Denotational semantics for the masses:

Calculating a Ledger, among other things

One spec to rule them all, one spec to find them, one spec to bring them all and in the darkness bind them

Christos 𝕂 Loverdos

2022-09-28
Version: `bd14bc5e (+dev)`

Abstract

We use a tagless-final approach to design an Embedded Domain-Specific Language (eDSL) and accompanying infrastructure for the specification and utilization of program semantics. Semantics defined in this way, for blockchain ledgers and other systems, are denotational, directly executable and amenable to multiple interpretations. We provide a layered compiler supported by two families of Abstract Syntax Trees (AST), the former as a direct translation of eDSL-based semantics and the latter suitable for immediate code generation. In addition, we provide a code generator for the Scala programming language as a backend for the compiler. Our design is in Scala, without relying on exotic features like macros or staging. We use the logic of Scala's implicits to encode algorithmic type reconstruction rules in a straightforward way, lifting the compiler's type computations at the value level, thus enabling powerful type-preserving code generation from the eDSL. We observe that we can support recursion directly, without resorting to the usual *fixpoint* combinators, by letting the compiler tie the knot. As a consequence of this and other design choices, the computations written using our approach are close to what a software engineer would write without thinking about "semantics" or using "too theoretical artifacts". Thus we deliberately try to bridge the gap between software engineering and formal methods and we provide a methodology for teams with different mindsets to collaborate effectively. The design space is vast but the current results are tangible and our hope is that by exploring it further we can reap many more benefits, helping making "semantics programming" more mainstream. The code is the spec is the code[a].

---

[a]is the doc, but this latter point can be debatable: If the reader really feels intrigued, let them please think whether the existence of this very document proves that maybe *code cannot always be the doc* ♥.

Contents

1 Introduction

1.1 Role of the document and target audience

We report on work that answers the question: *How can we write executable computational semantics in a way that is a) close to day-to-day software engineering, and b) flexible enough as to support different uses of the semantics other than execution?* Our answer is provided by following a tagless-final approach [1, 6] in the programming language Scala 3. Essentially, the present document serves both as a tutorial for the tagless-final approach to building eDSLs and expressing computational semantics, and an exposition of a rich eDSL. We have designed and implemented this eDSL in order to describe computational semantics of systems, with a blockchain ledger as the main inspiration. We build the tagless-final approach to ledger semantics progressively, with numerous examples, explaining details and rationale along the way. For the various needed details there are many designs we could have tried and followed, and indeed we have, but our full explorations are out of scope for this document.

We assume some familiarity with Scala 3. In case the reader is not familiar with Scala 3, as opposed to Scala 2, we do not anticipate much friction. Moreover, we avoid features that could be deemed exotic, such as macros and staging, although it would be interesting to use them in a continuation of this line of work[1]. Practitioners in other languages may need some exposure to Scala fundamentals in order to follow conveniently. We also assume some familiarity with notions from the Cardano Ledger Spec (CLS) [2] and also general blockchain notions, such as a ledger.

The full source code can be found as open-source on GitHub [3]. In a sense, the present document, written in LaTeX[2], also serves as a technical discussion for the source code in the repository.

1.2 Motivation

During the inaugural Plutusfest [4] in 2018, the author suggested the idea that the accumulated experience and expertise in designing and implementing blockchains at IOHK [5], should be able to

---

[1]And, in fact, we started with macros but then we decided that we wanted to pursue the challenges of the more straightforward approach, which is the one we present here.

[2]See https://github.com/input-output-hk/ce-blockchain-dsl/tree/master/doc/report

guide us come up with a DSL to describe the computational aspects and formal properties perhaps of the whole of a blockchain. Later on, the author became aware of the PDF specification of the Cardano Ledger [2].

This specification is in essence a text with the necessary definitions, algorithms and transition systems used to implement the Cardano blockchain ledger. One drawback of the current form of the specification, which reinforced the DSL idea, is that an implementer of the Cardano Ledger will have to translate the text to actual code, as many times as the chosen programming languages. At IOHK we have implemented part of the specification in Rust or Scala as well, starting from scratch in each case. In addition, the specification was written by one group of people who then had to explain it to other groups, whose task was to come up with the actual implementation. Formal method experts were also involved, in order for them to formalize the desired properties in theorem provers and related systems, generating alternative "formal specifications". Moreover, the properties and invariants that the specification defines have been translated to test suites or even theorems in an ad-hoc and manual way.

It is clear from the above that it would be beneficial to remove the burden of translating the specification manually and many times to different formats. This would be not only *cost effective* but would also indicate and approach of *higher-assurance*. The ideal situation, which we set out to explore, would be to have one source of truth from which to automatically generate all the necessary artifacts: from code, to tests, to documentation to necessary theorem prover and model checker inputs.

Of course, we started out with an exact use-case but the approach is quite general – there is nothing that ties it to a blockchain ledger. In addition we have been interested in the overarching question of "What can we do about this situation?" more than advocating a particular path. The work described here is one such path but there are many ways to devise a similar DSL and many ways to design and implement other DSLs, not necessarily *embedded* ones.

1.3 Goals

When we set out to explore the design space we had some very clear goals in mind.

- The approach is going to be intentionally *close to programming concepts and idioms*. For instance, we do not start by using an abstract language of total or partial functions and sets. Or any other abstract mathematical language. Since programming is proving [9], we choose programming as the starting point. Our premise is that if it is not enough, then either make it enough or just move on, but only after having tried.

- The approach needs to be *familiar to people that are in the end going to program the actual system*. So there is no room for theorem provers and other "formalities". The latter could be possible some day in the future, when the majority or just a critical mass of programmers will be more familiar with formal methods and tools. Note, though, that we do not exclude the use of such systems! What we are saying here is that these systems are not going to be the primary means of expression as our starting point. But we definitely aim to interface with them, as a direct continuation of this line of work.

- We start with and take inspiration from a concrete use-case (a blockchain ledger) but we always keep in mind how the outcome can be used in other contexts, that is it should have more general applicability.

- The outcome is going to be both a concrete way of defining a system and a pattern of how to approach such definitions.

- We should be able to specify some computational aspects. The users of the product, who are engineers, should be able to define algorithms and algorithmic systems.

- We should be able to specify ways to model data. The users of the product should be able to define their required types and data structures.

- We wish to explore the design space of an embedded domain-specific language in a specific meta language, Scala 3. Scala 3 was chosen because of familiarity and because of new features from Scala 2, which we wanted to explore.

- Finally, we would like to suggest this approach as a gateway drug to writing formal specifications of programs in more exotic ways.

1.4 Approach and contributions

So how would one design a language in order to express computational notions suitable for describing a ledger? Domain expertise is crucial here, which means that leveraging prior exposure to such systems is going to be beneficial. This leads to the conclusion that *given* existing ledger designs, formal or otherwise, one could abstract away their constituent parts and come up with fundamental building blocks. We are particularly interested in being able to build everything bottom-up, so we need to define almost everything that will be needed, and assume as few things as we can, just enough to make our approach practical.

For instance, we need to introduce fundamental data types, like an *integer*. In fact, we will have to introduce different kinds of integers, like the *natural numbers* $\mathbb{N}$, or the full range of integers, $\mathbb{Z}$. In addition, constrained version of those may be further necessary, like all positive, non-zero integers, $\mathbb{N}^+$. We also need to introduce fundamental data structures. Nowadays it is common to assume the existence of dictionaries (hash maps), sets, and sequences (lists, vectors).

The current description in the Cardano Ledger Spec uses a mathematical language from set theory, and abstracts things by leveraging the idea of (partial) functions, which can serve as maps, since they are *mappings*, as well. These notions are not difficult to grasp but not in everyday use by software engineers. Sometimes the notation, though not of complexity, can obscure the intention or meaning. Take for example the case of function ubalance from Figure 14: "Functions used in UTxO rules" in [2], reproduced here:

$$\text{ubalance} \in \text{UTxO} \to \text{Coin}$$
$$\text{ubalance } utxo = \sum_{(\_ \mapsto (\_, c)) \in utxo} c$$

So, just to give you an idea of what to expect, here is how we can use the eDSL to (re)define it:

```scala
def ubalance(utxo: R[UTxO]): R[Coin] =
  val coin_mapper = f["coin_mapper"] :=
    p["txout"] --> ( (txout: R[TxOut]) => txout.get(TxOut.coins) )

  val coin_seq  = utxo.values().map(coin_mapper)
  val coin_sum  = coin_seq.sum(coin_zero, coin_add)

  coin_sum
end ubalance
```

Some definitions in both cases are elided. The former is a mathematical one-liner in the language of sets, the latter expresses the denotational semantics of ubalance in a way that is both directly executable and amenable to other "interpretations" from *a single source of truth* that a real programming language compiler can understand.

Our contributions are:

- We elaborate on a tagless-final design in Scala 3 that provides a rich set of core types, data structures and their corresponding operations. The result is a non-trivial eDSL, unlike the usual presentations of tagless-final. This paper is an extensive tutorial of tagless-final in Scala, probably the first of its kind.

- The eDSL natively supports properties in the form of function preconditions and postconditions. We have designed and implemented standard boolean properties and universal quantification properties over integer intervals. More property types can be added easily and the choice of the previous two was just driven by use-cases.

- The design is implemented and the full source code is open-source on GitHub.

- We show what can be done without exotic features, such as macros. The motivation for this approach has mostly been a desire to see how far we can go in a more "constrained" environment.

- We explore design decisions beyond *standard* or *textbook* tagless-final, notably in the case of let bindings (how to define variables)[3].

- We provide two kinds of semantic interpretations: the *direct* one uses the Scala compiler directly, in order to provide implementations for all the defined computations. The *AST-based* one encodes the computational semantics as values that can be further manipulated.

- We design and implement a type reconstruction engine, which follows precisely what the Scala compiler type inference engine does, in order to lift types at the value level. This is important for the semantics interpretation that is based on our AST, since it needs to know all the types used in the user-defined programs. Based on Scala's implicits (in the new Scala 3 form of `given` statements), our type reconstruction rules are straightforward, declarative and algorithmic.

- We provide two kinds of ASTs. The first AST is a Generalized Algebraic Data Type (GADT). It is closely tied to the eDSL and faithfully reflects all the operations and corresponding types, as dictated by the tagless-final eDSL. The second AST is an ADT (so not Generalized), suitable for direct code generation.

- Based on the second AST, we provide an extensible, visitor-based, code generation framework and use it to implement a Scala 3 code generator. With the latter, we can reconstruct the original Scala source code almost faithfully, and we open up possibilities for automatic generation of property tests suites.

- We suggest a way for semantics to be built up gradually in a pragmatic way. We assume the semantics of some foundational, underlying computational notions (primitive types and operations) and build our own ledger (and other systems) semantics on top. This could provide a way on how to approach formalization in similar ways, that is compositionally.

- Whereas types and operations in the Cardano Ledger Spec may left as abstract or implicit, everything in our approach is explicit. If we need to use some operation, then the nature and semantic details must be known, regardless of whether the details may be an approximation. In essence, we explicitly declare what we need, and we have the option to refine it later.

- From the design point of view, we show that the tagless-final representation, which parametrizes the eDSL at the type level, can also be present outside the eDSL as well. In essence, we show how to seamlessly combine code in the tagless-final style and code not in the tagless-final style. This exercise is crucial because shows what can or should be done externally to the tagless-final eDSL: not everything needs to be embedded.

- Although the idea of denotational semantics is very explicit and well-defended in the papers we base our work on, here we make it the starting, and even foundational, point. If this is a mistake, it is ours only. But we are very clear and loud: Denotational semantics is a practical way to semantics engineering, and tagless-final is it's best ambassador.

---

[3]We also welcome the headaches that our explorations give rise to.

1.5 Organization

We organize the rest of the paper as follows:

Section 2 briefly outlines the tagless-final approach, using Scala 3 as the implementation language.

Section 3 unfolds our current tagless-final design, by building the core abstractions. Here, we explore the design space in Scala 3. Together with section 2, section 3 provides a complete tagless-final tutorial that builds a non-trivial eDSL.

Section 4 uses the core abstractions to build new ones, related to a blockchain[4].

Section 5 presents the two ASTs: the first one provides an AST-based interpretation of the eDSL, and the second one drives code generation.

Section 6 provides details about the code generation process, the final stage of a pipeline that starts with writing semantic definitions in the eDSL.

Section 7 makes some concluding remarks.

Appendix A has some more source code that sheds light on particular aspects of the overall design and implementation. The full details are of course in the source code repository.

2 Tagless-final

> » In this section we explain the tagless-final approach by encoding it in Scala 3, using simple examples. This exposition forms the basis of the subsequent, more elaborate developments.

The tagless-final approach [1, 6] prescribes a way to define and interpret embedded domain-specific languages (eDSLs) by specifying and composing procedural abstractions [10]. In the literature, a language that we set out to implement as an eDSL is called the *object language*, and the language that we use as an implementation vehicle for the eDSL is called the *meta language*. For instance, if we use Scala to embed a ledger-related eDSL, then Scala is the meta language and the ledger-related eDSL is the object language. In essence, we re-use the facilities of the meta language, in order to provide some guarantees (like type-safety) and fast-track the implementation of the eDSL.

The primary means for expressing the computational abstractions are functions, as opposed to using algebraic data types (ADTs), which we subsequently transform or interpret. By using procedural abstractions via the tagless-final approach, we essentially provide the semantics of our computational constructs once and in a compositional way, and then we can then interpret the semantics in different domains. The beauty of tagless final is that it is very straightforward to create interpretations, extend and combine them. Tagless-final can be thought as a *gateway drug* to denotational semantics[11][7][5].

In order to explain concretely how we model an eDSL with the tagless-final approach, we are going to use a simple language that supports booleans, integers, a negation operation for the booleans and an addition operation for the integers. We proceed directly with the code in Figure 1 on the following page. Let's dissect what is happening there, in *three steps*.

First of all, `Core` is our eDSL, which, as we discussed, is going to work on booleans and integers. One of the fundamental aspects of tagless-final is the eDSL generalization by `R[_]`[6], which denotes the abstract representation of terms. So, for instance, in the definition of the `bool()` function[7], the

---

[4]Throughout this document we may use the terms *blockchain* and *ledger* interchangeably, although technically they are not the same.

[5][7, Section 3.1.4] gives a beautiful explanation of why this approach to semantics is actually denotational, and also provides further references.

[6]For those familiar with the original tagless-final papers, `R[_]` or `R[A]` in Scala correspond to `type 'a repr` and `repr a` in OCaml and Haskell respectively.

[7]Technically, due to Scala's and the JVM's object-oriented nature this is a *method* but there is no harm using the term *function* in the present context.

```
1  trait Core[R[_]]:
2    def int(i: Int): R[Int]
3    def bool(b: Boolean): R[Boolean]
4    def not(b: R[Boolean]): R[Boolean]
5    def add(a: R[Int], b: R[Int]): R[Int]
6  end Core
```

Figure 1: Tiny eDSL core in tagless-final style.

```
1  type Id[T] = T
2
3  trait DirectCore extends Core[Id]:
4    def int(i: Int): Int = i
5    def bool(b: Boolean): Boolean = b
6    def not(b: Boolean): Boolean = !b
7    def add(a: Int, b: Int): Int = a + b
8  end DirectCore
```

Figure 2: Direct interpretation of the tiny eDSL core.

type `Boolean` is the native type that Scala provides, and `R[Boolean]` is an abstract representation of booleans for any interpreter `R` of our eDSL[8].

Second, the function definition:

```
def bool(b: Boolean): R[Boolean]
```

lifts a boolean value from the meta language (Scala) to the object language (`Core`), so that we can further manipulate it with the operations of our DSL in the context of a potential interpretation. The function definition:

```
def int(i: Int): R[Int]
```

plays the same role for integers. In essence, what we do here is to state that our eDSL supports two types, booleans and integers. For that reason, we give the end-user of the eDSL (the programmer) the means to bootstrap computations on those types by using their native counter-parts in Scala.

Third, we have the remaining functions, `not()` and `add()`, which operate on values of boolean and integer `R`epresentations. We do not a-priori know what these representations are: it depends on what kind of interpreters we want to provide, as implementations of the `Core` trait. For instance, we can provide what is usually called a *direct* interpretation, which amounts to just reusing the *native* Scala types and operations.

We materialize the latter observation in Figure 2 on the current page, where we declare that all types are represented directly by setting:

```
type Id[T] = T
```

and making sure that operations on booleans and integers have their usual meaning, i.e. `!b` in Scala is the usual boolean negation and `a + b` is the expected integer addition. We note at this point that a direct interpretation is what renders any computational semantics defined by using `Core` as *directly executable*. Now the question becomes obvious: how do we compute with `Core`? Or, similarly, should we write the computations using `Core` or should we do it based on `DirectCore`?

Figure 3 on the following page implements a small application that defines a bunch of integers and some booleans over any domain of representation `R`[9]. Notice that we choose to keep `R` generic, passing it on to `Core`. This means that regardless of any interpretation we choose, whatever term we define in `App` carries with it its computational semantics, doing so in a denotational way. For instance, `two` is defined compositionally using the previous definition of `one` and the `add` operation.

---

[8]Notice how we used `R` for both the representation and an interpreter that uses that representation. Technically speaking, we can have more than one interpreters R_1, R_2, etc for the same representation `R`.

[9]Essentially, `R` is universally quantified: Everything that `App` defines is valid ∀ `R`. We note that this universal quantification of the generic parameter `R` is a Scala feature, not a tagless-final feature.

```scala
1  trait App[R[_]] extends Core[R]:
2    val one: R[Int] = int(1)
3    val two: R[Int] = add(one, one)
4
5    val True: R[Boolean] = bool(true)
6    val False: R[Boolean] = not(True)
7  end App
```

Figure 3: An application using the tiny eDSL core, without assuming any interpretation.

As trivial as it may seem, essentially we have a computational definition that corresponds to what one would write as "2 = 1 + 1", a recipe to get "2" provided that "1" and "+" are given, only that we do not fix either the representation of "1" or what "+" computes and how! We only fix the compositional way to get a "2" from "1" and "+". As a side note, the type annotations for the terms we defined in Figure 3 are not necessary, since the Scala compiler can infer them. They are there for clarity.

This is all nice and abstract but how can we get a concrete result? Very easily, we can compose the direct interpretation of Figure 2 with our application of Figure 3:

```scala
object DirectApp extends App[Id] with DirectCore
@main def print_direct_two() = println(DirectApp.two)
```

and indeed if we run `print_direct_two()`[10] we get:

2

as expected, since `DirectCore` computes values directly!

Now let's see how we can interpret addition in a different way and what we will get as a computed value for the `two` term. This time, instead of computing using the native types and operations that Scala provides, we will construct an AST. Figure 4 on the next page gives the details. Our AST is a Generalized Algebraic Data Type (GADT). An integer, which is data, is represented by the

```scala
case IntT(i: Int) extends AST[Int]
```

constructor, and the operation of addition is represented by the

```scala
case AddT(a: AST[Int], b: AST[Int]) extends AST[Int]
```

constructor. The `extends AST[Int]` part is what it makes this a GADT, giving each AST node the more precise type of `AST[T]` where `T=Int`. If we had defined the AST as `enum AST` instead of `enum AST[T]`, then it would be possible to construct a term that adds two booleans. So GADTs make our design more precise, which leads to extra type safety during compilation time.

Our AST interpretation, as opposed to the direct one that we have seen previously, is about constructing AST nodes from every operation that the `Core` eDSL defines. Let's see what we get by composing the application we implemented (Figure 3 on this page) with the AST interpretation:

```scala
object ASTApp extends App[AST] with ASTCore
@main def print_ast_two() = println(ASTApp.two)
```

Now, running `print_ast_two()`, we obtain:

```scala
AddT(IntT(1),IntT(1))
```

as expected, since `ASTCore` builds up AST nodes. Remember that, in contrast, `DirectCore` "interprets" computations directly.

We have shown an interpretation that builds the AST in a straightforward manner; that is, as implementers of the AST interpretation we use the algebraic data type (ADT) constructors such as `IntT()` and `AddT()` to build up trees. Of course, there are other approaches. For instance, we could

---

[10]The `@main` annotation is handy, as it instructs the Scala compiler to create a *main* function that is directly executable; this behaves like the `main()` function in the C language.

```scala
1  enum AST[T]:
2    case IntT(i: Int)                      extends AST[Int]
3    case BoolT(b: Boolean)                 extends AST[Boolean]
4    case AddT(a: AST[Int], b: AST[Int]) extends AST[Int]
5    case NotT(b: AST[Boolean])             extends AST[Boolean]
6  end AST
7
8  trait ASTCore extends Core[AST]:
9    import AST.*
10
11   def int(i: Int): AST[Int] = IntT(i)
12   def bool(b: Boolean): AST[Boolean] = BoolT(b)
13   def not(b: AST[Boolean]): AST[Boolean] = NotT(b)
14   def add(a: AST[Int], b: AST[Int]): AST[Int] = AddT(a, b)
15 end ASTCore
```

Figure 4: AST interpretation of the tiny eDSL core.

use macros to write actual Scala code that is then translated to an AST, perhaps leveraging Scala's Typed Abstract Syntax Trees (TASTY)[11] or the Scala compiler's intermediate representation (IR)[12].

3   Core abstractions

» In this section we present a tagless-final eDSL, enriched with a variety of primitive types, data structures, and operations on them. We support let bindings (albeit with a new encoding) and recursion on functions. This eDSL also gives us enhanced capabilities, like user-defined properties that a software engineer can attach to functions as pre- and post-conditions, as well as user-defined data structures. The latter, "lift" Scala's *case classes* at the object language level. We begin with a discussion about the essence of the approach and we attempt to tickle the brain of the category-theory-minded (CAT) people by showing a diagram that looks ...familiar. We avoid making the diagram commutative, in order to keep the non-CAT readers engaged as well.

3.1   The essence

One may wonder: What is the essence of all this? The definitions in `Core` and `App` are completely independent of any interpretation for booleans, integers and their operations. In fact, `Core` *defines* what booleans and integers are, by providing both the means to construct them in a any interpretation, and the applicable operations. In addition, the direct interpretation plays the role of a reference implementation, which is the intended one, by design. In this respect, with the introduction of `DirectCore` the denotational semantics of any computation that uses the facilities of `Core` are directly executable. The AST interpretation then is the gateway to producing new *artifacts*: documentation from the code, e.g. a PDF like the Ledger Specification [2], properties and invariants that can subsequently be used to create a test suite or can be fed to a theorem prover.

Alternatively, we could just generate code again, perhaps in a more optimized way than what the direct interpretation provides, and see if the elaboration of AST to executable code is faithful[13] to the direct interpretation semantics. We show the latter possibility in the diagram in Figure 5 on the next page. In there `App`, `DirectApp`, `ASTApp` represent what we introduced in section 2, while `ASTCodeGenerator` and `ASTGeneratedApp` are at the core of the present work. The dashed

---

[11]Scala 3 introduced TASTY as a high-level interchange format, which contains *all* the information from the source code and even more information than JVM's `.class` files.

[12]These avenues are worth exploring as a continuation of the present work.

[13]"Faithful", of course, is up to definition: It can be "source code is the same", or "parsed syntax tree is the same" or "behavior up to property checking is the same" ...
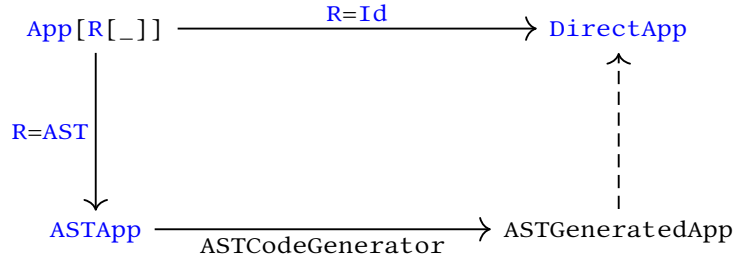
Figure 5: Executable semantics from two different paths.

line from `ASTGeneratedApp` to `DirectApp` indicates that we introduce an opportunity to check behavioral equivalence between the two. In such case the diagram would become a commutative one. Investigating such possibility is enabled by the present work but is out of scope.

3.2 More than `Core`

The `Core` presented in Figure 1 on page 7 is anemic. In a realistic scenario where the intention is to build an actual system, as opposed to showing the capabilities of the tagless-final approach, we will need at least the following:

- More types than just booleans and integers. Even the term "integer" is overloaded and usually we distinguish natural numbers in $\mathbb{N}$ from *the integers* in $\mathbb{Z}$. Clearly, Scala's 64-bit signed integer type (`Int`) is too concrete for a generalized approach.

- The full tower of the expected operations on numbers, assuming their usual semantics: for instance addition is associative and commutative etc.

- A few fundamental data structures. Sets, maps and lists are ubiquitous.

- An adequate set of operations on the data structures.

- Going beyond numbers and data structures, first-class functions are also important in order to enable higher-order programming.

- Support for binding values to labels. Essentially the ability to have variables in the object language

- Support for recursion. Quite intentionally, at this point we do not enter the discussion about *bounded* versus *unbounded* recursion, totality etc.

- Support for control constructs, like `if/then/else`.

- Support for user defined data structures. We can go very far with sets, maps, lists, but at some point we will need to create our own *structs* and *records*[14].

Our goal is to be able to have both a direct interpretation and generate more artifacts from an AST interpretation. For the current approach, we name two kinds of constructs:

- *Primitive constructs* that are defined by the `Core` system. In the direct interpretation, these primitives rely on the meta-language (Scala 3 in our case) facilities. In the AST interpretation, where we also assume we have one or more AST-based code generators, each code generator is responsible to provide the correct source code for the primitives. We have implemented a Scala source code generator, which opens up the possibilities explained in Figure 5.

---

[14]We use the terms *struct* and *record* interchangeably, as being the same concept named differently in different contexts and programming languages.

Maybe mention *protocol params* from the PDF spec, where the type assumed is a Map with existentially quantified keys. We can make things simpler – and indeed we have – by just creating a record with the needed protocol params.

Figure 5: Executable semantics from two different paths.

line from `ASTGeneratedApp` to `DirectApp` indicates that we introduce an opportunity to check behavioral equivalence between the two. In such case the diagram would become a commutative one. Investigating such possibility is enabled by the present work but is out of scope.

3.2 More than `Core`

The `Core` presented in Figure 1 on page 7 is anemic. In a realistic scenario where the intention is to build an actual system, as opposed to showing the capabilities of the tagless-final approach, we will need at least the following:

- More types than just booleans and integers. Even the term "integer" is overloaded and usually we distinguish natural numbers in $\mathbb{N}$ from *the integers* in $\mathbb{Z}$. Clearly, Scala's 64-bit signed integer type (`Int`) is too concrete for a generalized approach.

- The full tower of the expected operations on numbers, assuming their usual semantics: for instance addition is associative and commutative etc.

- A few fundamental data structures. Sets, maps and lists are ubiquitous.

- An adequate set of operations on the data structures.

- Going beyond numbers and data structures, first-class functions are also important in order to enable higher-order programming.

- Support for binding values to labels. Essentially the ability to have variables in the object language

- Support for recursion. Quite intentionally, at this point we do not enter the discussion about *bounded* versus *unbounded* recursion, totality etc.

- Support for control constructs, like `if/then/else`.

- Support for user defined data structures. We can go very far with sets, maps, lists, but at some point we will need to create our own *structs* and *records*[14].

Our goal is to be able to have both a direct interpretation and generate more artifacts from an AST interpretation. For the current approach, we name two kinds of constructs:

- *Primitive constructs* that are defined by the `Core` system. In the direct interpretation, these primitives rely on the meta-language (Scala 3 in our case) facilities. In the AST interpretation, where we also assume we have one or more AST-based code generators, each code generator is responsible to provide the correct source code for the primitives. We have implemented a Scala source code generator, which opens up the possibilities explained in Figure 5.

---

[14]We use the terms *struct* and *record* interchangeably, as being the same concept named differently in different contexts and programming languages.

Maybe mention *protocol params* from the PDF spec, where the type assumed is a Map with existentially quantified keys. We can make things simpler – and indeed we have – by just creating a record with the needed protocol params.

```
1  type Bool   = scala.Boolean
2  type String = scala.Predef.String
3  type Unit   = scala.Unit
4  type Char   = scala.Char
5
6  // We use this to define identifiers
7  // (variables, parameters etc)
8  type Name = String & Singleton
9
10 type Option[T]   = scala.Option[T]
11
12 type Pair[A, B] = (A, B)
13 type Map[K, V] = scala.collection.immutable.Map[K, V]
14 type Set[K] = scala.collection.immutable.Set[K]
15 type Seq[K] = scala.collection.immutable.IndexedSeq[K]
16
17 val Map = scala.collection.immutable.Map
18 val Set = scala.collection.immutable.Set
19 val Seq = scala.collection.immutable.IndexedSeq
20
```

Figure 6: Primitive types (a). "Simple" types and fundamental collections.

- *User-defined constructs*, mostly data structures and functions that a user of the `Core`, a software engineer, defines on top of `Core`.

In the following subsections, we explore what the enhanced `Core` provides. From now on, `Core` refers to the extended version needed in order to implement our goals, and not the introductory example of section 2.

3.3 Primitive types

We show the supported primitive types in Figures 6, 7, and 8. As far a direct interpretation is concerned, they will be the exact types used. So, in programs built using the eDSL, a `Map[K,V]` will be Scala's immutable `Map[K,V]`. Recall that in the direct interpretation, we use `type Id[T] = T` for the representation `R[_]` appearing in `Core[R[_]]`, so by simple equational-style reasoning:

$$
\begin{aligned}
\texttt{R[Map[K, V]]} &= \texttt{Id[Map[K, V]]} & \text{(by Figure 2)} \\
&= \texttt{Map[K, V]} & \text{(by Figure 2)} \\
&= \texttt{scala.collection.immutable.Map[K, V]} & \text{(by Figure 6)}
\end{aligned}
$$

One consequence of these being the exact types used in a direct interpretation, which renders our denotation semantics executable, is that they also define remaining evaluation and other semantics: evaluation order is the usual Scala evaluation order, a `Map` is assumed to have the algorithmic complexity of Scala's immutable `Map`, and of course being immutable imposes further constraints on the necessary API for `Map`s. For instance, adding a new key-value pair to a `Map` means that a new `Map` instance is returned. As a consequence, if we would like another kind of interpretation, such as the one that creates ASTs to be directly comparable to the direct interpretation, the code generated from the ASTs should have comparable semantics. Figure 5 on the previous page is relevant in this discussion, especially its dashed line on the right side.

In Figure 7 on the following page, note how we use:

- Operator overloading (<, +), in order to make programming with the eDSL feel natural. In this way, numeric computations in any interpretation will be able to use the usual operators.

- Scala 3's *extension methods*[15]. In this context we use extention methods as an implementation vehicle for operator overloading.

---

[15]See https://docs.scala-lang.org/scala3/book/ca-extension-methods.html

```
1 object Nats:
2   opaque type Nat = BigInt
3
4   object Nat:
5     def apply(i: Int): Nat =
6       scala.Predef.require(i >= 0)
7       BigInt(i)
8     end apply
9
10    extension (n: Nat):
11      def v: BigInt = n
12      def +(that: Nat): Nat = Nat(n.v + that.v)
13      def <(that: Nat): Bool = n.v < that.v
14      def ===(that: Nat): Bool = n.v == that.v
15      // ... more defs
16    end Nat
17 end Nats
```

Figure 7: Primitive types (b): Natural numbers.

```
1 type Fun0[R[_],          Z] = ()                 => R[Z]
2 type Fun1[R[_],       A, Z] = (R[A])             => R[Z]
3 type Fun2[R[_],    A, B, Z] = (R[A], R[B])       => R[Z]
4 type Fun3[R[_], A, B, C, Z] = (R[A], R[B], R[C]) => R[Z]
5 // Fun4 etc
```

Figure 8: Primitive types (c): Type aliases for first-class functions.

- Opaque type aliases[16], with the help of which we hide the implementation detail that a natural number `Nat` is actually Scala's `BigInt`. In this way, all the eDSL-based computational will rely on the eDSL's `Nat` operations. Of course, `BigInt` semantics are assumed, but without advertising the particular `BigInt` implementation.

In addition to natural numbers (ℕ), we also support the integers (ℤ) with a corresponding `Zat` opaque type, but we elide the definitions for brevity. They are almost identical to the presentation of `Nat` in Figure 7 on the current page.

Figure 8 on this page presents type aliases that are handy in defining and using functions as first-class values. Note how we always refer to values whose type is abstracted by some representation `R[_]`.

3.4 Primitive operations

The so called "primitive operations" are functions related to our primitive types and data structures. These operations are given as part of the eDSL implementation, and they are available to eDSL-based user programs. We have already shown some operations on `Nat`s in Figure 7 on the current page: addition (+) and comparison (<).

In relation to Figure 8, Figure 9 on the following page presents operations to define 0-, 1- and 2-parameter functions as well as operations for their corresponding function application. We omit definitions of bigger arities for brevity. The extension methods related to function application (via `app0()`, `app1()`, `app2()`) make use of Scala's `apply()` convention: If an object has an `apply()` method defined, then the object is callable, that is we can use it as a function[17].

So, how do we create `R[_]`-abstracted first-class functions? We need to lift a Scala method to a `Core` function, and that is precisely what `fun1()`, `fun2()` etc do. For instance, the following

---

[16]These are introduced in Scala 3 and they are like `newtype`s in Haskell. The documentation page https://dotty.epfl.ch/docs/reference/other-new-features/opaques.html provides further details.

[17]See https://www.scala-lang.org/files/archive/spec/2.13/06-expressions.html, in particular Section *6.6 Function Applications*. This is for Scala 2 but also holds for Scala 3.

```scala
1  def fun0[Z: Ty](
2    f: Fun0[R, Z]
3  ): R[Fun0[R, Z]]
4
5  def fun1[A: Ty, Z: Ty](a: Name,
6    f: Fun1[R, A, Z]
7  ): R[Fun1[R, A, Z]]
8
9  def fun2[A: Ty, B: Ty, Z: Ty](a: Name, b: Name,
10   f: Fun2[R, A, B, Z]
11 ): R[Fun2[R, A, B, Z]]
12
13 def app0[Z: Ty](f: R[Fun0[R, Z]]): R[Z]
14
15 def app1[A: Ty, Z: Ty](f: R[Fun1[R, A, Z]],
16   a: R[A]): R[Z]
17
18 def app2[A: Ty, B: Ty, Z: Ty](f: R[Fun2[R, A, B, Z]],
19   a: R[A], b: R[B]): R[Z]
20
21 extension[Z: Ty](f: R[Fun0[R, Z]])
22   def apply(): R[Z] = app0(f)
23
24 extension[A: Ty, Z: Ty](f: R[Fun1[R, A, Z]])
25   def apply(a: R[A]): R[Z] = app1(f, a)
26
27 extension[A: Ty, B: Ty, Z: Ty](f: R[Fun2[R, A, B, Z]])
28   def apply(a: R[A], b: R[B]): R[Z] = app2(f, a, b)
```

Figure 9: Primitive operations for functions.

```scala
1  trait Core[R[_]]:
2    // ...
3    def str(s: String): R[String]
4    def str_len(s: R[String]): R[Nat]
5    def str_char_at(s: R[String], i: R[Nat]): R[Char]
6    def str_concat(a: R[String], b: R[String]): R[String]
7
8    extension (s: R[String])
9      def apply(i: R[Nat]): R[Char] = str_char_at(s, i)
10     def length: R[Nat] = str_len(s)
11     def ++(t: R[String]): R[String] = str_concat(s, t)
12   // ...
13 end Core
```

Figure 10: Primitive operations for strings.

method:

```scala
def impl_triangle(a: R[Nat], b: R[Nat]): R[Nat] =
  a * a + b * b
```

is lifted by `fun2()`:

```scala
val triangle = fun2("a", "b", impl_triangle)
```

and then can be used with `app2()`:

```scala
app2(triangle, nat(0), nat(1))
```

or with the `apply()` extension method:

```scala
triangle(nat(0), nat(1))
```

The latter feels more natural in Scala. `nat()` is from Figure 11 on the next page.

In Figure 10 on this page we present operations related to strings. Of course we could have included dozens of string operations but one crucial idea of our approach is we build up what is necessary and extend as needed: our approach is a minimalistic one. The `length()` extension method enables the familiar *dot* (`.`) syntax that object-oriented programmers are so accustomed

```scala
trait Core[R[_]]:
   // ...
   def nat(x: scala.Int): R[Nat]
   def nat_add(a: R[Nat], b: R[Nat]): R[Nat]
   def nat_lt (a: R[Nat], b: R[Nat]): R[Bool]

   extension (n: R[Nat])
     def +(n2: R[Nat]): R[Nat] = nat_add(n, n2)
     def <(n2: R[Nat]): R[Bool] = nat_lt(n, n2)
   // ...
end Core
```

Figure 11: Primitive operations for natural numbers.

```scala
trait Core[R[_]]:
   // ...
   def seq_new[K: Ty](elems: R[K]*): R[Seq[R[K]]]
   def seq_add[K: Ty](seq: R[Seq[R[K]]], elem: R[K]): R[Seq[R[K]]]

   def seq_map[K: Ty, L: Ty](
     seq: R[Seq[R[K]]],
     f: R[R[K] => R[L]]
   ): R[Seq[R[L]]]

   def seq_fold_left[K: Ty, L: Ty](
     seq: R[Seq[R[K]]],
     initial: R[L],
     f: R[Fun2[R, L, K, L]]
   ): R[L]

   inline def seq_sum[K: Ty](
     seq: R[Seq[R[K]]],
     k_zero: R[K],
     seq_add: R[Fun2[R, K, K, K]]
   ): R[K] =
     seq_fold_left[K, K](seq, k_zero, seq_add)
   // ...
end Core
```

Figure 12: Primitive operations for sequences.

to[18].

   Notice how

```scala
def str(s: String): R[String]
```

lifts a value that exists at the meta-language level (Scala) to a value at the object-language level
(`Core[R[_]]`). Here, `String` is whatever we have defined as being a string (Figure 6 on page 11) in
our Scala programs but `R[String]` can be anything in the eDSL-based programs. In a direct inter-
pretation of the eDSL programs, strings can retain the exact `String` representation and semantics,
but in other interpretations they can be nodes to some AST. We can make similar observations for
`Nat`ural numbers in Figure 11.

   As a bit of a more involved demonstration of primitive operations, Figure 12 on this page gives
the ones we support for `Seq`uences. Given that the signatures may appear a bit involved, this is a
fine opportunity to dissect a few of them and understand more about:

1. The use of the `R[_]` abstract representation.

2. The computational semantics that are implied by our definitions. We need to take into account
   those semantics when designing alternative interpretations of programs based on the `Core`
   eDSL.

---

3. The use of higher-order functions, through the use of the previously defined function type aliases of Figure 8 on page 12.

First, let's start with the construction of sequences:

```scala
def seq_new[K: Ty](elems: R[K]*): R[Seq[R[K]]]
```

The idea here is that given some values of type `K` in the `R[_]` representation, we can construct a sequence of those values, and the sequence will be in the `R[_]` representation as well. In Scala the construct:

```scala
elems: R[K]*
```

used for function parameters, means we can call the function with any number of arguments of the same type[19]. So, the function `seq_new()`, both taking inputs and producing an output in the `R[_]` representation, is actually a function at the object-language level, not the meta-language level. But our object-language does not have first-class support for generics, and we need to rely on the meta-language. This is exactly the reason that this is provided as a primitive and is not constructed by other object-language facilities. The same observation on generics holds for other data structures as well.

Second, let's have a look at how we add items to a sequence:

```scala
def seq_add[K: Ty](seq: R[Seq[R[K]]], elem: R[K]): R[Seq[R[K]]]
```

We need (a) a sequence and (b) and element in order to get (c) a new sequence that includes the new element. All of those values, whether being the inputs or the output, are in the `R[_]` representation. In essence, we assume persistent/immutable[20] sequences, hence the returned instance is a new one. This assumption is part of the eDSL semantics, and the direct interpretation follows it exactly. Figure 6 on page 11 actually prescribes that the used sequence is an immutable one:

```scala
type Seq[K] = scala.collection.immutable.IndexedSeq[K]
```

Third, let's look at the ubiquitous left *fold*:

```scala
def seq_fold_left[K: Ty, L: Ty](
  seq: R[Seq[R[K]]],
  initial: R[L],
  f: R[Fun2[R, L, K, L]]
): R[L]
```

and in particular the type of its third parameter:

```scala
R[Fun2[R, L, K, L]]
```

If we use Figure 8 on page 12 to unpack it, we get

```scala
R[(R[L], R[K]) => R[L]]
```

Note how the function type

```scala
(R[L], R[K]) => R[L]
```

has been lifted in the `R[_]` representation. So, `seq_fold_left()` is a fully generic (via `K` and `L`) left fold on sequences that "live" in an `R[_]` representation. How do we construct those sequences in the first place? Of course, by using the `seq_new()` eDSL API in Figure 12 on the preceding page.

A final note regarding `seq_fold_left()`. One might argue that we could have obtained it based on a more fundamental "primitive operation", namely "recursion". There are two reasons he have not done it this way:

- By making fold a primitive, we can define and even demand once and for all it's complexity and computational semantics. Yes, this may be adhoc, but we believe it is worth experimenting on this design front and treat fold as a fundamental operation.

- The fold came into existence as a solution through the following requirement (▤) and thought (♡) process path:

  ▤ ⟶ We need to compute a sum $\sum$ over some values[21].

---

[19] The same idea is called *varargs* in the C language.

[20] We freely use "persistent" and "immutable" interchangeably.

[21] The *actual* use-case for this was function `ubalance` in Figure 14, page 22, Section 8.1 of [2], originally accessed at https://hydra.iohk.io/build/18363060/download/1/ledger-spec.pdf and archived at https://archive.ph/zx8CU.

```scala
1  trait Core[R[_]]:
2    // ...
3    // A more specific Representation, used for bindings
4    type Bind[T] <: R[T]
5
6    def bind[T: Ty](name: Name, v: R[T]): Bind[T]
7
8    def name(s: scala.Predef.String): Name
9    def name[S <: Name : ValueOf]: Name = valueOf[S]
10
11   // syntactic convenience for function parameter names
12   def p[S <: Name : ValueOf]: Name = name[S]
13
14   // syntactic convenience for function names
15   def f[S <: Name : ValueOf]: Name = name[S]
16
17   // syntactic convenience for variable names
18   def v[S <: Name : ValueOf]: Name = name[S]
19
20   extension(a: Name)
21     def :=[T: Ty](v: R[T]): Bind[T] = bind(a, v)
22
23     def -->[A: Ty, B: Ty](
24       f: Fun1[R, A, B]
25     ): R[Fun1[R, A, B]] = fun1(a, f)
26
27   extension(ab: (Name, Name))
28     def -->[A: Ty, B: Ty, Z: Ty](
29       f: Fun2[R, A, B, Z]
30     ): R[Fun2[R, A, B, Z]] = fun2(ab._1, ab._2, f)
31   // ...
32 end Core
```

Figure 13: Primitive operations for bindings.

💡 ⟶ Let's generalize the $\sum$ to a sequential[22] traversal + a general computation constrained by the types.

💡 ⟶ This is a fold, no need to re-invent the wheel!

Introducing recursion at this point would mean yet another abstraction. This is definitely intriguing from the theoretical point of view, and also a solved problem in tagless-final: the encoding of fix is known. We decided to stop at one abstraction. Should more requirements arise that warrant the introduction of recursion, then yes we would consider it.

3.5 Binding names to values

Core[R[_]] would be quite limited without support for variables. Especially when our goal is to provide a modern and familiar "user experience" (UX)[23]. Figure 13 on the current page presents our eDSL facilities for binding values to names. The intention is to re-construct Scala's

```scala
val name = value
```
statement in the eDSL. We will see further in the paper that such a desire for familiar syntax creates complications for the AST-based backend[24]. Nevertheless, it has been a fruitful design decision.

Let's concentrate on Figure 13. The Name type comes from Figure 6 on page 11 and its introduction is to serve bindings: they need a *name* after all[25], so this type is for identifiers. Then, bind() associates the name with the given value. In eDSL-based code this means that in order for an engineer to create a variable for the zero natural number, they would write:

```scala
val zero = bind(name["zero"], nat(0))
```

---

[22]We humbly ask for Guy Steele's forgiveness, https://www.youtube.com/watch?v=JbvnhuiHYxA.

[23]Remember that the user in our case is the software engineer.

[24]In addition, the astute reader may have noticed that our introduction of *let* bindings does not follow the usual tagless-final design. We'll come to this matter in a subsequent section of the paper.

[25]Since we are targeting human engineers, the idea of a *De Bruijn index* is alien.

where the `nat()` function comes from Figure 11 on page 14. We can use the `:=` extension to make the value assignment look like:

```
val zero = v["zero"] := nat(0)
```

Whether someone prefers the former versus the latter is probably a matter of taste; still the options are on the table.

In case of a function, let's say we have a Scala (meta-language) definition with the following signature:

```
def impl_min_fee(pparams: R[PParams], tx: R[Tx]): R[Coin]
```

If, for the time being, we ignore what `PParams`, `Tx` and `Coin` are, we can define an eDSL (object-language) variable, so as to be able to re-use the function as first-class within the object-language:

```
val min_fee = f["min_fee"] := (p["pparams"], p["tx"]) --> impl_min_fee
```

Notice how:

```
(p["pparams"], p["tx"]) --> impl_min_fee
```

resembles a function type with two parameters. Unpacking it, we see that it is equal to calling `fun2()` from Figure 9 on page 13:

```
fun2(p["pparams"], p["tx"], impl_min_fee)
```

and then the binding declaration:

```
f["min_fee"] := (p["pparams"], p["tx"]) ...
```

is equivalent to:

```
bind(f["min_fee"], fun2(p["pparams"], p["tx"], impl_min_fee))
```

Again, which one to use is a matter of preference. The idea here is to be able to experiment with syntax by providing a couple of options (but not too many!), so that teams can decide which style they prefer.

3.5.1 To name or not to name

You may be wondering what is the role of names as strings in all these bindings. A related question is: If you are building a compiler, won't the compiler be able to automatically generate names for all the bindings? In fact the latter is true but the idea is to use names that are meaningful to the software engineer, instead of relying on randomly-looking names. After all, we aim for readability even in the generated source code. We need to stress that the user of the eDSL is responsible to provide names that will not lead to errors: for instance, accidentally using the same name in the same scope. The backend we describe in a subsequent section does not catch such errors, so this can be an area of future work.

3.5.2 To let or not to let

The reader may have noticed that our support for bindings is unconventional or non-standard. Indeed, let bindings introduce lexical scope, either explicitly, as in OCaml's `let name = value in` construct or implicitly, as in Scala's `val name = value` statement; the latter implies that `name` is visible and has the given `value` in subsequent statements in the current lexical scope. So, the standard way to introduce let bindings is to leverage functions, since they are act as providers of lexical scope for their parameters:

```
trait Core[R[_]]:
  // ...
  def _let[A: Ty, Z: Ty](bind: Bind[A], in: Fun1[R, A, Z]): R[Z]

  inline def let[A: Ty, Z: Ty](bind: Bind[A])(in: Fun1[R, A, Z]): R[Z] =
    _let(bind, in)
  // ...
end Core
```

The second definition is just syntactic convenience, leveraging Scala's multiple parameter lists feature. So, with let bindings, this function:

17

```
def impl_min_fee(pparams: R[PParams], tx: R[Tx]): R[Coin] =
  val a = v["a"] := pparams.get(PParams.a)
  val b = v["b"] := pparams.get(PParams.b)

  val n_txsize = v["n_txsize"] := tx_size(tx)
  val txsize   = v["txsize"]   := zat_of_nat(n_txsize)

  val z_coins = v["z_coins"] := a * txsize + b
  val coins   = v["coins"]   := coin_of_z(z_coins)

  coins
end impl_min_fee
```

can be re-written as:

```
def impl_min_fee2(pparams: R[PParams], tx: R[Tx]): R[Coin] =
  let( v["a"]       := pparams.get(PParams.a)     )( a =>
  let( v["b"]       := pparams.get(PParams.b)     )( b =>
  let( v["n_txsize"] := tx_size(tx)               )( n_txsize =>
  let( v["txsize"]   := zat_of_nat(n_txsize) )( txsize =>
  let( v["z_coins"]  := a * txsize + b       )( z_coins =>
  let( v["coins"]    := coin_of_z(z_coins)   )( coins =>
    coins
  )))))) // Quite LISP-y, don't you think?
end impl_min_fee2
```

In the accompanying implementation, our emphasis has been to explore the first, non-standard pattern. By the way, we have just shown the precise denotational semantics of "minimum fee", translated from Figure 9, page 14 of [2]. Here, though, the definition is directly executable. This is exactly what we want to have as source of truth for the definition of a computation.

3.5.3   To be or not to be

Let's move on, please, there is no such question.

3.6   User-defined structs

We now present a way to introduce user-defined data structures. One can think of them as a means to create something like Scala's *case classes* but at the eDSL (object language) level. We use the term *structs* to refer to these data *struct*ures and for their implementation we leverage extensive support in Scala 3 for *tuple* type inference. Appendix A provides the necessary types, data, and operations to define and use such structs.

Let's build a user-defined struct bottom-up. Assume we need to create a `Slot`[26] struct, which contains one field named v that has a type of `Nat`. The field represents just the slot value number.

First, we introduce what is called a *bare field*. A bare field is re-usable, in the sense that we can include it in the definition of more than one structs. We call it "bare" because it is not a-priori attached to any struct. This is how we create v:

```
val v_Nat = bare_field["v", Nat]
```

Then, we reuse the meta-language case class construct as a container of the object-language struct:

```
case class SlotStruct(v: R[Nat]) extends AnyStruct
```

where `AnyStruct` is just a marker trait that we use for all the user-defined structs. The `Struct` suffix in `SlotStruct` is our programming convention to differentiate and recognize these special case classes. The direct interpretation is going to use `SlotStruct` as it is, without any extra overhead.

The next step is to define our `Slot` struct and for that we need its fields and a constructor definition, the latter being responsible for struct instantiation. In our design, we designate the struct fields by gathering all the bare fields that apply to a particular struct, and the constructor is automatically derived by using the `struct_constructor()` primitive operation (in Appendix A):

```
val Fields = Tuple1(v_Nat)
```

---

[26] A *slot* in the Cardano protocol denotes a time period.

18

```
1  trait Chain[R[_]] extends Core[R]:
2    object Slot {
3      case class SlotStruct(v: R[Nat]) extends AnyStruct
4      val Fields = Tuple1(bf.v_Nat)
5      val Constr = struct_constructor[SlotStruct]
6
7      val Struct = def_struct(Fields, Constr)
8      val v      = def_field(Struct, bf.v_Nat, _.v)
9    }
10   type Slot = Slot.SlotStruct
11 end Blockchain
```

Figure 14: A user-defined `Slot` struct.

```
1  trait Chain[R[_]] extends Core[R]:
2    // ...
3    def slot_of_n(n: R[Nat]): R[Slot] =
4      Slot.Struct(Tuple1(n))
5
6    def impl_slot_gte(a: R[Slot], b: R[Slot]): R[Bool] =
7      a.get(Slot.v) >= b.get(Slot.v)
8    end impl_slot_gte
9
10   lazy val slot_gte =
11     f["slot_gte"] := (p["a"], p["b"]) --> impl_slot_gte
12
13   extension(a: R[Slot])
14     def >=(b: R[Slot]): R[Bool] = impl_slot_gte(a, b)
15   end extension
16   // ...
17 end Chain
```

Figure 15: Instantiating and using `Slot`s.

```
val Constr = struct_constructor[SlotStruct]
```
We are now ready to define the struct along with its field as first-class objects in the eDSL:
```
val Struct = def_struct(Fields, Constr)
```
```
val v = def_field(Struct, v_Nat, _.v)
```
Since we assume there will be more than one user-defined structs in a program that is based on the eDSL, we use Scala objects as namespaces for all the necessary definitions. Figure 14 gathers the previous definitions according to this object-based namespace pattern. In there, note how we use a new `trait` to extend the `Core` eDSL. `Chain` denotes a blockchain or any of its parts, such as a ledger. Also, `slot_of_n()` acts as a constructor function for `Slot`s.

Finally, Figure 15 shows how we can construct a `Slot` instance and define a `Slot` comparison function. In there,
```
Slot.Struct(Tuple1(n))
```
is essentially equivalent to
```
Slot.Struct.apply(Tuple1(n))
```
which uses the `apply()` extension method (see Appendix A) to construct an instance, and
```
a.get(Slot.v) >= b.get(Slot.v)
```
uses the `get()` extension method (see Appendix A) to retrieve `Slot`'s v field from a previously constructed instance.

3.7 If Then Else

Support for the if/then/else control construct is straightforward and is given by these two primitive operations:

```scala
trait Core[R[_]]:
  // ...
  def ifte[T: Ty](i: R[Bool], t: => R[T], e: => R[T]): R[T]
  def if_then_else[T: Ty](i: => R[Bool])
                         (t: => R[T])
                         (e: => R[T]): R[T] = ifte(i, t, e)
  // ...
end Core
```

The second is just syntactic convenience for the first one. In the direct interpretation, the implementation looks like:

```scala
trait DirectCore extends Core[Id]:
  // ...
  def ifte[T: Ty](cond: Bool, t: => T, e: => T): Id[T] =
    if cond then t else e
  // ...
end DirectCore
```

Notice how we use `=> R[T]` instead of `R[T]`, so as to delay evaluation of the branches. In effect, when an eDSL user writes:

```scala
if_then_else(condition)(then_branch)(else_branch)
```

none of the `then_branch`, `else_branch` will be evaluated at call site. This is standard Scala *call-by-name* semantics.

3.8 Recursive functions

The standard way to handle recursion in tagless-final is via a *fixpoint combinator*, usually introduced as the `fix()` primitive operation. This is beautiful from the theoretical point of view and quite general in nature but we decided to introduce it only when needed[27] and use other means if they are available. And indeed they are. In fact, we let the Scala compiler tie the knot *in its usual way*, and we only require a syntactic convention for recursion to work as expected with our eDSL. We make these clear with the following example.

Let's just use the *factorial* function: $\forall n > 0 : n! = n * (n-1)!, 0! = 1$. A definition in the meta language (Scala) is:

```scala
def fact(n: Int): Int =
  if n == 0 then 1 else n * fact(n - 1)
```

A corresponding definition in the object language (`Core`) is:

```scala
def impl_fact(n: R[Nat]): R[Nat] =
  if_then_else[Nat](n === 0)(1)(n * fact(n - 1))
lazy val fact = f["fact"] := (p["n"]) --> impl_fact
```

Apart from the types (e.g. `Int` vs `R[Nat]`), there are two notable differences, which generalize to respective requirements when defining recursive functions.

1. A recursive function in `Core` is represented by a Scala method, and an eDSL variable. The method is Scala's native way of doing this kind of computations, and the variable is how the computation is lifted to the `Core` level. This is actually a general pattern for all functions, not only recursive ones. An implementation note: the use of `lazy val` instead of `val` is related to how initialization happens in Scala traits, and is a defensive mechanism to avoid the dreaded `NullPointerException`.

2. Within the Scala method, a recursive call uses only the `Core` variable `fact`, not the Scala method name `impl_fact`. You can think about it like this: we need object language (eDSL) symbols not meta language (Scala) compiler symbols, so that the resulting eDSL code can

---

[27]See the discussion about fold at the end of subsection 3.4.

be properly analyzed: this distinction is crucial for non-direct interpretations like the AST-based one of the present work. The expected direct interpretation can ignore (or forgive!) such a distinction. So, for instance, we could use `impl_fact()` instead of `fact()` in the `if_then_else()` branch, with almost identical computational results, the difference being the introduction of a function variable instead of using the actual function.

3.9 Properties

We now proceed to showing how one can model properties[28] for the functions defined using the eDSL. Usually, this kind of properties are not supported natively by most of the programming languages, so they are encoded either with extension facilities, like annotations, or within test suites. A notable exception is the programming langue Eiffel, which natively supports these facilities, promoting the *design by contract* approach. Our intention is to enable this "formal" design approach from within our eDSL. In essence, we will give an eDSL user the ability to define such properties and attach them to respective functions. As an educational example, we will use the well-studied[29] `leftpad()`[30] function.

One of the possible ways to write `leftpad()` in Scala is like this:

```scala
def leftpad(s: String, n: Int, c: Char): String =
  require(n >= 0)

  if s.length >= n then s else leftpad(c.toString ++ s, n, c)
end leftpad
```

where we can see a precondition that uses Scala's built-in `require()` function. In Figure 16 on the next page we translate the meta language `leftpad()` definition to an object language definition, `_leftpad()`. The translation is straightforward. Now, let's say we want to also attach the `require(n >= 0)` precondition, as long as two more postconditions to the definition of `_leftpad()`. In particular we want these three properties:

- *Precondition 1*: $n \geqslant 0$.
  The requested total length of the resulting string, after "padding", must be non-negative.

- *Postcondition 1*: $length(result) = max(n, length(s))$.
  The length of the resulting string, after padding, must be either the requested total length or the length of the original string, whichever is bigger.

- *Postcondition 2*: $\forall i \in [0, N), N = max(0, n - length(s)) \implies result[i] = c$.
  In the resulting string, each character inserted "on the left" must be equal to the input character used for padding.

In the above, *length* computes the length of a string, *result* is the return value of `_leftpad()`, and *result*[i] is the 0-indexed *i*-th character from the return value. Also, these are not the only properties we can define. You may also want to encode one more:

- *Postcondition 3*: *(exercise left for the reader)*.
  All the characters from the original string will be the same characters and in the same position in the final string as well.

Figure 17 on the following page has all the details. We define each property in a separate function. Note how a *property function* related to the precondition has the same *input* type signature as

---

[28]"Properties" here are as in "property based testing" à la QuickCheck. Preconditions and postconditions have been the inspirational use-case. See also wikipedia: https://en.wikipedia.org/wiki/Precondition and https://en.wikipedia.org/wiki/Postcondition.

[29]See https://www.google.com/search?q=let's+prove+leftpad.

[30]See https://www.google.com/search?q=leftpad+fiasco. `leftpad()` came into prominence in 2016, after a supply chain attack. You may also want to check https://archive.ph/dcBNO, which is an archived version of http://left-pad.io/.

```
1  trait LeftPad[R[_]] extends Core[R]:
2    def _leftpad(s: R[String], n: R[Nat], c: R[Char]): R[String] =
3      if_then_else( s.length >= n ) {
4        s
5      } {
6        leftpad(c.to_str ++ s, n, c)
7      }
8    end _leftpad
9  end Leftpad
```

Figure 16: Definition of `leftpad()` (a): Core implementation.

```
1  trait LeftPad[R[_]] extends Core[R]:
2    // continued
3    def precond1(s: R[String], n: R[Nat], c: R[Char]): R[Prop[R]] =
4      prop_bool( n >= nat(0) )
5
6    def postcond1(s: R[String], n: R[Nat], c: R[Char],
7      result: R[String]
8    ): R[Prop[R]] =
9      prop_bool( result.length === max(n, s.length) )
10
11   def postcond2(s: R[String], n: R[Nat], c: R[Char],
12     result: R[String]
13   ): R[Prop[R]] =
14     prop_forall_in["i", Nat](
15       (nat(0), max(0, n - s.length)),
16       UpperBoundIsExclusive,
17       (i) => c === result(i)
18     )
19 end Leftpad
```

Figure 17: Definition of `leftpad()` (b): Properties.

the *function of interest*, `_leftpad()`. This is expected, since preconditions should know about the function input but nothing more. In contrast, postconditions should also have access to the *output* of the function of interest, hence the introduction of an extra parameter, `result`. We will explain the return type `R[Prop[R]]` and its helper functions `prop_bool()`, and `prop_forall_in()` shortly.

Precondition 1: $n \geqslant 0$

This is a simple *boolean property* that translates to:

```
prop_bool(n >= nat(0))
```

Everything is straightforward and works as expected here. The eDSL provides us with a way to lift Scala's zero to a `Nat` zero (see Figure 11 on page 14), and the overloaded comparison operator `>=` makes programming feel natural. The new piece of the puzzle is `prop_bool()`, so it is time to explain how we model properties in `Core`.

Figure 18 on the following page shows the fundamental type that we have introduced to model properties: `Prop[R[_]]`. The meaning of `R[_]` is the same as that in `Core[R[_]]`. Currently, we model two kinds of properties:

1. Boolean properties, given as any boolean predicate. This is the `BoolP` constructor of the `Prop` algebraic data type (ADT).

2. Universal quantifier (∀) properties, with an extra constraint. This is the `ForAllInRangeP` constructor. The extra constraint is that values of the corresponding variable are taken to be in a specific interval, and there is provision on whether the upper bound is inclusive or

22

```
1  enum Prop[R[_]]:
2    case BoolP(predicate: R[Bool])
3
4    // The lower bound (first in the tuple) is always inclusive, while
5    // the upper bound can be either inclusive or exclusive.
6    case ForAllInRangeP[X <: (Nat | Zat), R[_]](
7      name: Name,
8      ty: Ty[X],
9      bounds: (R[X], R[X]),
10     upper_bound_is: BoundIs,
11     predicate: R[Fun1[R, X, Bool]]
12   ) extends Prop[R]
13 end Prop
```

Figure 18: The `Prop[R[_]]` type that supports the definition and use of properties.

exclusive. In this definition, values are either of `Nat` or `Zat` types. Recall that we have not defined other types of numbers in `Core`.

Appendix A provides details on how to construct these two kinds of properties in any interpretation. Other kinds of properties can be easily added. For instance, a property of the second kind, where the constraint can be more general than interval inclusion could be defined as:

```
enum Prop[R[_]]:
  // ...
  case ForAllP[X <: (Nat | Zat), R[_]](
    name: Name,
    ty: Ty[X],
    constraint: R[Fun1[R, X, Bool]],
    predicate: R[Fun1[R, X, Bool]]
  ) extends Prop[R]
```

We can easily see that a `ForAllP` can express a `ForAllInRangeP`, provided that any interval bounds are based on the known property inputs and the new universally quantified variable[31].

Postcondition 1: $length(result) = max(n, length(s))$

Here, we have a boolean property again, which translates to:

```
prop_bool( result.length === max(n, s.length) ).
```

Notable features from the eDSL point of view is the use of Scala's extension methods, in order to:

- Make `result` appear like an object with a `length` field. In reality, this delegates to the `str_length()` primitive operation (see Figure 10).

- Have a dedicated `Nat` equality primitive operation `===` (see Figure 7).

The `max()` function is a handy utility method defined in `Core` as:

```
trait Core[R[_]]:
  // ...
  def impl_nat_max(a: R[Nat], b: R[Nat]): R[Nat] =
    if_then_else(a >= b)(a)(b)
  end impl_nat_max

  lazy val nat_max = f["nat_max"] := (p["a"], p["b"]) --> impl_nat_max

  inline def max(a: R[Nat], b: R[Nat]): R[Nat] = nat_max(a, b)
  // ...
end Core
```

where `if_then_else()` was described in subsection 3.7. The Scala 3 keyword `inline` instructs the compiler to *always inline*, as opposed to the pre-existing `@inline` annotation, which is provisional.

---

[31]Otherwise a `Fun1`, that is a function of one parameter, would not be enough.

```
1  trait LeftPad[R[_]] extends Core[R]:
2    // continued
3    val params      = (p["s"], p["n"], p["c"])
4    val params_pre  = params
5    val params_post = params ++ Tuple1(p["result"])
6
7    lazy val leftpad =
8      ( f["leftpad"] := params --> _leftpad ).
9        add_precond3 ( params_pre  --> precond1 ).
10       add_postcond3( params_post --> postcond1 ).
11       add_postcond3( params_post --> postcond2 )
12 end Leftpad
```

Figure 19: Definition of `leftpad()` (c): Bringing it all together.

Postcondition 2: $\forall i \in [0, N), N = max(0, n - length(s)) \implies result[i] = c$

This is a *universal quantifier property*, as explained previously. It translates to:

```
prop_forall_in["i", Nat](
  (nat(0), max(0, n - s.length)),
  UpperBoundIsExclusive,
  (i) => result(i) === c
)
```

As with the simpler boolean properties, we can see that the translation of a universal quantifier property is syntactically very close to the defining mathematical expression.

We now have all the ingredients to present the final definition of `leftpad()`. Figure 19 on the current page creates the proper binding, and attaches to it all the relative properties. The primitive operations `add_precond3` and `add_postcond3` are responsible for this kind of "attachment". We elide their definition for brevity. Also, notice how the parameter list of a postcondition is just the parameter list of a precondition, with the `p["result"]` parameter appended to it, as expected:

```
val params_post = params ++ Tuple1(p["result"])
```

These definitions related to parameters are given in order to avoid repetition. There is no other essential requirement for their existence. One could even complain that

```
val params_pre = params
```

is purely aesthetic, but such is sometimes the nature of programming [8].

3.10 Types, finally

There are two aspects related to types that are crucial to providing a correct user experience using the eDSL. *The first one* is about providing a safe API that users can program against. When we add two values of type `Nat`, the signature of the addition operation reflects the intention and constraints what is possible:

```
def +(a: R[Nat], b: R[Nat]): R[Nat]
```

Or, when we need a condition for the `if_then_else()` construct:

```
def if_then_else[T: Ty](i: => R[Bool])(t: => R[T])(e: => R[T]): R[T]
```

we know that the condition should be a representation of a `Bool` and not a representation of a `Nat`, and the Scala compiler verifies this statically. Computational semantics encoded in this way are given the respective safety guarantees by design.

The second aspect is about carrying these types along to all interpretations, and making them available as first-class citizens. The reason for this may not be obvious at first sight, since we are used to the fact that the only "thing" to worry about types is the Scala compiler. When we write `a + b` for some variables, a, b:

24

```
val a: R[Nat]
val b: R[Nat]
```
we know that we do not particularly have to think much about how a direct interpretation is carried out, since `R[Nat]` is projected to just `Nat` and then `+` is a standard operation, compiled and available for execution. Our program is "compiled away" by Scala, and then `R[Nat]` and `Nat` are types that the Scala compiler has to worry about, not our eDSL infrastructure.

But the moment we need to think about other interpretations, the question arises: Is the actual type `R[Nat]` still an implementation detail that we do not need to worry about? Or does it need special provision as part of the design of the eDSL and its accompanying infrastructure? We can actually make the reasoning a bit more interesting, if we use a function that is parametrically polymorphic.

Let's recall Figure 12 on page 14, and in particular the polymorphic primitive operation that constructs a sequence embedded in `R[_]`:
```
def seq_new[K](elems: R[K]*): R[Seq[R[K]]]
```
This operation seems to be able to construct a sequence for *any* type `K`. Now, let's make a thought experiment: Imagine an interpretation that for a particular use of `seq_new[K]()`, needs to know *all* the information about `K`. For instance, if `K` is instantiated as the concrete `String`, this piece of information should be known because the particular interpretation could be a code generator with strict rules about knowing all the exact types of all the expressions it needs to handle.

In essence, if we want the eDSL to be as general as possible, *all* the information that the Scala compiler has about types must be forwarded somehow to *every* interpretation! But the type information the Scala compiler has lives during Scala compilation, and what we ask for leads, by necessity, to type information *out-living* the Scala compilation phase, essentially entering the runtime phase of a Scala program. The solution is to lift types to values and pass them along.

The astute reader may have noticed that in the `seq_new()` signature we have cheated a little bit. The *actual* signature is:
```
def seq_new[K: Ty](elems: R[K]*): R[Seq[R[K]]]
```
where the difference is in `[K]` versus `[K: Ty]`. In fact, this `Ty` constraint on a generic type `K`, also called a *context bound*, has appeared previously in a multitude of places, such as Figure 8 on page 12, Figure 12 on page 14, and Figure 13 on page 16.

In Scala, a function with a context bound, is desugared to a function with an extra parameter list that contains an implicit parameter[32], so that `seq_new()` becomes:
```
def seq_new[K](elems: R[K]*)(using Ty[K]): R[Seq[R[K]]]
```
The idea here is that when such a function is used, the Scala compiler needs to have in the lexical scope a *witness* of type `Ty[K]`, that is a value of `Ty[K]`. This value of `Ty[K]` is precisely our internal representation of the Scala type `K` that the eDSL infrastructure carries along to every interpretation.

What are these witnesses? We need *two* things to make the respective description complete: the actual representation of `Ty` values and a mechanism to make these values.

As far as the representation is concerned, Figure 20 on the following page shows the `Ty[K]` algebraic data type. The fundamental idea behind this ADT is that all the types of interest at the object language level (eDSL) should have a representation in `Ty[K]`. We can see, for instance, that `NatTy` is the singleton witness for the `Nat Ty`pe:
```
case NatTy extends Ty[Nat]
```
Generic types, such as a `Seq[A]`, can have a witness given a witness of their generic type parameter `A`:
```
case SeqTy[A](a: Ty[A]) extends Ty[Seq[A]]
```
In the above, `a: Ty[A]` is a witness for `A`. Similar reasoning extends to a `Map[A, B]`.

And now, on to the constructive part. We'll proceed by example, without providing an exhaustive list. Hopefully, the idea will become clear. Each item in the following list has three parts: *(a)* A notation that resembles a type inference rule, which will be familiar to some readers; we do not attempt to be rigorous here, just familiar. *(b)* A textual description of the rule, that is how to obtain

---

[32]See https://docs.scala-lang.org/scala3/reference/contextual/index.html for a full discussion on implicits.

```scala
 1  enum Ty[K]:
 2    case NatTy    extends Ty[Nat]
 3    case BoolTy   extends Ty[Bool]
 4    case CharTy   extends Ty[Char]
 5
 6    case RTy[X, R[_]](inner: Ty[X]) extends Ty[R[X]]
 7
 8    case PropTy[R[_]]() extends Ty[Prop[R]]
 9
10    case Fun0Ty[Z]    (z: Ty[Z])              extends Ty[() => Z]
11    case Fun1Ty[A, Z](a: Ty[A], z: Ty[Z]) extends Ty[(A) => Z]
12    // ...
13
14    case MapTy[A, B](a: Ty[A], b: Ty[B])  extends Ty[Map[A, B]]
15    case SetTy[A]    (a: Ty[A])              extends Ty[Set[A]]
16    case SeqTy[A]    (a: Ty[A])              extends Ty[Seq[A]]
17
18    case StructTy[Struct <: AnyStruct](tag: scala.reflect.ClassTag[Struct])
19        extends Ty[Struct]
20  end Ty
```

Figure 20: Types at the value level.

---

a witness for a type, given some assumptions. *(c)* A translation of the above into the computational language of Scala implicits.

- $\boxed{\text{NAT-TY-W}}$ $\quad \dfrac{\text{Nat} : Type}{w(\text{Nat}) = \text{Ty.NatTy}}$

  Given the type `Nat`, I can create its singleton witness, `Ty.NatTy`.

  ```scala
  given given_NatTy : Ty[Nat] = Ty.NatTy
  ```

- $\boxed{\text{BOOL-TY-W}}$ $\quad \dfrac{\text{Bool} : Type}{w(\text{Bool}) = \text{Ty.BoolTy}}$

  Given the type `Bool`, I can create its singleton witness, `Ty.BoolTy`.

  ```scala
  given given_BoolTy: Ty[Bool] = Ty.BoolTy
  ```

- $\boxed{\text{R-TY-W}}$ $\quad \dfrac{w(\text{T}) : \text{Ty[T]}, \quad \text{R[\_]} : Type}{w(\text{R[T]}) : \text{Ty[R[T]]} = \text{Ty.RTy}(w(\text{T}))}$

  Given a value of `Ty[T]` as a witness of a generic type `T`, and a particular (known) representation `R[_]`, I can create a witness for `R[T]`, `R[T]` being the lifted type `T` in the representation.

  ```scala
  given given_RTy[T](using w_t: Ty[T]): Ty[R[T]] = Ty.RTy(w_t)
  ```

- $\boxed{\text{FUN1-TY-W}}$ $\quad \dfrac{w(\text{A}) : \text{Ty[A]}, \quad w(\text{Z}) : \text{Ty[Z]}}{w(\text{A => Z}) : \text{Ty[A => Z]} = \text{Ty.Fun1Ty}(w(\text{A}), w(\text{Z}))}$

  Given a function type `A => Z` and witnesses for its input/output types `A` and `Z`, I can create a witness for the function type.

  ```scala
  given given_Fun1Ty[A, Z](using a: Ty[A], z: Ty[Z]): Ty[A => Z] =
    Ty.Fun1Ty(a, z)
  ```

- $\boxed{\text{SEQ-TY-W}}$ $\quad \dfrac{w(\text{T}) : \text{Ty[T]}}{w(\text{Seq[T]}) : \text{Ty[Seq[T]]} = \text{Ty.SeqTy}(w(\text{T}))}$

  Given a value of `Ty[T]` as a witness of type `T`, I can create a witness for the type `Seq[T]` of sequences.

  ```scala
  given given_SeqTy[T](using aw_t Ty[T]): Ty[Seq[T]] = Ty.SeqTy(w_t)
  ```

26

In essence, the above algorithmic rules reconstruct the types that appear in our eDSL code, at the value level. The rules follow the Scala compiler's type inference process, creating witnesses of the inferred types. This way we transcend compile-time and move on to runtime in a completely declarative way. We model this *type inference* process using a trait parameterized by the interpretation `R[_]`. Below we show a glimpse of this model, while the reader can find more rules in Appendix A.

```scala
trait TypeInference[R[_]]:
  given given_NatTy : Ty[Nat]   = Ty.NatTy
  given given_RTy[X](using x_ty: Ty[X]): Ty[R[X]] = Ty.RTy(x_ty)
  given given_Fun1Ty[A, Z](using a: Ty[A], z: Ty[Z]): Ty[A => Z] =
    Ty.Fun1Ty(a, z)
  given given_SeqTy[A](using a: Ty[A]): Ty[Seq[A]] = Ty.SeqTy(a)
  // ...
end TypeInference
```

`Core` uses `TypeInference` like this:

```scala
trait Core[R[_]]:
  object TypeInference extends TypeInference[R]
  import TypeInference.given
  // ...
end Core
```

The `object TypeInference extends TypeInference[R]` statement "fixes", within `Core`, type inference for whatever `R[_]` was given to it, and the `import TypeInference.given` statement brings all the rules into scope, so that primitive operations in `Core` can proceed with proper type inference and respective witness generation in place. Definitions like the previously shown:

```scala
def seq_new[K: Ty](elems: R[K]*): R[Seq[R[K]]]
```

can now be compiled and used for all `K`s ($\forall$ `K`) that our type inference process cares about.

4   Blockchain abstractions

Having laid the groundwork, it is now straightforward to provide definitions for a lot of data types and computations from the Cardano Ledger Spec, our originating use-case and motivation. In page 3 of [2] there are definitions like these:

Sequences Given a set X, $X*$ is the set of sequences having elements taken from X...

Functions $A \rightarrow B$ denotes a **total function** from A to B. Given a function f we write fa for the application of f to argument a.

Maps and partial functions $A \mapsto B$ a partial function from A to B, which can be seen as a map (dictionary) with keys in A and values in B...

Option type An option type in type A is denoted as $A^? = A + \lozenge$. The A case corresponds to the case when there is a value of type A and the $\lozenge$ case corresponds to the case when there is no value.

In our approach, these definitions are replaced with corresponding computational notation, which also has a meaning in a programming language, and is therefore familiar to software engineers. For instance, we have seen that sequences are represented by the `Seq[X]` primitive type, and we can make, if we desire so, assumptions about how they behave. These assumptions can be derived from the semantics and the algorithmic complexity of Scala's `immutable.Seq` and are applicable to any direct interpretation: in a sense, the direct interpretation we choose to implement can guide us to a "reference computational semantics". We can use this reference semantics as a basis to "complete" the diagram of Figure 5 on page 10, in the sense of making it a commutative one. It's up to us to specify the "base" computational semantics that every representation `R[_]` should adhere to. Following a similar reasoning as with `Seq`s, `Map`s have a corresponding primitive type as well. Functions are the usual functions and we can model total ones by the type `A => Option[B]`, where:
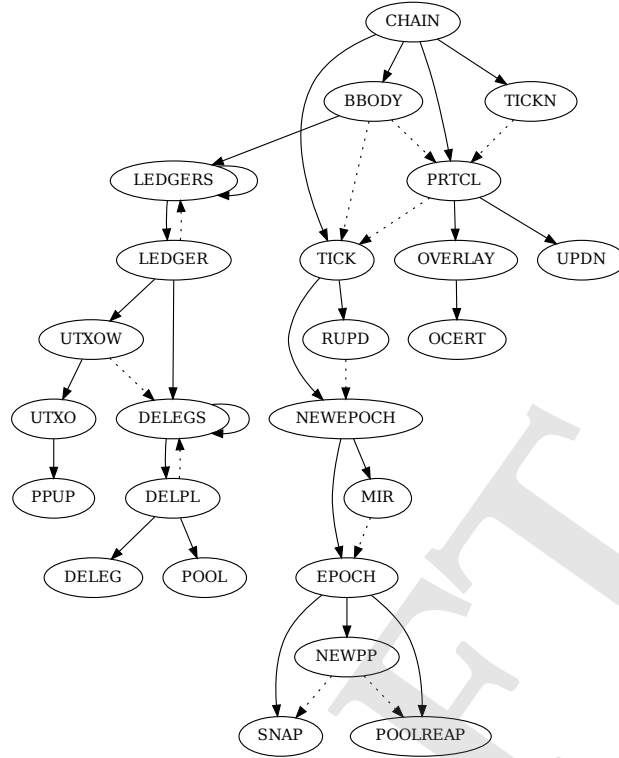
Figure 21: Figure 78 "STS Rules, Sub-Rules and Dependencies" from the Cardano Ledger Spec.

```
type Option[T] = scala.Option[T]
```

and two corresponding primitive operations ("constructors") can be:

```
def option_none[T: Ty](): R[Option[R[T]]]
def option_some[T: Ty](value: R[T]): R[Option[R[T]]]
```

These programming notions are more familiar to programmers than the mathematical notions and can be, from the mathematical point of view, as precise as we like them to be.

Of course, we never set out to fully (re)define what is in the Cardano Ledger Spec but, even at the beginning of our journey, we had to make a decision on where to start from. Fortunately, the Spec authors had provided a dependency diagram in their Figure 78: "STS Rules, Sub-Rules and Dependencies"[33], reproduced here in Figure 21 on this page. We picked UTXO , as it seemed to combine two features: a) minimum dependencies, which means we would have to worry about fewer things, and b) anything related to UTxOs is an interesting enough use-case, since UTxOs are so fundamental.

We model blockchain abstractions by extending `Core` with `Chain`:

```
trait Chain[R[_]] extends Core[R]:
  import TypeInference.given
  // ...
end Chain
```

The repetition of the `import` statement is necessary due to Scala's scoping rules, that is the implicits represented by `TypeInference.given` are not automatically inherited from `Core` to `Chain`. From now on in this section, we will assume that code is within the `Chain` definition, unless ex-

---

[33]See https://archive.ph/j5ghH.

plicitly stated otherwise. Also, any Figure references in the code correspond to the Cardano Ledger Spec (designated in the code as `ledger-spec`), not this document. By necessity, some familiarity with respective notions from the Cardano Ledger Spec is necessary, in order to follow this section comfortably.

4.1 Warm-up

GenesisDelegation

We can start with something simple, in fact one of our initial attempts:

```
// fig.2, p.5, ledger-spec
// Hash of a key
type KeyHash = String
type KeyHash_VRF  = KeyHash
type KeyHash_Pool = KeyHash
type KeyHash_G    = KeyHash

// fig.12, p.19 ledger-spec
// genesis delegations
type GenesisDelegation =
  Map[R[KeyHash_G], R[Pair[R[KeyHash], R[KeyHash_VRF]]]]
```

where `String` and `Map` are `Core`'s primitive types. You can see successive "applications" of `R[_]` to compose types from other types, in a bottom-up fashion:

$$
\begin{aligned}
\text{KeyHash, KeyHash\_VRF} &\rightarrow \text{R[KeyHash], R[KeyHash\_VRF]} \\
&\rightarrow \text{Pair[R[KeyHash], R[KeyHash\_VRF]]} \\
&\rightarrow \text{R[Pair[R[KeyHash], R[KeyHash\_VRF]]]} \\
&\rightarrow \text{Map[R[KeyHash\_G], R[Pair[R[KeyHash], R[KeyHash\_VRF]]]]} \\
&\rightarrow \text{GenesisDelegation} \\
&\rightarrow \text{R[GenesisDelegation]}
\end{aligned}
$$

The ideas of "bottom-up" and "compositional" are fundamental within the context of denotational semantics and tagless-final. We have shown it here for types, but it will also be evident in computations.

Please note that we have defined:

`type GenesisDelegation = Map...`

and not:

`type GenesisDelegation = R[Map...]`. It is easy to see that a `Chain` function that needs an instance of `GenesisDelegation` will be encoded as:

`def gd_fun(gd: R[GenesisDelegation]): ....`

and not as:

`def gd_fun(gd: GenesisDelegation): ....`

unless it is the operation that lifts a `GenesisDelegation`, so a `Map`, to an `R[_]` value:

`def gd_new(gd: GenesisDelegation): R[GenesisDelegation]`

in the same way that

`def zat(scala.Int): R[Zat]`

does the lifting for integers (`Zat` is to $\mathbb{Z}$ what `Nat` is to $\mathbb{N}$).

Why `type KeyHash = String` and not some more specific type? In particular, this type alias in Scala gives `KeyHash` all the `String` operations, so clearly the domain of `String`s is leaking into the domain of `KeyHash`es. If `KeyHash = String`, can we then use `str_concat()` from Figure 10 on `KeyHash`es? From the point of view of the Scala compiler, we can but the domain semantics are wrong. In essence, we need an opaque type, like what we did in Figure 7 on page 12 for `Nat`. Nevertheless, we presented the approach as being a fast translation from the Cardano Ledger Spec figures to some executable Scala, and it has served its purpose well towards our proof-of-concept. We have, though, another ace up our sleeve: user-defined structs.

4.2 A little bit further

KESPeriod

The encoding of user-defined structs in its present form is the outcome of a deliberate exercise in design. Let's see how we can put them to use for ledger semantics. Here is KESPeriod:

```scala
// Bare fields
object bf:
  def F[N <: FieldName: ValueOf, F: Ty]: BareField[N, F] =
    bare_field[N, F]

  // A Nat value named "v" in generated code
  val v_Nat = F["v", Nat]
  // ... more bare fields
end bf

// fig.7, p.13 ledger-spec
object KESPeriod {
  case class KESPeriodStruct(v: R[Nat]) extends AnyStruct
  val Fields = Tuple1(bf.v_Nat)
  val Constr = struct_constructor[KESPeriodStruct]

  val Struct = def_struct(Fields, Constr)
  val v      = def_field(Struct, bf.v_Nat, _.v)
}
type KESPeriod = KESPeriod.KESPeriodStruct
```

Here is Slot, which we will need shortly, together with KESPeriod:

```scala
// Absolute slot
// p.10 ledger-spec
object Slot {
  case class SlotStruct(v: R[Nat]) extends AnyStruct
  val Fields = Tuple1(bf.v_Nat)
  val Constr = struct_constructor[SlotStruct]

  val Struct = def_struct(Fields, Constr)
  val v      = def_field(Struct, bf.v_Nat, _.v)
}
type Slot = Slot.SlotStruct
```

From page 11 of the Cardano Ledger Spec we have the following, *emphasis is ours*:

> Note that *Slot is an abstract type*, while the constants are integers. *We use multiplication and division symbols on these distinct types without being explicit about the types and conversion.*

Clearly, there is a difference from the above statement and the approach we take here. We are explicit about the types and their operations. "Explicit" does not mean unable to handle abstractions. The very existence of the direct and AST interpretations, as potentially two ends of a spectrum, shows we do not shy away from abstractions: they are at the heart of tagless-final.

With KESPeriod and Slot defined, we can proceed to the denotation semantics of the KES period calculation of a slot:

```scala
// fig.9, p.14 ledger-spec
// Denotational semantics of: KES period of a slot
protected def impl_kes_period(slot: R[Slot]): R[KESPeriod] =
  val n_slot = v["n_slot"] := n_of_slot(slot)
  val n_kesp = v["n_kesp"] := n_slot / Slots_Per_KES_Period
  val kesp   = v["kesp"]   := kes_period_of_n(n_kesp)

  kesp
end impl_kes_period

lazy val kes_period = f["kes_period"] := p["slot"] --> impl_kes_period
```

where:

```
// Absolute slot
// p.10 ledger-spec
inline def n_of_slot(s: R[Slot]): R[Nat] = s.get(Slot.v)

// fig.8, p.14 (constants)
protected def impl_Slots_Per_KES_Period: R[Nat]

lazy val Slots_Per_KES_Period =
  v["Slots_Per_KES_Period"] := impl_Slots_Per_KES_Period

protected def impl_kes_period_of_n(n: R[Nat]): R[KESPeriod] =
  KESPeriod.Struct(Tuple1(n))

lazy val kes_period_of_n =
  f["kes_period_of_n"] := p["n"] --> impl_kes_period_of_n
```

Now, `kes_period` is a symbol that can be used anywhere else in eDSL-based code that requires the particular calculation. This means that `impl_kes_period()` is an implementation detail the eDSL-based user code should not care about.

As a side note, the expression `s.get(Slot.v)` typechecks because `Slot.v` is a `Slot`'s field that is derived from the bare field `bf.v_Nat`. Although the bare field stands on its own and is re-usable, `Slot.v` is created in a way that the type of the respective struct is attached to it's type, so that it cannot be used in an irrelevant context. Using **??** on page **??**, `s.get(Slot.v)` is desugared to `dot(s, Slot.v)`.

---

pv_can_follow()

As yet another example, we specify the `pv_can_follow()` function:

```
// fig.7, p.13 ledger-spec
// protocol version
type ProtVer = Pair[R[Nat] /* m */, R[Nat] /* n */]

// fig.12, p. 19
protected def impl_pv_can_follow(a: R[ProtVer], b: R[ProtVer]): R[Bool] =
  // (m, n)
  val m  = v["m"]  := pair_first(a)
  val n  = v["n"]  := pair_second(a)

  // (m_, n_), which is (m', n') in the PDF
  val m_ = v["m_"] := pair_first(b)
  val n_ = v["n_"] := pair_second(b)

  // (m + 1,  0) == (m', n')
  ((m + 1) === m_) && (0 === n_) ||
  // (m,   n + 1) == (m', n')
  (m === m_) && ((n + 1) === n_)
end impl_pv_can_follow

lazy val pv_can_follow =
  f["pv_can_follow"] := (p["a"], p["b"]) --> impl_pv_can_follow
```

where the `Core` primitive type `Pair` and its operations `pair_first()` and `pair_second()` are the "expected" ones:

```
// outside Core
type Pair[A, B] = (A, B)

trait Core[R[_]]:
  // ...
  def pair_new    [A: Ty, B: Ty](a: R[A], b: R[B]): R[Pair[R[A], R[B]]]
  def pair_first [A: Ty, B: Ty](pair: R[Pair[R[A], R[B]]]): R[A]
  def pair_second[A: Ty, B: Ty](pair: R[Pair[R[A], R[B]]]): R[B]
  // ...
end Core
```

5 The AST-based interpretation

If `Core` and `Chain` qualify as the "front-end", we certainly need to have a look at the "back-end". The direct interpretation is rather ...direct and we will not present it further in this paper. Here we are concerned about our two ASTs. The next section will comment on the code generation part.

5.1 The first AST

The first AST is close to the eDSL and follows the flow of the types via the operations. It is a Generalized Algebraic Data Type (GADT):

```scala
enum AST[T](val ty: Ty[T]):
  case ...
  case ...
  // ...
end enum
```

It has been introduced to serve the AST-based interpretation:

```scala
trait ASTCore extends Core[AST]:
  // ...
end ASTCore
```

where the representation `R[T]` is now fixed to `AST[T]`. As a reminder, in the direct representation we have `R[T] = Id[T] = T`, so each type is itself. Each node in the AST has a type `T` and for that type we require a witness `val ty: Ty[T]`. The latter is obtained via the type inference mechanism explained in subsection 3.10 on page 24. All primitive operations are modeled as `AST[T]` node constructors, where `T` represents the return type of an operation.

So, for instance, the `Map` constructor in `Core`:

```scala
def map_new[K: Ty, V: Ty](): R[Map[R[K], R[V]]]
```

is given a direct interpretation of:

```scala
def map_new[K: Ty, V: Ty](): Map[K, V] = Map()
```

For its AST interpretation we just construct the corresponding node:

```scala
def map_new[K: Ty, V: Ty](): AST[Map[AST[K], AST[V]]] =
  val k_ty = summon[Ty[K]]
  val v_ty = summon[Ty[V]]
  val ty = summon[Ty[Map[AST[K], AST[V]]]]

  AST.MapNewT(k_ty, v_ty, ty)
end map_new
```

The summon`[Ty[K]]` Scala 3 built-in *summons a witness* for our eDSL's `Ty[K]` inferred type. `MapNewT`, as a constructor of the `AST[T]` GADT, is defined as:

```scala
case MapNewT[K, V](
  k_ty: Ty[K],
  v_ty: Ty[V],
  override val ty: Ty[Map[AST[K], AST[V]]]
) extends AST(ty)
```

For other operations, we proceed in a similar fashion. For instance, binding values to names and defining parameters to functions:

```scala
enum AST[T](val ty: Ty[T]):
  // ...
  case BindT[T](
    name: String, value: AST[T],
    override val ty: Ty[T],
    creation: BindTCreation,
    pre_cond: List[AST[?]] = Nil,
    post_cond: List[AST[?]] = Nil
  ) extends AST(ty)

  case ParamT[T](name: String, override val ty: Ty[T]) extends AST(ty)
  // ...
end AST
```

or representing properties, functions, and function application:

```scala
enum AST[T](val ty: Ty[T]):
  // ...
  case PropT(prop: Prop[AST]) extends AST(Ty.PropTy[AST]())

  case Fun1T[A, Z](
    a: ParamT[A],
    f_applied: () => AST[Z], // f(a), computed on demand
    f: Fun1[AST, A, Z],
    override val ty: Ty[Fun1[AST, A, Z]]
  ) extends AST(ty)

  case App1T[A, Z](
    f: AST[Fun1[AST, A, Z]],
    a: AST[A],
    override val ty: Ty[Z]
  ) extends AST(ty)
  // ...
end AST
```

Even our fold (see subsection 3.4) gets its node:

```scala
case SeqFoldLeftT[K, L](
  seq: AST[Seq[AST[K]]],
  initial: AST[L],
  f: AST[Fun2[AST, L, K, L]],
  k_ty: Ty[K],
  l_ty: Ty[L]
) extends AST(l_ty)
```

Interpreting `seq_fold_left()` of Figure 12 on page 14 in the `AST[T]`-based semantics requires building the proper node:

```scala
def seq_fold_left[K: Ty, L: Ty](
  seq: AST[Seq[AST[K]]],
  initial: AST[L],
  f: AST[Fun2[AST, L, K, L]]
): AST[L] =

  val k_ty = summon[Ty[K]]
  val l_ty = summon[Ty[L]]

  SeqFoldLeftT(seq, initial, f, k_ty, l_ty)
end seq_fold_left
```

Introspection, recursion and laziness

A usual "complaint" about tagless-final is we cannot introspect into the function bodies. For the `Core` operation:

```scala
def fun1[A: Ty, Z: Ty](a: Name, f: Fun1[R, A, Z]): R[Fun1[R, A, Z]]
```

the direct interpretation can be as straightforward as:

```scala
def fun1[A: Ty, Z: Ty](a: Name, f: Fun1[Id, A, Z]): Fun1[Id, A, Z] = f
```

or, in its desugared form:

```scala
def fun1[A: Ty, Z: Ty](a: Name, f: A => Z): A => Z = f
```

It is clear that in the usual direct interpretation an eDSL function is just a lambda abstraction from the meta language, so one does not have access to the function body.

But introspection *is* possible, and the trick is to craft a representation that helps towards this direction, in our case `AST[T]`. An `AST[T]` node is a value that we can look at and manipulate at runtime. We'll also add some more ingredients along the way:

```scala
def fun1[A: Ty, Z: Ty](a_name: Name, f: Fun1[AST, A, Z]):
  AST[Fun1[AST, A, Z]] =

  val a_ty = summon[Ty[A]]
  val ty   = summon[Ty[Fun1[AST, A, Z]]]

  val a: ParamT[A] = ParamT[A](a_name, a_ty)
  val f_applied = () => f(a)

  Fun1T(a, f_applied, f, ty)
end fun1
```

The crucial details are in the two lines 7 and 8 above. We synthesize a parameter a of the correct type and then we apply the function f to the parameter, albeit in a *delayed* fashion, as `() => f(a)` indicates. Now, `f_applied` can compute the body of the function! The reason we do not do that in an immediate way:

```scala
val f_applied = f(a)
```

is to avoid an infinite loop if the respective function calls itself. By taking into account the discussion of subsection 3.8 on page 20, everything is in place for introspecting all details of an `AST[T]` node, including recursive function definitions. Furthermore, pattern-matching does not need any machinery other than the `AST[T]` nodes themselves. This leads us to the second AST.

5.2 The second AST

There are various reasons for the introduction of a second AST:

- The fully typed `AST[T]` is deliberately close to the syntax that `Core` exposes via its operations: conceptually, it is almost a *concrete* instead of an *abstract* syntax tree. A code generator does not necessarily need the information in the same form. For instance, information can be consolidated:

  - Combinations of `BindT` (Figure 13) with `Fun1T`, `Fun2T` etc. (Figure 8 and Figure 9) can be collapsed to one `FunDefT` node for function definitions. Generating code from one such general function definition ensures that the right code is concentrated in one place. We will show `FunDefT` shortly.

  - Binary operations that are represented by different `AST[T]` nodes, such as `BoolBinOpT` for boolean logic operators (or, and), `ComparisonBinOpT` for comparison operators (<, =), and `ArithBinOpT` for arithmetic operators (+, ∗), can all be represented by one `BinOpExprT` node. We will show `BinOpExprT` shortly.

- While the "frontend" (eDSL) does not separate bindings for functions from bindings for non-functions, a code generation backend may need to know which case it is dealing with because code can look different in each case. For instance, in Scala, we may emit a `def` for a function (method) and a `val` for other variables. The eDSL, though, models `bind()`-ings without distinction, and this is a feature. So some post-processing of the `AST[T]` may be needed.
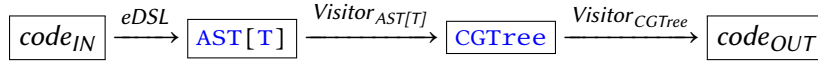
Figure 22: Code generation end-to-end pipeline.

- At some point we realized that our code generation framework for Scala became too tangled to the rest of the `AST[T]`-related code (post-)processing. What if we needed to introduce a code generator for another language? This is of course "Software Engineering 101" but in any case, the power of a Minimum Viable Product (MVP) cannot be under-estimated. Our code generation library was using a visitor already. This is a modular approach but only as far as the *unprocessed input* is concerned: given `AST[T]` nodes as input, if you need to transform it before producing the code generation output, then a good strategy is to write the transformation and code generation as two separate operations that are composed together in a modular way.

- When generating code for a target language, there are constructs that the backend needs to generate, which do not exist in the frontend in a literal form. For instance, `import` statements or all the primitive type and operations definitions. There is no corresponding `AST[T]` node for them, so we need a new construct.

The above led to our second AST, `CGTree`, where the `CG` prefix stands for *Code Generation*. `CGTree` is a *simple* ADT, not a GADT:

```scala
enum CGTree:
  // Expressions (user-defined)
  case LiteralExprT(value: Literal)
  case BinOpExprT  (op: BinOp,    a: CGTree, b: CGTree)
  case CallExprT   (f: CGTree, types: Tys, args: CGTrees)
  // ...

  // Struct operations
  case StructNewExprT(name: String, fields: NameCGTrees)
  case DotExprT      (prefix: CGTree, field: NameType)

  // Definitions
  // val name: ty = value
  case ValDefT(name_ty: NameType, value: CGTree, is_lazy: Bool)
  // def name(params): ret_ty = body
  case FunDefT(
    name: String,
    params: NameTypes,
    ret_ty: Ty[?],
    body: CGTrees,
    inline_is: InlineIs,
    pre_cond: List[CGProp],
    post_cond: List[CGProp]
  )
  // ...
end CGTree
```

where we can see the previously mentioned `FunDefT` and `BinOpExprT`. The full definition can be found in the source code repository.

6   Code generation

We mentioned that we use the visitor pattern to process `AST[T]` nodes. We do the same for `CGTree` nodes. The respective end-to-end pipeline is shown in Figure 22 on this page. Both *code$_{IN}$* and *code$_{OUT}$* are Scala 3 in our case. The choice for *code$_{IN}$* is rigid and has already been made in the context of the present work, since our eDSL design is using Scala 3 for the role of the meta language. On the other hand, in the current design the choice for *code$_{OUT}$* is completely open. We just picked

Scala 3 again, in order to be able to directly compare eDSL code with generated code *for the same semantics definition*. Figure 22 is also related to Figure 5 on page 10.

6.1 Dependencies

In going from AST[T] to CGTree, we use *Visitor*$_{AST[T]}$ implementations more than once. The obvious goal is of course to create CGTree nodes. In order to do this cleanly in one pass, we re-use the visitor infrastructure once more (but with a different implementation) to compute some needed dependencies. Let's see how this relates to binding names to values and the subtle dependency details we need to handle. Remember (subsection 3.5 on page 16) that we have chosen for eDSL code to resemble *standard* Scala code as much as possible, so binding is done with :=, an extension method delegating to Core's bind() (Figure 13 on page 16).

```
def impl_duration_of_slots(a: R[Slot], b: R[Slot]): R[Duration] =
  val n_a  = v["n_a"]  := n_of_slot(a)
  val n_b  = v["n_b"]  := n_of_slot(b)
  val diff = v["diff"] := n_b - n_a

  Duration.Struct(Tuple1(diff))
end impl_duration_of_slots

lazy val duration_of_slots =
  f["duration_of_slots"] := (p["a"], p["b"]) --> impl_duration_of_slots
```

where Duration is defined like Slot (subsection 4.2 on page 30):

```
// fig.7, p.13 ledger-spec
object Duration {
  case class DurationStruct(v: R[Nat]) extends AnyStruct
  // ...
}
type Duration = Duration.DurationStruct
```

and n_of_slot() translates a Slot to a Nat (again, subsection 4.2). In the AST interpretation, the body of duration_of_slots() essentially[34] is this AST[T] node:

```
1  StructNewT[Duration](
2    BindT[Nat](
3      "diff",
4      ArithBinOpT[Nat](
5        "-",
6        BindT[Nat](
7          "n_b",
8          StructDotT[Slot, Slot.v](
9            ParamT[Slot]("b"),
10           Slot.v
11         )
12       )
13       BindT[Nat](
14         "n_a",
15         StructDotT[Slot, Slot.v](
16           ParamT[Slot]("a"),
17           Slot.v
18         )
19       )
20     )
21   )
22 )
```

So, in the eyes of the visitor, the body of duration_of_slots() is just one expression! Where have the val statements gone? Simply, there are no statements, we only have type-full expressions linked to other expressions. Clearly, we need to process these expressions further, in order to recover the original intention of the programmer. We need to calculate dependencies *and* make sure we emit them in the correct order: This is exactly what the second *Visitor*$_{AST[T]}$ does. The net result of the

---

[34]This is not an exact string representation of the AST, we omit or change some details in order to save space, without compromising on the essence.

compositions of the two visitors is a list of `CGTree` nodes, which represent the (re)generated body of `duration_of_slots()`:

```
1  List(
2    ValDefT(                              val n_a = v["n_a"] := n_of_slot(a)
3      "n_a", NatTy,
4      DotExprT(
5        RefT("a", StructTy[Slot]),
6        Slot.v, NatTy
7      )
8    ),
9
10   ValDefT(                              val n_b = v["n_b"] := n_of_slot(b)
11     "n_b", NatTy,
12     DotExprT(
13       RefT("b", StructTy[Slot]),
14       Slot.v, NatTy
15     )
16   ),
17
18   ValDefT(                              val diff = v["diff"] := n_b - n_a
19     "diff", NatTy,
20     BinOpExprT(
21       "-",
22       RefT("n_b", NatTy),
23       RefT("n_a", NatTy)
24     )
25   ),
26
27   StructNewExprT(                       Duration.Struct(Tuple1(diff))
28     "Duration",
29     RefT("diff", NatTy)
30   )
31 )
```

Now we have clear definitions for `n_a`, `n_b` and `diff`, and any reference to them is given by a `RefT()` node. The resemblance to the specification of `duration_of_slots()`, with respective fragments copied on the right column, is clear. Note also how we have gone from a fully typed AST node (`BindT[Nat]()`), to a node where the type is represented only by its witness at the value level (`ValDefT(NatTy)`). The former is an `AST[T]` node, the latter is a `CGTree` node.

6.2 Primitive operations

We know that the `Core` requires, among other things, the presence of an addition operator for the `Nat` type, but how do we make sure a particular code generator will provide it? The best we can do is to simply enumerate all the requirements to the generator, so that at least there is no excuse about forgetting to emit some code. This is the role of the `PrimFun` enum:

```
enum PrimFun:
  case Nat          // Nat constructor
  case Str_Len      // length of a string
  case Map_New      // Map constructor
  case Map_Put      // Map.put(key, value) operation
  case Seq_Fold_Left // (left) fold is a primitive !
  // ...
end PrimFun
```

Then, we require that each code generator implements a corresponding visitor for primitive operations:

```
abstract class PrimFunVisitor:
  type Ctx = SourceCodeWriter
  type Result = Unit

  def visit_Nat(pf: PrimFun, ctx: Ctx): Result
  def visit_Str_Len(pf: PrimFun, ctx: Ctx): Result
  def visit_Map_New(pf: PrimFun, ctx: Ctx): Result
  def visit_Map_Put(pf: PrimFun, ctx: Ctx): Result
  def visit_Seq_Fold_Left(pf: PrimFun, ctx: Ctx): Result

  // ...
end PrimFunVisitor
```

6.3 Properties

Each property is compiled to its own function, where a proper `require()` or `ensure()` is emitted, depending on whether this is a pre- or a post- condition. Using the `leftpad()` definitions from subsection 3.9, we now come full circle with everything: from the original description to the generated code:

Precondition 1: $n \geqslant 0$

```
def leftpad_pre_cond1(s: String, n: Nat, c: Char): Unit =
  val n_upper_value: Nat = n
  val n_lower_value: Nat = nat(0)

  require(
    (n_upper_value >= n_lower_value),
    "require(n_upper_value >= n_lower_value)"
  )
end leftpad_pre_cond1 //(s: String, n: Nat, c: Char): Unit
```

Postcondition 1: $length(result) = max(n, length(s))$

```
def leftpad_post_cond1(s: String, n: Nat, c: Char, result: String): Unit =
  val res_length: Nat = str_len(result)
  val s_length: Nat = str_len(s)

  ensure(
    (res_length == nat_max(n, s_length)),
    "ensure(res_length == nat_max(n, s_length))"
  )
end leftpad_post_cond1 //(s: String, n: Nat, c: Char, result: String): Unit
```

Postcondition 2: $\forall i \in [0, N), N = max(0, n - length(s)) \implies result[i] = c$

```
def leftpad_post_cond2(s: String, n: Nat, c: Char, result: String): Unit =
  val lower: Nat = nat(0)
  val pad_length: Nat = nat_max(nat(0), (n - str_len(s)))
  val upper: Nat = pad_length

  for i <- lower until upper do
    val result_i: Char = str_char_at(result, i)

    ensure(
      char_eq(c, result_i),
      "ensure(char_eq(c, result_i))"
    )
  end for
end leftpad_post_cond2 //(s: String, n: Nat, c: Char, result: String): Unit
```

The fact that with the AST interpretation we can fully manipulate code that we write in `Core` or `Chain` cannot be understated. Note, for instance, how the `require()` and `ensure()` calls give,

as their string description, the very requirement expression itself. In this particular case we use the code generator twice or, more precisely, the code generator (re)uses itself.

7 Concluding remarks

We have presented a pragmatic way where day-to-day software engineering can go hand in hand with semantics engineering. This is just the tip of the iceberg. We believe that we need languages and environments, where semantics notions truly unite with the usual programming notions and are first-class. But this is a few PhDs (or software engineering milestones or both) further down the road...

Transition systems In the code-base, there is an initial design for supporting state transition systems, as those described in the Cardano Ledger Spec. Historically, after having picked UTxO , as explained at the beginning of section 4, we started with a high-level design that can model any box in Figure 21, and then reasoned backwards to see what `Core` and `Chain` should look like. The transition system design should work with the direct interpretation but we have not extracted respective primitives in order to support the AST interpretation, and so, code generation. This observation leads us to some comments for more future work.

Some future work The list is by no means exhaustive, it is just indicative.

- The eDSL is comprehensive, yet still incomplete. More constructs could be provided.

- The Scala code generator needs to be a bit more robust. Also some implementations need to be filled in.

- We could explore creating more code generators, for instance WASM and JavaScript would be useful, given the ubiquitous nature of the Web and the amount of software that has started targeting these technologies.

- Another dimension to consider for code generators is about theorem provers, model checkers and all this family of formal systems. We could also support more property types, for instance invariants.

- All the ingredients for generating property test suites for ScalaCheck are in place, so an implementation should be straightforward. In fact, the code-base already includes an extension of `Core` with proper generators for some `Core` types.

8 Acknowledgments

References

[1] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009. 2, 6

[2] Jared Corduan, Polina Vinogradova, and Matthias Güdemann. A formal specification of the cardano ledger, "Shelley Ledger Spec SL-D5 v.1.0". https:

`//hydra.iohk.io/build/18363060/download/1/ledger-spec.pdf`, `https://archive.ph/zx8CU`, 2019. 2, 3, 4, 9, 15, 18, 27

[3] IOHK Creative Engineering. "Denotational Semantics for the masses" code repository. `https://github.com/input-output-hk/ce-semantics-dsl`, 2022. 2

[4] IOHK. Launching Plutus and Marlowe at the inaugural Plutusfest, Edinbrough. `https://archive.ph/xaFXI`, 2018. 2

[5] IOHK. Web site. `https://iohk.io`, 2022. 2

[6] Oleg Kiselyov. Typed tagless final interpreters. *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, pages 130–174, 2012. 2, 6

[7] Oleg Kiselyov and KC Sivaramakrishnan. Eff directly in OCaml. *Electronic Proceedings in Theoretical Computer Science*, 285:23–58, Dec 2018. 6

[8] Donald E. Knuth. Computer programming as an art. `https://doi.org/10.1145/1283920.1283929`, 1974. 24

[9] ncatlab.org. Proofs as programs. `https://ncatlab.org/nlab/show/proofs+as+programs`, 2019. 3

[10] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, pages 309–317, 1978. 6

[11] Wikipedia. Denotational semantics. `https://en.wikipedia.org/wiki/Denotational_semantics`, 2022. 6

A   Appendix

Types and data supporting user-defined structs

```scala
type FieldName = Name /*& Singleton*/

// Marker trait for user-defined structs
trait AnyStruct extends Product

case class StructDef[
  Struct <: AnyStruct,
  // definition of fields, as obtained from new_bare_field()
  Fields <: Tuple,
  // constructor, as obtained from struct_constructor()
  Args   <: Tuple
](
  name: Name,
  ty: Ty[Struct],
  fields: Fields,
  constr: Args => Struct
)

type StructDefs = List[StructDef[?, ?, ?]]

case class FieldDef[
  Struct <: AnyStruct,
  N <: FieldName,
  F,
  R[_]
](struct: StructDef[Struct, _ <: Tuple, _ <: Tuple],
  field: BareField[N, F],
  getter: Struct => R[F]
)

case class BareField[N <: FieldName, F](name: N, ty: Ty[F])
```

Operations supporting user-defined structs

```scala
trait Core[R[_]]:
  // ...
  import scala.deriving.Mirror

  def bare_field[N <: FieldName: ValueOf, F:Ty]: BareField[N,F] =
    BareField(valueOf[N], summon[Ty[F]])

  transparent inline def struct_constructor[Struct <: AnyStruct](
    using StructM: Mirror.ProductOf[Struct]
  ) =
    StructM.fromProduct(_: StructM.MirroredElemTypes)

  transparent inline def struct_constructor[Struct <: AnyStruct](
    using StructM: Mirror.ProductOf[Struct]
  ) =
    StructM.fromProduct(_: StructM.MirroredElemTypes)

  transparent inline def def_struct[Struct <: AnyStruct : Ty,
    Fields <: Tuple, Args <: Tuple
  ](
    fields: Fields,
    constr: Args => Struct
  ): StructDef[Struct, Fields, Args] =
    val struct_ty = summon[Ty[Struct]]
    val name = struct_ty.repr

    StructDef(name, struct_ty, fields, constr)
  end def_struct

  transparent inline def def_field[Struct <: AnyStruct,
    Fields <: Tuple, Args <: Tuple, N <: FieldName, F
  ](
    struct: StructDef[Struct, Fields, Args],
    field: BareField[N, F],
    getter: Struct => R[F]
  ): FieldDef[Struct, N, F, R] =
    FieldDef(struct, field, getter)
  end def_field
  // ...
end Core
```

Instantiating user-defined structs

```scala
trait Core[R[_]]:
  // ...
  def struct_new[
    Struct <: AnyStruct: Ty,
    Fields <: Tuple,
    Args <: Tuple
  ](
    struct_def: StructDef[Struct, Fields, Args],
    args: Args
  ): R[Struct] /* implementation is R[_]-specific */

  // struct(field_def)
  extension [
    Struct <: AnyStruct: Ty,
    Fields <: Tuple,
    Args <: Tuple
  ](struct_def: StructDef[Struct, Fields, Args])
    def apply(args: Args): R[Struct] =
      struct_new(struct_def, args)
  // ...
end Core
```

42

Accessing fields of user-defined structs

```scala
trait Core[R[_]]:
  // ...
  // struct.dot(field_def)
  def dot[Struct <: AnyStruct: Ty, N <: FieldName, F: Ty](
    struct: R[Struct],
    field_def: FieldDef[Struct, N, F, R]
  ): R[F]

  // struct.get(field_def)
  extension [Struct <: AnyStruct: Ty](struct: R[Struct])
    def get  [N <: FieldName, F: Ty](
      field_def: FieldDef[Struct, N, F, R]
    ): R[F] = dot(struct, field_def)

  // field_def(struct)
  extension [Struct <: AnyStruct: Ty, N <: FieldName, F: Ty](
    field_def: FieldDef[Struct, N, F, R])
    def apply(struct: R[Struct]): R[F] = struct(field_def)
  // ...
end Core
```

Primitive operations for properties

```scala
trait Core[R[_]]:
  // ...
  def prop(p: Prop[R]): R[Prop[R]]

  // This is meant to be used internally, as a building block block for the eDSL itself
  // TODO enforce the nature of its use via Scala facilities.
  // See also AST transformation and code generation for properties.
  def prop_bool(expr: R[Bool]): R[Prop[R]] =
    prop( Prop.BoolP(expr) )
  end prop_bool

  // This is meant to be used internally, as a building block block for the eDSL itself
  // TODO enforce the nature of its use via Scala facilities.
  // See also AST transformation and code generation for properties.
  def prop_forall_in[X <: (Nat | Zat) : Ty](
    name: Name,
    bounds: (R[X], R[X]),
    upper_bound_is: BoundIs,
    predicate: Fun1[R, X, Bool]
  ): R[Prop[R]] =
    val predicate_f = fun1(name, predicate)
    val ty = summon[Ty[X]]

    prop( Prop.ForAllInRangeP(name, ty, bounds, upper_bound_is, predicate_f) )
  end prop_forall_in

  def prop_forall_in[N <: Name : ValueOf, X <: (Nat | Zat) : Ty](
    bounds: (R[X], R[X]),
    upper_bound_is: BoundIs,
    predicate: Fun1[R, X, Bool]
  ): R[Prop[R]] =
    val name = valueOf[N]
    prop_forall_in(name, bounds, upper_bound_is, predicate)
  end prop_forall_in
  // ...
end Core
```

Algorithmic type inference rules, encoded in the logic of Scala implicits, using the Scala 3 `given` syntax

```scala
trait TypeInference[R[_]]:
  given given_Unit  : Ty[Unit]   = Ty.UnitTy
  given given_NatTy : Ty[Nat]    = Ty.NatTy
  given given_ZatTy : Ty[Zat]    = Ty.ZatTy
  given given_BoolTy: Ty[Bool]   = Ty.BoolTy
  given given_StringTy: Ty[String] = Ty.StringTy
  given given_CharTy: Ty[Char]   = Ty.CharTy

  given given_RTy[X](using x_ty: Ty[X]): Ty[R[X]] = Ty.RTy(x_ty)

  given given_PropTy: Ty[Prop[R]] = Ty.PropTy[R]()

  given given_Fun0Ty[Z]          (using z: Ty[Z]): Ty[() => Z] =
    Ty.Fun0Ty(z)
  given given_Fun1Ty[A, Z]       (using a: Ty[A], z: Ty[Z]): Ty[(A) => Z] =
    Ty.Fun1Ty(a, z)
  given given_Fun2Ty[A, B, Z]    (using a: Ty[A], b: Ty[B], z: Ty[Z]): Ty[(A, B) => Z] =
    Ty.Fun2Ty(a, b, z)
  given given_Fun3Ty[A, B, C, Z](using a: Ty[A], b: Ty[B], c: Ty[C], z: Ty[Z]): Ty[(A, B, C) => Z] =
    Ty.Fun3Ty(a, b, c, z)
  given given_Fun4Ty[A, B, C, D, Z]    (using a: Ty[A], b: Ty[B], c: Ty[C], d: Ty[D], z: Ty[Z]):
    Ty[(A, B, C, D) => Z] = Ty.Fun4Ty(a, b, c, d, z)
  given given_Fun5Ty[A, B, C, D, E, Z](using a: Ty[A], b: Ty[B], c: Ty[C], d: Ty[D], e: Ty[E], z: Ty[Z]):
    Ty[(A, B, C, D, E) => Z] = Ty.Fun5Ty(a, b, c, d, e, z)

  given given_OptionTy[A]   (using a: Ty[A]                ): Ty[Option[A]]   = Ty.OptionTy(a)
  given given_PairTy  [A, B](using a: Ty[A], b: Ty[B]): Ty[Pair[A, B]]  = Ty.PairTy(a, b)

  given given_MapTy[A, B](using a: Ty[A], b: Ty[B]): Ty[Map[A, B]] = Ty.MapTy(a, b)
  given given_SetTy[A](using a: Ty[A]): Ty[Set[A]] = Ty.SetTy(a)
  given given_SeqTy[A](using a: Ty[A]): Ty[Seq[A]] = Ty.SeqTy(a)

  given [S <: AnyStruct](using tag: ClassTag[S]): Ty[S] = Ty.StructTy(tag)
end TypeInference
```