Daml：一种用于安全自动化现实世界多方业务工作流的智能合约语言

Alexander Bernauer、Sofia Faro、Rémy Hämmerle、Martin Huschenbett、Moritz Kiefer、Andreas Lochbihler、Jussi Mäki、Francesco Mazzoli、Simon Meier、Neil Mitchell、Ratko G. Veprek Digital Asset Switzerland

arXiv:2303.03749v1 [cs.PL] 7 Mar 2023

摘要

分布式账本技术（也称为企业区块链）有望显著降低自动化多方业务工作流的高昂成本。我们认为，用于编写此类链上逻辑的编程语言应满足三项要求：(1) 提供用于捕捉支配现实世界业务工作流的法律规则的概念。(2) 提供用于指定访问与授权策略的简单方法。(3) 支持将简单工作流组合成复杂工作流，即使这些简单工作流已经部署。

我们介绍了基于采用严格求值的 Haskell 构建的开源智能合约语言 Daml。Daml 通过提供用于表示、访问和修改账本上数据的新型原语来满足这些要求，这些原语模拟了当今法律体系中的原语。这些原语中包含了稳健的访问与授权策略，而 Daml 内置的授权规则支持委托，这是实现工作流可组合性的关键。这些特性使 Daml 非常适合协调多个彼此不同且异构的参与方之间的业务工作流。

Daml合约可在以下环境中运行：(1) 由数据库支撑的中心化账本上；(2) 采用拜占庭容错共识的分布式部署中；以及 (3) 作为第二层，通过原子提交协议运行于传统区块链之上。

1 引言

通过分布式应用实现自动化，可以提高在不同信任域中运营的多个实体之间开展业务的效率，并降低成本。分布式账本技术（DLT），也称为企业区块链，承诺通过提供具有共享执行逻辑的数字账本，以及在所有参与方之间处理共享数据时提供原子性和一致性保证，大幅降低构建此类自动化的高昂成本。共享逻辑通常由用户提供的程序定义，这些程序称为智能合约，以专用编程语言编写。为了实现为了兑现轻松构建此类自动化的承诺，我们认为智能合约语言应满足三个要求：

现实世界适配性。所有业务工作流都在一个或多个支撑性法律系统的上下文中执行，其法典规定了开展业务的基础规则。例如，大多数法典要求通过要约与承诺订立合约[32], ，并规定在违反合同条款时的救济与损害赔偿。该语言应提供能够表达此类规则的概念，以便将其无缝自动化。

特别是，“代码即法律”这一观点与迄今为止采用的法律体系不相容 [19]：除非能够说服社会认同分布式账本技术状态代表了人们意图发生的事情，否则账本上的状态便无法在法院强制执行。虽然有人尝试相应地调整法律体系（适用于范围有限的用例），但另一种解决方案要求分布式账本技术能够在发生不可预见的现实世界突发事件时支持人工干预，以纠正局面 [25]，即使这会违反编码规则。因此，智能合约语言应明确说明，在执行过程中可以在何处、以何种方式以及由谁进行干预，同时不要求预先了解干预的具体类型，也不削弱任何已承诺的安全保证。

安全。 智能合约为恶意的分布式账本技术参与方提供了广泛的攻击面。不断增长的公开已知智能合约漏洞清单充分证明了这一点[3, 8]。我们认为，一种安全的智能合约语言应满足以下要求：

- 清晰且不存在令人意外的边界情况的语言语义，有助于程序员避免常见的安全陷阱。例如，在以太坊虚拟机（EVM）上发生的重入等非预期执行，至今仍是一个持续存在的安全问题。
- 对数据变更进行适当的授权检查，可确保行为不当的实体无法超出预定范围篡改共享数据。授权策略应与

智能合约代码一并指定，以便二者轻松保持同步。

- 语言的设计应保障机密性，使人们能够轻松、自然地定义存储数据的特定访问策略，并让智能合约程序员能够轻松理解和维护这些策略。

可组合性。通过将已有工作流组合成更复杂的工作流，应能轻松地单方面扩展账本功能。可组合性促进分布式账本技术解决方案的自然发展，并有助于管理复杂性。我们发现了三个挑战：

- 即使工作流已部署并在账本上启动，组合也必须能够实现。
- 需要进行权限委托，以便某个实体能够执行其他实体已预先授权的子工作流。
- 工作流的验证必须自动聚焦于相关子事务，并忽略上下文。

本文介绍 Daml，这是一种函数式智能合约语言，旨在满足上述要求。Daml 是开源的，可从 daml.com 获取。它源自 Haskell，因此继承了该语言的许多特性，例如代数数据类型、类型类、参数多态、单子和高阶函数。

Daml 增添了用于表示、访问和修改账本上数据的新原语。在设计这些原语时，我们从当今法律体系背后的原则中汲取灵感，从而确信常见的业务工作流可以在 Daml 中自然地表示出来。此外，我们刻意避免引入当今法律体系中没有对应概念的新概念。例如，无信任需求的持有者代币和全局状态不变量，在纸面合约的世界中都没有对应物，因此不受支持。

在 Daml 中，授权和访问策略与数据及智能合约代码一同作为原语的一部分进行指定。程序员会为每项账本数据标注一组所有者，以及一组可依据智能合约代码更改该数据的控制方。随后，语义会确保所有者都能看到这些更改。此外，通过将这些原语封装在带有单子操作的 Update 类型构造器中，我们获得了清晰的语言语义；与 Haskell 中的 IO 单子一样，Update 单子有助于将本地 pure 计算与依赖账本状态并对其进行更新的代码分离。

Daml 会编译为核心语言 Daml-LF。Daml-LF 基于 Girard 的 System F_ω [13]。编译器复用格拉斯哥 Haskell 编译器（GHC）的前端来解析、类型检查并对 Daml 代码进行语法脱糖。与 Haskell 不同，Daml 使用严格的按值调用求值，更易于推理和实现；在分布式账本技术（DLT）环境中，惰性求值的好处不大，因为频繁同步无论如何都会触发求值。Daml-LF 代码组织为模块和包，可在 Daml 中通过模块化编译使用。代码通过基于内容的寻址进行引用，即使用加密哈希值唯一标识所引用的代码。

账本使用 Daml 运行时解释 Daml-LF 代码；该运行时实现为一台带有外部账本状态的 CEK 机器。Update 语句会生成一棵交易树，以层级化的方式描述账本状态的变更。树结构使工作流组合能够在语义层面实现：现有工作流的树会成为组合后工作流的子树。在交易验证时，每个组成工作流都可以专注于自己的子树，而忽略其余部分。

Daml 可在多种账本实现上运行，同时确保应用在这些实现之间具有可移植性。由关系型数据库系统（例如 Postgres、Oracle）支持的集中式实现，采用门槛低，开发周期也短。对于去中心化部署，Daml 可在 VMware Blockchain 上支持拜占庭容错状态机复制 [29]；该实现基于 SBFT 协议 [14], , 适用于 Hyperledger Fabric[2], ；此外，Daml 也可用于 Hyperledger Besu [18], （一种基于 IBFT [26], 的企业级以太坊版本）以及 Corda [24]。撰写本文时，领先的金融服务、医疗保健和供应链管理企业已在此类账本上部署 Daml 智能合约，每日处理的交易量达 1000 亿美元+。这表明 Daml 语言能够满足实际需求。此外，由于相同的 Daml 智能合约可在所有这些实现上运行，Daml 还使应用能够在这些不同技术之间移植。

我们的主要贡献如下：

- Daml 和 Daml-LF 中的智能合约原语抽象了当今法律体系中的原则，使程序员能够轻松地用智能合约代码编码现实世界的业务工作流。
- Daml-LF 的数据所有权处理方式及其针对账本变更的授权规则，简化了链上数据读写访问控制的定义。
- 将交易树作为账本变更模型，以便在部署后实现工作流组合。
- 适用于生产环境的 Daml 实现。

本文结构安排如下：我们通过示例介绍 Daml 的主要特性，见 第2节。Daml-LF 的语法、类型系统、语义和授权规则见 第 3 节。我们介绍 Daml 编译器、Daml 运行时和 Daml

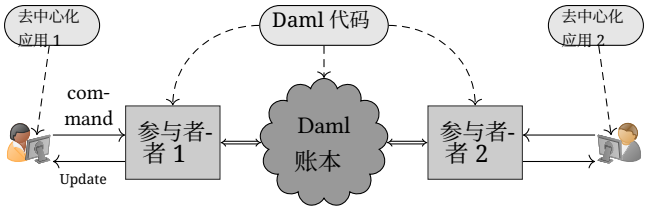

图 1. Daml账本的系统模型

账本，见 第4节。 第5节 讨论相关工作；第6节总结全文。

2 Daml 示例

Daml 是一种开源函数式智能合约语言，采用类似 Haskell 的语法。本节将简要概述 Daml账本的系统模型（第2.1节），然后通过现金借据（“I Owe You”，即“我欠你”）的简单模型，介绍 Daml 语言背后的核心概念和思想（第 2.2–2.5 节）。完整的官方文档可在线查阅：docs.daml.com。

2.1 系统模型

图 1 展示了典型 Daml 账本的系统模型。我们区分分布式应用（去中心化应用）的共享规则和用户策略。只有共享规则通过 Daml 实现，而每个用户自行决定如何表达和维护其在账本上进行操作的策略。策略部分（包括去中心化应用前端）通常使用 Java 或 TypeScript 等主流语言实现。用户通过向账本客户端（称为参与方）发送 Daml 命令来与 Daml 账本交互。参与方通过运行 Daml 智能合约代码来解释此类命令，生成账本更新，然后将其转发至 Daml 账本。Daml 账本负责在其信任假设下执行 Daml 语义。更具体地说，Daml 账本（1）验证账本更新是否符合 Daml 语义；（2）将成功账本更新所影响的参与方告知相关信息，而这些参与方随后会通知其用户的去中心化应用。我们将在第4.3节讨论 Daml 账本实现。目前，只需将 Daml 账本理解为彼此同步的参与方 API 端点即可。

2.2 数据建模

我们从如何在 Daml 中对借据建模开始介绍 Daml。我们使用下面通过关键字 **data** 定义的记录类型Cash 来表示现金金额。该记录类型由记录构造器 Cash 和两个字段组成：类型为 Text （即字符串）的currency ，以及类型为 Decimal （即定点数）的amount 。该定点数的小数点前有 28 位，后有 10 位。

```
数据现金 = 现金 其中货币：文本 金额：十进制数
```

借据表示为智能合约定义的一个实例，即 Daml **模板**。模板以存储在合约实例中的数据（即合约参数）作为参数。对于 SimpleIou，这些数据包括借据的发行方 、借据当前的所有者 ，以及发行方欠所有者的现金 。**ensure** 子句规定了合约参数上的一个不变量，即现金金额必须为正数。

Daml 类型 参与方 表示一种可以在账本上执行操作的身份，也就是说，他们可以签署合约并提交交易。参与方的此类操作如何以密码学方式记录、验证并提供证据，取决于实现。通常，所有 Daml 账本都必须将每笔交易关联到某种密码学证据，以用于授权和不可否认性。将证据与现实世界实体关联起来是另一个独立的问题，可以通过 Daml 工作流来实现。这样，我们就能支持该领域中的多种需求，而这些需求无法通过单一方法满足。

```
模板SimpleIou 其中发行方：参与方 所有者：参与方 现金：现金其中 确保现金.金额 > 0.0观察者所有者签署方发行方选择 SimpleTransfer：合约标识符 SimpleIou 其中newOwner：参与方控制者所有者执行 创建此合约其中所有者 = newOwner
```

观察者和**签署方**子句分别规定了模板合约实例的通知规则和授权规则：创建或归档合约实例时，观察者参与方会收到通知。非空的签署方集合指定了其授权是创建或归档合约所必需的参与方。通常，Daml 账本会保证，每当某个参与方的授权被使用时，该参与方都会收到通知。对于SimpleIou，发行方是签署方，所有者是观察者。无需将发行方另行指定为观察者，因为无论如何，签署方都会在创建和归档时收到通知，因为其授权会被使用。

2.3 编码业务逻辑

在我们的示例中，借据的所有者可以将借据转让给另一参与方。然而，Daml合约使用经过修改的 UTXO 模型，因此具有不可变性。因此，借据转让会归档旧的 SimpleIou 实例，并在一笔 Daml 事务中以原子方式为新所有者创建新的SimpleIou 实例。不过，所有者需要发行方对归档和创建两者的授权，因为

发行方是 SimpleIou合约的签署方。为此，SimpleIou 模板定义了一个 **选择** SimpleTransfer。选择是智能合约的代码入口点，用于定义规定合约实例如何演变的规则。在此，SimpleTransfer 选择返回一个 合约标识符 SimpleIou，即新创建的 SimpleIou 合约实例的标识符。该选择将新所有者作为参数，而关键字 **控制者** 表示执行此选择（即调用此入口点）需要所有者的授权。**do** 代码块包含具体实现，即创建当前借据 this 的副本，并将其 所有者字段更新为 newOwner。**create** 函数返回账本分配给新创建实例的合约 ID。

所有者的授权足以执行借据上的SimpleTransfer 选择，但创建转让后的借据需要发行方的授权（因为发行方是签署方）。因此，这展示了一种权限委托：创建旧借据时，发行方已预先授权模板中各项选择所产生的后果。因此，在选择实现中可以使用发行方的授权，从而使该工作流获得充分授权。

每个模板都会隐式定义一个无参数的归档 选择，并将所有签署方设为控制者，该选择仅会归档一个合约实例。默认情况下，用户定义的选择（例如SimpleTransfer ）也会归档其所作用的合约实例。归档意味着无法再对该合约执行其他选择。此归档 选择确保原则上每个合约实例都可以归档，并随后从账本中删除，也就是说，不会出现“卡住”的 Daml 合约。因此，签署方可以通过共同归档过时的合约实例，并以适当的方式创建新实例，来模拟对账本状态进行的任何现实世界干预。

然而，此归档选择允许发行方随时单方面归档SimpleIou ，因为发行方是唯一的签署方。理想情况下，如果一份借据被从所有者手中收回，或被交给所有者，则所有者应对此表示同意，这遵循合约形成中的提议-接受模式。我们可以在 Daml 中通过让所有者也成为借据的签署方来对此进行建模，如Iou 模板所示。Iou 与SimpleIou 有两点不同：

- 所有者也是签署方。
- 转让选择有两个控制方，即原所有者和新所有者。

执行该选择需要两个控制方的共同授权。这确保选择实现中有足够的授权来**创建**新借据：发行方和新所有者都是签署方。

```
模板借据 with发行方:
参与方 所有者：参与
方 现金：现金
whereensure 现金.金额 > 0.0signatory 发行
方、所有者choice 转让：合约标识符 借据
withnewOwner：参与方controller 所有者、
newOwnerdo create this with 所有者 =
newOwner
```

2.4 提议-接受模式

Iou 合约实例代表一份正式的双边协议，相当于纸面上的法律合同。为达成该协议，发行方可以提出协议，未来的所有者则可以接受该提议。我们在 Daml 中将此类工作流建模为一个独立的IouProposal 模板，其中只有发行方是签署方。借据的条款是Iou类型的模板参数，该类型是同名模板合约参数的记录。拟议借据的所有者控制两个选择：接受会创建Iou 并返回其合约 ID；而拒绝仅归档该提议，并返回单例()。隐式的归档选择允许发行方在其既不接受也不拒绝该提议时撤回提议。

```
模板IouProposal 包含iou : Iou其中
observer iou.owner签署方 iou.issuer
选择 接受: ContractId Iou 控制者
iou.owner执行创建iou选择 拒绝: () 控制
者 iou.owner执行 pure()
```

对于一次转让，借据的原所有者和新所有者可以运行类似的提议工作流，以共同获得行使该选择所需的授权。若要频繁转让，最好在角色合约模板中预先设置好工作流，例如IouSendRole。在这里，接收方 同意接收来自指定发送方的任意借据 转让，并使用发送 选择。关键字**非消耗型** 表示发送选择不会归档IouSendRole 合约实例，因此可以重复使用任意次数。非消耗型选择有助于减少类似 UTXO 的模型中的争用。如果发送 选择是消耗型的，选择体就必须重新创建角色合约，才能转让另一张借据 。这样一来，发送方就无法并发发送多张借据，因为他们必须等到看到分配给重新创建的角色合约的新合约 ID 后才能发送。

do 代码块中的选择体使用 Haskell 风格的 do 记法，依次执行对账本状态的操作。**fetch** 原语会查找给定合约 ID iouId对应的合约参数；如果合约不存在或已归档，则会失败。**assert** 会检查发送方是否确实拥有其想要转让的借据。而

exercise 会使用给定参数，在提供的合约 ID `iouId` 上调用选择项 `Transfer`；如果合约已归档，**exercise** 会像 **fetch** 原语一样失败。

模板IouSendRole **with**发送方：参与方 接收方：参与方**其中观察者为**发送方**签署方为**接收方**非消耗式选择**Send：合约标识符 Iou **with**iouId：合约标识符 Iou**控制者为**发送方**执行**iou <-**获取**iouId**断言**(iou.owner == 发送方)**执行** iouId Transfer**with**newOwner = 接收方

模板TradeProposal **其中**initiator : Party initId : ContractId Iou expected : Iou**其中 let** responder =expected.owner**观察方** responder**签署方** initiator**选择** Settle: (ContractId Iou, ContractId Iou) **其中**respId : ContractId Iou**控制者** responder**执行**responderIou <- **获取**respId**断言**(expected == responderIou) newInitId <- **exercise**respId Transfer**其中** newOwner = initiator newRespId <- **exercise**initId Transfer**其中**newOwner = responder**pure**(newInitId, newRespId)**非消耗式选择** DiscloseIou : Iou**观察方** responder**控制者** initiator**执行 获取**initId

2.5 可组合性

Daml 代码按模块和包组织，包可以使用**import** 关键字依赖其他包。这使得开放式业务架构成为可能，不同实体可以像使用构建模块一样组合其他实体的工作流。例如，金融机构 A 可以在模块`BasicAssets` 中提供带有 `Transfer` 选择的`Iou` 智能合约，并将其作为（已编译的）Daml 包`Basic`发布。用户已经可以创建`Iou`并转让它们。之后，另一个实体 B 可以通过导入`Basic`，在另一个包 `Swap` 中构建两个借据的原子交换。`Swap`一经部署，用户就可以原子交换现有且未更改的`Iou`。尤其是，A 无需升级其服务。这个例子展示了 Daml 的可组合性。

假设两个参与方——发起方和响应方——希望以原子方式交换各自的借据，例如，将美联储发行的 100 美元借据换成由瑞士国家银行支持的 90 瑞士法郎借据。原子性对于降低交易对手风险至关重要：如果没有原子性，一张借据先于另一张转让，那么前一张借据的原所有者就要承担风险：另一方可能在转让第二张借据之前违约或停业，也可能实施欺诈，永远不转让第二张借据。

`转让` 选择已实现单张借据的转让。因此，我们只需在单个 Daml 事务中组合两次此类转让。`TradeProposal` 模板组合了两次此类转让，同样遵循提议-接受模式。发起方指定其借据的合约 ID，以及其希望交换获得的`借据` 条件。响应方，也就是预期`借据`的所有者，可以接受交易提议，并立即使用其 ID 为`respId`的`借据` 结算交易，其中两次**exercise** 调用`转让` 将两次转让汇总为单个事务。末尾的关键字**pure** 定义了主体的结果，即两个新的合约 ID。

在接受提议之前，响应方应检查 `initId` 是否引用了发起方的一张借据，并且响应方愿意接收该借据。由于 Daml 的保密模型，响应方不能直接解析合约 ID 并查看合约参数，因为其中可能包含保密的业务数据。发起方可以通过使用`DiscloseIou` 非消耗型选择向响应方披露借据，并将响应方声明为选择观察者。Daml 会确保每当某项选择被执行时，都会通知选择观察者（这与通知模板中声明的观察者合约创建和归档的方式类似）。因此，响应方会观察到对 `initId` 的**获取**，从而记住合约 ID 与合约参数之间的映射。该映射不可变，因此在执行 `Settle` 选择时仍然有效；所以无需检查 `initId` 是否仍指向相同的借据条款。

至此，本文对 Daml 语言特性的介绍告一段落。Daml 还提供其他基本类型，例如字符串、日期、时间戳、泛型映射，以及多种数值类型，尤其包括用于内部计算的不可序列化任意精度数。本文未涉及的其他语言特性包括代数数据类型、Haskell 风格类型类、带有 try-catch 块的异常处理，以及用于按值而非按 ID 引用合约的合约键。相关文档请参阅 docs.daml.com。

3 Daml-LF

Daml 代码会编译为 Daml-LF（LF 是 ledger fragment，即账本片段的缩写），并由 Daml 账本解释执行。下面介绍 Daml-LF 的核心特性、交易树的语义域，以及授权和可见性规则。Daml-LF 的完整规范可在线查阅：

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x :: \tau} \qquad \frac{\Gamma, x : \tau \vdash t :: \sigma}{\Gamma \vdash \lambda x : \tau.\, t :: \tau \Rightarrow \sigma}$$

$$\frac{\Gamma, \alpha : k \vdash t :: \tau}{\Gamma \vdash \Lambda \alpha : k.\, t :: \forall \alpha : .\, \tau} \qquad \frac{\Gamma \vdash t_1 :: \tau \Rightarrow \sigma \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash t_1\ t_2 :: \sigma}$$

$$\frac{\Gamma \vdash \tau :: k \qquad \Gamma \vdash t_1 :: \forall \alpha : k.\, \sigma}{\Gamma \vdash t_1 \circledcirc \tau :: \sigma[\alpha \mapsto \tau]}$$

图 2。System F 的类型规则$_\omega$

github.com/digital-asset/daml/blob/main/daml-lf/spec/daml-lf-1.rst. 我们将在稍后讨论编译，见第4节.1。

3.1 无类型级λ抽象的 System F$_\omega$

Daml-LF 基于 System F$_\omega$，这是 Girard [13]提出的简单类型 λ 演算的扩展。此外，它深受 GHC Core 的启发。GHC Core 是 GHC [27]使用的中间语言。与 GHC Core 类似，Daml-LF 省略了 System F$_\omega$中的类型级λ抽象，并加入了用户定义数据类型。因此，类型表达式 τ，σ 由类型变量 α、函数箭头 $\Rightarrow$、量化 $\forall \alpha{:}k.\,\tau$、类型应用 $\tau\ \sigma$和基本类型构造器 T构成。种类k 用于对类型表达式分类，并排除格式错误的类型表达式，具体形式化为良构种类判断 $\Gamma \vdash \tau :: k$。

$$\frac{\alpha : k \in \Gamma}{\Gamma \vdash \alpha :: k} \qquad \frac{}{\Gamma \vdash \Rightarrow :: \star \to \star \to \star} \qquad \frac{\Gamma, \alpha : k \vdash \tau :: k'}{\Gamma \vdash \forall \alpha : k.\, \tau :: k'}$$

$$\frac{\Gamma \vdash \tau :: k_1 \to k_2 \qquad \Gamma \vdash \sigma :: k_1}{\Gamma \vdash \tau\ \sigma :: k_2} \qquad \frac{T_{\text{Prim}} \text{ has kind } k}{\Gamma \vdash T_{\text{Prim}} :: k}$$

基本类型构造器 T_{Prim} 包括单例类型Unit、布尔值Bool、64位整数Int64、定点小数Decimal、字符串文本、日期Date和时间戳Timestamp，其种类均为$\star$；以及列表的类型构造器List，其种类为$\star \to \star$。智能合约的基本类型构造器将在下文介绍，见3.2。

项 t 由变量 x、函数抽象$\lambda x{:}\ \tau.\, t$、函数应用 $t_1\ t_2$、类型抽象 $\Lambda \alpha{:}\ k.\, t$、类型应用 $t@\tau$以及基本常量构成 C。所有抽象均以类型或种类标注，从而使类型检查是可判定的。针对 $\Gamma \vdash t :: \tau$ 的类型规则是标准的（图 2）。

良构种类判断和良构类型判断是相对于用户在 Daml-LF 模块中声明的一组自定义 Daml-LF 类型和常量而言的。Daml-LF 确保每个包都有一个全局唯一的名称（包中的模块也一样）：包标识符是该包 Daml-LF 代码的密码学哈希，也就是说，它是一种基于内容的寻址方式。为便于阅读，本文省略包标识符和模块名称，只使用未限定的 Daml 名称。例如，以下是一个带标签叶节点的二叉树的 Daml 数据类型声明

```
数据Tree a= Leaf 带叶:
a| Node带左: Treea 右:
Tree a
```

由三个 Daml-LF 类型构造器表示，它们的种类均为 $\star \to \star$：

- 一个记录类型 Tree.Leaf，具有一个字段 leaf，用于作为 `Leaf` 构造器的参数。
- 记录类型 Tree.Node，具有字段 left 和 right，用于作为 `Node` 构造器的参数。
- 具有变体 Leaf 和 Node 的变体类型 Tree。

每个记录都带有一个记录构造器、字段投影和字段更新操作。变体类型带有构造器和模式匹配支持。类型规则确保这些操作均被完全应用；如有必要，Daml 编译器 η会对部分应用的出现进行展开。这一约定确保所有构造器都被完全应用于其参数，从而简化数据类型的实现。这同样与 GHC Core 保持一致。例如，记录构造器 Tree.Node 和变体构造器 Node 的类型规则如下：

$$\frac{\Gamma \vdash \tau :: \star \qquad \Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash \text{Tree.Node} \circledcirc \tau\ e_1\ e_2 :: \text{Tree.Node} \circledcirc \tau}$$

$$\frac{\Gamma \vdash \tau :: \star \qquad \Gamma \vdash e :: \text{Tree.Node} \circledcirc \tau}{\Gamma \vdash \text{Node} \circledcirc \tau\ e :: \text{Tree} \circledcirc \tau}$$

3.2 智能合约的原语

Daml-LF 提供了三种用于定义智能合约的类型构造器：

- Party：$\star$ 用于参与方，即字符串的子集
- ContractId：$\star \to \star$ 用于合约标识符（ID）
- Update：$\star \to \star$ 用于账本状态的更新

合约 ID 使用类似于 Daml 中的模板类型进行参数化；它们能够唯一标识该模板的合约实例。对于 Daml 程序而言，合约 ID 是不透明的，因此 Daml 程序的语义与 Daml 账本如何分配合约 ID 无关。合约 ID 只能通过 create 命令在 Daml-LF 中创建，但也可以将任意值作为序列化参数传入，详见 第 3.3 节。原始比较操作 $\leq_{\text{ContractId}}$ 允许对合约 ID 进行排序，因此合约 ID 可用于函数式数据结构，例如搜索树。

$$\frac{\Gamma \vdash \tau :: \star \qquad \Gamma \vdash e_1 :: \text{ContractId}\ \tau \qquad \Gamma \vdash e_2 :: \text{ContractId}\ \tau}{\Gamma \vdash \leq_{\text{ContractId}} \circledcirc \tau\ e_1\ e_2 :: \text{Bool}}$$

类型构造器 Update（种类$\star \to \star$）表示账本更新的单子。单子操作 pure 和 bind $x{:}\tau \leftarrow e$在 e'中将值注入单子，并将相互依赖的账本更新串联起来。共有三个基本单子

操作：create@T e、fetch @T e和 exercise@TCh e_1 e_2。它们分别表示：(1) 使用参数 e创建模板 T的合约实例；(2) 查找 e的合约参数；(3) 对 e_1 执行选择 Ch，并使用选择参数e_2。与记录和变体一样，我们要求这些操作完全应用。在下面的类型规则中，T遍历 Daml-LF 程序中的模板。

$$\frac{\Gamma \vdash \tau :: \star \qquad \Gamma \vdash e :: \tau}{\Gamma \vdash \mathsf{pure}\ @\,\tau\ e :: \mathsf{Update}\ \tau}$$

$$\frac{\Gamma \vdash \tau_1 :: \star \qquad \Gamma \vdash e_1 :: \mathsf{Update}\ \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 :: \mathsf{Update}\ \tau_2}{\Gamma \vdash \mathsf{bind}\ x : \tau_1 \leftarrow e_1\ \mathsf{in}\ e_2 :: \mathsf{Update}\ \tau_2}$$

$$\frac{\Gamma \vdash e :: T}{\Gamma \vdash \mathsf{create}\ @\,T\ e :: \mathsf{Update}\ (\mathsf{ContractId}\ T)}$$

$$\frac{\Gamma \vdash e :: \mathsf{ContractId}\ T}{\Gamma \vdash \mathsf{fetch}\ @\,T\ e :: \mathsf{Update}\ T}$$

$$\frac{\Gamma \vdash e_1 :: \mathsf{ContractId}\ T \qquad \Gamma \vdash e_2 :: \tau \qquad \mathsf{Choice}\ Ch\ \text{has argument type}\ \tau\ \text{and return type}\ \sigma}{\Gamma \vdash \text{执行}\ {}_{@T}\text{第}\ \ e_1\ e_2 :: \mathsf{Update}\ \ \sigma}$$

3.3 可序列化类型

Daml-LF 区分可序列化类型和不可序列化类型。可序列化类型是一阶数据，不包含计算。换句话说，可序列化类型的值保证不包含任何函数。合约参数、选择参数和选择结果都必须是可序列化的。这确保合约涉及的所有逻辑都编码在其定义中，而不是由其参数提供。虽然 Daml-LF 理论上可以支持高阶合约参数，但出于以下原因，这种限制是可取的：

- 人们可以审计 Daml-LF 合约，并充分理解其影响，而无需对合约参数作出假设。如果参数包含逻辑（即函数），这将无法实现。例如，如果允许选择参数中出现函数，控制者就可以在签署方的授权下运行任意 Daml 代码。

- 代码与数据之间的这种解耦，也使我们能够更自由地演进解释器和底层账本。例如，如果我们希望在解释之前加入优化遍和即时编译，同时将数据存储在合约参数中，那么此类优化必须保留足够的数据，以重建函数未经优化的 Daml-LF 表示，从而使所有相关参与方都能独立于其优化设置来验证存储数据。同样，如果我们未来要支持零知识证明，将 lambda 编码为算术电路似乎也颇具挑战。

- 将类型集合限制在一个由少量一阶类型构成的范围内，使我们能够将合约数据存储在各种数据库中，从集中式 SQL 服务器到更传统的区块链；并可与每个平台现有的数据类型建立映射。例如，Daml-LF 记录可以转换为 SQL 列，进而实现高效查询。

对模板声明进行类型检查，可确保模板参数类型和选择参数类型均为可序列化类型。基本类型是可序列化的；如果列表和合约 ID 的类型参数是可序列化的，那么它们本身也可序列化。对于 Daml-LF 记录类型 R，其类型参数和每个字段的类型都必须是可序列化的。变体类型 V 也同样如此：其类型参数以及每个变体的参数类型都必须是可序列化的。

例如，第 2 节中的 Daml 记录 `Cash` 之所以可序列化，是因为字段 `currency` 和`amount` 具有可序列化的类型。对于 `Tree`，构造器参数为可序列化当且仅当类型参数可序列化时，多态记录 Tree.Leaf 和 Tree.Node 才可序列化；变体 Tree 也同样如此。因此，Tree Int64 是可序列化的。

由于不可序列化性会通过类型构造器传播，Tree (Int64 ⇒ Bool) 也不可序列化。账本更新类型 Update α 也不可序列化；如下文所述，账本更新只能针对账本执行。

3.4 语义

Daml-LF 采用按值调用语义，该语义通过针对闭项的两个大步求值关系定义。首先，表达式求值 $e \Downarrow r$会产生一个结果 r，即表达式值 Val v或致命错误 Err err，并附带错误消息 err。表达式值 v包括 λ 抽象以及内置数据类型、合约 ID、参与方等的字面值。当且仅当所有参数都是表达式值时，完全应用的记录和变体构造器才构成表达式值。例如，

$$\mathsf{Tree.Leaf}\ @\ (\mathsf{Int64} \Rightarrow \mathsf{Int64})\ (\lambda x : \mathsf{Int64}.\ x + 1)$$

这是一个值，因为 lambda 抽象是一个值，而且 Tree.Leaf 已完全应用。表达式求值也可能返回 Update 表达式 u，它同样被视为表达式值。Update 表达式有五种形式：

- 创建 @T v，用于创建模板 T的合约实例，并以值 v作为合约参数
- Exercise @TCh v_{cid} v_{a}，用于对合约 ID v_{cid} 上的模板 T的选择 Ch 执行 Exercise，并使用选择参数v_{a}
- 获取 @T v，用于查找合约 ID v对应的合约参数
- pure @τ v，表示类型为 τ的纯值 v
- bind $x : \sigma \leftarrow v$ in e 用于计算（受阻的）账本变更

其次，Update 解释关系 $\langle u, s\rangle \Downarrow R$ 解释上述形式的账本变更，并对其子项进行表达式求值。它依赖于当前账本状态 s，该状态是从合约 ID 到以下合约实例相关信息的映射：

- 模板标识符 T
- 作为可序列化类型的表达式值的合约参数 v
- 签署方 S 和观察者 O
- 合约状态 cs（活动或已归档）

如果 s对于 cid 未定义，则合约 ID cid 在 s中是新的。更新结果 R可能是一个值v、一个事务 tx 和一个更新后的状态 s'，或是带有错误消息 err 的致命错误 Err err。

交易是 Daml 对账本更新的层次表示。交易 tx 是操作列表，而操作 a 采用以下形式之一：

- Create cid $T\ v$ S O 表示创建一个模板 T 的合约实例，其参数为 v，合约 ID 为 cid，签署方为 S，观察者为 O。
- Exercise cid T.Ch $v\ K$ C Q S tx 表示在模板 T 中，对合约 ID 为 cid 的合约执行选择 Ch，选择参数为 v。选择类型 K 可以是消耗型或非消耗型。集合 C 表示控制方，Q 表示选择观察者，S 表示被执行合约的签署方。Exercise 操作包含将该选择的实现作为子事务 tx 执行所得的结果，我们将其称为该操作的后果。
- Fetch cid T S O 表示查找合约 ID cid 后，找到了一个模板 T的合约实例，其签署方为 S，观察者为 O。

因此，操作是树的根，而交易是一片森林。例如，假设 Alice 提交一条命令，将她持有的由 Bank 发行、金额为 100 美元的 `SimpleIou` 转让给 Bob，其合约 ID 为 #1。该命令会转换为执行以下 Daml-LF 表达式；这里省略了 Daml 编译器引入的所有记录冗余：

调用@ SimpleIou.SimpleTransfer #1 Bob

解释此表达式会生成事务，如 图3所示，其中只有一个根节点，其后果显示为子节点。控制者集合以下划线标出，签署者集合以粗体显示，观察者集合不加修饰，空集合则完全省略；默认情况下，所有执行操作都是消耗型的。该 Exercise 会在根节点生成消耗型 Exercise 节点，而 **创建** 操作位于 `SimpleTransfer` 选择的主体中，会生成 Create 子节点。

Exercise #1 SimpleIou。SimpleTransfer
Bob {Alice } {Bank}
⇓
创建 #2 SimpleIou
(SimpleIou Bank Bob (现金 美元 100)) {Bank} {Bob}

图3. Alice 向 Bob 转让 100 美元的`SimpleIou`的交易树

更新解释 $\langle u, s\rangle \Downarrow R$ 的工作方式如下。这里我们只讨论成功情形；如果任何检查或子求值失败，解释过程会生成相应的错误并中止。

对于创建 $@T\ v$:

1. 查找 T 的模板定义。2. 使用 $\Downarrow$ 对 v 计算**确保** 谓词，并检查结果是否为 Val True。3. 计算 T的**签署方** 和**观察者** 函数，并将生成的参与方列表转换为集合 S 和 O。4. 选择一个在 s中全新的合约 ID cid。5. 解释结果由值 cid、事务 [Create cid $T\ v$ S O], 和更新后的状态 s[cid $\mapsto (T, v,$ S, O, Active)] 组成。

对于 exercise $@T$Ch $v_{\text{cid}}\ v_{\text{arg}}$:

1. 在状态 s 中查找合约数据，例如 $s(v_{\text{cid}}) = (T' v_{\text{c}},$ S, O, cs)，并检查 $T' = T$ 且 cs 为 =Active。如果 Ch 是消耗型，则设置 $K =$ 为消耗型；否则设置 $K =$ 为非消耗型。2. 针对合约参数 v_{c} 和选择参数 v_{arg}，计算选择 Ch 的控制者和观察者表达式，并将生成的参与方列表转换为集合 C 和 Q。3. 将 v_{c} 和 v_{arg} 替换为选择体表达式中的模板参数和选择参数，并对其求值，得到 Update 表达式 u'。4. 在状态 s' 中解释 u'；如果 Ch 是消耗型，则状态为 $s' = s[v_{\text{cid}} \mapsto (T, v_{\text{c}},$ S, O, Archived)]，否则保持原状态。解释 u' 的结果由值 v_r、事务 tx 和新状态 s'' 组成。5. 解释 Exercise 的结果由 v_r 、事务 Exercise $v_{\text{cid}}\ T$.Ch $v_{\text{arg}}\ K$C Q S tx 和状态 s''组成。

对于获取 $@T\ v$:

1. 在 s 中查找合约数据，例如 $s(v) = (T, v_{\text{c}},$ S, O cs)，并检查 $T = T'$和 cs =活动。2. 结果由值 v_{c}、事务[Fetch $v_{\text{cid}}\ T$S O], 以及未更改的状态 s 组成。

对于 pure$@\tau\ v$，结果为 $(v,$ [], $s)$。顺序组合 bind x: $\sigma \leftarrow v$ in e 会组合产生的事务：

1. 在 s 中解释 v，例如 $\langle v,$ $s\rangle \Downarrow (v_1,$ tx$_1,$ $s_1)$。

2. 将 v_1 替换为 x，然后对其求值，$e[x \mapsto v_1] \Downarrow u$。3. 在 s_1 中解释 u，例如 $\mathrm{h}u,\ s_1\mathrm{i} \Downarrow (v_2,\ \mathrm{tx}_2,\ s_2)$。4. 结果为 $(v'',\ \mathrm{tx}_1 \mathrm{+\!+} \mathrm{tx}_2,\ s'')$，其中 ++ 将两个列表连接起来。

可以看出，合约 ID 的账本状态最多更新两次：一次是在为新的合约实例选取该合约 ID 时，另一次是在对其执行消耗性选择时。这种不可变的合约实例让人联想到比特币的 UTXO 模型中的事务输出，不同之处在于，Daml 合约实例可以存储任意数据，而不只是金额和验证函数。我们会在第5节中更详细地比较 Daml 的执行模型与相关工作。

3.5 图灵完备性

Daml-LF 表达式不包含递归绑定符，例如 Haskell 的 `let` 或 OCaml 的 `let rec`。相反，Daml-LF 模块可以包含任意数量的顶层值定义，将名称绑定到 Daml-LF 表达式，而非定义合约。值定义可以相互递归，因此赋予 Daml-LF 不受限制的递归能力，使其达到图灵完备。

此外，Daml-LF 允许使用非单调数据类型，这些类型也可用于定义不需要显式递归的不动点组合子，如下方的不动点组合子 `fix` 所示：Z

```
数据 Mu a = Mu 带有 unroll : Mu a -> a

fix : ((a -> a) -> (a -> a)) -> a ->a fix f =令 x =
Mu (\y -> f (\v -> (unroll y) y v))其中 f (\v ->
(unroll x) x v)
```

使 Daml-LF 成为全函数语言并无实际优势——这与基于类型理论的定理证明器不同，后者的可靠性依赖于一种全函数、非图灵完备的语言。对于分布式账本技术应用而言，重要的是计算一个函数所需的时间；而全函数语言并不能避免编写需要数千年才能完成计算的函数。相反，账本实现可以为每个 Daml-LF 求值步骤分配成本，并使用该成本模型进行资源管理，类似于以太坊的 Gas 模型 [31]。

3.6 授权规则

在第 2 节中，我们概述了 Daml 的授权规则，其中涉及签署方和控制者。然而，上一节的求值语义并不检查授权。相反，我们在交易上定义授权规则。如果操作 $\vdash_{\mathrm{a}} a$ 和 $\mathcal{A} \vdash_{\mathrm{a}} tx$ 符合授权规则，a 或者事务 tx 在授权上下文 $\mathcal{A}$ 中符合授权规则，我们就写作 $\mathcal{A}$，其中 A 是参与方集合。当参与方提交 Daml 命令

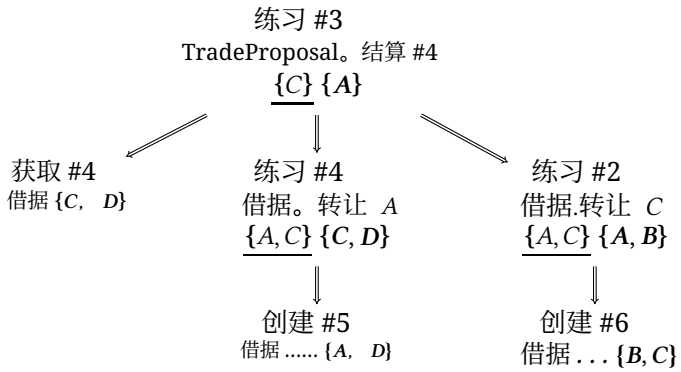

图4. 事务：C 对结算 选择进行 Exercise，在 `TradeProposal` 上，由 A（合约 ID #3）执行，以交换由 B 签发的借据和由 D 签发的借据（#4）

到账本时，整个结果交易的授权上下文仅由 $\{A\}$ 构成。不过，对于 Exercise 操作的子（交易）动作，授权上下文可以扩展并发生变化。

形式化授权规则如下。若上下文 A 包含所有签署方，则创建动作在 A 中是经过良好授权的。对于获取，A 必须包含被获取合约的至少一名签署方或观察者。Exercise 的规则实现了委托：首先，A 必须包含所有控制方，这符合预期。其次，在签署方和控制方的上下文中检查后果是否获得良好授权。这表示签署方在创建合约实例时，已预先授权这些选择的后果。不过，委托并非传递性的，因为 Exercise 操作的授权上下文 A 不会传播到后果。最后，如果事务的所有（顶层）操作都获得良好授权，则该事务也获得良好授权。

$$\frac{\mathrm{S} \subseteq \mathrm{A}}{\mathrm{A} \vdash_{\mathrm{a}} \text{创建 cid } T\ v\ \mathrm{S}\ \mathrm{O}} \qquad \frac{\text{一个} \cap (\mathrm{S} \cup \mathrm{O}) \neq \{\}}{\mathrm{A} \vdash_{\mathrm{a}} \text{获取 cid } T\ \mathrm{S}\ \mathrm{O}}$$

$$\frac{\mathrm{C} \subseteq \mathrm{A} \qquad \mathrm{S} \cup \mathrm{C} \vdash_{\mathrm{a}} \mathrm{tx}}{\text{一个} \vdash_{\mathrm{a}} \text{Exercise cid } T.\mathrm{Ch}\ v\ K\ \mathrm{C}\ \mathrm{Q}\ \mathrm{S}\ \mathrm{tx}}$$

$$\frac{\forall a \in tx.\ \mathcal{A} \vdash_{\mathrm{a}} a}{\mathcal{A} \vdash_{\mathrm{a}} tx}$$

委托不具传递性，是 Daml 中一项刻意的设计选择。这简化了对 Daml 程序授权是否充分的推理，因为选择实现的授权上下文完全由模板定义决定，因此独立于该选择的调用上下文。一方面，这有助于 Daml 程序员判断选择实现是否拥有足够的授权。另一方面，这也是一项重要的安全特性，有助于以模块化方式组合工作流：当工作流构建

块（即选择）如 结算 选择所示组合成更复杂的工作流时，构建块的授权安全分析仍不受影响。相反，如果在扩展上下文 A ∪S ∪ C 中检查 Exercise 操作的后果，那么 A 中的额外参与方可能会使某个子操作获得充分授权，而该操作单独执行时却不会。

如果需要传递委托，则必须在选择体中将所需的签署方声明为**Exercise**d 选择的显式控制方。这种模式可见于结算 选择，其中发起方的授权通过对**Exercise** 的 `respId` 的`newOwner` 控制者，传递给转让 选择，进而用于创建新的借据。 图4展示了这样一笔事务： *C* 在一份 结算 选择上执行操作，该选择属于一份 `TradeProposal` （合约 ID #3）；在该合约中， *A* 提出用 *B* 发行的 ID 为 #2 的借据，交换*D* 发行的借据 （#4）。在上下文 {*C*} 中，这笔事务获得了充分授权。关键在于， *A*作为 `TradeProposal` 合约 #3 的签署方所拥有的授权，与 *C*作为 Exercise 控制者所拥有的授权，在 Exercise #4 的控制方 {*A*, *C*} 中合并，因此创建 #5 时即可使用 *A*的授权。

4 实现

现在我们介绍如何实现将 Daml 编译为 Daml-LF 的编译器（第 4.1 节）和 Daml-LF 运行时（第 4.2 节），并简要说明如何将 Daml-LF 运行时集成到不同的账本中（第4.3节）。

4.1 Daml 编译器

Daml 编译器会将 Daml 包中的 Daml 模块编译成 .dalf 文件，其中以 Protobuf 格式编码了生成的 Daml-LF 代码。编译器使用格拉斯哥 Haskell 编译器（GHC）前端，将 Daml 代码解析、进行类型检查并去除语法糖，转换为 GHC 的中间语言 GHC Core （ System FC 的一种变体） [30]。

最初，我们从头开始实现 Daml 编译器，包括设计语言、编写解析器、类型检查器和去语法糖器。然而，添加各种语法便利特性和语言功能是一项艰巨的挑战。每增加一项新功能，都会引入新的错误，尤其是在解析和类型检查方面。因此，我们改变了方向，决定重用 GHC 的前端。GHC 是一款成熟的编译器，历经数十年的发展。得益于此，Daml 现在具备了 Haskell 的许多语言特性和语法便利功能，能让开发者更高效地工作，例如类型类，以及带守卫的模式匹配。

与此同时，我们也认识到 Haskell 存在一些明显的不足，在开发 Daml 时尤为突出。因此，我们着手改进这些方面，让 Daml 用户和 Haskell 用户都能受益。具体来说，与其他语言相比，Haskell 的记录系统长期以来一直被认为相对薄弱，因此我们启动了

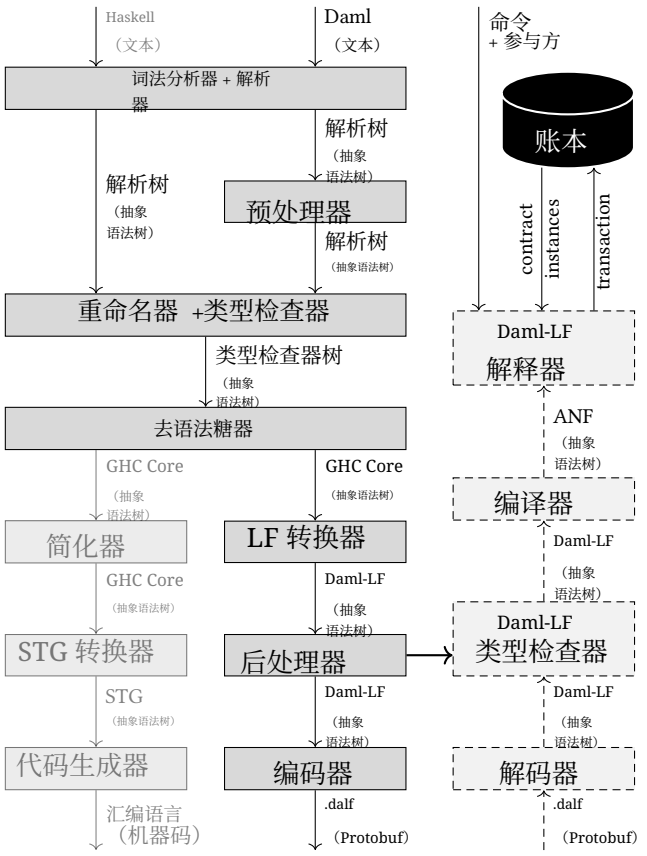

图 5. GHC（左）、Daml 编译器（中）和 Daml-LF 运行时（右）的流水线

`RecordDotSyntax` 提案，并与 Haskell 开源社区的许多其他成员共同推进。该提案现已包含在最新版本的 GHC（9.2 版）中。[12]我们还开发了一个名为 ghcide 的 Daml/Haskell 集成开发环境。此项目后来与另一个集成开发环境项目合并，形成了 Haskell Language Server [21], , 如今它已成为 Haskell 的标准集成开发环境。

图 5 从左至右展示了 GHC、Daml 编译器和 Daml-LF 运行时的流水线，我们将在第 4.2 节中对此进行讨论。流水线阶段以矩形表示，箭头标注了从一个阶段流向下一阶段的数据。宽矩形表示 GHC 和 Daml 编译器共用的部分，即词法分析器、解析器、重命名器、类型检查器和去语法糖器。重命名器、类型检查器和去语法糖器完全相同。

对于 Daml 的 **模板** 和 **选择** 语法，我们为 GHC 词法分析器和解析器扩展了相应的解析规则。修改后的解析器会将模板和选择 desugar 为 Haskell 类型类，使后续阶段保持不变。每种模板操作都有自己的 Haskell 类型类，这样在添加更多操作时就能轻松升级。

类 HasEnsure t**其中** ensure :: t -> **Bool** **类** HasObserver t **其中** observer :: t -> [参与方]

类型类 HasSignatory t **其中** signatory :: t -> [参与方]**类型类** HasCreate t**其中**
创建 :: t -> Update (ContractId t)**类** HasFetch t **其中** 获取 :: ContractId t ->Update t**类** HasExercise tcr |t c -> r **其中** exercise :: ContractId t -> c -> Updater

HasExercise 类型类由模板 t、选择参数 c 和选择结果 r参数化，其中 t 与c 的组合唯一确定r。这是因为每个选择都会为其选择项参数生成一个独立的记录类型。解析器还会为所有已声明的模板和选择生成相应的实例声明。

例如，SimpleIou 模板会变成表示模板参数的 Haskell 数据类型，并带有 HasEnsure、HasObserver 等类型类实例。原始Update操作会以类型级字符串的形式保留，直到 LF 转换阶段将其替换为相应的 Daml-LF 原语。SimpleTransfer 选择同样会生成一个表示选择参数的数据类型，并带有一个HasExercise实例。

数据SimpleIou = SimpleIou，其中 issuer：参与方 owner：参与方 cash：现金

实例 HasEnsure SimpleIou **其中** ensure this = (cash.amount this) > 0.0**实例**HasObserver SimpleIou **其中**observer this = [owner this]**实例**HasSignatory SimpleIou **其中**signatory this = [issuer this]**实例** HasCreate SimpleIou**其中** create = primitive @ "UCreate"**实例** HasFetch SimpleIou **其中**fetch = primitive @"UFetch"

数据SimpleTransfer = SimpleTransfer，其中 newOwner : 参与方

实例 HasExercise SimpleIou SimpleTransfer (ContractId SimpleIou)**其中** exercise = 原语 @"UExercise"

新的预处理器会提前对 Daml 中不可用的 Haskell 特性发出警告和错误，例如不受支持的语言扩展，如广义代数数据类型。它还会将 Daml 专用导入注入模块，以便后续阶段能够处理生成的模板代码。

接下来，抽象语法树（AST）会经过重命名器、类型检查器和去语法糖器的处理。尤其是去语法糖器会将类型类替换为字典[15]。此后，AST 将处于 GHC Core 中，Daml 编译器也将脱离 GHC。用于优化 GHC Core 的 GHC 简化器无法使用，因为 Haskell 的惰性求值已深深植入优化规则之中。相反，LF 转换器会将 GHC Core 转换为 Daml-LF。这样，它会将这些类型类实例转换回 Daml-LF 模板和选择，同时还会将 原语 占位符替换为 Daml-LF 中相应的原语操作。如果 GHC Core 使用了 Daml-LF 中不存在的 System FC 特性（例如广义代数数据类型），转换就会失败。

流水线的其余部分大多采用标准流程：后处理器首先清理生成的 Daml-LF 代码，并应用 Daml-LF 专用优化。然后，它会对生成的 Daml-LF 代码进行类型检查，以额外防范编译器各阶段中的错误。最后，编码器会将代码序列化为二进制 Protobuf 消息，并将其存储为 .dalf 文件。

我们已确保使用 Daml 编译器进行的构建在单线程模式下运行时具有可重复性。也就是说，如果使用同一版本的 Daml 编译器编译相同的 Daml 输入程序，就会生成相同的 .dalf 文件。这对于 Daml-LF 中基于内容的寻址至关重要，因为如果不想信任共享 .dalf 文件的来源，任何实体都可以直接重新编译 Daml 包。

4.2 Daml-LF 运行时

Daml-LF 解释器位于 Daml-LF 运行时的核心。如右侧的图 5 所示，其输入是形如**创建** T v 或**Exercise** cid Ch v 的命令，分别用于创建模板T 的合约，或对合约 ID cid 上的选择Ch 进行 Exercise，并传入参数v。它还接收提交这些命令的参与方列表 A。在解释过程中，它会从 Daml 账本（或其缓存）中查询所需的合约实例。正常情况下，输出为一个事务；否则输出相应的错误。随后，该事务会被交给 Daml 账本进行验证和状态更新。

解释器通过软件包加载器按需加载被引用的 .dalf 包及其依赖项。软件包加载器首先解码经 Protobuf 编码的 Daml-LF 代码，并再次对其进行类型检查。随后，另一编译步骤会进一步将 Daml-LF 转换为一种更简单、不可序列化的管理范式中间语言 [11]。

加载完所有必需的包后，解释器会将命令转换为 Update 表达式，并使用高效的 CEK 机器实现 [10]，按照 Daml-LF 语义 $h_{c,\_i} \Downarrow_{\_}$ 对它们进行求值。如果所有求值均成功，最终输出将由拼接后的交易组成。

在构建事务的过程中，也会动态检查授权规则。如果解释过程尝试在不具备所需授权的情况下 Exercise 某个选择、获取某个合约，或创建某个合约，则解释会在开始评估相关智能合约之前因授权错误而失败。此顺序至关重要，因为否则诸如 **断言**或 **ensure** 子句失败之类的解释错误，可能会通过错误消息向提交者泄露敏感信息。

4.3 账本集成

如图 1所示，Daml 智能合约运行在 Daml 账本上。去中心化应用通过参与方提供的账本 API（以 gRPC 和 HTTP JSON 服务的形式）访问账本：它们可以提交 Daml 命令，并接收账本状态的同步和异步更新。基于统一账本 API 的这种事件驱动编程模型，使得 Daml 去中心化应用能够跨不同技术移植，因为相同的 Daml 智能合约可在所有集成方案上运行。为满足不同用例，现有与不同账本技术的集成方案可供选择，它们具有不同的性能、安全和保密性特性。我们仅列出用于验证交易和持久化账本的可用选项；有意义的比较超出了本文的范围。

- 由单一可信实体运营的 SQL 数据库，例如 Postgres 和 Oracle。
- 由一个或多个实体运营的分布式节点网络，使用状态机复制来运行确定性的 Daml-LF 解释器，例如 VMware Blockchain [29], HyperledgerFabric 和 Besu。
- 作为区块链或共识算法之上的第二层，合约实例仅本地存储在参与方处，数据在参与方之间传输时采用端到端加密。在此，底层账本技术仅用于对更新排序和协调提交。

5 相关工作

Daml 的账本状态让人联想到比特币的 UTXO 模型[22]：合约创建对应于一个由合约 ID 标识的“交易输出”，而消耗性选择的执行可以“花费”该输出。在此期间，Daml 合约可通过非消耗式选择任意多次使用，从而减少争用。

Chakravarty 等人 [7] 提出了扩展 UTXO（eUTXO）执行模型，该模型由运行在 Cardano 区块链[5]上的 Plutus [6]智能合约语言采用。在 eUTXO 中，用于花费交易输出的验证函数会接收一个额外的输入参数——数据，其中包含任意的合约特定数据。用 Daml 的术语来说，该数据对应于合约参数，而验证器函数则由相应模板的选择表示。Daml 合约更加灵活，因为它们不受加密货币价值（即交易输出）的限制，而且既可以消耗，也可以非消耗地使用。由于 eUTXO 交易是扁平的，验证器必须手动找出相关的交易组成部分。Daml 交易则通过树结构对其进行编码，从而确保可组合性。

以太坊智能合约基于具有可变状态的账户，而非 UTXO。账户避免了合约 ID 变得过时的问题：Update 会归档合约实例，并重新创建更新后的实例。相反，账户的可变性也会带来问题，因为交易提交者无法预先确定确切的效果，例如，当事务依赖于在提交和执行之间发生变化的数据时。在 Daml 中，可以通过引用存储数据的部分内容而非合约 ID 来避免悬空的合约 ID（本文不作讨论）。

Corda [24] 是一种分布式账本技术平台，其账本状态与 Daml 一样，是一组不可变的合约实例。合约实例通过 JVM 类定义，这些类既规定实例的数据，也规定如何验证创建、引用或消耗此类实例的交易。Corda 要求程序员将验证指定为针对完整交易的布尔谓词，这为错误留下了很大空间：检查过于宽松会导致安全漏洞，而检查过于严格则会妨碍工作流可组合性。Corda 将合约代码打包为 .jar 文件，并在 JVM 上执行。JVM 上的合约执行可能具有非确定性，进而导致账本参与者的状态不同步；Daml 从设计上就是确定性的，因此避免了这一问题。

在本节余下部分，我们将 Daml 与密切相关的智能合约语言进行比较。Tolmach 等人 [28]对更广泛的领域作了很好的概述。

Peyton Jones 等人 [23]提出了一个由小型构建块和组合子组成的函数式库，用于对双边金融合约建模，以进行估值。Bahr 等人 [4] 的合约同样着眼于估值和风险分析，但还能够对多方参与的复杂投资组合建模。Daml 沿用了将简单工作流组合成更复杂工作流的这一理念：选择对应于备选项，而选择主体则将各个构建块汇总为更大的原子交易。

Hvitved 等人 [1, 16, 17] 进一步发展了 Peyton Jones 等人的构想，提出了合约规范语言 CSL。这是一种用于事件驱动交易系统的过程演算，已由 Deon Digital 实现 [9]。CSL 将合约实例（包括 λ 表达式）作为过程项存储在账本上。发生事件时，过程项会根据过程演算规则规约为另一个过程项。这种格式为实例化前组合工作流提供了很大的灵活性。相比之下，Daml 要求程序员在新模板中编码组合，并生成和部署 .dalf 包。作为回报，账本上的 Daml 合约实例始终绑定到一个模板，这有助于理解账本状态。对于 CSL，要弄清楚部分求值后的合约实例代表什么要困难得多，因为必须研究原始过程项。

虽然 CSL 和 Daml 都声称具有可组合性，但 CSL 对这一概念的定义更为受限：CSL 的组合算子只能在合约实例部署之前应用；

在账本上基于现有合约实例构建工作流是不可能的。因此，必须在一开始就设想合约所有可能的后续用例。例如，只有在首次创建借据合约时就作出决定，才能在 TradeProposal 工作流中使用 借据；后续的任何用途也同样如此。相比之下，Daml 允许组合已部署合约实例的工作流；TradeProposal 可以与已部署的借据配合使用。事实上，得益于 Daml 的分层事务，借据工作流甚至不必知道自己属于更大的工作流。

在 Rainfall 中， [20],账本存储带标签事实（Daml 中的模板 ID 和合约参数）构成的多重集合，并单独存储转换这些事实的规则。事实会标注授权方和观察方，这分别对应 Daml 模板中的签署方和观察者。规则会对输入事实进行模式匹配，并可能消耗这些事实，然后生成新的输出事实；为此，规则可以从输入事实中获取授权方，这类似于 Daml 中签署方预先授权选择的后果。每个事实随附的使用集合决定了哪些规则可以使用该事实。这个使用集合实际上引入了模板的动态概念：所有具有相同标签和使用集合的事实都是同一模板的实例。Rainfall 和 Daml 的主要区别在于执行模型：Daml 通过唯一的合约 ID 引用数据，而 Rainfall 则按内容查找事实。因此，在 Rainfall 中，所有所有者、发行方和现金金额相同的借据都可以互换。

Plutus 语言和 Daml 一样，基于 System F_ω 。Plutus 编译器也使用 GHC 将 Plutus 编译为 Plutus Core，但采用了不同的设置：事务代码通过 Template Haskell 嵌入普通的 Haskell 程序中；在 GHC 后端之前，会有一个新阶段将这些嵌入的片段转换为 Plutus Core，以便在链上运行。外围程序则应在链下运行。对于 Daml，我们严格区分用 Daml 编写的链上代码和链下代码。为此，Daml SDK 会生成 Java 和 TypeScript 的绑定，这两种语言更广泛地用于构建链下应用程序。

6 结论

我们介绍了智能合约语言 Daml。**模板**定义了共享链上数据的数据模型，以及用于规定如何更改这些数据的**选择**。程序员会为所有数据和选择标注所需的授权者（**签署方** 和**控制者** ），以及会收到数据创建和修改通知的**观察者**。这些策略不仅控制访问权限，还会根据所有权对数据进行划分，从而支持并行处理。

Daml 生态系统不仅包括本文关注的 Daml 语言。Daml SDK 包含一款带有语法高亮、类型推断、跳转到定义等功能的 VS Code 集成开发环境。Daml Script 让程序员能够编写账本交互场景脚本，用作单元测试或进行调试。Daml Triggers 提供了一种简单的方法，可通过发送新命令来响应账本更新。更复杂的链下逻辑（包括去中心化应用的用户界面）通常使用主流编程语言编写，而非 Daml。去中心化应用通常使用 Daml 模板和选择的自定义绑定与 Daml 代码交互，目前可为 Java 和 TypeScript 自动生成这些绑定。

未来，我们计划扩展 Daml，以提供比显式使用 `DiscloseIou` 选择更灵活的方式，将合约实例披露给其他参与方，使他们能够执行委托选择。为了提高 Daml 的模块化程度，我们还在开发模板接口，用于指定可用的选择，而不涉及具体实现。对于 Daml 账本实现，我们计划向 Daml 代码开放底层区块链或分布式账本技术的一些功能。具体而言，我们计划支持 Daml 智能合约直接与原生区块链资产交互，而目前这只能通过可信第三方实现。

致谢。 我们感谢 Digital Asset 的所有员工，他们为 Daml 和 Daml-LF 作出了贡献，尤其感谢 Jost Berthold、Shayne Fletcher、Rafael Guglielmetti、Rohan Jacob- Rao、Shaul Kfir、Ben Lippmeier、Ognjen Marić、David Millar- Durrant 和 Sofus Mortensen。

参考文献

[1] Jesper Andersen、Patrick Bahr、Fritz Henglein 和 Tom Hvitved。2014年。企业系统领域专用语言。载于《形式化方法、验证与确认的应用：驾驭变革的技术（《计算机科学讲义》）》，Tiziana Mar- garia 和 Bernhard Steffen（编），第 8802 卷。Springer，73–95 页。https://doi.org/10.1007/978-3-662-45234-9_6[2] Elli Androulaki、Artem Barger、Vita Bortnikov、Christian Cachin、Konstantinos Christidis、Angelo De Caro、David Enyeart、Christo- pher Ferris、Gennady Laventman、Yacov Manevich、Srinivasan Mu- ralidharan、Chet Murthy、Binh Nguyen、Manish Sethi、Gari Singh、Keith Smith、Alessandro Sorniotti、Chrysoula Stathakopoulou、Marko Vukolic、Sharon Weed Cocco 和 J ason Yellick。2018年。Hyperledger Fabric：面向许可型区块链的分布式操作系统。载于*EuroSys 2018*，Rui Oliveira、Pascal Felber 和 Y. Charlie Hu（编）。ACM，30 1 30 15 10 1145 3190508 3190538：–：。[3] https://doi.org/./.Nicola Atzei、Massimo Bartoletti 和 Tiziana Cimoli。2017年。以太坊智能合约攻击综述（SoK）。载于《安全与信任原理（*POST 2017*）》（《计算机科学讲义》），Matteo Maffei 和 Mark Ryan（编），第 10204 卷。Springer，164–186 页。https://doi.org/10 1007 978 3 662 54455 6_8 ./---- -Patrick Bahr、Jost [4] Berthold 和 Martin Elsman。2015年。金融多参与方合约的经认证符号化管理。载于国际函数式编程会议（*ICFP 2015*）。ACM，315–327 页。https://doi.org/10 1145 2784731 2784747 ./.Cardano。 。https://cardano.org/。[5] 2021

[6] Cardano。2021年。Plutus。https://developers.cardano.org/docs/smart-contracts/plutus/ [7] Manuel M. T. Chakravarty、James Chapman、Kenneth MacKen- zie、Orestis Melkonian、Michael Peyton Jones 和 Philip Wadler。2020年。《扩展 UTXO 模型》。载于《金融密码学与数据安全》（*FC 2020*）（《计算机科学讲义》），Matthew Bernhard、Andrea Bracciali、L. Jean Camp、Shin'ichiro Matsuo、Alana Maurushat、Peter B. Rønne 和 Massimiliano Sala（编），第 12063 卷。Springer，525–539 页。https://doi.org/ 10 1007 978 3 030 54455 3_37 ./---- [8] 去中心化应用安全项目。2019年。十大项目。https://dasp.co/ [9] Deon Digital。2021年。https://www.deondigital.com/。[10] Mattias Felleisen 和 D. P. Friedman。1987年。《高阶语言中的赋值演算》。载于《编程语言原理》（*POPL 1987*）。ACM，314–325 页。https://doi.org/ 10 1145 41625 41654 ./. Cormac Flanagan、Amr Sabry、[11] Bruce F. Duba 和 Matthias Felleisen。1993年。《使用续延进行编译的本质》。载于《编程语言设计与实现》（*PLDI 1993*）。ACM，237–247 10 1145 155090 155113 。https://doi.org/./. GHC 提案。[12] 0282 2021 记录点语法。https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/028 2-record-dot-syntax.rst [13] Jean-Yves Girard。《高阶算术的直谓数解释与割消去》。博士论文。巴黎第七大学。[14] Guy Golan-Gueta、Ittai Abraham、Shelly Grossman、Dahlia Malkhi、Benny Pinkas、Michael K. Reiter、Dragos-Adrian Seredinschi、Orr Tamir 和 Alin Tomescu。2019 年。《SBFT：可扩展的去中心化信任基础设施》。载于《可靠系统与网络》（DSN 2019）。IEEE，568–580。https://doi.org/10.1109/DSN.2019.00063 [15] Cordelia Hall、Kevin Hammond、Simon Peyton Jones 和 Philip Wadler。1994 年。Haskell 中的类型类。载于《编程语言与系统》（ESOP 1994）（《计算机科学讲义》），Donald Sannella（编），第 788 卷。Springer，241 256 10 1007 3 540 57880 3_16 –。https://doi.org/。/--- [16] Tom Hvitved。2011 合同形式化与领域特定语言的模块化实现。博士论文。计算机科学系，哥本哈根大学。[17] Tom Hvitved、Felix Klaedtke 和 Eugen Zălinescu。2012 年。多方合约的基于迹的模型。《逻辑与代数编程杂志》81 卷第 2 期（2012 年），72–98。https://doi.org/ 10 1016 2011 04 010 。/j.jlap。 。 Hyperledger [18] Besu。企业级、基于 Java 2021 并采用 Apache 2.0 许可的以太坊客户端。https://github.com/hyperledger/besu。[19] Lawrence Lessig。2006 年。《代码与网络空间的其他法律》，2.0 版。Basic Books。[20] Ben Lippmeier、Amos Robinson 和 Andrae Muys。2019 年。智能合约作为授权产生式规则。载于《编程语言的原理与实践》（PPDP 2019），Ekaterina Komendantskaya（编）。ACM，14:1–14:14。https://doi.org/10.1145/3354166.3354179 [21] Neil Mitchell、Moritz Kiefer、Pepe Iborra、Luke Lau、Zubin Duggal、Hannes Siebe nhandl、Javier Neira Sanchez、Matthew Pickering 和 Alan Zimmerman。2020 年。在构建系统之上构建集成开发环境（IDE）。载于《函数式语言的实现与应用》（IFL 2020）（英国肯特大学），Olaf Chitil（编）。ACM，1 10 10 1145 3462172 3462180 –。[22] https://doi.org/。/。中本聪。2008 《比特币：一种点对点电子现金系统》。https://bitcoin.org/bitcoin.pdf。[23] Simon Peyton Jones、Jean-Marc Eber 和 Julian Seward。2000 年。合约组合：金融工程的一次探索（函数式珍品）。载于《国际函数式编程会议》（ICFP *2000* 280 292 10 1145 351240 351267 ）。ACM，–。[24] 3 2021 https://doi.org/。/。 R。 Corda。https://www.corda.net/。 [25] Carol M. Rose。1987–1988 年。《财产法中的水晶与泥泞》。斯坦福特法律评论 40，3（1987–1988），577–610。[26] Roberto Saltini 和 David Hyland-Wood。2019 年。IBFT 2.0：适用于最终同步网络的 IBFT 区块链共识协议的一种安全且活跃的变体。arXiv: 1909 10194 。Martin Sulzmann、[27] Manuel M. T. Chakravarty、Simon Peyton Jones 和 Kevin Donnelly。2007 年。带类型相等强制转换的 System F。载于 语言类型设计与实现会议（*TLDI 2007*）。ACM出版社， 53 66 10 1145 1190315 1190324 –。https://doi.org/ 。 / 。[28] Palina Tolmach、Yi Li、Shang-Wei Lin、Yang Liu 和 Zengxiang Li。2021年。一项关于智能合约形式化规范与验证的综述。*ACM* 计算调查54，7，文章 148（2021年），38 页。https://doi.org/ 10 1145 3464421 ./VMware。. VMware Blockchain。[29] 2021 https://www.vmware.com/products/blockchain.html。[30] Stephanie Weirich、Justin Hsu 和 Richard A. Eisenberg。2013 年。具有显式种类相等的 System FC。载于 国际函数式编程会议（*ICFP 2013*）（*SIGPLAN* 通讯），第 48 卷，第 9 期。ACM， 275 286 10 1145 2500365 2500599 –。https://doi.org/ 。 / 。[31] Gavin Wood。2014 。以太坊：安全、去中心化、通用化的交易账本。https://github.com/ethereum/yellowpaper。[32] MaxYoun g.2009. 理解 g 合同法. Routled ge.