Formalizing Operational Semantics in Logical Frameworks: A Case Study using LF / Elf

Henning Niss [*] John Hatcliff [†]
DIKU
Computer Science Department
University of Copenhagen [‡]

December 19, 1995

Abstract

We illustrate how definitions and correctness proofs for the familiar thunk-based simulation of call-by-name by call-by-value can be encoded in the logical framework LF and the logic programming language Elf. This work is part of a survey of logical frameworks applied to formalizing operational semantics and associated meta-theory.

1 Introduction

Logical frameworks are useful tools for specifying and formalizing meta-theoretic properties of operational semantics definitions. Recent work illustrates their use in mechanically verifying compiler correctness, correctness of type-based program analysis, and correctness of program specialization [2,3,6,8]. Several different frameworks have been used in these examples — with varying degrees of success. In an attempt to clarify relative strengths and weaknesses, we are conducting a survey of logical frameworks as they are used to encode and reason about operational semantics definitions.

We use the following criteria to evaluate the logical frameworks in our survey:

- simplicity of encoding of object language syntax (including the ability to identify terms up to α-equivalence),
- simplicity of encoding of standard forms for operational semantics (*e.g.*, big-step semantics, small-step semantics, abstract machines),

[*]E-mail: `hniss@diku.dk`,
WWW: `http://www.diku.dk/research-groups/topps/personal/hniss/`.
[†]E-mail: `hatcliff@diku.dk`,
WWW: `http://www.diku.dk/research-groups/topps/personal/hatcliff.html`.
This work is supported by the Danish Research Academy and by the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils.
[‡]Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark.

- ability to prototype (*e.g.*, interpreters, compilers, analyses) directly from encoded specifications, and
- degree to which associated meta-theory (*e.g.*, compiler correctness proofs) can be encoded and mechanically checked.

We apply all the frameworks to a single test case: formalization of the correctness proofs for the familiar thunk-based simulation of call-by-name by call-by-value [9] This test case is conceptually simple and contains many elements associated with classic presentations of operational semantics (*e.g.*, Plotkin's seminal work [20]):

- "small-step" call-by-name and call-by-value operational semantics of λ-terms,
- call-by-name and call-by-value program calculi (*i.e.*, equational theories for reasoning about observational equivalences), and
- correctness of program transformations (*i.e.*, operational adequacy, and preservation of the convertability relation of program calculi).

Encoding these elements in a logical framework requires formalization of various forms of induction (*e.g.*, induction on the structure of terms, induction on the structure of derivations in the operational semantics, induction on the number of steps in a derivation sequence), capture-free substitution, and substitution properties in translations.

The present work reports on part of the survey. We consider the logical framework LF [7] and its realization in Elf [16] — a logic programming language based on the LF type system.

The paper is organized as follows. Section 2 gives a brief overview of LF and Elf. Section 3 presents the syntax and semantics of λ-terms and associated Elf encodings. Section 4 presents the syntax and semantics of thunks and associated Elf encodings. Section 5 presents the formalization of the proofs of correctness for the thunk transformation. Section 6 gives an assessment of LF/Elf with respect to our evaluation criteria and briefly mentions relevant aspects of our ongoing survey of other logical frameworks.

Elf has been used previously to encode operational semantics [6,11] and λ-calculi [17]. Our encoding techniques and presentation style will follow this previous work closely. The organization of the present paper mirrors the original presentation of the proofs that we encode [10], and the reader may find it helpful to read the present paper with that presentation in hand. Space constraints force us to give only representative examples of the proof encodings. We refer the reader to the extended version of this paper [14] for the complete formalization and all associated Elf code.

2 LF and the Elf Programming Language

In this section we present a short overview of the Logical Framework LF and the logical programming language Elf. Harper, Honsell and Plotkin [7] is the canonical reference for LF, while Pfenning [16] is the canonical (technical) reference for Elf. For further examples of using Elf as a tool for encoding operational semantics and related theory the reader is referred to [11,17,19].

2.1 LF — a framework for defining logics

The LF calculus [7] has three levels: *objects*, *families*, and *kinds*. Families are classified by kinds, and objects are classified by *types*, *i.e.*, families of kind Type.

$$
\begin{array}{llcl}
\textit{Kinds} & K & ::= & \mathsf{Type} \mid \Pi x\!:\!A.\,K \\
\textit{Families} & A & ::= & a \mid \Pi x\!:\!A_1.\,A_2 \mid \lambda x\!:\!A_1\,.\,A_2 \mid A\,M \\
\textit{Objects} & M & ::= & c \mid x \mid \lambda x\!:\!A\,.\,M \mid M_1\,M_2
\end{array}
$$

Family-level constants are denoted by a, and object-level constants by c. $A_1 \to A_2$ abbreviates $\Pi x : A_1.\,A_2$ and $\Pi x : A.\,K$ when x does not appear free in A_2 or K, respectively. The typing rules for LF can be found in [7]. We take $\alpha\beta\eta$-equivalence as the notion of *definitional equality* in LF [7, Appendix A.3]. For all the languages we consider, we identify terms up to renaming of bound variables.

One defines a logic in LF by specifying a *signature* which declares the kinds of family-level constants a and types of object-level constants c. These constants are constructors for the logic's syntax, judgements, and deductions. Well-formedness is enforced by LF type-checking as described in [7, Section 2]. The LF type system can represent the conditions associated with binding operators, with schematic abstraction and instantiation, and with the variable occurrence and discharge conditions associated with rules in systems of natural deduction.

2.2 Elf — an implementation of LF

Elf is a logic programming language based on the LF type system, see Pfenning [15,16]. The syntax of Elf is as follows (the last column lists the corresponding LF term). Optional components are enclosed in $\langle\cdot\rangle$.

$$
\begin{array}{rcll}
\textit{kindexp} & ::= & \texttt{type} & \mathsf{Type} \\
 & \mid & \{\textit{id}\langle\texttt{:}\textit{famexp}\rangle\}\ \textit{kindexp} & \Pi x\!:\!A.\,K \\
 & \mid & \textit{famexp}\ \texttt{->}\ \textit{kindexp} & A \to K \\
\textit{famexp} & ::= & \textit{id} & a \\
 & \mid & \{\textit{id}\langle\texttt{:}\textit{famexp}_1\rangle\}\ \textit{famexp}_2 & \Pi x\!:\!A_1.\,A_2 \\
 & \mid & \texttt{[}\textit{id}\langle\texttt{:}\textit{famexp}_1\rangle\texttt{]}\ \textit{famexp}_2 & \lambda x\!:\!A_1\,.\,A_2 \\
 & \mid & \textit{famexp}\ \textit{objexp} & A\,M \\
 & \mid & \textit{famexp}_1\ \texttt{->}\ \textit{famexp}_2 & A_1 \to A_2 \\
 & \mid & \textit{famexp}_2\ \texttt{<-}\ \textit{famexp}_1 & A_1 \to A_2 \\
\textit{objexp} & ::= & \textit{id} & c \\
 & \mid & \texttt{[}\textit{id}\langle\texttt{:}\textit{famexp}\rangle\texttt{]}\ \textit{objexp} & \lambda x\!:\!A\,.\,M \\
 & \mid & \textit{objexp}_1\ \textit{objexp}_2 & M_1\,M_2
\end{array}
$$

The terminal *id* ranges over variables, and family and object constants. Bound variables and constants in Elf can be arbitrary identifiers, but free variables in a declaration or query must begin with an upper case letter. Free variables act as logic variables and are implicitly Π-abstracted. Elf's *term reconstruction phase* (preprocessing) infers these abstractions as well as appropriate arguments to these abstractions. It also fills in the omitted types in quantifications $\{x\}$ and abstractions $\texttt{[}x\texttt{]}$ and omitted types or objects indicated by an underscore `_`. The `<-` is used to improve the readability of some Elf programs. `B <- A` is parsed into the same representation as `A -> B`; `->` is right associative, while `<-` is left associative. An Elf program is a representation of an LF signature. Although we are implicitly encoding logics in LF, we give all encodings using the syntax of Elf.

In addition to the type checking LF terms Elf also implements a *search mechanism*[1]. Proof search proceeds by a depth-first search (in the style of Prolog) for a closed object of a specified type; that is, a type corresponds to a goal in Prolog. The query may contain free variables to be instantiated during the search. Pfenning [16] gives the operational semantics of the search mechanism.

[1]Note, however, that proof search is *external* to the LF calculus.

Object language:

$$\begin{array}{lcl} e & \in & \Lambda \\ e & ::= & \mathsf{b} \;\mid\; x \;\mid\; \lambda x\,.\,e \;\mid\; e_0\,e_1 \end{array}$$

Elf encoding:

```
exp : type.

b   : exp.
lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.
```

Figure 1: Abstract syntax of the language Λ

3 The lambda calculus

3.1 Syntax and semantics of λ-terms

Figure 1 presents the syntax of the object language Λ. Λ is a pure untyped λ-calculus with non-functional constants, λ-abstractions (*i.e.* functions) and applications. Note, that in [10] b represented a *family* of constants, whereas in our presentation `b` represents a *single* constant. We identify α-equivalent terms and adopt the convention that bound variables occurring in terms in definitions, proofs, *etc.* are chosen to be different from the free variables [1, p. 26]. If e_1 and e_2 are α-equivalent we write $e_1 \equiv_\alpha e_2$.

$FV(e)$ denotes the set of *free variables* of a term e. $e[x := e']$ denotes the capture-avoiding (possibly involving renaming) *substitution* of e' for all free occurrences of x in e. A *context* C is a term with a "hole" $[\cdot]$. The operation of *filling* the context C with a term e yields the term $C[e]$, possibly capturing some free variables of e in the process. *Contexts*$[l]$ denotes the set of contexts from some language l. Closed terms, *i.e.* terms with no free variables, are called *programs*. *Programs*$[l]$ denotes the set of programs for a language l.

Figure 1 presents the Elf signature used to represent object language syntax. The syntactic category Λ is represented as a type-family `exp`. Associated with `exp` is an object constant for each syntax constructor in Λ. For non-functional constants, abstractions, and applications, the associated object constant is parametrized by the subexpressions of the expression. Thus, `b` is a nullary constant since it has no subexpressions, and `app` is binary constant with one argument for each subexpression.

Identifiers do *not* have an associated object constant; the reason is the use of higher-order abstract syntax. This principle is originally due to Church [4], but the term *higher-order abstract syntax* is more recent [18]. The idea is to represent object-level variables by meta-level variables and binding constructs in the object-language, *e.g.* in this case λ-abstractions, by bindings in the meta-language. Thus, the constant `lam` should be applied to a function from `exp` to `exp` (compare the types of `app` which takes two subexpressions as arguments, and `lam` which takes a function as argument).

For example, the term $\lambda x\,.\,\lambda y\,.\,\mathsf{b}$ is represented in Elf by

```
lam [x] (lam [y] b).
```

Remember that `[x` $\langle$:*famexp*$\rangle$`]` e is the notation for the (LF) λ-abstraction $\lambda x\!:\!famexp\,.\,e$.

In general an (object-level) λ-abstraction is represented by

```
lam [x] E
```

where `E` is the representation of e (with occurrences of x represented by `x`).

We formalize the above notions of encoding by defining a *representation function* $\cdot$ that maps terms in Λ to their representations in Elf.

$$\begin{array}{rcl} \cdot & : & \Lambda \rightarrow \texttt{exp} \\ \mathsf{b} & = & \texttt{b} \\ x & = & \texttt{x} \\ \lambda x \,.\, e & = & \texttt{lam [x]} \;\; e \\ e_1\, e_2 & = & \texttt{app} \;\; e_1 \quad e_2 \end{array}$$

To be completely formal we should show that this (and every later defined) representation function is *adequate* (as illustrated in [7, Theorems 3.1 and 3.2]); that is it is a *compostional bijection*. The intuition is that *injectivity* gives us a *unique* Elf representation for every Λ-term; *surjectivity* ensures that there is no term in `exp` without a corresponding Λ-term; finally, *compositionality* ($e[x \;:=\; e'] \;\; = \;\; e \,[\; x \;\; := \;\; e' \;]$) allows us to use meta-level substitution for object-level substitution. We omit the straightforward proofs of adequacy for all representation functions we consider. Detailed proofs for various examples can be found in [7].

The real advantage of using higher-order abstract syntax shows up when implementing (object-level) α-equivalence and substitution. First, observe that since we encode object-level variables using meta-level variables and since α-equivalence is included in definitional equality, *we do not have to encode α-conversions explicitly.* For instance, asserting that e_1 and e_2 are α-equivalent (at the object-level) corresponds to asserting that e_1 and e_2 are α-equivalent (at the meta-level).

Second, the following reasoning show that object-level substitution can be "represented" by (meta-level) application and can be "carried-out" by (meta-level) β-reduction.

$$\begin{array}{rcl} e[x \;:=\; e'] & = & e \,[\texttt{x} \;:=\; e' \,] \\ & \equiv_{\alpha\beta\eta} & (\texttt{[x]} \; e \;) \; e' \end{array}$$

3.1.1 Values

Certain terms of Λ are designated as *values* (i.e. canonical terms). Intuitively, values correspond to the results of evaluations. The sets $\mathit{Values}_{\mathrm{n}}[\Lambda]$ and $\mathit{Values}_{\mathrm{v}}[\Lambda]$ below represent the set of values from the language Λ under call-by-name and call-by-value evaluation respectively.

$$\begin{array}{rcl} v & \in & \mathit{Values}_{\mathrm{n}}[\Lambda] \\ v & ::= & \mathsf{b} \mid \lambda x \,.\, e \end{array} \qquad \begin{array}{rcl} v & \in & \mathit{Values}_{\mathrm{v}}[\Lambda] \\ v & ::= & \mathsf{b} \mid x \mid \lambda x \,.\, e \end{array} \qquad \ldots \; \textit{where } e \in \Lambda$$

Note that identifiers are included in $\mathit{Values}_{\mathrm{v}}[\Lambda]$ since only values will be substituted for identifiers under call-by-value evaluation. We use v as a meta-variable for values and where no ambiguity results we will ignore the distinction between call-by-name and call-by-value values.

Specifying the values by grammars is not directly suitable for Elf. Rather we translate the grammars to judgements $\cdot\; \mathit{Value}_{cbn}$ and $\cdot\; \mathit{Value}_{cbv}$. This will be the general technique for representing (object-level) sets specified by grammars, but in the future encodings we will not go through the procedure again.

Giving only the definition for Value_{cbn} (since Value_{cbv} is similar) we have

$$\frac{}{\mathsf{b}\;\mathit{Value}_{cbn}} \qquad \frac{}{\lambda x \,.\, e\;\; \mathit{Value}_{cbn}}$$

Object language:

$$v \in Values_{\mathrm{n}}[\Lambda] \qquad v \in Values_{\mathrm{v}}[\Lambda]$$
$$v ::= \mathtt{b} \mid \lambda x \,.\, e \qquad v ::= \mathtt{b} \mid x \mid \lambda x \,.\, e \qquad \ldots \textit{ where } e \in \Lambda$$

Elf encoding:

Call-by-name:

```
cbn_val     : exp -> type.

cbn_val_b   : cbn_val b.
cbn_val_lam : cbn_val (lam E).
```

Call-by-value:

```
cbv_val     : exp -> type.

cbv_val_b   : cbv_val b.
cbv_val_lam : cbv_val (lam E).
```

Figure 2: Value judgements for terms in Λ

These judgements are encoded in Elf by declaring a new family-level constant `cbn_val` as

```
cbn_val : exp -> type.
```

parameterized by an expression. Intuitively, `cbn_val` acts as a predicate on expressions; the term `cbn_val E` conveys the information that the expression `E` satisfies `cbn_val`.

To say that `b` is a value under call-by-name we say

```
cnb_val_b : cbn_val b.
```

This states that the constant `cbn_val_b` has type `cbn_val b`, *i.e.*, it is a deduction of `cbn_val b`.

Similarly, for `lam E`

```
cbn_val_lam : cbn_val (lam E).
```

Figure 2 gives the complete signatures for the call-by-name and call-by-value value judgements. Note, that due to the use of higher-order abstract syntax the fact that x is a value is not directly expressed as a clause in the encoding, instead we use it as a hypothesis when using the predicates `cbn_val` and `cbv_val`.

3.1.2 Operational semantics

Figure 3 presents single-step call-by-name and call-by-value evaluation rules for closed Λ terms. The rules presented are *schematic* in subexpressions of the expression. The (call-by-name) β-rule, for instance, is schematic in e_0 and e_1, while the (call-by-name) apply-rule is schematic in e_0, e_0' and e_1.

The general technique (*e.g.*, Michaylov and Pfenning [11]) for encoding operational semantics is to represent the reduction relation, *e.g.* $\longmapsto_{\mathrm{n}}$, as a type-family, *e.g.* `-->n`, indexed by the representation of the Λ-term and its reduct. $e_1 \longmapsto_{\mathrm{n}} e_2$, then `E1 -->n E2` (where `E1` and `E2` are the representations of e_1 and e_2). Each inference rule is represented by an object constant which

Object language:

Call-by-name:

$$(\lambda x \,.\, e_0)\, e_1 \longmapsto_{\mathrm{n}} e_0[x := e_1] \qquad \frac{e_0 \longmapsto_{\mathrm{n}} e_0'}{e_0\, e_1 \longmapsto_{\mathrm{n}} e_0'\, e_1}$$

Call-by-value:

$$(\lambda x \,.\, e)\, v \longmapsto_{\mathrm{v}} e[x := v] \qquad \frac{e_0 \longmapsto_{\mathrm{v}} e_0'}{e_0\, e_1 \longmapsto_{\mathrm{v}} e_0'\, e_1}$$

$$\frac{e_1 \longmapsto_{\mathrm{v}} e_1'}{(\lambda x \,.\, e_0)\, e_1 \longmapsto_{\mathrm{v}} (\lambda x \,.\, e_0)\, e_1'}$$

Elf encoding:

Call-by-name:

```
-->n : exp -> exp -> type.      %infix none 10 -->n
                                %name  -->n    R

-->n_beta : (app (lam E0) E1) -->n (E0 E1).
-->n_app  :           E0 -->n E0'
          -> (app E0 E1) -->n (app E0' E1).
```

Call-by-value:

```
-->v : exp -> exp -> type.      %infix none 10 -->v
                                %name  -->v    R

-->v_beta :  cbv_val V
          -> (app (lam E) V) -->v (E V).
-->v_app1 :                 E1 -->v E1'
          -> (app (lam E0) E1) -->v (app (lam E0) E1').
-->v_app2 :           E0 -->v E0'
          -> (app E0 E1) -->v (app E0' E1).
```

Figure 3: Single-step evaluation rules for Λ

acts as a deduction constructor. For each rule, the corresponding constructor takes deductions of any rule hypotheses as arguments and yields a deduction of the rule conclusion as a result. Schematic rules are represented by Π-quantification over the schematic variables. Thus, the representation of the β-rule is quantified over e_0 and e_1, while the apply-rule is quantified over e_0, e_0' and e_1.

```
-->n      : exp -> exp -> type.

-->n_beta : {E0:exp -> exp}    {E1:exp}
                (app (lam E0) E1) -->n (E0 E1).
-->n_app  : {E0:exp} {E0':exp} {E1:exp}
                         E0 -->n E0'
            ->  (app E0 E1) -->n (app E0' E1).
```

Remember that `{E0:exp -> exp} t` is Elf's notation for the (LF) object-level expression $\Pi e_0 : \texttt{exp} \rightarrow \texttt{exp}.\, t$. Fortunately, the Elf front end is able to infer these quantifications: by convention each identifier starting with a capital letter is implicitly Π-quantified during type reconstruction. With this convention the above rules are written as follows.

```
   -->n : exp -> exp -> type.      %infix none 10 -->n
                                   %name  -->n    R

   -->n_beta : (app (lam E0) E1) -->n (E0 E1).
   -->n_app  :           E0 -->n E0'
            -> (app E0 E1) -->n (app E0' E1).
```

The annotation `%infix none 10 -->n` informs the Elf parser and the Elf search engine that `-->n` is an infix relation with no (syntactic) associativity and a precedence level of 10 (the higher the level, the tighter the binding). The annotation `%name --> n R` informs the Elf output functions that variables occurring in deductions constructed with `-->n` should be named `R`, `R1`, *etc.* Other possible relation type annotations are `prefix` and `postfix` and the associativities allowed are `left`, `right` or `none`.

From the single-step reduction rules we can define the reflexive, transitive closure (denoted $\longmapsto^*$) and the transitive closure (denoted $\longmapsto^+$). Defining and encoding these are straightforward (see for instance [17, Pages 7 and 8]). Figure 4 presents the definitions and encodings (we omit the cases for call-by-value since they are identical to the call-by-name cases).

Figure 5 defines (partial) evaluation functions $eval_n$ and $eval_v$ in terms of $\longmapsto^*$. The requirement that evaluation produces a value is enforced using the `cbn_val` and `cbv_val` judgements.

3.1.3 Executing the evaluator specifications

Using Elf's depth-first proof search mechanism, we obtain an executable evaluator directly from the specifications in the previous section. For instance, the call-by-name evaluation of the term $((\lambda x\,.\,\lambda y\,.\,x)\,\mathsf{b})\,(\lambda x\,.\,x)$ is expressed as

```
?- evaln (app (app (lam [x] (lam [y] x)) b) (lam [x] x)) E'.
Solving...

E' = b.

yes
```

By introducing a logic variable `P`, one obtains the deduction of the evaluation.

Object language:

$$\frac{}{e \longmapsto_{\mathrm{n}}^{*} e} \qquad \frac{e \longmapsto_{\mathrm{n}} e' \quad e' \longmapsto_{\mathrm{n}}^{*} e''}{e \longmapsto_{\mathrm{n}}^{*} e''}$$

$$\frac{e \longmapsto_{\mathrm{n}} e'}{e \longmapsto_{\mathrm{n}}^{+} e'} \qquad \frac{e \longmapsto_{\mathrm{n}} e' \quad e' \longmapsto_{\mathrm{n}}^{+} e''}{e \longmapsto_{\mathrm{n}}^{+} e''}$$

Elf encoding:

```
-->n* : exp -> exp -> type.      %infix none 10 -->n*
                                 %name  -->n*   R*
-->n*_refl  :   E -->n* E.
-->n*_trans :  E1 -->n  E2
            -> E2 -->n* E3
            -> E1 -->n* E3.

-->n+ : exp -> exp -> type.      %infix none 10 -->n+
                                 %name  -->n+   R+
-->n+_base  :  E  -->n  E'
            -> E  -->n+ E'.
-->n+_trans :  E1 -->n  E2
            -> E2 -->n+ E3
            -> E1 -->n+ E3.
```

Figure 4: The reflexive, transitive closure of $\longmapsto_{\mathrm{n}}$ in Λ

Object language:

$$\begin{array}{llll} \mathit{eval}_{\mathrm{n}}(e) = v & \text{iff} \quad e \longmapsto_{\mathrm{n}}^{*} v & \quad \ldots \textit{ where } v \in \mathit{Values}_{\mathrm{n}}[\Lambda] \\ \mathit{eval}_{\mathrm{v}}(e) = v & \text{iff} \quad e \longmapsto_{\mathrm{v}}^{*} v & \quad \ldots \textit{ where } v \in \mathit{Values}_{\mathrm{v}}[\Lambda] \end{array}$$

Elf encoding:

```
evaln : exp -> exp -> type.
evaln_def :  evaln E V
          <- E -->n* V
          <- cbn_val V.

evalv : exp -> exp -> type.
evalv_def :  evalv E V
          <- E -->v* V
          <- cbv_val V.
```

Figure 5: Evaluation functions

```
?- P : evaln (app (app (lam [x] (lam [y] x)) b) (lam [x] x)) E'.
Solving...

E' = b,
P =
   evaln_def cbn_val_b
      (-->n*_trans (-->n_app -->n_beta) (-->n*_trans -->n_beta -->n*_refl)).

yes
```

The form of `P` shows us that the deduction uses `evaln_def` (the basic definition of the evaluation function) applied to deductions `cbn_val_b` and (`-->n*_trans ...`). `cbn_val_b` expresses that `b` is a value (this is needed in order to conclude evaluation to `b`). `P'` = `-->n*_trans ...` shows how the multi-step reduction was constructed. The expression `P'` corresponds to the following deduction tree.

$$P' = \cfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{((\lambda x\,.\,\lambda y\,.\,x)\,\mathsf{b})\,(\lambda x\,.\,x) \;\longmapsto^*_{\mathrm{n}}\; \mathsf{b}}\;\mathsf{trans}$$

The subdeductions $\mathcal{D}_1$ and $\mathcal{D}_2$ are as follows.

$$\mathcal{D}_1 \quad = \quad \cfrac{\cfrac{}{(\lambda x\,.\,\lambda y\,.\,x)\,\mathsf{b} \;\longmapsto_{\mathrm{n}}\; \lambda y\,.\,\mathsf{b}}\;\mathsf{beta}}{((\lambda x\,.\,\lambda y\,.\,x)\,\mathsf{b})\,(\lambda x\,.\,x) \;\longmapsto_{\mathrm{n}}\; (\lambda y\,.\,\mathsf{b})\,(\lambda x\,.\,x)}\;\mathsf{app}$$

$$\mathcal{D}_2 \quad = \quad \cfrac{\cfrac{}{(\lambda y\,.\,\mathsf{b})\,(\lambda x\,.\,x) \;\longmapsto_{\mathrm{n}}\; \mathsf{b}}\;\mathsf{beta} \qquad \cfrac{}{\mathsf{b} \;\longmapsto^*_{\mathrm{n}}\; \mathsf{b}}\;\mathsf{refl}}{(\lambda y\,.\,\mathsf{b})\,(\lambda x\,.\,x) \;\longmapsto^*_{\mathrm{n}}\; \mathsf{b}}\;\mathsf{trans}$$

`P'` is obtained by reading the names off the tree from bottom to top.

4 Thunks

4.1 Thunk introduction

To establish the simulation properties of thunks, we extend the language Λ to the language Λ_τ that includes suspension operators.

$$\begin{array}{rcl} e & \in & \Lambda_\tau \\ e & ::= & \ldots \;\;|\;\; \mathsf{delay}\; e \;\;|\;\; \mathsf{force}\; e \end{array}$$

To encode thunks in Elf we define a new syntactic category `texp` for Λ_τ. `texp` includes the constants of `exp` with the following additional declarations:[2]

```
...
tdelay : texp -> texp
tforce : texp -> texp
```

Note, that in order to avoid confusing the constants of the two syntactic categories during type reconstruction, we need two separate type families with associated constants. We use the

[2]In this section we only give significant differences to the encodings of syntax, values and operational semantics already giving in Section 3.1. The full encodings are presented in [14].

Object language:

$$\begin{array}{rcl}
\mathcal{T} & : & \Lambda \to \Lambda_{\mathcal{T}} \\
\mathcal{T}\langle\!|\mathsf{b}|\!\rangle & = & \mathsf{b} \\
\mathcal{T}\langle\!|x|\!\rangle & = & \mathsf{force}\, x \\
\mathcal{T}\langle\!|\lambda x \,.\, e|\!\rangle & = & \lambda x \,.\, \mathcal{T}\langle\!|e|\!\rangle \\
\mathcal{T}\langle\!|e_0\, e_1|\!\rangle & = & \mathcal{T}\langle\!|e_0|\!\rangle\, (\mathsf{delay}\, \mathcal{T}\langle\!|e_1|\!\rangle)
\end{array}$$

Elf encoding:

```
trans: exp -> texp -> type.     %name trans R

trans_b:  trans b tb.

trans_lam:  ({x : exp} {y: texp}
               (trans x (tforce y)) -> (trans (E x) (T y)))
         -> trans (lam E) (tlam T).

trans_app:  trans E0 T0
         -> trans E1 T1
         -> trans (app E0 E1) (tapp T0 (tdelay T1)).
```

Figure 6: Thunk introduction

convention that the representation of a term in $\Lambda_{\mathcal{T}}$ is prefixed with a `t` and that variables in `texp`-terms are called `y` instead of `x`.

The operator delay suspends the computation of an expression (*i.e.*, it creates a *thunk*) — thereby coercing an expression to a value. The operator force then forces the evaluation of a delayed computation. Therefore, $\mathsf{delay}\, e$ is added to the value sets in $\Lambda_{\mathcal{T}}$.

$$\begin{array}{lcl}
v & \in & \mathit{Values}_{\mathrm{n}}[\Lambda_{\mathcal{T}}] \\
v & ::= & \ldots \mid \mathsf{delay}\, e
\end{array}
\qquad
\begin{array}{lcl}
v & \in & \mathit{Values}_{\mathrm{v}}[\Lambda_{\mathcal{T}}] \\
v & ::= & \ldots \mid \mathsf{delay}\, e
\end{array}
\qquad \ldots\ \textit{where}\ e \in \Lambda_{\mathcal{T}}$$

The encodings of these sets are based on the value sets of Λ with the following additional declaration for delay (the call-by-value case is similar).

```
tcbn_val_delay : tcbn_val (tdelay E).
```

Figure 6 presents the definition of the translation function $\mathcal{T}$ between terms in Λ and terms in $\Lambda_{\mathcal{T}}$. The LF framework does not have inductive types (in contrast to *e.g.*, HOL [5, Chapter 20]) and hence we cannot formalize the translation function as a recursive function over the structure of Λ terms.

The standard solution when using Elf is to represent such functions as relations, *i.e.* a function f from domain A to range B is represented as a relation in $A \times B$, where x and y are related iff $f(x) = y$. This technique is (implicitly) used whenever we represent object-level functions in Elf.

In this particular case, we encode $\mathcal{T}$ as a relation `trans` in `exp` $\times$ `texp`, such that when e and t are related by `trans`, `trans(`e`,`t`)`, we have that $\mathcal{T}\langle\!|e|\!\rangle = t$. The relation is represented by a type-family `trans` indexed by its two arguments

```
trans : exp -> texp -> type.
```

Using this technique it is easy to encode the translation of constants and applications. In the case of constants we simply translate between the two different Elf constants, and in the case of application we appeal to recursive calls for each of the subexpressions. Note, that the recursive nature of $\mathcal{T}$ results in a recursive definition of the `trans` relation.

λ-bindings and variables, however, need a more careful treatment. This is best illustrated by recasting the translation function (we only show the case for abstractions) as an inference rule system. The rule that translates an abstraction $\lambda x \,.\, e$ to a thunked expression employs a *hypothetical* judgement. That is, in translating e we assume that we can translate x to $\mathsf{force}\, x$ using the identifier rule for $\mathcal{T}$. In the style of *natural deduction* (Prawitz [21]) this is written as follows.

$$\dfrac{\dfrac{}{\text{trans}\; x\; (\mathsf{force}\; x)}\,u \\ \vdots \\ \text{trans}\; e\; t}{\text{trans}\; (\lambda x \,.\, e)\; (\lambda x \,.\, t)}\,u$$

where the label u indicates where the hypothesis is introduced and discharged.

Deductions of hypothetical judgements are encoded in Elf as functions from deductions to deductions. Therefore, a deduction of the premise in the above rule is a function from a deduction of trans $x\,(\mathsf{force}\, x)$ to a deduction of trans $e\, t$.

4.2 Reduction of thunked terms

The evaluation rules for Λ_τ are obtained by adding the following rules to both the call-by-name and call-by-value evaluation rules of Figure 3.

$$\frac{e \longmapsto e'}{\mathsf{force}\; e \longmapsto \mathsf{force}\; e'} \qquad \mathsf{force}\; (\mathsf{delay}\; e) \longmapsto e$$

When formalizing this, we add

```
t-->n_fd   : (tforce (tdelay E)) t-->n E.

t-->n_force :           E t-->n E'
            -> (tforce E) t-->n (tforce E').
```

to the encodings of the call-by-name operational semantics (`-->n`) for Λ. Similar additions are made to `-->v` to obtain `t-->v`.

Plotkin [20] used the β and β_{v} calculi for reasoning about programs under call-by-name and call-by-value evaluation respectively. For reasoning about Λ_τ programs, we consider the following notion of τ-reduction: $\mathsf{force}\,(\mathsf{delay}\, e) \longrightarrow_\tau e$. We refer the reader to [9] for the full definitions of these calculi.

5 A thunk-based simulation of call-by-name

In this section we state and prove the correctness theorem for the thunked-based simulation of call-by-name under call-by-value evaluation. In the theorem below, $E_1 \simeq E_2$ ($E_1 \simeq_\tau E_2$) holds if E_1 and E_2 are both undefined, or else both defined and denote α-equivalent (τ-equivalent) terms.

Theorem 1 *For all $e \in Programs[\Lambda]$ and $e_1, e_2 \in \Lambda$,*

1. **Indifference:** $eval_{\mathrm{v}}(\mathcal{T}\langle\!\lbrace e \rbrace\!\rangle) \simeq eval_{\mathrm{n}}(\mathcal{T}\langle\!\lbrace e \rbrace\!\rangle)$

$$e_0 \overset{\mathcal{T}}{\sim} \mathcal{T}\langle\!\langle e_0 \rangle\!\rangle \qquad e_1 \overset{\mathcal{T}}{\sim} t_1^j \qquad e_2 \overset{\mathcal{T}}{\sim} t_2^j \qquad\qquad e_m \overset{\mathcal{T}}{\sim} t_m^j$$

$$\begin{tikzcd}
\mathcal{T}\langle\!\langle e_0 \rangle\!\rangle \arrow[r, mapsto, "\beta_{\mathrm{v}}"'] & t_1^1 \arrow[r, mapsto, "*"', "\tau"] & t_1^{i_1} \arrow[r, mapsto, "\beta_{\mathrm{v}}"'] & t_2^1 \arrow[r, mapsto, "*"', "\tau"] & t_2^{i_2} \arrow[r, mapsto, "\beta_{\mathrm{v}}"'] & \cdots\cdots \arrow[r, mapsto, "\beta_{\mathrm{v}}"'] & t_m^1 \arrow[r, mapsto, "*"', "\tau"] & t_m^{i_m} \\
e_0 \arrow[rr, mapsto, "\mathrm{n}"'] & & e_1 \arrow[rr, mapsto, "\mathrm{n}"'] & & e_2 \arrow[r, mapsto, "\mathrm{n}"'] & \cdots\cdots \arrow[r, mapsto, "\mathrm{n}"'] & e_m
\end{tikzcd}$$

Figure 7: Evaluation of thunked terms

2. **Simulation:** $\mathcal{T}\langle\!\langle eval_{\mathrm{n}}(e) \rangle\!\rangle \simeq_{\mathcal{T}} eval_{\mathrm{v}}(\mathcal{T}\langle\!\langle e \rangle\!\rangle)$

3. **Translation:**

$$\lambda\beta \vdash e_1 = e_2 \text{ iff } \lambda\beta_{\mathrm{v}}\tau \vdash \mathcal{T}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle \text{ iff } \lambda\beta\tau \vdash \mathcal{T}\langle\!\langle e_1 \rangle\!\rangle = \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle$$

Indifference shows that terms in the image of the translation are evaluation order independent. **Simulation** shows how call-by-name evaluation of Λ terms is mirrored by call-by-value evaluation in the image of the translation. Finally, **Translation** shows that the translation preserves and reflects Λ convertability.

For space reason this paper only gives proofs and encodings for **Simulation**. The additional proofs and encodings are given in full in the companion technical report [14].

5.1 Simulation

5.1.1 Motivating the simulation proof

Simulation, $\mathcal{T}\langle\!\langle eval_{\mathrm{n}}(e) \rangle\!\rangle \simeq_{\mathcal{T}} eval_{\mathrm{v}}(\mathcal{T}\langle\!\langle e \rangle\!\rangle)$, can be proved by showing that for each e in Λ,

- if $eval_{\mathrm{n}}(e)$ is defined then so is $eval_{\mathrm{v}}(\mathcal{T}\langle\!\langle e \rangle\!\rangle)$ and $\mathcal{T}\langle\!\langle eval_{\mathrm{n}}(e) \rangle\!\rangle =_{\mathcal{T}} eval_{\mathrm{v}}(\mathcal{T}\langle\!\langle e \rangle\!\rangle)$
- if $eval_{\mathrm{n}}(e)$ is undefined then so is $eval_{\mathrm{v}}(\mathcal{T}\langle\!\langle e \rangle\!\rangle)$,

In this paper we will only a single-step version of item (1) above. This is known as the *one-step simulation* property — it shows how each call-by-name step in Λ is mirrored by a call-by-value step in the image of the translation. One would hope that

$$e_1 \longmapsto_{\mathrm{n}} e_2 \quad \text{implies} \quad \mathcal{T}\langle\!\langle e_1 \rangle\!\rangle \longmapsto^{+}_{\mathrm{v}} \mathcal{T}\langle\!\langle e_2 \rangle\!\rangle$$

but unfortunately this doesn't hold. Consider, for instance, the term $e \overset{\mathrm{def}}{=} (\lambda x\,.\,\lambda y\,.\,x)\,\mathsf{b}$ and the corresponding term $t \overset{\mathrm{def}}{=} \mathcal{T}\langle\!\langle e \rangle\!\rangle \equiv_\alpha (\lambda x\,.\,\lambda y\,.\,\mathsf{force}\,x)\,(\mathsf{delay}\,\mathsf{b})$ in $\Lambda_{\mathcal{T}}$. Then call-by-name reduction on e yields

$$(\lambda x\,.\,\lambda y\,.\,x)\,\mathsf{b} \longmapsto_{\mathrm{n}} \lambda y\,.\,\mathsf{b} \overset{\mathrm{def}}{=} e_1$$

whereas call-by-value reduction on t yields

$$(\lambda x\,.\,\lambda y\,.\,\mathsf{force}\,x)\,(\mathsf{delay}\,\mathsf{b}) \longmapsto_{\mathrm{v}} \lambda y\,.\,(\mathsf{force}\,\mathsf{delay}\,\mathsf{b}) \overset{\mathrm{def}}{=} t_1,$$

and the two reducts are *not* α-convertible, *i.e.*, $\mathcal{T}\langle\!\langle e_1 \rangle\!\rangle \not\equiv_\alpha t_1$.

The problem is that reduction in $\Lambda_{\mathcal{T}}$ might result in "left-over" force-delay pairs. The solution is to define a relation $\overset{\mathcal{T}}{\sim}$ such that $e \overset{\mathcal{T}}{\sim} t$ when t and $\mathcal{T}\langle\!\langle e \rangle\!\rangle$ only differ by force-delay pairs.

Object language:

$$\overset{\mathcal{T}}{\sim}.1 \quad \mathsf{b} \overset{\mathcal{T}}{\sim} \mathsf{b} \qquad\qquad \overset{\mathcal{T}}{\sim}.4 \quad \frac{e_0 \overset{\mathcal{T}}{\sim} t_0 \quad e_1 \overset{\mathcal{T}}{\sim} t_1}{e_0\, e_1 \overset{\mathcal{T}}{\sim} t_0\, (\mathsf{delay}\, t_1)}$$

$$\overset{\mathcal{T}}{\sim}.2 \quad x \overset{\mathcal{T}}{\sim} \mathsf{force}\, x$$

$$\overset{\mathcal{T}}{\sim}.3 \quad \frac{e \overset{\mathcal{T}}{\sim} t}{\lambda x\,.\,e \overset{\mathcal{T}}{\sim} \lambda x\,.\,t} \qquad\qquad \overset{\mathcal{T}}{\sim}.5 \quad \frac{e \overset{\mathcal{T}}{\sim} t}{e \overset{\mathcal{T}}{\sim} \mathsf{force}\, (\mathsf{delay}\, t)}$$

Elf encoding:

```
~: exp -> texp -> type.          %infix none 10 ~
                                 %name ~ R

~_b:      b ~ tb.

~_lam:  ({x : exp} {y : texp}
             x ~ (tforce y) -> (E x) ~ (T y))
     -> (lam E) ~ (tlam T).

~_app:           E0 ~ T0
     ->          E1 ~ T1
     -> (app E0 E1) ~ (tapp T0 (tdelay T1)).

~_fd:  E ~ T
    -> E ~ (tforce (tdelay T)).
```

Figure 8: The $\overset{\mathcal{T}}{\sim}$ relation

With this relation, the steps involved in $\mathcal{T}\langle\!|eval_{\mathrm{n}}(e)|\!\rangle$ and $eval_{\mathrm{v}}(\mathcal{T}\langle\!|e|\!\rangle)$ can be pictured as in Figure 7 (in the figure, $\longmapsto_{\tau}$ and $\longmapsto_{\beta_{\mathrm{v}}}$ denote $\longmapsto_{\mathrm{v}}$ steps which correspond to τ and β_{v} reduction, respectively).

Figure 8 presents the (inductive) definition of $\overset{\mathcal{T}}{\sim}$. Simple inductions show that $e \overset{\mathcal{T}}{\sim} \mathcal{T}\langle\!|e|\!\rangle$, and that $e \overset{\mathcal{T}}{\sim} t$ implies $\mathcal{T}\langle\!|e|\!\rangle$ is τ-equivalent to t. The encoding of the relation (Figure 8) is similar to the `trans` judgement.

Now to prove the one-step simulation property we need two auxillary properties: a substitution property, and a formal understanding of force-delay reductions. These are addressed in the following two sections.

5.1.2 The substitution property

The substitution property shows how $\overset{\mathcal{T}}{\sim}$ interacts with substitution. We write $\mathcal{D} :: J$ when $\mathcal{D}$ is a deduction of judgement J.

Property 1 *For all $e_0, e_1 \in \Lambda$ and $t_0, t_1 \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$,*

$$e_0 \overset{\mathcal{T}}{\sim} t_0 \;\wedge\; e_1 \overset{\mathcal{T}}{\sim} t_1 \quad\Rightarrow\quad e_0[x := e_1] \overset{\mathcal{T}}{\sim} t_0[x := \mathsf{delay}\, t_1]$$

Computationally, this states that given deductions of $\mathcal{R}_0 :: e_0 \overset{\mathcal{T}}{\sim} t_0$ and $\mathcal{R}_1 :: e_1 \overset{\mathcal{T}}{\sim} t_1$ we can construct a deduction $\mathcal{R}$ of

$$e_0[x := e_1] \overset{\mathcal{T}}{\sim} t_0[x := \mathsf{delay}\, t_1],$$

i.e., we can consider the property as a *function* from $\mathcal{R}_0$ and $\mathcal{R}_1$ to $\mathcal{R}$. Thus we represent the proof in Elf as a relation, encoded using the type family

```
subst   : ({x:exp} {y:texp} x ~ (tforce y) -> E0 x ~ T0 y)
        ->    E1 ~ T1
        -> E0 E1 ~ T0 (tdelay T1)
        -> type.
```

based on the observations 1) that the representation of $\mathcal{R}_0$ may contain `x` and `y` free (since it proves a property about the body of the abstraction) and hence may refer to `x ~ tforce y`, and 2) that object-level substitution is represented as meta-level application (Section 3.1, page 5).

Proof by induction over the structure of the derivation $\mathcal{R}_0$ of $e_0 \overset{\mathcal{T}}{\sim} t_0$

Case $\mathcal{R}_0 :: \mathsf{b} \overset{\mathcal{T}}{\sim} \mathsf{b}$: Then

$$\begin{array}{rcll} \mathsf{b}[x := e_1] & \equiv_\alpha & \mathsf{b} & \\ & \overset{\mathcal{T}}{\sim} & \mathsf{b} & \ldots\ \textit{by}\ \overset{\mathcal{T}}{\sim}.1 \\ & \equiv_\alpha & \mathsf{b}[x := \mathsf{delay}\, t_1] & \end{array}$$

This case is represented in Elf as

```
subst_b   :  subst ([x] [y] [fx:x ~ (tforce y)] ~_b) R1
               ~_b.
```

Case $\mathcal{R}_0 :: x' \overset{\mathcal{T}}{\sim} \mathsf{force}\, x'$, $x' = x$: To construct $\mathcal{R}$ such that

$$\mathcal{R} :: x'[x := e_1] \overset{\mathcal{T}}{\sim} \mathsf{force}\, x'[x := \mathsf{delay}\, t_1]$$

we reason as follows

$$\begin{array}{rcll} x'[x := e_1] & \equiv_\alpha & e_1 & \\ & \overset{\mathcal{T}}{\sim} & \mathsf{force}\, (\mathsf{delay}\, t_1) & \ldots\ \textit{by}\ \overset{\mathcal{T}}{\sim}.5\ \textit{and}\ e_1 \overset{\mathcal{T}}{\sim} t_1 \\ & \equiv_\alpha & (\mathsf{force}\, x')[x := \mathsf{delay}\, t_1] & \end{array}$$

since $x' = x$.

In the encoding we appeal to the hypothesis that $x \overset{\mathcal{T}}{\sim} \mathsf{force}\, x$ indicating that $x' = x$.

```
subst_fx  :  subst ([x] [y] [fx:x ~ (tforce y)] fx) R1
               (~_fd R1).
```

Case $\mathcal{R}_0 :: x' \overset{\mathcal{T}}{\sim} \mathsf{force}\, x'$, $x' \neq x$: We have

$$\begin{array}{rcll} x'[x := e_1] & \equiv_\alpha & x' & \\ & \overset{\mathcal{T}}{\sim} & \mathsf{force}\, x' & \ldots\ \textit{by}\ \mathcal{R}_0 \\ & \equiv_\alpha & (\mathsf{force}\, x')[x := \mathsf{delay}\, t_1] & \end{array}$$

and thus $\mathcal{R}$ can be set to $\mathcal{R}_0$.

This case is not represented as an actual clause. Instead, it is represented as an assumption about the behaviour of the cases involving identifiers, *i.e.*, the case for abstractions. Pfenning [17, Section 5] elaborates.

Case $\mathcal{R}_0 :: \lambda x' . e \overset{\mathcal{T}}{\sim} \lambda x' . t$ because $e \overset{\mathcal{T}}{\sim} t$: Applying the inductive hypothesis to $e \overset{\mathcal{T}}{\sim} t$ yields

$$e[x := e_1] \overset{\mathcal{T}}{\sim} t[x := t_1].$$

Then, $\overset{\mathcal{T}}{\sim}$.3 used on this gives (since we assume $x' \neq x$ by our variable convention)

$$\begin{array}{rcll} (\lambda x' . e)[x := e_1] & \equiv_\alpha & \lambda x' . (e[x := e_1]) & \\ & \overset{\mathcal{T}}{\sim} & \lambda x' . (t[x := \mathsf{delay}\, t_1]) & \quad \textit{... by ind. hyp.} \\ & \equiv_\alpha & (\lambda x' . t)[x := \mathsf{delay}\, t_1] & \end{array}$$

Assuming that $x' \neq x$ we appeal to the prevous case in the premise for the body of the abstractions.

```
subst_lam :  subst ([x] [y] [fx:x ~ (tforce y)]
                      ~_lam (R0' x y fx)) R1
                 (~_lam R')
          <- ({z:exp} {z':texp} {fz:z ~ (tforce z')}
                subst ([x] [y] [fx1:x ~ (tforce y)] fz) R1 fz
                -> subst ([x] [y] [fx2:x ~ (tforce y)]
                           (R0' x y fx2 z z' fz)) R1
                      (R' z z' fz)).
```

Case $\mathcal{R}_0 :: e\,e' \overset{\mathcal{T}}{\sim} t\,t'$ because $e \overset{\mathcal{T}}{\sim} t$ and $e' \overset{\mathcal{T}}{\sim} t'$: The inductive hypothesis (on $e \overset{\mathcal{T}}{\sim} t$ and $e' \overset{\mathcal{T}}{\sim} t'$) gives us proofs $\mathcal{R}'$ and $\mathcal{R}''$ of

$$\begin{array}{r} \mathcal{R}' :: e[x := e_1] \overset{\mathcal{T}}{\sim} t[x := \mathsf{delay}\, t_1] \\ \mathcal{R}'' :: e'[x := e_1] \overset{\mathcal{T}}{\sim} t'[x := \mathsf{delay}\, t_1] \end{array}$$

By definition of substitution we have

$$\begin{array}{rcll} (e\,e')[x := e_1] & \equiv_\alpha & (e[x := e_1])\,(e'[x := e_1]) & \\ & \overset{\mathcal{T}}{\sim} & (t[x := \mathsf{delay}\, t_1])\,(\mathsf{delay}\,(t'[x := t_1])) & \quad \textit{... by } \overset{\mathcal{T}}{\sim}\text{.4}, \mathcal{R}', \mathcal{R}'' \\ & \equiv_\alpha & (t[x := \mathsf{delay}\, t_1])\,((\mathsf{delay}\, t')[x := \mathsf{delay}\, t_1]) & \\ & \equiv_\alpha & (t\,(\mathsf{delay}\, t'))[x := \mathsf{delay}\, t_1] & \end{array}$$

The encoding in Elf uses straightforward applications of the inductive hypothesis

```
subst_app :  subst ([x] [y] [fx:x ~ (tforce y)]
                      ~_app (R0' x y fx) (R0' x y fx)) R1
                 (~_app R0'' R0'')
          <- subst R0' R1 R'
          <- subst R0' R1 R''.
```

Case $\mathcal{R}_0 :: e \overset{\mathcal{T}}{\sim} \mathsf{force}\,(\mathsf{delay}\, e)$: is similar. ∎

5.1.3 Initial force-delay-reductions

When e and t are related by $\overset{\mathcal{T}}{\sim}$, we know that t has a shape similar to $\mathcal{T}\langle\!\langle e \rangle\!\rangle$ except for occurrences of force-delay-pairs ("inserted" by $\overset{\mathcal{T}}{\sim}$.5). In particular, t might have some *initial* force-delay-pairs

$$\mathsf{force}\,(\mathsf{delay}\,(\ldots)).$$

The following property shows how reduction in Λ_τ eliminates these initial force-delay-pairs. After these reductions, the shape of the reduct corresponds to the shape of the image $\mathcal{T}\langle\!\langle e \rangle\!\rangle$ at the top-most syntax constructor (note, however, that inside the term there may still be force-delay-pairs).

Property 2 *For all $e \in \mathit{Programs}[\Lambda]$ and $t \in \mathit{Programs}[\mathcal{T}\langle\!|\Lambda|\!\rangle^*]$ such that $e \overset{\mathcal{T}}{\sim} t$,*

$$\begin{array}{llcccccl}
\text{a)} & e & \equiv & \mathsf{b} & \overset{\mathcal{T}}{\sim} & t & \Rightarrow & t \longmapsto_{\mathrm{v}}^{*} b \\
\text{b)} & e & \equiv & \lambda x \,.\, e_0 & \overset{\mathcal{T}}{\sim} & t & \Rightarrow & t \longmapsto_{\mathrm{v}}^{*} \lambda x \,.\, t_0 \quad \textit{where } e_0 \overset{\mathcal{T}}{\sim} t_0 \\
\text{c)} & e & \equiv & e_0\, e_1 & \overset{\mathcal{T}}{\sim} & t & \Rightarrow & t \longmapsto_{\mathrm{v}}^{*} t_0\,(\mathsf{delay}\, t_1) \quad \textit{where } e_0 \overset{\mathcal{T}}{\sim} t_0 \textit{ and } e_1 \overset{\mathcal{T}}{\sim} t_1
\end{array}$$

The proof of each of the subproperties are very similar — we show only the cases for abstractions.

The formalization (again) expresses the proof as a function from a deduction of $\lambda x \,.\, e_0 \overset{\mathcal{T}}{\sim} t$ to deductions of $t \longmapsto_{\mathrm{v}}^{*} \lambda x \,.\, t_0$ and $e_0 \overset{\mathcal{T}}{\sim} t_0$.

```
    fdred_lam :  ((lam E0) ~ T)
              -> (T t-->v* (tlam T0))
              -> ({x:exp} {y:texp} {fx:x ~ tforce y} E0 x ~ T0 y)
              -> type.
```

Note, that since the bodies e_0 and t_0 may contain free occurrences of x the deduction of $e_0 \overset{\mathcal{T}}{\sim} t_0$ may refer to `x ~ tforce y`.

Proof by induction over the derivation of $e \overset{\mathcal{T}}{\sim} t$

Case $\overset{\mathcal{T}}{\sim}.3$ :: $\lambda x \,.\, e_0 \overset{\mathcal{T}}{\sim} \lambda x \,.\, t_0$: immediate.

The encoding is simple

```
fd_lam_base : fdred_lam (~_lam R) t-->v*_refl
                     ([x] [y] [fx:x ~ tforce y] (R x y fx)).
```

Case $\overset{\mathcal{T}}{\sim}.5$ :: $\lambda x \,.\, e_0 \overset{\mathcal{T}}{\sim} \mathsf{force}\,(\mathsf{delay}\, t')$: Then

$$\begin{array}{lll}
\mathsf{force}\,(\mathsf{delay}\, t') & \longmapsto_{\mathrm{v}} & t' \\
 & \longmapsto_{\mathrm{v}}^{*} & \lambda x \,.\, t_0 \qquad \text{where } e_0 \overset{\mathcal{T}}{\sim} t_0 \qquad \textit{... by ind. hyp}
\end{array}$$

The encoding simply appeals to the inductive hypothesis *via* a recursive call to `fred_lam`

```
fd_lam_ind  :  fdred_lam R1 S1 R1'
            -> fdred_lam (~_fd R1)
                         (t-->v*_trans t-->v_fd S1) R1'.
```

Observe, how we explicitly construct a reduction *via* transitivity of `t-->v*` (recall, that a deduction of `t-->v*_trans` consists of a `t-->v` and a `t-->v*` deduction). This technique of "gluing" together subdeductions is also required later. ∎

5.1.4 One-step simulation

The one-step simulation property show how each call-by-name step in Λ is mirrored by a call-by-value step (between appropriate τ-equivalence classes) in Λ_τ.

Property 3 ($\mathcal{T}$ — one step simulation)
For all $e_0, e_1 \in \mathit{Programs}[\Lambda]$ and $t_0 \in \mathit{Programs}[\mathcal{T}\langle\!|\Lambda|\!\rangle^*]$ such that $e_0 \overset{\mathcal{T}}{\sim} t_0$,

$$e_0 \longmapsto_{\mathrm{n}} e_1 \;\Rightarrow\; \exists t_1 \in \mathcal{T}\langle\!|\Lambda|\!\rangle^* \,.\; t_0 \longmapsto_{\mathrm{v}}^{+} t_1 \;\wedge\; e_1 \overset{\mathcal{T}}{\sim} t_1$$

Again, it is important to view this property as a function from deductions of $e_0 \overset{\mathcal{T}}{\sim} t_0$ and $e_0 \longmapsto_{\mathrm{n}} e_1$ to deductions of $t_0 \longmapsto_{\mathrm{v}}^{+} t_1$ and $e_1 \overset{\mathcal{T}}{\sim} t_1$. Note, that the existential quantifier is *not* represented explicitly in the encoding. Viewing the property as a function from deductions to deductions implies that we never consider actual terms, all manipulations are based on deduction objects.

```
simul1 :  (E0 -->n E1)
       -> (E0 ~ T0)
       -> (T0 t-->v+ T1)
       -> (E1 ~ T1)
       -> type.
```

The formalization of this proof in Elf is particularly interesting because it follows the informal proof very closely, while at the same time it shows that the informal proof really is informal, *i.e.*, it relies on unspoken intuition.

Proof by induction over the derivation of $e_0 \longmapsto_{\mathrm{n}} e_1$

Case $(\lambda x \,.\, e_a)\, e_b \longmapsto_{\mathrm{n}} e_a[x := e_b]$: We have

$$\begin{array}{llll} t_0 & \longmapsto_{\mathrm{v}}^{*} \; t'_a\,(\mathsf{delay}\,t_b) & \text{where } \lambda x \,.\, e_a \overset{\mathcal{T}}{\sim} t'_a \text{ and } e_b \overset{\mathcal{T}}{\sim} t_b & \textit{... by Prop. 2, c)} \\ & \longmapsto_{\mathrm{v}}^{*} \; (\lambda x \,.\, t_a)\,(\mathsf{delay}\,t_b) & \text{where } e_a \overset{\mathcal{T}}{\sim} t_a & \textit{... by Prop. 2, b)} \\ & \longmapsto_{\mathrm{v}} \; t_a[x := \mathsf{delay}\,t_b] & & \textit{... by a } \beta_{\mathrm{v}}\textit{-step} \end{array}$$

The substitution property (Property 1) applied to $e_a \overset{\mathcal{T}}{\sim} t_a$ and $e_b \overset{\mathcal{T}}{\sim} t_b$ yields

$$e_a[x := e_b] \overset{\mathcal{T}}{\sim} t_a[x := \mathsf{delay}\,t_b],$$

completing this proof case.

The encoding follows the structure of the informal proof by first appealing to the initial force-delay-reductions property (Property 2) twice and then to the substitution property (Property 1) once. But in order to construct a deduction of

$$t_0 \longmapsto_{\mathrm{v}}^{+} t_a[x := \mathsf{delay}\,t_b]$$

we have to "glue" together the subdeductions. Firstly, being formal we actually apply Prop. 2, b) to t'_a and not the *entire* term $t'_a\,(\mathsf{delay}\,t_b)$, but by using compatibility of $\longmapsto_{\mathrm{v}}^{*}$ with respect to applications, we obtain the indicated reduction step. Secondly, we glue this deduction and the deduction of the first reduction step together, using transitivity of $\longmapsto_{\mathrm{v}}^{*}$, to obtain a deduction of

$$t_0 \longmapsto_{\mathrm{v}}^{*} (\lambda x \,.\, t_a)\,(\mathsf{delay}\,t_b).$$

Finally, this step and a β_{v}-step are glued togther to obtain the full deduction (the fact that we use a β_{v}-reduction allows us to conclude $\longmapsto_{\mathrm{v}}^{+}$, not just $\longmapsto_{\mathrm{v}}^{*}$).

```
simul1_lam :  simul1 (-->n_beta) Q R8 R9

           <- fdred_app Q R1 R2 R3
           <- fdred_lam R2 R4 R5
           <- t-->v*_app_comp R4 R6
           <- t-->v*_*trans R1 R6 R7
           <- t-->v+_*trans R7 (t-->v_beta tcbv_val_delay) R8
           <- subst R5 R3 R9.
```

In this we have used the formalization of some propeties about reductions: `t-->v*_app_comp` respectively `t-->v+_app_comp` shows compatibility of `t-->v*` respectively `t-->v+` with respect to applications. `t-->v*_*trans` constructs a deduction of `t-->v*` from two `t-->v*` deductions. `t-->v+_*+trans` takes a `t-->v*` and a `t-->v` deduction and yields a `t-->v+` deduction. Similarly, `t-->v+_*+trans` takes a `t-->v*` and a `t-->v+` deduction and yields a `t-->v+` deduction.

Case $e_a\, e_b \longmapsto_{\mathrm{n}} e'_a\, e_b$ because $e_a \longmapsto_{\mathrm{n}} e'_a$: Similar to the case above.

∎

6 Assessment

Based on techniques introduced in previous work [6,11], we have given another example of how LF/Elf can be used to encode operational semantics and associated meta-theory. We now give an evaluation of LF/Elf with respect to the criteria presented in the introduction.

6.1 Evaluation

Simplicity of encoding of object language syntax: LF was designed with a specific encoding methodology in mind. The methodology centers around the use of higher-order abstract syntax to represent binding constructs in the object language. Furthermore, the notion of what it means for the encoding to be correct (*i.e.*, *adequate*) has been clearly presented in the definition of the framework: an adequate encoding must be a compositional bijection [7].

This is in contrast to other logical frameworks and proof assistants where such notions are not emphasized or mentioned at all. In other frameworks where higher-order abstract syntax is not used, showing that the encoding preserves the notion of object term equality (*e.g.*, α-equivalence) is often more complicated than in LF since the given notion of term equality must also be explicitly encoded in the framework by the user. In LF, this is not necessary (if object term is based on α-equivalence) since in using higher-order abstract syntax (*i.e.*, representing object language binding with meta-language binding), LF definitional equality is sufficient for expressing object term equality.

A disadvantage of emphasizing higher-order abstract syntax is that this seems to preclude the inclusion of inductive types in the framework. This implies a less than satisfactory representation of the meta-theory (as discussed below). On a more pragmatic note, the use of higher-order abstract syntax often complicates the presentation of meta-theory and "higher-level" judgements since one is never able to reason *directly* about object language variables, but instead must reason under hypotheses about properties that variables satisfy. This is the case *e.g.*., in the definition of the translation `trans` (Section 4.1) and in the definition of the substitution property `subst` (Section 5.1.2). Thus, the particular encoding style of object language syntax affects the presentation of all layers of formal reasoning "above" it.

Simplicity of encoding of standard forms for operational semantics: Previous work as well as this case study illustrate LF's ability to formalize big and small-step operational semantics, program calculi, program transformations, type systems, and type-based program analyses. LF's type system is rich enough to express well-formednes conditions on deductions in all of these logistic systems — checking well-formedness is reduced to type checking (which is decidable). This is in contrast to other systems such as Prolog, λ-Prolog, and Typol [12,13] that are frequently used to encode operational semantics. In those systems, well-formedness must be checked by *e.g.*, defining a function to traverse the data structure representing a deduction. To establish decidability of well-formedness, one must prove that every such predicate introduced is total.

Ability to prototype directly from encoded specifications: Section 3 illustrated how the specifications of the call-by-name and call-by-value evaluators could be directly executed using Elf's operational interpretation of types. For simple systems this works quite well. However, one must keep in mind the evaluation order imposed by Elf's search strategy. This is of particular relevance in non-deterministic systems where one may need to re-order Elf clauses to observe the full range of operational behaviour.

Another factor complicating the prototyping of large systems from LF specifications is that LF/Elf has no built in primitive data types such as numbers, lists, *etc.*. The new module facility to be released with the next version of Elf should make it easier to code larger examples.

Ability to encode and mechanically check associated meta-theory: The biggest weakness of Elf is its inability to completely formalize all the meta-theory (especially forms of induction) commonly associated with operational semantics. Section 5 illustrated how *individual cases* of induction proofs could be formalized, but that there is no way to guarantee (within LF) the function representing an inductive proof is total. Rohwedder and Pfenning [22] have recently proposed a system *external* to the LF logic which can automatically check the *mode* and *termination* of Elf predicates. This system should be useful, but in its current form it can still not detect that Elf predictates are total (*i.e.*, they *terminate* and *succeed*).

In contrast, many proof assistants (*e.g.*, HOL [5]) do provide an inductive definition facility and all the inductive proofs which we were unable to completely formalize here can be completely formalized in such systems.

6.2 Conclusion

Elf (along with the relatively weak logic LF) seems to lie in the middle of a continuum running from logic programming languages to proof assistents. On one side, logic programming languages give more efficient execution of logic programs than Elf (better prototyping abilities) but can seldom be used to machine-check meta-theory. On the other side, proof assistants (with stronger logics) formalize meta-theory better than Elf, but have little or no prototyping ability. Elf provides one alternative to this dichotomy. Another approach is to have some intermediate language for specifying operational semantics in proof assistants which allows easy compilation to a logic programming language (for the purpose of prototyping). This is the essence of ongoing work by Bertot and Fraer [2].

References

[1] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, 1984.

[2] Yves Bertot and Ranan Fraer. Reasoning with executable specifications. In TAPSOFT'95 [23], pages 531–545.

[3] Sandrine Blazy and Philippe Facon. Formal specification and prototyping of a program specializer. In TAPSOFT'95 [23], pages 666–680.

[4] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[5] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[6] John Hannan and Frank Pfenning. Compiler verification in LF. In *Proceedings of the Seventh Symposium on Logic in Computer Science*, pages 407–418. IEEE, 1992.

[7] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. A preliminary version appeared in the proceedings of the First IEEE Symposium on Logic in Computer Science, pages 194–204, June 1987.

[8] John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Proceedings of the Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, number 982 in Lecture Notes in Computer Science, pages 279–298, Utrecht, The Netherlands, September 1995.

[9] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, ?? (to appear).

[10] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. DIKU Report 95/3, University of Copenhagen, Copenhagen, Denmark, 1995.

[11] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. Report MPI-I-91-211, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, August 1991.

[12] Goplan Nadathur and Dale Miller. An overview of λ-prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, volume 1, pages 810–827, August 1988.

[13] Ulf Nilsson and Jan Maluszyński. *Logic, Programming and Prolog.* John Wiley, 1990.

[14] Henning Niss and John Hatcliff. Formalizing operational semantics in logical frameworks: A case study using LF / Elf (Extended version). DIKU Report ??, University of Copenhagen, Copenhagen, Denmark, 1995. To appear.

[15] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE Computer Society Press, June 1989.

[16] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[17] Frank Pfenning. A proof of the church-rosser theorem and its representation in a logical framework. Technical Report CMU-CS-92-186, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 1992. To appear in Journal of Automated Reasoning.

[18] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, pages 199–208, June 1988.

[19] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th Eleventh International Conference on Automated Deduction*, number 607 in Lecture Notes in Artificial Intelligence, pages 537–551, Saratoga Springs, New York, 1992. Springer-Verlag.

[20] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[21] Dag Prawitz. *Natural Deduction.* Almquist and Wiksell, Uppsala, 1965.

[22] Ekke Rohwedder and Frank Pfenning. Mode and termination analysis for higher-order logic programs. Unpublished manuscript, 1995.

[23] *TAPSOFT '95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, Aarhus, Denmark, May 1995.