Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 196 (2008) 37–51
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Induction on Concurrent Terms

Anders Schack-Nielsen[1]

Programming, Logics and Semantics
IT University of Copenhagen
Denmark

Abstract

This paper considers MiniML equipped with a standard big-step semantics and a destination-passing semantics both represented in concurrent LF (CLF) and prove the two semantics equivalent. The proof is then examined yielding insights into the issues concerning induction on concurrent terms. We conclude by outlining some of the difficulties that one will need to address when designing a meta-logic for CLF.

Keywords: CLF, logical frameworks, induction, destination-passing style.

1 Introduction

CLF [1] is a logical framework with several interesting applications including adequate representations of the π-calculus, protocols and programming languages employing state, concurrency, lazy computations and more. Furthermore, a large subset of these semantic specifications can currently be run with LolliMon [3] which implements parts of CLF. However, CLF currently has no notion of meta-logic and it is therefore not possible to reason about CLF representations within CLF. In this paper we will consider an initial case study in order to shed light on some of the difficulties that one will need to address when designing a meta-logic for CLF.

CLF is a dependently typed lambda calculus extended by linear types and monadic types inhabited by concurrent terms, which makes it a conservative extension of the dependently typed logical framework LF. Therefore CLF supports the same “judgments as types, derivations as terms” methodology as LF. The Twelf system [5] implements LF and provides a meta-logic for reasoning about LF representations. Twelf is well-suited for formalizing functional programming languages, their operational semantics and type systems, as well as classical and intuitionistic

[1] Email: `anderssn@itu.dk`

doi:10.1016/j.entcs.2007.09.016

logics. However, imperative and concurrent language features are hard to implement and reason about using Twelf since e.g. state has to be modelled and reasoned about explicitly.

The presence of concurrent terms in CLF allows for a new representation methodology compared to the way e.g. operational semantics has been represented in Twelf. In Twelf the methodology is a goal-oriented approach focusing on proof-search via backward chaining bearing much resemblance to logic programming, whereas in CLF the canonical representation methodology is context-oriented, employing forward chaining inside the monad. As CLF is a conservative extension to LF it allows both styles of representation to coexist.

The Twelf methodology provides means to represent meta-theory and its proofs as higher-level judgments describing relations between derivations, and these proofs can then be mechanically checked by checking the totality of the relation.

So the important question is whether the methodology of meta-theory representation and proof representation known from Twelf can be conservatively extended to deal with the new CLF representations and how. The CLF extensions over LF are linear and concurrent terms, so a conservative meta-logic for CLF would need to extend Twelf with induction on linear and concurrent terms. The importance of this question is emphasized by the fact that it is a main part of the uncharted CLF-territory and contains valuable insight on the directions in which CLF could be further developed.

The case study that we will consider in this paper is the equivalence proof of two semantics for MiniML. On the one hand we can represent a big-step semantics completely within the LF fragment of CLF, and on the other hand we can also represent the semantics in destination-passing style employing the distinct features of CLF. This style of representation is based on multiset rewriting with names (destinations) representing the holes in evaluation contexts. Furthermore, destination-passing style is a natural way to represent semantics in CLF and it allows for easy extension of the MiniML semantics to include lazy evaluation, futures, mutable references and concurrency [2]. Given these two styles of semantics, the equivalence proof will bridge the two different representation methodologies, and we will use the proof to outline some of the difficulties that one will need to address when designing a meta-logic for CLF.

2 CLF

2.1 Syntax

In the CLF type theory we have objects, types and kinds. In order to simplify the meta-theory all terms are required to be in canonical form (i.e. completely beta-reduced and completely eta-expanded), and this invariant can be maintained by a suitable definition of substitution which performes the necessary reduction steps (hereditary substitution).

The CLF types are the ones known from LF (with $A \to B$ as syntactic sugar for $\Pi x : A.\, B$ as usual) and the linear connectives from LLF, i.e. linear implication

Kinds

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

Types

$$\begin{array}{rll}
A, B & ::= A \multimap B \mid \Pi x : A.\ B \mid A \mathbin{\&} B \mid \top \mid \{S\} \mid P & \textit{Asynchronous types} \\
P & ::= a \mid P\ N & \textit{Atomic type constructors} \\
S & ::= S_1 \otimes S_2 \mid 1 \mid \exists x : A.\ S \mid A & \textit{Synchronous types}
\end{array}$$

Objects

$$\begin{array}{rll}
N & ::= \widehat{\lambda} x.\ N \mid \lambda x.\ N \mid \langle N_1, N_2 \rangle \mid \langle\rangle \mid \{E\} \mid R & \textit{Normal objects} \\
R & ::= c \mid x \mid R \mathbin{^\frown} N \mid R\ N \mid \pi_1\ R \mid \pi_2\ R & \textit{Atomic objects} \\
E & ::= \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M & \textit{Expressions} \\
M & ::= M_1 \otimes M_2 \mid 1 \mid [N, M] \mid N & \textit{Monadic objects} \\
p & ::= p_1 \otimes p_2 \mid 1 \mid [x, p] \mid x & \textit{Patterns}
\end{array}$$

Contexts

$$\begin{array}{rll}
\Gamma & ::= \cdot \mid \Gamma, x : A & \textit{Unrestricted contexts} \\
\Delta & ::= \cdot \mid \Delta, x \mathrel{\hat{:}} A & \textit{Linear contexts}
\end{array}$$

Signatures

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

Fig. 1. CLF syntax

($\multimap$), additive product ($\&$) and top ($\top$). Then there is multiplicative product ($\otimes$), the multiplicative unit (1) and dependent pair ($\exists$) all of which are wrapped in a monadic type constructor $\{S\}$. The complete syntax is given in figure 1. The destinction between normal and atomic objects is simply there to enforce canonical forms.

Constructing objects inside the monad (i.e. expressions inside curly braces) is supposed to model concurrent computation, and any given term consisting of a sequence of let expressions denotes a trace of that computation. In order to facilitate this interpretation two terms will be considered equivalent if they only differ in the ordering of their let expressions. The equivalence $\equiv$ is defined as the smallest congruence relation satisfying

$$\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E \equiv \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ \mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E$$

where the bindings are independent: p_1 and p_2 must bind disjoint sets of variables, no variable bound by p_1 can appear free in R_2 and vice versa.

2.2 Computational interpretation of CLF

The representation of meta-theory in Twelf is based on a computational interpretation of LF signatures as logic programs. With this in place a meta-logic can then be used to state the totality of certain relations, which thereby represent constructive proofs.

The basis of computation is constructing a term of a given type, by the means of proof search. This consists of applying right-rules in the corresponding logic until the goal is reduced to an atomic type, at which point the different constructors of the type is tried one by one by backtracking from unsatisfiable subgoals.

The semantics of CLF is similar (it is implemented as the language LolliMon [2] and described in detail in [3]) except when encountering the monad type. At this point the computation goes from being goal-directed to being context-directed. The context-directed computation consists of a sequence of steps, each of which is a nondeterministic choice between either ending the context-directed mode and constructing the monadic object M directly or nondeterministically choosing a term in the context (or signature) and reduce it to its monadic head with left-rules at which point the context gets augmented with the newly constructed types using a let-binding: $\mathsf{let}\ \{p\} = R\ \mathsf{in}\ E$, where R is the computation step that was just taken, p is the binding of the newly constructed types and E is the rest of the computation.

These steps are considered atomic and are not undone, backtracking is only applied during the construction of the individual steps to make sure that the step can actually be completed before committing to the nondeterministic choice.

2.3 CLF meta-theory

In Twelf, proofs are by structural induction since whatever is represented in Twelf is represented as an inductively defined LF-term. Furthermore the proof objects themselves are inductively defined LF-terms. We expect this meta-level representational methodology to extend to CLF as well, since it is a conservative extension at both the object level and the semantic level. There are however several challenges, and the one we will focus on is how to extend the structural induction known from Twelf to one working with terms with implicit concurrency. [3]

3 MiniML

The primary object of study in this paper will be the semantics of MiniML represented in two different ways. The first representation is a big-step semantics represented entirely in the LF fragment of CLF as it would be done in Twelf. The second representation is done in destination-passing style employing the monad and

[2] LolliMon is not exactly CLF since for the monadic and linear types it only includes the corresponding logic and not the terms. But currently LolliMon is as close as one gets to an implementation of CLF and it is sufficient for execution of programs in the destination-passing semantics given in this paper.

[3] As a side note, notice that the monadic terms potentially allows for a “concurrent” proof built in a more algorithmic manner instead of the usual induction proofs. What this means is however still unclear.

the linear features of CLF (see [2]). The meta-theorem that we will be examining is the equivalence proof of these semantics.

Note that the destination-passing semantics does everything sequentially, but since it is within the monad the potential for concurrency is still enough to generate intersting observations as we will see below. Furthermore the destination-passing semantics can easily be extended with e.g. concurrency, mutable references, lazy evaluation, etc. (as shown in [2]). In section 5 we will discuss some of the complications of the proof in the context of concurrency.

3.1 Syntax

The fragment of MiniML that we will be considering include abstractions, applications, fixpoints and natural numbers with zero, successor and case.

$$e ::= x \mid \mathsf{z} \mid \mathsf{s}\ e \mid (\mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{z} \Rightarrow e_2 \mid \mathsf{s}\ x \Rightarrow e_3) \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{fix}\ x.e$$

The MiniML syntax is represented in CLF (and LF) as shown in figure 2.

```
exp : type.

z : exp.
s : exp → exp.
case : exp → exp → (exp → exp) → exp.
lam : (exp → exp) → exp.
app : exp → exp → exp.
fix : (exp → exp) → exp.
```

Fig. 2. MiniML syntax in CLF

3.2 Big-step semantics

The first semantics for MiniML is a standard call-by-value big-step semantics (figure 3) and has the standard representation where the type family `ev` E V is inhabited if and only if E evaluates to V. [4]

The given representation is not strictly a CLF signature as defined in [1] since it is not in canonical form. It can however easily be transformed into the equivalent canonical form by eta expansion. In the following I will freely use any form eta equivalent to a canonical form, since the more verbose canonical form can be obtained by mechanical eta expansions.

3.3 Destination-passing semantics

The second semantics for MiniML is a destination-passing semantics. Destination-passing style is based on multi-set rewriting and handles evaluation contexts (a.k.a.

[4] Note that this semantics as it is given is not suitable for Twelf execution, since Twelf solves subgoals "inside out". If the semantics should be executed in Twelf, one would therefore have to do a simple rewriting, reversing the order of the arguments of `ev_case_z`, `ev_case_s` and `ev_app`.

```
ev : exp → exp → type.

ev_z       : ev z z.
ev_s       : ΠE:exp. ΠV:exp. ev E V → ev (s E) (s V).
ev_case_z  : ΠE1:exp. ΠE2:exp. ΠE3:exp → exp. ΠV:exp.
              ev E1 z → ev E2 V → ev (case E1 E2 E3) V.
ev_case_s  : ΠE1:exp. ΠE2:exp. ΠE3:exp → exp. ΠV:exp. ΠV':exp.
              ev E1 (s V') → ev (E3 V') V → ev (case E1 E2 E3) V.
ev_lam     : ΠE:exp → exp. ev (lam E) (lam E).
ev_app     : ΠE1:exp. ΠE2:exp. ΠE1':exp → exp. ΠV:exp. ΠV2:exp.
              ev E1 (lam E1') → ev E2 V2 → ev (E1' V2) V
                 → ev (app E1 E2) V.
ev_fix     : ΠE:exp → exp. ΠV:exp. ev (E (fix E)) V → ev (fix E) V.
```

Fig. 3. Big-step semantics in CLF

continuations) implicitly by naming the context holes. The names of the holes in the evaluation contexts are called destinations [6]. With the logic programming semantics of CLF outlined above in mind, the destination-passing semantics of MiniML is defined as follows. We introduce a type of destinations `dest`[5], a type family `eval` E D representing the instruction to evaluate E and return the result in destination D and a type family `return` V D representing the returned value V in destination D. Now the type $\Pi d : \texttt{dest}.\ \texttt{eval}\ E\ d \multimap \{\texttt{return}\ V\ d\}$ is inhabited if and only if E evaluates to V.

The signature is given in figure 4. Notice how each constructor consumes an `eval` E D to produce either a `return` V D representing the result, or a new `eval` E' d' corresponding to the subexpression to be evaluated next along with a continuation in the form $\Pi V : \texttt{exp}.\ \texttt{return}\ V\ d' \multimap \{\dots\}$. Take for instance `eval_case`. Assuming that we have an `eval` E D in the context with $E = \texttt{case}\ E_1\ E_2\ E_3$ and we aim to construct a `return` V D in the monad. Then `eval_case` can be applied to yield a fresh destination d', an `eval` E_1 d' and a continuation which can only be applied when the result of evaluating E_1 has finished. The continuation is an additive product which means that we can only ever use one of the branches. The `eval` E_1 d' will trigger further rules and end up with a result in the form of `return` V_1 d' (assuming termination). If V_1 is `z` we can apply the first projection of the continuation and if V_1 is `s` V' we can apply the second projection. In both cases we end up with a new `eval` E' D designating the expression to be evaluated and this will in turn trigger further rules and if this terminates we will end up with the result in the form of a `return` V D.

[5] Notice how the type `dest` is empty since we initially have no evaluation context and thus no holes to name, i.e. all we will ever see are variables of type `dest`.

```
dest   : type.
return : exp → dest → type.
eval   : exp → dest → type.

eval_z    : ΠD:dest. eval z D ⊸ {return z D}.
eval_s    : ΠE:exp. ΠD:dest.
             eval (s E) D ⊸ {∃d':dest. eval E d' ⊗
                                 ΠV:exp. return V d' ⊸ {return (s V) D}}.
eval_case : ΠE1:exp. ΠE2:exp. ΠE3:exp → exp. ΠD:dest.
             eval (case E1 E2 (λx. E3 x)) D ⊸
                 {∃d':dest. eval E1 d' ⊗
                   ( (return z d' ⊸ {eval E2 D}) &
                     (ΠV':exp. return (s V') d' ⊸ {eval (E3 V') D})
                   ) }.
eval_lam  : ΠE:exp → exp. ΠD:dest.
             eval (lam (λx. E x)) D ⊸ {return (lam (λx. E x)) D}.
eval_app  : ΠE1:exp. ΠE2:exp. ΠD:dest.
             eval (app E1 E2) D ⊸
                 {∃d':dest. eval E1 d' ⊗
                   (ΠE1':exp → exp. return (lam (λx. E1' x)) d' ⊸
                     {∃d'':dest. eval E2 d'' ⊗
                       (ΠV2:exp. return V2 d'' ⊸ {eval (E1' V2) D})
                     }
                   )
                 }.
eval_fix  : ΠE:exp → exp. ΠD:dest.
             eval (fix (λx. E x)) D ⊸ {eval (E (fix (λx. E x))) D}.
```

Fig. 4. Destination-passing semantics

4 Equivalence of semantics

We would like to prove the equivalence of the two semantics presented above. More formally, we will prove the following theorem:

Theorem 4.1 *For all closed terms E and V of type* `exp`*, the type* `ev` E V *is inhabited if and only if the type* $\Pi d :$ `dest.` `eval` $E\ d \multimap \{$`return` $V\ d\}$ *is inhabited.*

The proof consists of two parts, each being a translation from one semantics to the other. We will start with the easy one: translating big-step into destination-passing style.

4.1 Translation from big-step to destination-passing style

4.1.1 The paper proof

Lemma 4.2 *Let E and V be closed terms of type* `exp` *and let P be a closed term of type* `ev` $E\ V$*. Then there exists a closed term C of type* $\Pi d :$ `dest.` `eval` $E\ d \multimap$

$\{\texttt{return}\ V\ d\}$.

Proof. The proof is a simple structural induction on P.

Case: $P = \texttt{ev\_z}$

We take $C = \texttt{eval\_z}$.

Case: $P = \texttt{ev\_s}\ E'\ V'\ P'$

In this case $E = \texttt{s}\ E'$, $V = \texttt{s}\ V'$ and P' is of type $\texttt{ev}\ E'\ V'$. We can therefore apply the induction hypothesis to P' to get a C'. Now let

$$\begin{aligned} C = \lambda d.\widehat{\lambda} u : \texttt{eval}\ (\texttt{s}\ E')\ d.\ \{\mathsf{let}\ \{[d', (p : \texttt{eval}\ E'\ d') \\ \otimes (f : \Pi V.\texttt{return}\ V\ d' \multimap \{\texttt{return}\ (\texttt{s}\ V)\ d\})]\} \\ = \texttt{eval\_s}\ E'\ d\ \hat{}\ u\ \mathsf{in} \\ \mathsf{let}\ \{r' : \texttt{return}\ V'\ d'\} = C'\ d'\ \hat{}\ p\ \mathsf{in} \\ \mathsf{let}\ \{r : \texttt{return}\ (\texttt{s}\ V')\ d\} = f\ V'\ \hat{}\ r'\ \mathsf{in} \\ r\}. \end{aligned}$$

Case: $P = \texttt{ev\_case\_z}\ E_1\ E_2\ E_3\ V\ P_1\ P_2$

In this case P_1 is of type $\texttt{ev}\ E_1\ \texttt{z}$ and P_2 is of type $\texttt{ev}\ E_2\ V$. We apply the induction hypothesis to P_1 and P_2 yielding C_1 and C_2. Now let

$$\begin{aligned} C = \lambda d.\widehat{\lambda} u.\{&\mathsf{let}\ \{[d', (p_1 : \texttt{eval}\ E_1\ d') \otimes f_1]\} = \texttt{eval\_case}\ E_1\ E_2\ E_3\ d\ \hat{}\ u\ \mathsf{in} \\ &\mathsf{let}\ \{r' : \texttt{return}\ \texttt{z}\ d'\} = C_1\ d'\ \hat{}\ p_1\ \mathsf{in} \\ &\mathsf{let}\ \{p_2 : \texttt{eval}\ E_2\ d\} = (\pi_1\ f_1)\ \hat{}\ r'\ \mathsf{in} \\ &\mathsf{let}\ \{r : \texttt{return}\ V\ d\} = C_2\ d\ \hat{}\ p_2\ \mathsf{in} \\ &r\}. \end{aligned}$$

The remaining cases are similar. The induction hypothesis is applied to all subterms representing subevaluations (i.e. subterms of type $\texttt{eval}\ E\ V$ for some E and V), after which C is easily constructed. □

4.1.2 Representation of the proof in CLF

Since the above proof only relies on straigtforward induction on LF-terms it should be easy to represent in CLF for any conservative extension of the Twelf meta-theory to CLF. This is however still very speculative. More on this below in section 4.2.2.

4.2 Translation from destination-passing style to big-step

This part of the proof is a lot trickier. We cannot simply deconstruct a term of type $\texttt{eval}\ E\ D \multimap \{\texttt{return}\ V\ D\}$ into a constructor and subterms of the same type schema, since this among other things relies on the implicit ordering of the consumption of linear variables.

4.2.1 The paper proof

In order to complete the proof we will need to come up with a much stronger induction hypothesis. We will need to reason about the continuations that can occur in the linear context, and in order to make this precise, we will start with a definition of *normal* linear contexts to be the relevant linear implications from `return` ... into a monadic type:

Definition 4.3 A linear context Δ is called *normal* if it only consists of variables with the following types:

- $\Pi v : \texttt{exp}.\ \texttt{return}\ v\ D' \multimap \{\texttt{return}\ (\texttt{s}\ v)\ D\}$
- $(\texttt{return}\ \texttt{z}\ D' \multimap \{\texttt{eval}\ E_2\ D\})$
  $\&\ (\Pi v' : \texttt{exp}.\ \texttt{return}\ (\texttt{s}\ v')\ D' \multimap \{\texttt{eval}\ (E_3\ v')\ D\})$
- $\Pi e_1' : \texttt{exp} \rightarrow \texttt{exp}.\ \texttt{return}\ (\texttt{lam}\ (\lambda x.\ e_1'\ x))\ D' \multimap \{\exists d'' : \texttt{dest}.\ \texttt{eval}\ E_2\ d'' \otimes$
  $(\Pi v_2 : \texttt{exp}.\ \texttt{return}\ v_2\ d'' \multimap \{\texttt{eval}\ (e_1'\ v_2)\ D\})\}$
- $\Pi v_2 : \texttt{exp}.\ \texttt{return}\ v_2\ D'' \multimap \{\texttt{eval}\ (E_1'\ v_2)\ D\}$

for any instantiations of the free variables (written with capital letters).

Notice that these types correspond exactly to the continuations put in the context by `eval_s`, `eval_case` and `eval_app`. The latter is represented with two possible types, since the application of the continuation result in yet another continuation.

Now we can state the lemma:

Lemma 4.4 *Let Γ be a context of destinations, $\Gamma = d_1 : \texttt{dest}, \ldots, d_n : \texttt{dest}$, and let Δ be a normal linear context. Let E and V' be closed terms of type* `exp`. *Let d and d' be two (not necessarily distinct) destinations in Γ. And let C be a term with a typing $\Gamma; \Delta \vdash C : \texttt{eval}\ E\ d \multimap \{\texttt{return}\ V'\ d'\}$. Then there exists a closed term V of type* `exp`, *a closed term P of type $\texttt{ev}\ E\ V$, a context Γ' of destinations with $\Gamma \subseteq \Gamma'$ and a subterm R of C with a typing $\Gamma'; \Delta \vdash R : \texttt{return}\ V\ d \multimap \{\texttt{return}\ V'\ d'\}$.*

Proof. The proof is by induction on C. First of all C must have the form $\widehat{\lambda} u : \texttt{eval}\ E\ d.\{\ldots\}$. Secondly, since there is no way to construct a term of type `return` ... directly in the current context and signature, we know that C must consist of at least one computation step (let-term). This first step must be an application of one of the `eval_?`'s from the signature, since everything in the context constructing something monadic requires a term of type `return` ... to be present.

Now we can consider the different possibilities. The cases are very similar so we will only present the zero, successor and the application cases in detail.

Case: $C = \widehat{\lambda} u : \texttt{eval}\ \texttt{z}\ d.\{\mathsf{let}\ \{r\} = \texttt{eval\_z}\ d\ \widehat{\ }u\ \mathsf{in}\ R'\}$
In this case we can take $V = \texttt{z}$, $P = \texttt{ev\_z}$ and $R = \widehat{\lambda} r.\{R'\}$.

Case: $C = \widehat{\lambda} u : \texttt{eval}\ (\texttt{s}\ E_1)\ d.\{\mathsf{let}\ \{[d_0, p \otimes f]\} = \texttt{eval\_s}\ E_1\ d\ \widehat{\ }u\ \mathsf{in}\ C'\}$
We apply the induction hypothesis to $\Gamma_0; \Delta_0 \vdash \widehat{\lambda} p.\{C'\} : \texttt{eval}\ E_1\ d_0 \multimap \{\texttt{return}\ V'\ d'\}$ where $\Gamma_0 = \Gamma, d_0 : \texttt{dest}$ and $\Delta_0 = \Delta, f \,\widehat{:}\, \Pi v.\texttt{return}\ v\ d_0 \multimap \{\texttt{return}\ (\texttt{s}\ v)\ d\}$ to get $V_1 : \texttt{exp}$, $P_1 : \texttt{ev}\ E_1\ V_1$ and $\Gamma'; \Delta_0 \vdash R'$:

$\texttt{return}\ V_1\ d_0 \multimap \{\texttt{return}\ V'\ d'\}$. Now since $d_0 \neq d'$ then R' has to be of the form $\widehat{\lambda} r'.\{\mathsf{let}\ \{r\} = f\ V_1\ \hat{}\, r'\ \mathsf{in}\ R''\}$. Then we can take $V = \texttt{s}\ V_1$, $P = \texttt{ev\_s}\ P_1$ and $R = \widehat{\lambda} r.\{R''\}$.

Case: $C = \widehat{\lambda} u : \texttt{eval}\ (\texttt{case}\ E_1\ E_2\ E_3)\ d.\{\mathsf{let}\ \{[d_0, p \otimes f]\} =$
$\texttt{eval\_case}\ E_1\ E_2\ E_3\ d\ \hat{}\, u\ \mathsf{in}\ C'\}$

This is similar to the successor case above except that there is now two possible forms for R', each of which yields subcomputations with `eval`'s in the context; i.e. R' can be $\widehat{\lambda} r'.\{\mathsf{let}\ \{p'\} = (\pi_1\ f)\ \hat{}\, r'\ \mathsf{in}\ C_2'\}$ or $\widehat{\lambda} r'.\{\mathsf{let}\ \{p'\} = (\pi_2\ f)\ V''\ \hat{}\, r'\ \mathsf{in}\ C_3'\}$. The induction hypothesis can then be applied again on C_2' and C_3' yielding $\texttt{ev}\ E_2\ V$ and $\texttt{ev}\ (E_3\ V'')\ V$ respectively. Together with the big-step term from the first application of the induction hypothesis we can now create a term of type $\texttt{ev}\ (\texttt{case}\ E_1\ E_2\ E_3)\ V$ with either `ev_case_z` or `ev_case_s`.

Case: $C = \widehat{\lambda} u : \texttt{eval}\ (\texttt{lam}\ E')\ d.\{\mathsf{let}\ \{r\} = \texttt{eval\_lam}\ E'\ d\ \hat{}\, u\ \mathsf{in}\ R'\}$

This case is similar to the zero case; i.e. we take $V = \texttt{lam}\ E'$, $P = $ `ev_lam E'` and $R = \widehat{\lambda} r.\{R'\}$.

Case: $C = \widehat{\lambda} u : \texttt{eval}\ (\texttt{app}\ E_1\ E_2)\ d.\{\mathsf{let}\ \{[d_0, p_1 \otimes f_1]\} =$
$\texttt{eval\_app}\ E_1\ E_2\ d\ \hat{}\, u\ \mathsf{in}\ C_1\}$

We apply the induction hypothesis to $\Gamma_1; \Delta_1 \vdash \widehat{\lambda} p_1.\{C_1\} : \texttt{eval}\ E_1\ d_0 \multimap \{\texttt{return}\ V'\ d'\}$. This gives us $P_1 : \texttt{ev}\ E_1\ V_1$ and $\Gamma_1'; \Delta_1 \vdash R_1 : \texttt{return}\ V_1\ d_0 \multimap \{\texttt{return}\ V'\ d'\}$. Now R_1 has to be on the form $\widehat{\lambda} r.\{\mathsf{let}\ \{[d_0', p_2 \otimes f_2]\} = f_1\ E_1'\ \hat{}\, r\ \mathsf{in}\ C_2\}$. This implies that $V_1 = \texttt{lam}\ E_1'$. Since C_2 is a subterm of C we can apply the induction hypothesis on $\Gamma_2; \Delta_2 \vdash \widehat{\lambda} p_2.\{C_2\} : \texttt{eval}\ E_2\ d_0' \multimap \{\texttt{return}\ V'\ d'\}$. This gives $\vdash P_2 : \texttt{ev}\ E_2\ V_2$ and $\Gamma_2'; \Delta_2 \vdash R_2 : \texttt{return}\ V_2\ d_0' \multimap \{\texttt{return}\ V'\ d'\}$. Now R_2 has to be on the form $\widehat{\lambda} r.\{\mathsf{let}\ \{p_3\} = f_2\ V_2\ \hat{}\, r\ \mathsf{in}\ C_3\}$. Since C_3 is a subterm of C_2 which is a subterm of C we can apply the induction hypothesis on $\Gamma_3; \Delta \vdash \widehat{\lambda} p_3.\{C_3\} : \texttt{eval}\ (E_1'\ V_2)\ d \multimap \{\texttt{return}\ V'\ d'\}$. This gives $\vdash P_3 : \texttt{ev}\ (E_1'\ V_2)\ V_3$ and $\Gamma'; \Delta \vdash R_3 : \texttt{return}\ V_3\ d \multimap \{\texttt{return}\ V'\ d'\}$. Now we can set $V = V_3$, construct P from P_1, P_2 and P_3 using `ev_app` and set $R = R_3$.

Case: $C = \widehat{\lambda} u : \texttt{eval}\ (\texttt{fix}\ E')\ d.\{\mathsf{let}\ \{p\} = \texttt{eval\_fix}\ E'\ d\ \hat{}\, u\ \mathsf{in}\ C'\}$

This case follows directly from one application of the induction hypothesis.

□

Now we can apply the lemma to $d : \texttt{dest}; \cdot \vdash C : \texttt{eval}\ E\ d \multimap \{\texttt{return}\ V'\ d\}$. This gives $\Gamma'; \cdot \vdash R : \texttt{return}\ V\ d \multimap \{\texttt{return}\ V'\ d\}$, but since Δ is empty and $d = d'$, R has to be equal to $\widehat{\lambda} r.\{r\}$. This in turn implies $V' = V$ which gives us the sought $\vdash P : \texttt{ev}\ E\ V'$ completing the translation from destination-passing style to big-step semantics.

4.2.2 Representation of the proof in CLF

Currently CLF does not have a meta-theory to support the representation of proofs. So even though it is very speculative, it is still interesting to consider the representation of the above proof in CLF, as we will gain insight in some of the unresolved issues regarding the design of a meta-theory for CLF.

One of the main issues is how to adequately state the lemma (or theorems in general). Some of the problems that are related to linearity arise already in the context of linear LF (LLF) and are discussed in [7].

A natural first approach would be:[6]

```
lemma : ΠE:exp. ΠD₁:dest. ΠD₂:dest. ΠV₂:exp. ΠV₁:exp.
        (eval E D₁ ⊸ {return V₂ D₂})
        → ev E V₁
        → (return V₁ D₁ ⊸ {return V₂ D₂}) → type.
%mode lemma +E +D₁ +D₂ +V₂ -V₁ +C -P -R.
```

The line `%mode ...` is the Twelf way of specifying which arguments should be regarded as input (`+`) and which should be regarded as output (`-`). The type should thus be read as "Given E, D_1, D_2, V_2 and C where C has type `eval` E D_1 $\multimap$ $\{$`return` V_2 $D_2\}$, there exists V_1, P and R such that P has type `ev` E V_1 and R has type `return` V_1 D_1 $\multimap$ $\{$`return` V_2 $D_2\}$.

The zero case can be encoded without problems:

```
lemma_z : lemma z D₁ D₂ V₂ z
            (λ̂u:eval z D₁.{let {r’} = eval_z ^u in let {r} = R^r’ in r})
            ev_z R.
```

But we get in trouble with the successor case:

```
lemma_s : lemma (s E) D₁ D₂ V₂ (s V)
            (λ̂u:eval (s E) D₁.
                {let {[d’,p⊗f]} = eval_s ^u in
                 let {r} = C d’ ^f ^p in r})
            (ev_s P) R
     ← Πd’. Πf. lemma E d’ D₂ V₂ V (λ̂p. C d’ ^f ^p) P
            (λ̂r’.{let {r’’} = f V ^r’ in let {r} = R ^r’’ in r}).
```

There are two problems. The first is with `f`. One could imagine that a hypothetical CLF coverage checker doing output coverage[7] would not be able to see that `f` cannot occur in `R`. This is because the definition of the type family gives no indication of the relationship between the linear contexts of the given computation trace and the returned continuation, as opposed to the paper formulation in which we are able to state that they should be equal.

The second problem is the newly created destinations. Every time a new destination is created it stays in scope for the entire rest of the computation. This is handled in the paper proof above by stating that the continuation is typed in Γ', even though $\Gamma' \setminus \Gamma$ essentially is superfluous. But we cannot have a type family in which the different arguments are typed in different contexts; and realizing when the different destinations are no longer needed is not trivial by local observations. This problem manifests itself in the same way as the first, namely that a hypothetical coverage checker would not be able rule out the possibility of `d’` occurring in `R`.

[6] We will disregard the problem with the continuation being a subcomputation of the input, since that is already studied in the context of Twelf and can be solved.

[7] Output coverage checking is essentially checking the validity of inversion.

Another central issue is that of (input) coverage checking. Once we have all of the cases from the proof, a hypothetical coverage checker would need to figure out that all cases are indeed covered. This implies analyzing the possibilities of pattern matching monadic objects. If we disregard reordering of let-terms then coverage checking should not be much harder than for LF. But this is a very conservative solution and probably not what we want (see section 5 below).

As a side note, notice that the specification of normal contexts resembles the world declarations of Twelf.

5 Handling interleavings of let-bindings

The considered semantics are both sequential. Let us see what happens if we use the features of CLF to make the destination-passing style concurrent. Consider the following alternative, concurrent version of `eval_app`:

```
eval_app' : ΠE₁:exp. ΠE₂:exp. ΠD:dest.
                eval (app E₁ E₂) D ⊸
                    {∃d₁:dest. ∃d₂:dest. eval E₁ d₁ ⊗ eval E₂ d₂ ⊗
                        (ΠE₁':exp → exp. ΠV₂:exp.
                            return (lam (λx. E₁' x)) d₁ ⊸
                            return V₂ d₂ ⊸ {eval (E₁' V₂) D}
                        )
                    }.
```

This version differs from the previous by adding both `eval` E_1 d_1 and `eval` E_2 d_2 to the context at the same time. This means that the subsequent evaluations of E_1 and E_2 can happen in any order and the individual steps can be arbitrarily interleaved. However, since these two computations are essentially independent, the different traces representing different interleavings must all be equivalent modulo let-floating; but since this fact is not immediate the proof gets more complicated.

Let us see how a proof translating this concurrent version into big-step looks like. First of all, since there can now be multiple `eval`'s in the context we will have to modify the definition of normal contexts to accomodate this, i.e. allow variables with types `eval` E D and `return` V D to occur in a normal context. With this new definition lemma 4.4 can be reused in its exact same formulation. Notice that this singles out a particular `eval` E d to be the focus of the lemma.

Now in order to start the proof and split by cases like we did above we will need to argue that C does indeed begin with the consumption of the `eval` E D that the lemma focusses on. This is, however, no longer immediate. The computation trace C can just as well begin with the consumption of any of the other `eval`'s in the context or by the application of a $\Pi\ldots$ `return`$\ldots \multimap \{\ldots\}$ to a corresponding `return`. Therefore we will need a let-floating-lemma to state that any C with the type given is equivalent to a trace in which the particular `eval` E d is consumed first:

Lemma 5.1 (let-floating for `eval`'s) *Let* Γ *be a context of destinations,* $\Gamma = d_1 :$

$\texttt{dest}, \ldots, d_n : \texttt{dest}$*, and let* Δ *be a normal linear context. Let* E *and* V' *be closed terms of type* $\texttt{exp}$*. Let* d *and* d' *be two (not necessarily distinct) destinations in* Γ*. And let* C *be a term with a typing* $\Gamma; \Delta \vdash C : \texttt{eval}\ E\ d \multimap \{\texttt{return}\ V'\ d'\}$*. Then there exists a term* $C' \equiv C$*, such that* $C' = \widehat{\lambda} u.\{\mathsf{let}\ \{\ldots\} = \ldots\ \hat{}u\ \mathsf{in}\ C''\}$*.*

The dots in the form for C' covers all the different cases that the main proof subsequently splits into.

The proof of this let-floating-lemma relies on the fact that there can never be introduced anything in the (linear or unrestricted) contexts, which would allow the linear ressource $\texttt{eval}\ E\ d$ to be consumed in any different way.

With this in place we can reuse the cases of the proof for zero, lambda and fixpoint without changes. The other cases will however require their own let-floating-lemmas. Consider for instance the successor case; after the application of the induction hypothesis, we want to apply inversion to conclude that R' begins with the application of f, but this requires a specific let-floating-lemma stating that any R' of the corresponding type is equivalent to a term beginning with the application of f. Similarly for the other cases; each time inversion is used on the R resulting from the induction hypothesis we will need a specific let-floating-lemma.

Here are two of them:

Lemma 5.2 (let-floating for the successor case) *Let* Γ *be a context of destinations,* $\Gamma = d_1 : \texttt{dest}, \ldots, d_n : \texttt{dest}$*, and let* Δ *be a normal linear context. Let* V *and* V' *be closed terms of type* $\texttt{exp}$*. Let* d *and* d' *be two (not necessarily distinct) destinations in* Γ *and let* d'' *be a destination in* Γ *distinct from the other two. And let* R *be a term with typing* $\Gamma; \Delta, f \mathrel{\hat{:}} \Pi v.\ \texttt{return}\ v\ d'' \multimap \{\texttt{return}\ (\texttt{s}\ v)\ d\}, r' \mathrel{\hat{:}} \texttt{return}\ V\ d \vdash R : \{\texttt{return}\ V'\ d'\}$*. Then there exists a term* $R' \equiv R$*, such that* $R' = \{\mathsf{let}\ \{r\} = f\ V\ \hat{}r'\ \mathsf{in}\ R''\}$*.*

Lemma 5.3 (let-floating for the concurrent app case) *Let* Γ *be a context of destinations,* $\Gamma = d_1 : \texttt{dest}, \ldots, d_n : \texttt{dest}$*, and let* Δ *be a normal linear context. Let* V_1*,* V_2 *and* V' *be closed terms of type* $\texttt{exp}$*. Let* d *and* d' *be two (not necessarily distinct) destinations in* Γ *and let* d_1 *and* d_2 *be two distinct destinations in* Γ *distinct from the other two. And let* R *be a term with typing* $\Gamma; \Delta, f \mathrel{\hat{:}} \Pi e. \Pi v.\ \texttt{return}\ (\texttt{lam}\ (\lambda x.\ e\ x))\ d_1 \multimap \texttt{return}\ v\ d_2 \multimap \{\texttt{eval}\ (e\ v)\ d\}, r_1 \mathrel{\hat{:}} \texttt{return}\ V_1\ d_1, r_2 \mathrel{\hat{:}} \texttt{return}\ V_2\ d_2 \vdash R : \{\texttt{return}\ V'\ d'\}$*. Then* V_1 *is equal to* $\texttt{lam}\ E$ *for some* E *and there exists a term* $R' \equiv R$*, such that* $R' = \{\mathsf{let}\ \{p\} = f\ E\ V_2\ \hat{}r_1\ \hat{}r_2\ \mathsf{in}\ R''\}$*.*

If let-floating has to be reasoned about explicitly in CLF then we could probably just as well have represented the concurrent features explicitly as it would be done in Twelf. To get actual benefit from CLF it therefore seems likely that we would have to come up with a let-floating aware coverage-checker, such that the let-floating would be handled behind the scenes, much like substitution is handled behind the scenes in Twelf. More specifically, in a trace where A and B can occur in either order, we want to be able to implicitly assume that for instance A occurred first.

6 CLF signatures

All the proofs so far are working with a fixed signature and of course cannot be expected to work with arbitrary extensions to the signature. Extending MiniML in any way will naturally require extensions to the proofs as well. This is all good and reasonable. If we however extend the signature with something completely different i.e. new types and type families, we would expect the proofs to work without any changes. So far these are just the natural expectations coming from the way Twelf works.

In Twelf we know this is how things work, since execution is goal-oriented and adding a new type family does not add any new constructors to the old type families. In CLF execution works differently. When inside the monad, the execution semantics will simply nondeterministically perform any action possible given the current signature and context. And since the proofs at some point have to conclude that there can be no more computation, any signature allowing monadic objects — and thereby computation steps — to be constructed directly will disrupt the proofs.

Therefore I propose a simple restriction on CLF signatures which will hopefully simplify meta-theory representations a bit. Consider the number of terms N of type $\cdot;\cdot \vdash N : \{1\}$ in some signature. Of course we can have $N = \{1\}$. But if there are any other terms $N : \{1\}$ then any computation trace constructing any monadic type can have interjections of completely irrelevant, superfluous steps. The proposed restriction is therefore that there can be only one term N of type $\{1\}$ in the empty context. Adding stuff like

```
junk : type.
junk_intro : junk.
junk_elim : junk ⊸ {1}.
```

would therefore be considered an illegal signature.

A conservative approximation of this restriction which is easy to compute, is to simply start the proof search semantics looking for a term of type $\{1\}$. The first step after entering the monad is a nondeterministic choice depending on the signature. Now if the only option for this nondeterministic choice is to terminate the forward-directed mode and construct 1 directly then we are certain that the signature is legal, otherwise we reject the signature.

7 Conclusion and future work

We have proven a traditional big-step semantics equivalent to a destination-passing semantics by induction on terms with an equivalence relation capable of modelling concurrency. Examination of this proof has identified several problems regarding meta-theory representations in CLF.

First, there is the problem of scoping; during the course of a computation in the monad, every intuitionistically introduced term stays in the context. This means that subcomputations cannot easily be split, since the different parts are typed in increasingly larger contexts. One solution could perhaps be to represent proofs in a forward-directed manner in the monad, since this would allow $\exists$-introductions of variables instead of Π-introductions. In the case of the destinations they did not actually occur; if this is the common case, another solution might be to infer this by an automated analysis.

Second, there is the problem of linear contexts; this could though perhaps be solved at the CLF meta-level with some sort of extended world-declaration stating which terms should be linear in which arguments. Alternatively, the work on hybrid metalogical frameworks [7] might be applicable.

Third, there is the problem regarding coverage in the context of let-floating. There is a lot to be gained if a coverage checker could be devised in such a way that the overhead of let-floating-lemmas described in section 5 could be moved to the correctness proof of the coverage checker.

Furthermore it has been argued that restricting the CLF signatures in some way is necessary for a CLF implementation. Specifically it seems like a good idea to require that the type $\{1\}$ is only inhabited by a single term.

Acknowledgement

I thank Andrzej Filinski and my advisor Carsten Schürmann for helpful discussions.

References

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

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

[3] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. 2005. "Monadic concurrent linear logic programming". In Proceedings of the 7th ACM SIGPLAN international Conference on Principles and Practice of Declarative Programming (Lisbon, Portugal, July 11–13, 2005). PPDP '05. ACM Press, New York, NY, 35–46.

[4] Andrew McCreight and Carsten Schürmann. "A Meta-Linear Logical Framework". Proceedings of Logical Frameworks and Meta Languages, July 2004.

[5] Frank Pfenning and Carsten Schürmann. "System description: Twelf — a meta-logical framework for deductive systems". In Proceedings of the 16th International Conference on Automated Deduction (CADE-16), Trento, Italy, June 1999. H. Ganzinger, Ed. Lecture Notes In Computer Science, vol. 1632. Springer-Verlag, London, 202-206.

[6] Frank Pfenning. "Substructural operational semantics and linear destination-passing style". In W.-N. Chin, editor, Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04), page 196, Taipei, Taiwan, Nov. 2004. Springer-Verlag LNCS 3302.

[7] Jason Reed. "A Hybrid Metalogical Framework". Thesis Proposal Working Draft. Jan. 2007. http://www.cs.cmu.edu/~jcreed/papers/thesprop.pdf