CoLF Logic Programming as Infinitary Proof Exploration

Zhibo Chen
Carnegie Mellon University
zhiboc@andrew.cmu.edu

Frank Pfenning
Carnegie Mellon University
fp@cs.cmu.edu

Logical Frameworks such as Automath [de Bruijn, 1968] or LF [Harper et al., 1993] were originally conceived as metalanguages for the specification of foundationally uncommitted deductive systems, yielding generic proof checkers. Their high level of abstraction was soon exploited to also express algorithms over deductive systems such as theorem provers, type-checkers, evaluators, compilers, proof transformers, etc. in the paradigm of computation-as-proof-construction. This has been realized in languages such as λProlog [Miller et al., 1991] or Elf [Pfenning, 1991] based on backward chaining, and LolliMon [López et al., 2005] or Celf [Schack-Nielsen and Schürmann, 2008], which integrated forward chaining.

None of these early frameworks supported the direct expression of infinitary objects or proofs, which are available in the recently developed CoLF$^\omega$[Chen, 2023]. In this work-in-progress report, we sketch an approach to computation-as-proof-construction over the first-order fragment of CoLF$^\omega$ (called CoLF$^\omega_1$) that already includes infinitary objects and proofs. A key idea is the interpretation of logic variables as communication channels and computation as concurrent message-passing. This is realized in a concrete compiler from CoLF$^\omega_1$ to Sax, a proof-theoretically inspired parallel programming language based on the proof-reduction in the semi-axiomatic sequent calculus [DeYoung et al., 2020].

1 Introduction

Consider the set of natural numbers inductively generated by symbols `s` and `z`. The following two rules define the addition operation on natural numbers.

$$\frac{}{\texttt{add z A A}}\,\texttt{add\_z} \qquad\qquad \frac{\texttt{add A B C}}{\texttt{add (s A) B (s C)}}\,\texttt{add\_s}$$

The adequacy of representation dictates that a derivation of `add A B C` exists if and only if $A+B=C$. We can represent both the natural numbers and the addition relation in LF as follows.

```
nat: type.
z : nat.
s : nat -> nat.

add: nat -> nat -> nat -> type.  %mode add + + -.
add_z : add z A A.
add_s : add A B C -> add (s A) B (s C).
```

The computational interpretation of this signature in Twelf proceeds by searching for a derivation of of `add A B C`, given `A` and `B`. Mode checking guarantees that if the first two arguments are ground and proof search succeeds, then the third argument will also be ground. In this example, there is a unique

Chaudhuri, Nantes-Sobrinho (Eds.): International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2025)
EPTCS ??, 2025, pp. 1–8, doi:10.4204/EPTCS.??.??

C such that the relation `add A B C` holds, but in general proof search may backtrack and thereby either fail to terminate or enumerate multiple solutions.

Backward-chaining proof search in this style presents multiple difficulties in the infinitary settings of $CoLF^{\omega}$. One is termination if the inputs are infinitary. Infinitary objects also interact poorly with backtracking because we may never definitively fail. Related is a problem with unification, which is guaranteed to terminate only over rational (that is, circular) terms, while in many applications either objects or proofs are not rational in this sense.

What we would like is a dynamics where outputs are computed incrementally from inputs, evoking the image of transducers between streams (even if terms generally have the shape of potentially infinite trees, not just streams).

In order to avoid backtracking, we use mode and uniqueness checking to statically enforce that there will be at most one proof for each unique input. As a (perhaps surprising) consequence we can then exploit and-parallelism between multiple premises, reducing synchronization between them to communication between shared variables.

For example, consider the following program that computes the product of two conatural numbers (that is, potentially infinite numbers). We continue to use `s` and `z` as constructors. Note that both `add` and `mult` are cotypes, because we want to allow infinitary derivations for addition and multiplication.

```
conat: cotype.
z : conat.
s : conat -> conat.

add: conat -> conat -> conat -> cotype.  %mode add + + -.
add_z : add z A A.
add_s : add A B C -> add (s A) B (s C).

mult : conat -> conat -> conat -> cotype.  %mode mult + + -.
mult_z : mult z A z.
mult_s : mult A B C -> add B C D -> mult (s A) B D.
```

Here, the two premises `mult A B C` and `add B C D` can be evaluated in parallel, with some interesting flow of information. For example, in a lazy setting, where the output `D` is revealed step by step, we don't need to evaluate `mult A B C` for the first `B` steps since `C` remains unchanged.

In summary, we explore a logic programming language based on $CoLF^{\omega}_{1}$ where proofs are represented as infinitary terms, proof construction does not backtrack, and premises of rules are evaluated in parallel using shared logic variables for communication. In the rest of this paper, we give definitions of streams and stream transducers, and explain their operational semantics.

2 Simple Examples

We give an overview of the intended meaning of CoLF logic programs through examples. The examples we primarily consider are streams of conatural numbers. The current semantics of CoLF handles only the coinductive fragment, contrary to the mixed inductive and coinductive case [Chen, 2023, 2021].

The data signature is given below, where `s` and `z` are the constructors of conatural numbers, and `cons` is the only constructor for streams.

```
conat: cotype.
```

```
z: conat.
s: conat -> conat.

stream: cotype.
cons: conat -> stream -> stream.
```

We may use notational definitions to define several streams. Notational definitions are elaborated into relations during compilation. We define `repeat n` to be the stream that repeats the number n infinitely, and `up n` to be the stream that counts up from n. Notice that the syntax for the above definitions follows that of Twelf [Pfenning and Schürmann, 1998], where λ-abstractions are written using square brackets.

```
repeat : conat -> stream = [N] cons N (repeat N).
up : conat -> stream = [N] cons N (up (s N)).
```

The definition of `repeat` is that `repeat n` is `cons n (repeat n)`, a stream whose head is `n` and whose tail is the same stream. The definition of `up` is that `up n` is `cons n (up (s n))`, a stream whose head is `n` and whose tail is the stream that counts up from `s n`. The above two definitions are equivalent to the following relational definitions. The compiler performs this translation automatically as part of the compilation process. We leave the universal quantification of the free variables (in uppercase) implicit.

```
repeat : conat -> stream -> cotype.  %mode repeat + -.
repeat_def : repeat N R -> repeat N (cons N R).

up : conat -> stream -> cotype.  %mode up + -.
up_def : up (s N) U -> up N (cons N U).
```

We can also view the relation as the following rules for constructing infinitary proofs.

$$\frac{\texttt{repeat}\,\texttt{N}\,\texttt{R}}{\texttt{repeat}\,\texttt{N}\,(\texttt{cons}\,\texttt{N}\,\texttt{R})}(\texttt{repeat\_def}) \qquad \frac{\texttt{up}\,(\texttt{s}\,\texttt{N})\,\texttt{U}}{\texttt{up}\,\texttt{N}\,(\texttt{cons}\,\texttt{N}\,\texttt{U})}(\texttt{up\_def})$$

The rules mimic the recursive definitions: an infinitary proof expansion of `repeat N R` will equate `R` with `cons N (cons N (cons N ...))`, and an infinitary proof expansion of `up N U` will equate `U` with `cons N (cons (s N) (cons (s (s N)) ...))`. We show a partial expansion below.

$$\cfrac{\cfrac{\cfrac{\texttt{repeat}\,\texttt{N}\ldots}{\texttt{repeat}\,\texttt{N}\,(\texttt{cons}\,\texttt{N}\ldots)}}{\texttt{repeat}\,\texttt{N}\,(\texttt{cons}\,\texttt{N}\,(\texttt{cons}\,\texttt{N}\ldots))}}{\texttt{repeat}\,\texttt{N}\,(\texttt{cons}\,\texttt{N}\,(\texttt{cons}\,\texttt{N}\,(\texttt{cons}\,\texttt{N}\ldots)))} \qquad \cfrac{\cfrac{\cfrac{\texttt{up}\,(\texttt{s}\,(\texttt{s}\,(\texttt{s}\,\texttt{N})))\ldots}{\texttt{up}\,(\texttt{s}\,(\texttt{s}\,\texttt{N}))\,(\texttt{cons}\,(\texttt{s}\,(\texttt{s}\,\texttt{N}))\ldots)}}{\texttt{up}\,(\texttt{s}\,\texttt{N})\,(\texttt{cons}\,(\texttt{s}\,\texttt{N})\,(\texttt{cons}\,(\texttt{s}\,(\texttt{s}\,\texttt{N}))\ldots))}}{\texttt{up}\,\texttt{N}\,(\texttt{cons}\,\texttt{N}\,(\texttt{cons}\,(\texttt{s}\,\texttt{N})\,(\texttt{cons}\,(\texttt{s}\,(\texttt{s}\,\texttt{N}))\ldots)))}$$

In fact, we are able to run this program in our implementation by providing a main function. For example, we may be interested in seeing the evaluating `up z` and we may write a main definition (or relation) as follows.

```
main : stream = up z.
```

The main function may also be defined as a relation. This process is automatically carried out by the compiler.

```
main : stream -> cotype.  %mode main -.
main_def : up z U -> main U.
```

The compiler will then compile and perform infinitary proof expansion. It will stop after the result is ground up to a predetermined depth. Here is the result from the compiler, where ... indicates a term that has not been computed yet.

```
(cons z
(cons (s z)
(cons (s (s z))
(cons (s (s (s z)))
(cons (s (s (s (s z)))) ...))))))
```

For space reasons, we truncated the output above. The effect of computing up to a certain term depth is more prominent when we compute the infinite stream of natural number ω as follows.

```
omega : conat = s omega.
main : stream = repeat omega.
```

Notice here that the argument to `repeat` is also defined recursively, and the compiler correctly elaborates them into definitions.

```
omega : conat -> cotype.  %mode omega -.
omega_def : omega O -> omega (s O).
main : stream -> cotype.  %mode main -.
main_def : omega O -> repeat O R -> main R.
```

Here, something interesting happens in the definition of `main`. Since `omega O` and `repeat O R` are two separate premises, they may be evaluated in parallel. Moreover, although semantically `omega O` is outputting a stream `O` to be read by `repeat O R`, since `repeat` is a recursive definition that does not look at the structure of its first argument, there is no strict evaluation order on whether `repeat` or `omega` should be evaluated first.

Running the interpreter, we get the following stream as the result. In this case, the stream is computed up to depth 5, and we can see that the first element is computed up to depth 4, the second element is computed up to depth 3, and etc.

```
(cons (s (s (s (s ...))))
(cons (s (s (s ...)))
(cons (s (s ...))
(cons (s ...)
(cons ...)))))
```

3 More Complex Relations

We now turn to more complex relations that include case analysis on input arguments to a relation.

As in the LF logical framework, term abstractions cannot perform a case analysis on their arguments but may only use them "parametrically". To analyze the input arguments, we need to define a relation and provide different branches for each input argument case.

For example, consider the following addition relation defined on conatural numbers. The rules should be straightforward as we have seen these for several times so far.

```
add: conat -> conat -> conat -> cotype.  %mode add + + -.
add_z : add z A A.
add_s : add A B C -> add (s A) B (s C).
```

The core question is: what does this program mean computationally?

Rather than giving a full formal semantics, we explain informally the program behavior of `add X Y Z`. The idea is to treat each argument as a channel of communication. The mode declaration specifies whether channels are inputs (mode $+$) or outputs (mode $-$). For instance, the program `add X Y Z` operates on two input channels `X` and `Y`, and one output channel `Z`. The behavior of the program is specified by the clauses, directly modeling infinitary proof construction.

The first clause `add_z` states that the program `add X Y Z` *reads* the first input channel `X`, and if the value read is `z`, the program *forwards* the input channel `Y` to output channel `Z`.

The second clause `add_s` states that the program `add X Y Z` *reads* the first input channel `X`. If the value read is `s A`, for some input channel `A`, the program *allocates* a fresh channel `C`, and *writes* to the output channel `Z` the value `s C`, then we continue as the program `add A Y C`.

The proposed informal semantics has the following properties:

1. The program operates on list of input channels and a single output channel
2. At each step, the program consists of interacting processes that may perform one of the following actions:
   (a) Read a value from an input channel
   (b) Write a value to an output channel
   (c) Forward an input channel to an output channel
   (d) (During writing) Allocate fresh channels
   (e) Spawn a new process (see stream processor example below)
   (f) Continue as some process
3. No backtracking (see below)

The property of no backtracking states that at each program point, the process has at most one possible action. In other words, the clauses for a relation have to agree on the action taken at each step. In the case of `add` above, we see that the first action in both clauses should be to read from the first input channel `X`. Then, based on different values of `X`, each clause may take different actions. The compiler checks uniqueness during compilation, and signals an error if more than one action is possible at some program point.

We now define the pointwise addition of two streams.

```
add_stream: stream -> stream -> stream -> cotype.  %mode add_stream + + -.
add_stream_def : add A B C -> add_stream R S T ->
  add_stream (cons A R) (cons B S) (cons C T).
```

The clause `add_stream_def` states that the program `add_stream X Y Z` will carry out the following actions in order:

1. Read from the first input channel `X` a stream `cons A R`.
2. Read from the second input channel `Y` a stream `cons B S`.
3. Allocate a fresh channel `C`, and spawn a new process `add A B C`.
4. Allocate a fresh channel `T`, and spawn a new process `add_stream R S T`.
5. Write to the output channel `Z` the value `cons C T`.

There are two things to notice here. First, we dictate that the program read the first input channel `X` before reading the second input channel `Y`. Strictly speaking, this is not necessary in this case, but when we get to the case of multiple clauses, this is needed because we would like to ensure unique decision at every program point. Second, steps (3) (4) and (5) can happen simultaneously, given that fresh channels `C` and `T` are allocated. However, due to the fact that we do not allow backtracking, we still ensure a sequential order of actions where the spawned processes can execute in parallel with the main process.

With the help of stream addition, as an example, we may define the stream of even numbers by adding together two streams that count up from 0. Note here that `even` cannot be a constant definition but has to be a relation, because `stream_add` needs to analyze its arguments.

```
even : stream -> cotype.  %mode even -.
even_def : add_stream (up z) (up z) E -> even E.
```

We may evaluate the `even` predicate through a `main` relation. Since `even` is a relation, `main` also needs to be a relation, not a constant definition.

```
main : stream -> cotype.  %mode main -.
main_def : even E -> main E.
```

And we get the expected result when executing the program.

```
(cons z
(cons (s (s z))
(cons (s (s (s (s z))))
(cons (s (s (s (s (s (s z))))))
(cons (s (s (s (s (s (s ...)))))))  ...)))))
```

In a similar way, we may define the Fibbonacci stream by beginning with 0 and 1, and then adding the stream with its tail.

```
tail : stream -> stream -> cotype.  %mode tail + -.
tail_def : tail (cons N F) F.

fib : stream -> cotype.  %mode fib -.
fib_def : fib F -> tail F G ->
    add_stream F G H -> fib (cons z (cons (s z) H)).

main : stream -> cotype.  %mode main -.
main_def : fib F -> main F.
```

Here, the channel represented by variable F is an output if `fib` and an input to two other processes (`tail` and `add_stream`).

We can test this program by running the interpreter.

```
(cons z
(cons (s z)
(cons (s z)
(cons (s (s z))
(cons (s (s (s z)))
(cons (s (s (s (s (s z)))))
(cons (s (s (s (s (s (s (s z)))))))) ...)))))))
```

Another interesting operation we can define is the integration operation (inspired by Budiu et al. [2024]), which calculates the cumulative sum of a stream. Integration is done by taking a sum of the stream with the shifted output.

```
integrate : stream -> stream -> cotype.  %mode integrate +A -B.
integrate/def : integrate D B -> add_stream (cons z B) D R -> integrate D R.
```

We can check that integrating the stream that counts up from 0 gives the stream $0, 1, 3, 6, 10....$

```
main : stream -> cotype.  %mode main -.
main_def : integrate (up z) G -> main G.
```

```
(cons z
(cons (s z)
(cons (s (s (s z)))
(cons (s (s (s (s (s (s z))))))
(cons (s (s (s (s (s (s (s (s (s z)))))))))) ...)))))
```

4 Conclusion

We have sketched an interpreter for CoLF^{ω}_{1}. It statically performs mode checking and uniqueness checking and then executes the program concurrently, initially with a single process with just one output channel. During this execution it also constructs a partial proof object using the constructors naming the rules. When new processes are spawned, each is responsible for sending along a single output channel while receiving from possibly several input channels. This matches the computational model of Sax under its message-passing interpretation [DeYoung et al., 2020, Pfenning and Pruiksma, 2023]. Our implementation therefore generates Sax code from the CoLF^{ω}_{1} source and then evaluates it to a certain depth.

The first-order kinds are interpreted as data types, and the clauses of first-order kinds are interpreted as data constructors. The higher-order kinds are interpreted as relations, which are seen as logic programming recipes for constructing outputs from inputs with mode specifications. Clauses of relations are interpreted as concrete steps on how to read inputs and write outputs. Premises of a clause become concurrently executing subprocesses, and recursive definitions are seen as a special instance of relations.

Besides a formal description of compilation which is beyond the scope of this work-in-progress report, our preliminary investigations leave room for multiple generalizations towards full CoLF^{ω}, such as higher-order terms, dependently typed terms (not just at the level of proof objects), and the mix of inductive and coinductive types. We are encouraged by experiments with our preliminary compiler and the strong proof-theoretic foundations for both CoLF^{ω} and Sax.

References

Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Incremental computation on streams and its applications to databases. *SIGMOD Record*, 53(1):87–95, March 2024. doi: 10.1145/3665252.3665271.

Zhibo Chen. Towards a mixed inductive and coinductive logical framework. Technical Report CMU-CS-21-144, Department of Computer Science, Carnegie Mellon University, 2021.

Zhibo Chen. A logical framework with infinitary terms. *CoRR*, abs/2312.05919, 2023. doi: 10.48550/ARXIV.2312.05919. URL `https://doi.org/10.48550/arXiv.2312.05919`.

N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, pages 29–61, Versailles, France, December 1968. Springer-Verlag LNM 125. doi: 10.1016/s0049-237x(08)70200-3.

Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-axiomatic sequent calculus. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 29:1–29:22, Paris, France, June 2020. LIPIcs 167. doi: 10.4230/LIPICS.FSCD.2020.29.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993. doi: 10.1145/138027.138060.

Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In A.Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46, Lisbon, Portugal, July 2005. ACM Press. doi: 10.1145/1069774.1069778.

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991. doi: 10.1016/0168-0072(91)90068-W.

Frank Pfenning. *Logic Programming in the LF Logical Framework*, page 149–181. Cambridge University Press, USA, 1991. ISBN 0521413001. doi: 10.1017/cbo9780511569807.008.

Frank Pfenning and Klaas Pruiksma. Relating message passing and shared memory, proof-theoretically. In S. Jongmans and A. Lopes, editors, *25th International Conference on Coordination Models and Languages (COORDINATION 2023)*, pages 3–27, Lisbon, Portugal, June 2023. Springer LNCS 13908. doi: 10.1007/978-3-031-35361-1_1. Notes to an invited talk.

Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In D. Galmiche, editor, *Proceedings of the CADE Workshop on Proof Search in Type-Theoretic Languages*. Electronic Notes in Theoretical Computer Science, July 1998. doi: 10.1016/S1571-0661(05)01181-3.

Anders Schack-Nielsen and Carsten Schürmann. Celf - a logical framework for deductive and concurrent systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326, Sydney, Australia, August 2008. Springer LNCS 5195. doi: 10.1007/978-3-540-71070-7_28.