System Description: Celf – A Logical Framework for Deductive and Concurrent Systems

Anders Schack-Nielsen and Carsten Schürmann

IT University of Copenhagen
Denmark

Abstract. CLF (Concurrent LF) [CPWW02a] is a logical framework for specifying and implementing deductive and concurrent systems from areas, such as programming language theory, security protocol analysis, process algebras, and logics. Celf is an implementation of the CLF type theory that extends the LF type theory by linear types to support representation of state and a monad to support representation of concurrency. It relies on the judgments-as-types methodology for specification and the interpretation of CLF signatures as concurrent logic programs [LPPW05] for experimentation.
Celf is written in Standard ML and compiles with MLton and MLKit. The source code and a collection of examples are available from http://www.twelf.org/∼celf.

1 Introduction

The Celf system is a tool for experimenting with deductive and concurrent systems prevalent in programming language theory, security protocol analysis, process algebras, and logics. It supports the specification of object language syntax and semantics through a combination of deductive methods and resource-aware concurrent multiset transition systems. Furthermore it supports the experimentation with those specifications through concurrent logic programming based on multiset rewriting with constraints.

Many case studies have been conducted in Celf including all of the motivating examples that were described in the original CLF technical report [CPWW02b]. In particular, Celf has been successfully employed as a tool for experimenting with concurrent ML, in particular its type system and a destination passing style operational semantics. Our Celf encoding provides Haskell-style suspensions with memoizations, futures, mutable references, and concurrency omitting negative acknowledgments. Furthermore, we used Celf to experiment with the design of security protocols, especially the widely studied and well-understood Needham-Schroeder authentication protocol, which will be our running example (see Sec. 2). The ease with which we applied Celf to this example sheds some light on the range of other protocols that could be studied using Celf. Other examples include various encodings of the π-calculus, petri-nets, etc.

CLF is a conservative extension over LF [HHP93], which implies that Celf's functionality is compatible with that of Twelf [PS99]. With a few syntactic modifications Twelf signatures can be read and type checked, and queries can be

executed. Celf does not yet provide any of the meta-theoretic capabilities that sets Twelf apart from its predecessor Elf, such as mode checking, termination checking, coverage checking, and the like, which we leave to future work. In this presentation we concentrate on the two main features of Celf.

Specification. CLF was designed with the objective in mind to simplify the specification of object languages by internalizing common concepts used for specification and make them available to the user. Celf supports dependent types for the encoding of judgments as types, e.g. operational relations between terms and values, open and closed terms, derivability, and logical truth. It also supports the method of *higher-order abstract syntax*, which relieves the user of having to specify substitutions and substitution application. In CLF, every term is equivalent to a unique inductively defined β-normal η-long form modulo α-renaming and let-floating providing an induction principle to reason about the adequacy of the encoding. In addition, CLF provides linear types and concurrency encapsulating monadic types in support of the specification of resource aware and concurrent systems. Examples include operational semantics for languages with effects, transition systems, and protocol stacks.

Experimentation. Celf provides a logic programming interpreter that implements a proof search algorithm for derivations in the CLF type theory in analogy to how Elf implements a logical programming interpreter based on uniform proof search. Celf's interpreter is inspired (with few modifications) by Lollimon [LPPW05], an extension of Lolli, the linear sibling of λ-Prolog. The interpreter implements backward-chaining search within the intuitionistic and linear fragment of CLF and switches to forward-chaining multiset rewriting search upon entering the monad. Celf programs may jump in and out of the concurrency monad and can therefore take advantage of both modes of operation. In addition, the operational semantics of Celf is conservative over the operational semantics of Elf, which means that any Twelf query can also be executed in Celf leading to the same result.

In the remainder of the paper, we illustrate the features of Celf. In Section 2, we describe the Needham-Schroeder protocol followed by a brief overview in Section 3 of the CLF type theory and the protocol specification in Celf [CPWW02b]. Finally, we comment on the implementation and conclude in Section 4.

2 Example

As a small running example, we consider the Needham-Schroeder protocol [NS78]. The protocol serves the authentication of two principals, A and B, and is characterized by the following simplified message exchange

$$A \to B : \{N_a, A\}_{K_b} \tag{1}$$

$$B \to A : \{N_a, N_b\}_{K_a} \tag{2}$$

$$A \to B : \{N_b\}_{K_b} \tag{3}$$

where K_a and K_b are the public keys of A and B, respectively. We write $\{\cdot\}_K$ for the encryption of a message by key K. Two messages may be concatenated using

Kinds

$$K ::= \texttt{type} \mid \texttt{Pi}\ x : A.\ K \qquad \textit{Kinds}$$

Types

$$\begin{array}{rll} A, B ::= & A\ \texttt{-o}\ B \mid \texttt{Pi}\ x : A.\ B \mid A\ \&\ B \mid \top \mid \{S\} \mid P & \textit{Asynchronous types} \\ P ::= & a \mid P\ N & \textit{Atomic type constructors} \\ S ::= & S_1\ \texttt{@}\ S_2 \mid 1 \mid \texttt{Exists}\ x : A.\ S \mid A & \textit{Synchronous types} \end{array}$$

Objects

$$\begin{array}{rll} N ::= & \backslash\hat{}\ x.\ N \mid \backslash x.\ N \mid \langle N_1, N_2\rangle \mid \langle\rangle \mid \{E\} \mid & \\ & c \mid x \mid N\ \hat{}N \mid N\ N \mid N\ \texttt{\#}_1 \mid N\ \texttt{\#}_2 & \textit{Objects} \\ E ::= & \texttt{let}\ \{p\} = N\ \texttt{in}\ E \mid M & \textit{Expressions} \\ M ::= & M_1\ \texttt{@}\ M_2 \mid 1 \mid [N, M] \mid N & \textit{Monadic objects} \\ p ::= & p_1\ \texttt{@}\ p_2 \mid 1 \mid [x, p] \mid x & \textit{Patterns} \end{array}$$

Signatures

$$\Sigma ::= \cdot \mid a : K.\ \Sigma \mid c : A.\ \Sigma \qquad \textit{Signatures}$$

Fig. 1. Celf syntax

",". N_a and N_b are nonces, randomly generated messages, which are created in line (1) and (2) and compared for identity in line (2) and (3), respectively. We think of A as the initiator of the message exchange and B as the responder. From the point of view of the initiator, two actions are necessary to participate in the protocol.

1. Create a new nonce N_a. Send message $\{N_a, A\}_{K_b}$ (1). Remember B, k_b, and N_a.
2. Recall B, k_b, and N_a. Receive message $\{N'_a, N_b\}_{K_a}$ (2). Check that N_a is identical to N'_a. Send message $\{N_b\}_{K_b}$ (3).

Correspondingly, the responder needs to execute two actions.

1. Receive message $\{N_a, A\}_{K_b}$ (1). Create a new nonce N_b. Send message $\{N_a, N_b\}_{K_a}$ (2). Remember A and N_b.
2. Recall A and N_b. Receive $\{N'_b\}_{K_b}$ (3). Check that N_b is identical to N'_b.

A successful run of the protocol initializes initiator and responder and causes then the initiator to send the first message, the responder to reply, and so forth.

3 Celf

The basis of the Celf system is the CLF type theory [CPWW02a]. The CLF type theory is a dependently typed λ-calculus extended by linear functions, additive

and multiplicative pairs, additive and multiplicative units, dependent pairs, and concurrent objects. The syntax of Celf is shown in Fig. 1 and explained below:

Lolli, A `-o` B, is linear implication with linear lambda, $\backslash\widehat{\ }$, as introduction form and linear application, $\widehat{\ }$, as elimination form.

Pi, `Pi` $x : A.\ B$, is dependent intuitionistic implication, which can also be written A `->` B in the non-dependent case. It has lambda, $\backslash$, and application (juxtaposition) as introduction and elimination forms.

And, $A \ \& \ B$, is additive conjunction with $\langle \cdot, \cdot \rangle$ as introduction form, the projections $\#_i$ as elimination forms, and *Top*, `T`, as unit.

Tensor or multiplicative conjunction, A `@` B, and *Existential*, `Exists` $x : A.\ S$, are only available inside the monad and can only be deconstructed by the pattern in a `let`-construct. Their introduction forms are tensor, `@`, and dependent pair, $[\cdot, \cdot]$.

Monad, $\{S\}$, is the concurrency monad and represents concurrent computation traces (sequences of `let`-bindings) with a result described by S.

One, 1, is unit for the multiplicative conjunction, and *Top*, `T`, is unit for the additive conjunction. Their introduction forms are 1 and $\langle\rangle$ respectively. Combining one with existential quantification allows us to encode the intuitionistic embedding known from linear logic as: $!A = \texttt{Exists}\ x : A.\ 1$. Another common use of one is in the type $\{1\}$ which is the type of concurrent traces in which all linear resources have been consumed. In contrast, $\{\top\}$ is the type of *any* concurrent trace.

CLF has important meta-theoretical properties including decidability of typechecking and the existence of canonical forms. The notion of definitional equality on CLF terms, types, and kinds is induced by the usual β- and η-rules modulo α-renaming along with one additional rule: Inside the concurrency monad, a sequence of `let`-bindings is allowed to permute as long as the permutation respects the dependencies among bound variables (i.e. permutation is disallowed if it causes a bound variable to escape its scope). This equivalence is the foundation for specifying concurrency in object languages: If a sequence of `let`-bindings represents a concurrent trace of an operational semantics or a protocol communication exchange then CLF will only distinguish between those traces that can lead to observably different results. In other words if two independent events occur within one trace then CLF is completely unaware about the order of their occurrence.

We return to our running example the Needham-Schroeder protocol described in the previous section and illustrate how to specify it in Celf. We follow hereby closely Section 6 of Cervesato et al. [CPWW02b] and refer the interested reader to this technical report for an in depth discussion on how one can derive this encoding and how to reason about its adequacy.

Figure 2 depicts the Celf code specifying the syntactic categories of principals, nonces, public, and private keys in the left column. The right column gives the Celf encoding of messages, where the first two constructors embed principals and nonces into messages, `+` concatenates two messages, and `pEnc` encrypts a message with the public key of principal `A`. Celf's type reconstruction algorithm

```
principal : type.                msg  : type.
nonce     : type.                p2m  : principal -> msg.
pubK      : principal -> type.   n2m  : nonce -> msg.
privK     : pubK A -> type.      +    : msg -> msg -> msg.
                                 pEnc : pubK A -> msg -> msg.
```

Fig. 2. Specification of syntactic categories in Celf.

```
net    : msg -> type.            init : principal -> type.
rspArg : type.                   resp : principal -> type.
rsp    : rspArg -> type.
```

Fig. 3. Specification of the network, memory, identity in Celf

infers the types of all undeclared uppercase arguments (here `A`), and builds an implicit `Pi`-closure.

The left column of Fig. 3 defines `net` which represents messages being sent on the network. Recall from Section 2 that a principal may need to remember the name of the principal it is trying to authenticate with or specific nonces. In the encoding, the type `rspArg` is used for expressing what the principals can remember, and the type `rsp` for what the principals currently are remembering. In a slight deviation from [CPWW02b] we use two type families `init` and `resp` to assign roles to principals. The corresponding Celf declarations are depicted in the right column of Fig. 3.

What follows below are the two Celf declarations that define initiator and responder of a Needham-Schroeder protocol interaction. The initiator is guarded by `init A` and the responder by `resp B`.

```
nspkInit : init A -o { Exists L : Pi B : principal.
                         pubK B -> nonce -> rspArg.
           Pi B : principal. Pi kB : pubK B.
              { Exists nA : nonce. net (pEnc kB (+ (n2m nA) (p2m A)))
                                   @ rsp (L B kB nA) }
         @ Pi B : principal. Pi kB : pubK B. Pi kA : pubK A.
           Pi kA' : privK kA. Pi nA : nonce. Pi nB : nonce.
              net (pEnc kA (+ (n2m nA) (n2m nB)))
              -o rsp (L B kB nA)
              -o { net (pEnc kB (n2m nB)) }}.

nspkResp : resp B -o { Exists L : principal -> nonce -> rspArg.
           Pi kB : pubK B. Pi kB' : privK kB.
           Pi A : principal. Pi kA : pubK A. Pi nA : nonce.
              net (pEnc kB (+ (n2m nA) (p2m A)))
              -o { Exists nB : nonce. net (pEnc kA (+ (n2m nA) (n2m nB)))
                                      @ rsp (L A nB) }
         @ Pi A : principal. Pi kB : pubK B. Pi kB' : privK kB.
           Pi nB : nonce.
              net (pEnc kB (n2m nB)) -o rsp (L A nB) -o { 1 }}.
```

Note that both principals introduce a new parameter `L` to remember the other's identity and their nonce. In addition, the initiator stores the responder's public key to be able to encrypt the second message. The respective nonces are modeled via higher-order abstract syntax, and dynamically created as new and fresh parameters using the `Exists`. Both, messages and memory, are modeled using linear assumptions. Each principal introduces two rules, separated by the top level tensors `@`, which correspond literally to the ones outlined in Section 2. Protocol traces are represented as monadic objects as evidenced by the fact that the declarations end in monadic type `{ · }`.

To experiment with the design in Celf, we use its logic programming engine. For example, in order to find a valid trace of a communication between principal `a`, with public key `ka` and private key `ka'`, and principal `b` with public key `kb` and private key `kb'` we query if it is possible to derive the empty linear context `{1}` from assumptions `init a` and `init b`. We obtain as answer to the query `#query init a -o resp b -o {1}` the term below, which includes six lets. The first two initiate initiator and responder, and the remaining four correspond to the message exchange of the authentication protocol.

```
Solution: \^ X1. \^ X2. {
    let {[L: Pi B: principal. pubK B -> nonce -> rspArg,
            X3: Pi B: principal. Pi kB: pubK B.
                {Exists nA: nonce. net (pEnc kB (+ (n2m nA) (p2m a))) @ rsp (L B kB nA)}
          @ X4: Pi B: principal. Pi kB: pubK B. Pi kA: pubK a. privK kA -> Pi nA: nonce.
                Pi nB: nonce. net (pEnc kA (+ (n2m nA) (n2m nB)))
                -o rsp (L B kB nA) -o {net (pEnc kB (n2m nB))}]} = nspkInit ^ X1 in
    let {[L': principal -> nonce -> rspArg,
            X5: Pi kB: pubK b. privK kB -> Pi A: principal. Pi kA: pubK A. Pi nA: nonce.
                net (pEnc kB (+ (n2m nA) (p2m A)))
                -o {Exists nB: nonce. net (pEnc kA (+ (n2m nA) (n2m nB))) @ rsp (L' A nB)}
          @ X6: Pi A: principal. Pi kB: pubK b. privK kB -> Pi nB: nonce.
                net (pEnc kB (n2m nB)) -o rsp (L' A nB) -o {1}]} = nspkResp ^ X2 in
    let {[nA: nonce, X7: net (pEnc kb (+ (n2m nA) (p2m a))) @ X8: rsp (L b kb nA)]}
          = X3 b kb in
    let {[nB: nonce,
            X9: net (pEnc ka (+ (n2m nA) (n2m nB)))
          @ X10: rsp (L' a nB)]} = X5 kb kb' a ka nA ^ X7 in
    let {X11: net (pEnc kb (n2m nB))} = X4 b kb ka ka' nA nB ^ X9 ^ X8 in
    let {1} = X6 a kb kb' nB ^ X11 ^ X10 in 1}
```

4 Conclusion

Celf is a system that implements the concurrent logical framework CLF. The implementation includes a type checking, type reconstruction, and proof search algorithm. The implementation employs explicit substitutions, logic variables, and spines and maintains canonical forms through hereditary substitutions.

Celf's type reconstruction algorithm permits programmers to omit inferable top-level `Pi`-quantifiers in declarations of constants. The implicitly bound variables are identified by uppercase names that occur free in declarations. Any use of constants with implicit `Pi`s are then implicitly applied to the correct number of arguments, which are subsequently inferred by Celf via unification.

Free variables in queries play a slightly different role. Uppercase variables occurring free in a query will not be `Pi`-quantified but will instead be considered logic variables and their instantiations are printed for each solution.

The operational semantics of Celf (i.e. the proof search algorithm) works in two modes: when searching for an object of an asynchronous type the algorithm proceeds by a backwards, goal-directed search (resembling pure Prolog), but when searching for an object of a synchronous type the algorithm shifts to an undirected, non-deterministic, forward-chaining, and concurrent execution, using committed choice instead of backtracking. This forward-chaining search is essentially a concurrent multiset rewriting engine.

Both the type reconstruction algorithm and the proof search algorithm rely on logic variables and unification. The implemented unification algorithm works on general CLF terms and handles all relevant aspects: general higher-order terms, linearity, and automatic reordering of bindings inside the monad. For unification problems inside the pattern fragment, which do not have multiple logic variables of monadic type bound by the same sequence of `let`s, the algorithm will always be able to find a most general unifier in case a unifier exists. Any unification problems that fall outside this fragment will be postponed as constraint, and if they are not resolved by later unifications, they are reported as leftover constraints. Our empirical experience has shown that this condition characterizes a sufficiently large decidable fragment of higher-order concurrent unification for the application of Celf as a specification and experimentation environment.

References

[CPWW02a] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University. Department of Computer Science, 2002.

[CPWW02b] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Carnegie Mellon University. Department of Computer Science, 2002.

[HHP93] 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.

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In Pedro Barahona and Amy P. Felty, editors, *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 35–46, Lisbon, Portugal, 2005.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.