{l}
[\![\texttt{Add (Val n) (Val m)}]\!] \\
= \quad \{\text{规则 } [V\text{-Add}]\} \\
[\![\texttt{Val n}]\!] + [\![\texttt{Val m}]\!] \\
= \quad \{\text{规则 } [V\text{-Val}]\} \\
n + m \\
= \quad \{\text{规则 } [V\text{-Add}]\} \\
[\![\texttt{Val (n + m)}]\!]
\end{array}
$$

——

$$
\begin{array}{l}
[\![\texttt{Add x y}]\!] \\
= \quad \{\text{规则 } [V\text{-Add}]\} \\
[\![\texttt{x}]\!] + [\![\texttt{y}]\!] \\
= \quad \{\text{归纳假设 } [\![\texttt{x}]\!] = [\![\texttt{x'}]\!]\} \\
[\![\texttt{x'}]\!] + [\![\texttt{y}]\!] \\
= \quad \{\text{规则 } [V\text{-Add}]\} \\
[\![\texttt{Add x' y}]\!]
\end{array}
$$

——

$$
\begin{array}{l}
[\![\texttt{Add x y}]\!] \\
= \quad \{\text{规则 } [V\text{-Add}]\} \\
[\![\texttt{x}]\!] + [\![\texttt{y}]\!] \\
= \quad \{\text{归纳假设 } [\![\texttt{y}]\!] = [\![\texttt{y'}]\!]\} \\
[\![\texttt{x}]\!] + [\![\texttt{y'}]\!] \\
= \quad \{\text{规则 } [V\text{-Add}]\} \\
[\![\texttt{Add x y'}]\!]
\end{array}
$$

本节最后我们补充两点。首先，和第 3 节用结构归纳法所作的证明相比，以上证明更为干净利落：在使用结构归纳法时，对于基准情况 $e = \texttt{Val n}$，我们要利用“值没有转移规则”这一点，通过实质

蕴涵“若前提不成立，则蕴涵式恒成立”的性质来说明 [P-Small-Step] 恒成立；而对于归纳情况 $e = \texttt{Add x y}$，我们则需要就三条推理规则 [E-Add-Val]、[E-Add-L] 和 [E-Add-R] 各自适用的情况作分类讨论。而运用规则归纳法，证明可以直接在转移规则的结构上进行，而不是在表达式的语法结构上进行。

其次，和结构归纳法的证明过程一样，规则归纳法的证明中通常也不会详细地逐条定义性质并陈述所用的归纳原理。例如，上述证明通常会被简化成：简单说一句“对转移 $e \longrightarrow e'$ 进行规则归纳”，然后立刻给出上述三个条件并加以验证。

延伸阅读 Wright (2005) 展示了如何运用规则归纳法来验证简单表达式语言的小步操作语义和大步操作语义的等价性。同样的思想也可以应用于更通用的语言，例如计数求值步骤的 λ 演算 (Hope, 2008) 和支持某种形式的非确定选择的 λ 演算 (Moran, 1998)。

6. 语境语义

第 4 节中所讲论的表达式的小步语义中有一条用于加法的基本归约规则 [E-Add-Val]，以及两条允许在较大表达式中执行加法的结构规则 [E-Add-L] 和 [E-Add-R]。将这两种形式的规则分离，便产生了*语境语义 (contextual semantics)*，又称*归约语义 (reduction semantics)* (Felleisen & Hieb, 1992)。非正式地讲，语境是一个带有空位的词项，空位通常用 $[-]$ 表示，这个空位随后可以被另一个词项填充。在语境语义中，这个空位代表了在词项内执行单个基本步骤的位置。例如，考虑以下小步语义中的转移：

$$(1 + 2) + (3 + 4) \longrightarrow 3 + (3 + 4)$$

在这个例子中，加法在表达式的左子项中进行。这个想法可以精确地表述为：我们可以在语境 $[-] + (3 + 4)$ 中执行基本步骤 $1 + 2 \longrightarrow 3$，这里空位 $[-]$ 表示加法发生的位置。对于算术表达式，其语境 C 可由如下文法形式化地定义：

$$C ::= [-] \mid C + E \mid E + C$$

和之前一样，为了在语法和语义之间划清界限，我们将这个文法翻译成 Haskell 数据结构：

```
data Con = Hole | AddL Con Expr | AddR Expr Con
```

这种语境风格被称为“由外而内”，因为定位空位需要从外向内逐步抽丝剥茧。例如，在语境 c 中用表达式 e 填充空位（记作 $c[e]$）的概念可以如是定义：

$$\begin{aligned}
&\texttt{Hole} && [e] = [e] \\
&(\texttt{AddL}\ c\ \texttt{r}) && [e] = \texttt{Add}\ (c\ [e])\ \texttt{r} \\
&(\texttt{AddR}\ \texttt{l}\ c) && [e] = \texttt{Add}\ \texttt{l}\ (c\ [e])
\end{aligned}$$

也就是说，如果一个语境是一个空位，则我们简单地返回给定的表达式 e；否则，我们适当地在加法的左侧或者右侧递归。请注意，以上是空位填充的数学定义，只是借用了 Haskell 的语境和表达式语法。和前几节一样，我们稍后就会看到如何在 Haskell 中实现它。

利用空位填充这一思想，我们现在可以通过以下两条推理规则，以语境风格重新定义表达式的小步语义：

$$\frac{}{\texttt{Add (Val n) (Val m)} \rightarrowtail \texttt{Val (n + m)}}\ [EC\text{-Add-Val}] \qquad \frac{e \rightarrowtail e'}{c[e] \longrightarrow c[e']}\ [EC\text{-Hole-Fill}]$$

规则 [EC-Add-Val] 定义了归约关系 $\rightarrowtail$，它描述了加法的基本行为；而规则 [EC-Hole-Fill] 定义了转移关系 $\longrightarrow$，它允许在任何语境中——也就是对加法的任意一个参数——应用规则 [EC-Add-Val]。以这种方式，我们可以将小步语义重构为一条归约规则和一条结构规则，而如果

我们要进一步用其他特性扩展这一语言，通常只需要增加新的归约规则并扩展语境的定义，而无需添加新的结构规则。

现在语境语义可以被翻译成 Haskell 代码了。定义空位填充只须将数学定义用 Haskell 语法重写：

```
fill :: Con -> Expr -> Expr
fill Hole       e = e
fill (AddL c r) e = Add (fill c e) r
fill (AddR l c) e = Add l (fill c e)
```

而与之对偶的运算便是将表达式拆分成所有可能的由语境和表达式组成的对子 *(pair)*，可用列表推导式语法如下定义：

```
split :: Expr -> [(Con, Expr)]
split e = (Hole, e) : case e of
    Val n   -> []
    Add l r -> [(AddL c r, x) | (c, x) <- split l] ++
               [(AddR l c, x) | (c, x) <- split r]
```

这一函数的行为可被形式化地描述为：`split e` 所返回的每个对子 `(c, x)` 都满足 `fill c x = e`。用这两个函数，语境语义可被翻译为如下 Haskell 函数，它返回经一步归约或一步转移可抵达的所有表达式的列表：

```
reduce :: Expr -> [Expr]
reduce (Add (Val n) (Val m)) = [Val (n + m)]
reduce _                     = []

trans :: Expr -> [Expr]
trans e = [fill c x' | (c, x) <- split e, x' <- reduce x]
```

函数 `reduce` 实现了加法的归约规则，而 `trans` 如是实现了语境规则：首先将给定表达式拆分成所有可能的语境-表达式对子，然后对这些对子中的每个表达式执行任何可能的归约，最后将归约得到的表达式重新填入语境。

本节最后我们再补充两点。首先，尽管在定义语义时，效率通常不是首要考量，但从计算量的角度来看，以原始形式和语境形式定义的表达式小步操作语义都相当低效。在这些语义中对表达式求值都要重复这一过程：找到下一个可以进行归约的地方，执行归约，然后将产生的表达式填充回去。这显然是一种效率极低的求值方式。

其次，和前面的小步语义一样，语境语义也没有指定加法的求值顺序，因此是非确定性的。不过如果我们确实想指定一个求值顺序，所需的修改也直截了当。例如，对语境的文法作如下修改（并相应地修改空位填充的定义）就能确保加法的第一个参数先于第二个求值：

$$C ::= [-] \mid C + E \mid \mathbb{Z} + C$$

这个版本的语境语义同时也满足唯一分解 *(unique decomposition)* 性质，也就是说任何不是值的表达式 e 都可以被唯一地分解成 $e = c[x]$ 的形式，也就是说，任意表达式都只有至多一个转移。

唯一分解性可通过对表达式 e 归纳来证明。对于基准情况 $e =$ `Val n`，性质显然成立，因为这个表达式已经是一个值了。而对于归纳情况 $e =$ `Add l r`，我们对表达式的两个参数 `l` 和 `r` 作分类讨论：

- 若 `l` 和 `r` 都是值，则 $c = [-], x =$ `Add l r` 是 $e =$ `Add l r` 唯一的分解，因为 e 的两个子项都已经是值了，无法进一步归约。

- 若 `l` 是一个加法表达式，则根据归纳假设，l 可以被唯一分解为 $\texttt{l} = c'[x']$ 的形式。因为语境的文法规定，只有在 `l` 是值的时候才能分解 `r`，故 $c = c' + \texttt{r}$ 和 $x = x'$ 就是 $e = \texttt{Add l r}$ 唯一的分解。
- 最后，若 `l` 形如 `Val n`，而 `r` 是一个加法表达式，则根据归纳假设，`r` 可以被唯一分解为 $\texttt{r} = c'[x']$ 的形式。而 `l` 已经是一个无法继续分解的值了，故 $c = \texttt{n} + c'$ 和 $x = x'$ 就是 $e = \texttt{Add l r}$ 唯一的分解。

稍后在第 8 节中，在讲论将语义变换为抽象机时，我们会看到另一种指定求值顺序的方法。抽象机也为表达式的求值提供了一种更高效的小步方法。

延伸阅读 语境与程序设计和语义学中的许多概念息息相关，包括但不限于用续延显式地表示控制流 (Reynolds, 1972)、使用拉链 *(zipper)* 在数据结构中导航 (Huet, 1997)、从求值器推导出抽象机 (Ager 等, 2003a) 以及对数据结构进行微分 *(differentiating)* (Abbott 等, 2005) 和剖析 *(dissecting)* (McBride, 2008) 的思想。我们会在稍后讲论抽象机时讨论其中的某些主题。

7. 大步语义

小步语义关注每个执行步骤，而大步操作语义 *(big-step operational semantics)* 则规定了如何在一大步中完全地执行词项。形式化地说，对于由语法项构成的语言 T，其大步语义或称“自然语义” (Kahn, 1987) 由两部分组成：一个值集 V，以及一个 T 与 V 之间的求值关系 *(evaluation relation)*，该关系将每个词项与其完全求值后所得到的值关联起来。若词项 t 与值 v 处于该关系中，则称 t 可求值为 v，记作 $t \Downarrow v$。

算术表达式有一个非常简单的大步操作语义：设值集 V 为 Haskell 的整数类型 `Integer`，并按以下两条推理规则定义 `Expr` 和 `Integer` 之间的求值关系：

$$\frac{}{\texttt{Val n} \Downarrow \texttt{n}}\ [E\text{-Val}] \qquad \frac{\texttt{x} \Downarrow \texttt{n} \qquad \texttt{y} \Downarrow \texttt{m}}{\texttt{Add x y} \Downarrow \texttt{n} + \texttt{m}}\ [E\text{-Add}]$$

规则 $[E\text{-Val}]$ 声明一个值 `Val n` 求值的结果是其中的整数 `n`，而规则 $[E\text{-Add}]$ 声明若表达式 `x` 求值为 `n` 且表达式 `y` 求值为 `m`，则 `Add x y` 的求值结果为整数 $\texttt{n} + \texttt{m}$。

和小步语义的转移关系一样，大步语义的求值关系可以翻译成如下 Haskell 函数，该函数返回完全执行给定表达式所能达到的所有值的列表：

```
eval :: Expr -> [Integer]
eval (Val n)   = [n]
eval (Add x y) = [n + m | n <- eval x, m <- eval y]
```

就这个简单的表达式语言而言，大步语义看似与第 3 节所讲论的指称语义如出一辙，不过是把等式换成了推理规则。然而，组合性是指称语义的关键性质，而大步语义不必是组合性的，当讨论更复杂的语言时，这一区分就会变得尤为明显。例如，Bahr 和 Hutton 的 λ 演算编译器 (2015) 就基于一个大步形式的非组合性语义。

我们可以将表达式语言的指称语义和大步语义之间的等价性形式化地写作：

$$[\![e]\!] = n \quad \Leftrightarrow \quad e \Downarrow n$$

也就是说，若表达式可求值为某个整数值，则这个值就是该表达式的指称。要证明这一性质，我们分别讨论两个方向。要证明 $[\![e]\!] = n \Rightarrow e \Downarrow n$，首先用假设 $n = [\![e]\!]$ 替换结论 $e \Downarrow n$ 中的 n，得到 $e \Downarrow [\![e]\!]$，这一性质可通过对表达式 e 运用结构归纳法证明。对于基准情况 $e = \texttt{Val n}$，有：

$$
\begin{aligned}
&\texttt{Val n} \Downarrow [\![\texttt{Val n}]\!] \\
\Leftrightarrow\ &\{\text{规则 } [V\text{-Val}]\} \\
&\texttt{Val n} \Downarrow \texttt{n} \\
\Leftrightarrow\ &\{\text{规则 } [E\text{-Val}]\} \\
&\text{Reflexivity}
\end{aligned}
$$

而对于归纳情况 $e = \texttt{Add x y}$，可作如下推理：

$$
\begin{aligned}
&\texttt{Add x y} \Downarrow [\![\texttt{Add x y}]\!] \\
\Leftrightarrow\ &\{\text{规则 } [V\text{-Add}]\} \\
&\texttt{Add x y} \Downarrow [\![\texttt{x}]\!] + [\![\texttt{y}]\!] \\
\Leftrightarrow\ &\{\text{规则 } [E\text{-Add}]\} \\
&\texttt{x} \Downarrow [\![\texttt{x}]\!] \quad \wedge \quad \texttt{y} \Downarrow [\![\texttt{y}]\!] \\
\Leftrightarrow\ &\{\text{应用归纳假设}\} \\
&\text{Tautology}
\end{aligned}
$$

而对于另一方向 $e \Downarrow n \Rightarrow [\![e]\!] = n$，我们可以先用第 5 节引入的简写形式，将其写作 $\forall e \Downarrow n.\, [\![e]\!] = n$，这一性质可通过对表达式的大步语义运用规则归纳法证明：

$$
\begin{aligned}
&\forall e \Downarrow n.\, [\![e]\!] = n \\
\Leftrightarrow\ &\{\text{定义 } P(e,n) \Leftrightarrow [\![e]\!] = n\} \\
&\forall e \Downarrow n.\, P(e,n) \\
\Leftrightarrow\ &\{\text{对} \Downarrow \text{作规则归纳}\} \\
&P(\texttt{Val n}, \texttt{n}) \quad \wedge \quad \forall x \Downarrow \texttt{n}, y \Downarrow \texttt{m}\ .\, P(x,\texttt{n}) \wedge P(y,\texttt{m}) \Rightarrow P(\texttt{Add } x\ y, \texttt{n} + \texttt{m}) \\
\Leftrightarrow\ &\{\text{展开 } P \text{ 的定义}\} \\
&[\![\texttt{Val n}]\!] = \texttt{n} \quad \wedge \quad \forall x \Downarrow \texttt{n}, y \Downarrow \texttt{m}\ .\, [\![x]\!] = \texttt{n} \wedge [\![y]\!] = \texttt{m} \Rightarrow [\![\texttt{Add } x\ y]\!] = \texttt{n} + \texttt{m}
\end{aligned}
$$

最后的两个条件都可以简单地应用规则 $[V\text{-Val}]$ 和 $[V\text{-Add}]$ 来验证。

延伸阅读 当我们只关注执行的最终结果而不关心执行的具体细节时，大步语义很有用。本文主要关注指称性和操作性的语义学方法，但还有很多其他的语义学方法，包括公理 *(axiomatic)* 语义 (Hoare, 1969)、代数 *(algebraic)* 语义 (Goguen & Malcolm, 1996)、模块化 *(modular)* 语义 (Mosses, 2004)、动作 *(action)* 语义 (Mosses, 2005) 和游戏 *(game)* 语义 (Abramsky & McCusker, 1999)。

8. 抽象机

目前为止，我们的所有例子都侧重于解释语义学概念。而本节将展示如何利用整数和加法的语言来帮助发现语义概念：将这一语言用作基础，发现如何实现抽象机 (Landin, 1964)，从而以明确定义的求值顺序对表达式求值。回顾第 3 节中的简单求值函数：

```
eval :: Expr -> Integer
eval (Val n)   = n
eval (Add x y) = eval x + eval y
```

如前所述，定义并未指定 `Add x y` 中两个参数的求值顺序——求值顺序是元语言 Haskell 的实现决定的。若有必要，可以构建一个用于求值表达式的抽象机来明确求值顺序。

形式化地说，抽象机通常由一组语法重写规则定义，这些规则明确地描述了求值过程中的每一步。在 Haskell 中，可以在适当的数据结构上定义一组一阶尾递归函数来实现抽象机。本节将展示两个重要的语义概念——续延和去函数化，并展示如何使用基于这两个概念的两步过程，从求值函数系统地推导出抽象机。这一方法由 Danvy 及其合作者率先提出 (Ager 等, 2003a)。

8.1. 第一步 – 添加续延

为表达式语言构建抽象机的第一步，就是在语义中明确求值顺序。实现此目标的标准技术是将语义重写为续延传递风格 *(continuation passing style, CPS)* (Reynolds, 1972)。在我们的框架下，续延是将被应用于求值结果的函数。例如，根据表达式的语义，有等式：

$$\texttt{eval (Add x y)} = \texttt{eval x} + \texttt{eval y}$$

当第一个递归调用 `eval x` 被求值时，等式中右边剩下的部分 $+$ `eval y` 可被视为这一求值的续延，也就是说它是一个函数，将会被应用于 `eval x` 的结果。

更形式化地说，对于语义 `eval :: Expr -> Integer`，续延是一个类型为 `Integer -> Integer` 的函数，它会被应用于某步求值所得的整数，并返回一个新的整数。这一类型可进一步泛化为 `Integer -> a`，不过此处暂且不需要这种通用性。以下类型声明表达了这种续延的概念：

```
type Cont = Integer -> Integer
```

接下来要定义新的语义函数 `eval'`，它比 `eval` 多接受一个 `Cont` 类型的参数，这个参数（续延）会被应用于表达式求值的结果。也就是说，函数具有如下类型：

```
eval' :: Expr -> Cont -> Integer
```

而 `eval'` 应有的行为可由以下等式描述：

$$\texttt{eval'}\ e\ c \quad = \quad c\ (\texttt{eval}\ e) \qquad [P\text{-}\texttt{eval'}]$$

也就是说，将 `eval'` 应用于一个表达式和一个续延，相当于将 `eval` 应用于表达式，得到表达式的值，再对该值应用续延。

在大多数演示中，此时会给出 `eval'` 的递归定义，并由此证明上述等式。然而，我们也可以将上述等式视作函数 `eval'` 的一个规范，并在此基础上寻找或者演算出一个满足该规范的定义。需要注意的是，上述规范存在多种可能的解，因为原始语义并未指定求值顺序。下文将展示一种可能的解，但其他解也同样存在。

要求解 `eval'` 的定义，我们从规范 $[P\text{-}\texttt{eval'}]$ 出发，对表达式 e 应用结构归纳法。在每个情况下，我们都从词项 `eval'` $e\ c$ 出发，逐步应用等式推理，将其变换为一个不引用原始语义函数 `eval` 的词项 t，这样我们就可以将 `eval'` $e\ c = t$ 作为 `eval'` 在该情况下的一个定义性等式。对于基准情况 $e =$ `Val n`，演算只须两步：

$$\begin{aligned}
&\texttt{eval' (Val n)}\ c \\
&= \quad \{\text{规范}\ [P\text{-}\texttt{eval'}]\} \\
&c\ (\texttt{eval (Val n)}) \\
&= \quad \{\texttt{eval}\ \text{的定义}\} \\
&c\ \texttt{n}
\end{aligned}$$

如是我们便发现了 `eval'` 在基准情况下的如下定义：

$$\texttt{eval' (Val n)}\ c \quad = \quad c\ \texttt{n}$$

也就是说，若表达式是一个值，则简单地将续延应用于这个值。对于归纳情况 $e = \texttt{Add x y}$，我们也以同样的方式着手：

$$
\begin{aligned}
&\texttt{eval' (Add x y)}\ c \\
&= \quad \{\text{规范 } [P\text{-}\texttt{eval'}]\} \\
&c\,(\texttt{eval (Add x y)}) \\
&= \quad \{\text{展开 eval 的定义}\} \\
&c\,(\texttt{eval x} + \texttt{eval y})
\end{aligned}
$$

此时无法进一步应用定义。不过，因为我们是在进行归纳演算，所以我们还有参数表达式 x 和 y 的归纳假设可用：对于任意续延 c', c''，有 $\texttt{eval' x}\,c' = c'\,(\texttt{eval x})$ 和 $\texttt{eval' y}\,c'' = c''\,(\texttt{eval y})$。要运用这些假设，我们须重写词项中的某些部分，使其形如 $c'\,(\texttt{eval x})$ 和 $c''\,(\texttt{eval y})$。这可以通过用 λ 表达式对 eval x 和 eval y 抽象来实现。有了这些想法，余下的演算便势如破竹：

$$
\begin{aligned}
&c\,(\texttt{eval x} + \texttt{eval y}) \\
&= \quad \{\text{对 eval x 抽象}\} \\
&(\lambda n \to c\,(n + \texttt{eval y}))\ \texttt{eval x} \\
&= \quad \{\text{应用 x 的归纳假设}\} \\
&\texttt{eval' x}\ (\lambda n \to c\,(n + \texttt{eval y})) \\
&= \quad \{\text{对 eval y 抽象}\} \\
&\texttt{eval' x}\ (\lambda n \to (\lambda m \to c\,(n + m))\,(\texttt{eval y})) \\
&= \quad \{\text{应用 y 的归纳假设}\} \\
&\texttt{eval' x}\ (\lambda n \to \texttt{eval' y}\ (\lambda m \to c\,(n + m)))
\end{aligned}
$$

最后所得的词项确是我们需要的形式——其中不包含 eval。如是我们便发现了 eval' 在归纳情况下的如下定义：

$$
\texttt{eval' (Add x y)}\ c \quad = \quad \texttt{eval' x}\ (\lambda n \to \texttt{eval' y}\ (\lambda m \to c\,(n + m)))
$$

也就是说，若表达式形如 Add x y，则先求值其第一个参数 x，记其值为 n，再求值其第二个参数 y，记其值为 m，最后将续延 c 应用于 n 和 m 的和。如此，语义中求值的顺序便明确了。综上所述，我们演算得到了如下定义：

```
eval' :: Expr -> Cont -> Integer
eval' (Val n) c   = c n
eval' (Add x y) c = eval' x (λn -> eval' y (λm -> c (n + m)))
```

最后，若将恒等续延 $\lambda n \to n$ 替换规范 $[P\text{-}\texttt{eval'}]$ 中的续延 c，便可从我们的新语义中还原出原本的语义。也就是说，原本的语义函数 eval 现可定义为：

```
eval :: Expr -> Integer
eval e = eval' e (λn -> n)
```

8.2. 第二步 – 去函数化

在明确求值顺序后，我们距离抽象机更近了；但语义现在变成了一个接受续延作为参数的高阶函数，这却使我们离抽象机更远了。因此第二步正是要尽除续延以恢复原本语义的一阶性，同时保持续延所引入的明确求值顺序。

去函数化 (Reynolds, 1972) 是消除高阶函数参数的标准技术。这一技术基于以下观察：我们不需要支持任意的高阶函数参数，因为被用作实参的高阶函数只有寥寥几种。因此，我们可以用纯数据类型代替函数类型，来表示我们实际所需的高阶函数参数。

函数 `eval` 和 `eval'` 的定义实际上只用到了三种续延：$\lambda n \to n$ 用于结束求值过程；$\lambda n \to$ `eval y' ...` 用于在加法的第一个参数求值完毕时继续求值过程；而 $\lambda m \to c\,(n+m)$ 用于将两个整数相加。我们先定义三个组合子 `halt`、`next` 和 `add`，用于构造这三种形式的续延：

```
-- type Cont = Integer -> Integer

halt :: Cont
halt = λn -> n

next :: Expr -> Cont -> Cont
next y c = λn -> eval' y (add n c)

add :: Integer -> Cont -> Cont
add n c = λm -> c (n + m)
```

每种续延中的自由变量都变成了组合子的参数。使用以上定义，续延语义现可重写为：

```
eval :: Expr -> Integer
eval e = eval' e halt

eval' :: Expr -> Cont -> Integer
eval' (Val n)   c = c n
eval' (Add x y) c = eval' x (next y c)
```

下一步便是定义一种一阶数据类型，其构造子对应于三种组合子：

```
data CONT where
  HALT :: CONT
  NEXT :: Expr -> CONT -> CONT
  ADD  :: Integer -> CONT -> CONT
```

注意到 `CONT` 的构造子和三种 `Cont` 组合子具有相同的名称和类型，只是现在名字换成了大写。而以下函数为数据类型 `CONT` 定义了一个指称语义，形式化地阐述了 `CONT` 类型的值如何用于表示 `Cont` 类型的续延：

```
exec :: CONT -> Cont
exec HALT       = halt
exec (NEXT y c) = next y (exec c)
exec (ADD n c)  = add n (exec c)
```

在文献中这一函数常被称作 `apply` (Reynolds, 1972)：若将 `exec` 的类型展开成 `CONT -> Integer -> Integer`，则这一函数可被视作将续延的表示应用于一个整数，以得到另一个整数。而本文选用 `exec` 的缘由稍后便会揭晓。

下一步就是定义一个新的语义 `eval''`，它和 `eval'` 的行为一致，但使用 `CONT` 类型的值，而不是 `Cont` 类型的续延：

```
eval'' :: Expr -> CONT -> Integer
```

而 `eval'` 的行为可由以下等式描述：

$$\texttt{eval''}\ e\ c \quad = \quad \texttt{eval'}\ e\ (\texttt{exec}\ c\ ) \qquad [P\text{-}\texttt{eval''}]$$

也就是说，将 eval'' 应用于一个表达式 e 和一个以 CONT 表示的续延 c，所得的结果应与将 eval' 应用于该表达式和续延 exec c 的结果相同。

和之前一样，我们对表达式 e 作结构归纳来得到 eval'' 的定义。基准情况 $e =$ Val n 的处理简单直接：

```
  eval'' (Val n) c
= {规范 [P-eval'']}
  eval' (Val n) (exec c )
= {eval' 的定义}
  exec c n
```

而归纳情况 $e =$ Add x y 中则须展开 exec 的定义，从而允许应用归纳假设：

```
eval'' (Add x y) c
= {规范 [P-eval'']}
eval' (Add x y) (exec c )
= {eval' 的定义}
eval' x (next y (exec  c ))
= {exec 的定义}
eval' x (exec (NEXT y  c ))
= {应用 x 的归纳假设}
eval'' x (NEXT y c )
```

然而，exec 的定义中有组合子 next，而 next 的定义中仍然包含 eval'。我们可以对 CONT 参数作分类讨论（无须归纳），为 exec 演算出一个使用 eval''（而非 eval'）的定义：

```
exec HALT n
= {exec 的定义}
halt n
= {halt 的定义}
n
————
exec (NEXT y c) n
= {exec 的定义}
next y (exec c) n
= {next 的定义}
eval' y (add n (exec c))
= {exec 的定义}
eval' y (exec (ADD n c))
= {规范 [P-eval'']}
eval'' y (ADD n c)
————
exec (ADD n c) m
= {exec 的定义}
add n (exec c) m
= {add 的定义}
exec c (n + m )
```

最后，原本的语义 `eval` 可由新语义 `eval''` 经如下演算还原：

$$
\begin{aligned}
&\texttt{eval}\ e \\
=\ &\{\texttt{eval}\ \text{的旧定义}\} \\
&\texttt{eval'}\ e(\lambda n \to n) \\
=\ &\{\texttt{halt}\ \text{的定义}\} \\
&\texttt{eval'}\ e\ \texttt{halt} \\
=\ &\{\texttt{exec}\ \text{的定义}\} \\
&\texttt{eval'}\ e\ (\texttt{exec}\ \texttt{HALT}) \\
=\ &\{\text{规范}\ [P\text{-}\texttt{eval''}]\} \\
&\texttt{eval''}\ e\ \texttt{HALT}
\end{aligned}
$$

总结即得如下新定义：

```
eval :: Expr -> Integer
eval e = eval'' e HALT

eval'' :: Expr -> CONT -> Integer
eval'' (Val n)   c = exec c n
eval'' (Add x y) c = eval'' x (NEXT y c)

exec :: CONT -> Integer -> Integer
exec HALT       n = n
exec (NEXT y c) n = eval'' y (ADD n c)
exec (ADD n c)  m = exec c (n + m)
```

这三个定义和 `CONT` 类型一共构成了一个用于求值表达式的抽象机。这四个组件可分别理解为：

- `CONT` 是控制栈 *(control stack)* 的类型，控制栈中的指令决定抽象机在求值当前表达式后应如何继续。因此，这种抽象机有时也被称为“求值/继续”机 *(“eval/continue” machine)*。控制栈的类型也可重构为一系列指令：

  ```
  type CONT = [INST]
  data INST = ADD Integer | NEXT Expr
  ```

  不过本文仍使用 `CONT` 原本的定义，因为它是以系统的方式得出的，并且只需要声明一种类型。
- `eval` 以给定的表达式和空控制栈 `HALT` 调用 `eval''`，将表达式求值为整数。
- `eval''` 以一个控制栈 `c` 为语境，对表达式求值。若表达式是整数值，则以该整数为参数执行 *(execute)* 控制栈。若表达式是一个加法，则首先求值其第一个参数 `x`，并将指令 `NEXT y` 置于控制栈顶，表示当 `x` 求值完毕时应求值第二个参数 `y`。
- `exec` 以一个整数参数 `n` 为语境，执行控制栈 。若控制栈为空，由指令 `HALT` 表示，则将整数参数作为执行的结果返回；若栈顶是指令 `NEXT y`，则求值表达式 `y` 并将 `ADD n` 置于栈顶，表示当 `y` 求值完毕时，应将当前整数 `n` 与之相加；最后，若栈顶是指令 `ADD n`，这表明加法的两个参数的求值均已完成，则将两数之和作为语境，执行余下的控制栈。

注意 `eval''` 和 `exec` 是互递归的，对应于抽象机的两种模式：抽象机的行动取决于表达式结构还是控制栈。例如，对表达式 $1 + 2$：

```
  eval (Add (Val 1) (Val 2))
= eval'' (Add (Val 1) (Val 2)) HALT
= eval'' (Val 1) (NEXT (Val 2) HALT)
= exec (NEXT (Val 2) HALT) 1
= eval'' (Val 2) (ADD 1 HALT)
= exec (ADD 1 HALT) 2
= exec HALT 3
= 3
```

总而言之，我们展示了如何演算出一个用于求值算术表达式的抽象机，所有实现机制都自然而然地从实现中涌现了出来。我们无须事先了解任何实现思路，因为这些思路是在演算过程中被系统地发现的。

最后我们补充一点：抽象机所用的控制栈和第 6 节语境语义中的语境具有相似的形式。若将控制栈写成普通的代数数据类型：

```
data CONT = HALT | NEXT Expr CONT | ADD Integer CONT
```

并以与第 6 节末尾相同的风格写出指定从左到右求值顺序的求值语境类型：

```
data Con = Hole | AddL Con Expr | AddR Integer Con
```

注意到这两个类型是同构的，也就是说其值之间存在一一对应关系。这一同构表明了求值语境即是去函数化的续延，这并不仅限于此例，而是揭示了一种深层次的语义联系。下文引用的多篇文章对此进行了深入探讨。

延伸阅读 Reynolds 的开创性论文 (1972) 引入了三项关键技术：定义性解释器 *(definitional interpreter)*、续延传递风格和去函数化。Danvy 和他的合作者后来揭示出 Reynolds 的论文实际上包含了从求值器推导出抽象机的蓝图 (Ager 等, 2003a)，并继续就相关主题发表了一系列有影响力的论文，包括从求值器推导出编译器 (Ager 等, 2003b)、从小步语义推导出抽象机 (Danvy & Nielsen, 2004) 以及去函数化的对偶性 (Danvy & Millikin, 2009)；更多参考文献可在 Danvy 的特邀论文 (2008) 中找到。McBride (2008) 利用数据类型剖析的思想，开发了一种将使用 `fold` 算子表示的指称语义转换为等价抽象机的通用方法。

本节基于 (Hutton & Wright, 2006; Hutton & Bahr, 2016)，这些文献也展示了如何演算出扩展后的表达式语言的抽象机，以及如何将两步转换融合成一步。类似的技术可用于为栈机 (Bahr & Hutton, 2015)、寄存器机 (Hutton & Bahr, 2017; Bahr & Hutton, 2020)、类型化语言 (Pickard & Hutton, 2021)，非终止语言 (Bahr & Hutton, 2022) 和并发语言 (Bahr & Hutton, 2023) 演算编译器。

9. 总结

本文用整数和加法的简单语言展示了一系列语义学概念，考察了多种语义学方法、归纳原理如何被用于分析语义，以及如何将语义转换为实现。最小语言的运用规避了更复杂的语言会带来的额外复杂性，使得简明扼要地阐述这些思想成为可能。

当然，使用简单的语言亦有局限。例如这一语言不足以阐述不同语义学方法之间的差异：在讨论算术表达式的大步语义时，我们注意到其与指称语义相去无几，不同之处仅仅是将等式换作推理规则。此外，简单的语言不会像更复杂的语言那样引出语义学上的问题和挑战。例如，从语义角度看，可变状态、变量绑定和并发等特性尤其有趣，当它们组合使用时尤其如此。

对于有兴趣深入了解语义学的读者，有很多优秀的教科书可供选择，例如 (Winskel, 1993; Reynolds, 1998; Pierce, 2002; Harper, 2016)。此外还有各种暑期学校，例如俄勒冈编程语言暑期学校 (OPLSS, 2023) 和米德兰兹研究生院 (MGS, 2022)，以及大量的在线资源。我们希望我们开发的

简单语言能够为其他人提供一个有用的入口和工具，以探索程序语言语义学的更多方面。在这个框架下，一切就像“数数手指一二三”一样简单。

致谢

我要感谢 Jeremy Gibbons, Ralf Hinze, Peter Thiemann, Andrew Tolmach 和众多匿名审稿人，他们给出了许多有用的评论和建议，显著提升了本文的质量。这项工作由 EPSRC 拨款 EP/P00587X/1 资助，项目名称为“关于程序正确性和效率的统一推理”。

补充材料

本文的补充材料参见 http://doi.org/10.1017/S0956796823000072。

参考文献

Abbott, M. G., Altenkirch, T., McBride, C. & Ghani, N. (2005) δ for data: Differentiating data structures. *Fundam. Inform.* 65(1-2), 1–28.

Abramsky, S. & Jung, A. (1994) Domain theory. In *Handbook of Logic in Computer Science*, vol. 3. Clarendon, pp. 1–168.

Abramsky, S. & McCusker, G. (1999) Game semantics. *Comput. Logic* 165, 1–55.

Ager, M. S., Biernacki, D., Danvy, O., & Midtgaard, J. (2003a) A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*.

Ager, M. S., Biernacki, D., Danvy, O. & Midtgaard, J. (2003b) *From Interpreter to Compiler and Virtual Machine: A Functional Derivation*. Research Report RS-03-14. BRICS, Department of Computer Science, University of Aarhus.

Bahr, P. & Hutton, G. (2015) Calculating correct compilers. *J. Funct. Program.* 25.

Bahr, P. & Hutton, G. (2020) Calculating correct compilers II: Return of the register machines. *J. Funct. Program.* 30.

Bahr, P. & Hutton, G. (2022) Monadic compiler calculation. *Proc. ACM Program. Lang.* 6(ICFP), 80–108.

Bahr, P. & Hutton, G. (2023) Calculating compilers for concurrency. *Proc. ACM Program. Lang.* 7(ICFP), 740–767.

Burstall, R. (1969) Proving properties of programs by structural induction. *Comput. J.* 12(1), 41–48.

Danvy, O. (2008) Defunctionalized interpreters for programming languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*.

Danvy, O. & Millikin, K. (2009) Refunctionalization at work. *Sci. Comput. Program.* 74(8), 534–549.

Danvy, O. & Nielsen, L. R. (2004) *Refocusing in Reduction Semantics*. Research Report RS-04-26. BRICS, Department of Computer Science, University of Aarhus.

Duignan, B. (2018) *Occam's Razor*. Encyclopedia Britannica. Available at: https://www.britannica.com/topic/Occams-razor.

Dybjer, P. (1994) Inductive families. *Formal Aspects Comput.* 6(4), 440–465.

Felleisen, M. & Hieb, R. (1992) The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci.* 103(2), 235–271.

Gibbons, J. & Jones, G. (1998) The under-appreciated unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*.

Goguen, J. & Malcolm, G. (1996) *Algebraic Semantics of Imperative Programs*. MIT.

Harper, R. (2016) *Practical Foundations for Programming Languages*, 2nd ed. Cambridge University.

Hoare, T. (1969) An axiomatic basis for computer programming. *Commun. ACM* 12, 576–583.

Hope, C. (2008) *A Functional Semantics for Space and Time*. Ph.D. thesis, University of Nottingham.

Hope, C. & Hutton, G. (2006) Accurate step counting. In *Implementation and Application of Functional Languages*. LNCS, vol. 4015. Berlin/Heidelberg: Springer, pp. 91–105.

Hu, L. & Hutton, G. (2009) Towards a verified implementation of software transactional memory. In *Trends in Functional Programming Volume 9*. Intellect, pp. 129–143.

Hu, L. & Hutton, G. (2010) Compiling concurrency correctly: Cutting out the middle man. In *Trends in Functional Programming Volume 10*. Intellect, pp. 17–32.

Huet, G. (1997) The zipper. *J. Funct. Program.* 7(5), 549–554.

Hutton, G. (1998) Fold and unfold for program semantics. In *Proceedings of the 3rd International Conference on Functional Programming*.

Hutton, G. & Bahr, P. (2016) Cutting out continuations. In *A List of Successes That Can Change the World*. LNCS, vol. 9600. Springer, pp. 187–200.

Hutton, G. & Bahr, P. (2017) Compiling a 50-year journey. *J. Funct. Program.* 27.

Hutton, G. & Wright, J. (2004) Compiling exceptions correctly. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*. LNCS, vol. 3125. Springer.

Hutton, G. & Wright, J. (2006) Calculating an Exceptional Machine. In *Trends in Functional Programming Volume 5*. Intellect, pp. 49–64.

Hutton, G. & Wright, J. (2007) What is the meaning of these constant interruptions? *J. Funct. Program.* 17(6), 777–792.

Kahn, G. (1987) Natural semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*.

Landin, P. (1964) The mechanical evaluation of expressions. *Comput. J.* 6, 308–320.

McBride, C. (2008) Clowns to the left of me, jokers to the right: Dissecting data structures. In *Proceedings of the Symposium on Principles of Programming Languages*.

McCarthy, J. & Painter, J. (1967) Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*. Proceedings of Symposia in Applied Mathematics, vol. 19. American Mathematical Society, pp. 33–41.

Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the Conference on Functional Programming and Computer Architecture*.

MGS. (2022) *Midlands Graduate School in the Foundations of Computing Science*. Available at: http://www.cs.nott.ac.uk/MGS/.

Milner, R. (1999) *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University.

Moran, A. (1998) *Call-By-Name, Call-By-Need, and McCarthy's Amb*. Ph.D. thesis, Chalmers University of Technology.

Mosses, P. (2004) Modular structural operational semantics. *J. Logic Algebraic Program.* 60-61, 195–228.

Mosses, P. (2005) *Action Semantics*. Cambridge University.

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology.

OPLSS. (2023) *Oregon Programming Languages Summer School*. Available at: https://www.cs.uoregon.edu/research/summerschool/archives.html.

Pickard, M. & Hutton, G. (2021) Calculating dependently-typed compilers. *Proc. ACM Program. Lang.* 5(ICFP), 1–27.

Pierce, B. (2002) *Types and Programming Languages*. MIT.

Plotkin, G. (1981) *A Structured Approach to Operational Semantics*. Report DAIMI-FN-19. Computer Science Department, Aarhus University, Denmark, pp. 3–15.

Plotkin, G. (2004) The origins of structural operational semantics. *J. Logic Algebraic Program.* 60-61.

Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*.

Reynolds, J. C. (1998) *Theories of Programming Languages*. Cambridge University.

Schmidt, D. A. (1986) *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc.

Scott, D. & Strachey, C. (1971) *Toward a Mathematical Semantics for Computer Languages*. Technical Monograph PRG-6. Oxford Programming Research Group.

Wadler, P. (1998) *The Expression Problem*. Available at: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

Wand, M. (1982) Deriving target code as a representation of continuation semantics. *ACM Trans. Program. Lang. Syst.* 4(3), 496–517.

Winskel, G. (1993) *The Formal Semantics of Programming Languages: An Introduction*. MIT.

Wright, J. (2005) *Compiling and Reasoning about Exceptions and Interrupts*. Ph.D. thesis, University of Nottingham.