Mechanically Verifying the Correctness of an Offline Partial Evaluator

John Hatcliff *

DIKU, Computer Science Department, Copenhagen University **

Abstract. We show that using deductive systems to specify an offline partial evaluator allows one to specify, prototype, and mechanically verify correctness *via* meta-programming — all within a single framework.

For a λ-mix-style partial evaluator, we specify binding-time constraints using a natural-deduction logic, and the associated program specializer using natural (aka "deductive") semantics. These deductive systems can be directly encoded in the Elf programming language — a logic programming language based on the LF logical framework. The specifications are then executable as logic programs. This provides a prototype implementation of the partial evaluator.

Moreover, since deductive system proofs are accessible as objects in Elf, many aspects of the partial evaluator correctness proofs (*e.g.*, the correctness of binding-time analysis) can be coded in Elf and mechanically checked.

1 Introduction

Offline partial evaluation consists of two phases: a binding-time analysis phase (where information is gathered about which parts of the source program depend on known or unknown data), and a specialization phase (where constructs depending on known data are reduced away) [3,15]. Recent work specifies the analysis phase using *type systems* [7] and the specialization phase using *operational semantics* [14,25,26]. The type system and operational semantics formalisms can be unified if one emphasizes their logical character: a type-based analysis is a logic for deducing program properties, and an operational semantics is a logic for deducing computational steps or input/output behaviour of programs. However, in program specialization systems that use these formalisms, this logical character has neither been emphasized nor exploited.

In this paper, we exploit this logical character and obtain a uniform framework for specifying, prototyping, and mechanically verifying the correctness of program specialization systems. Specifically, we consider offline partial evalua-

* 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, 2100 Copenhagen Ø, Denmark. E-mail: hatcliff@diku.dk

tion in the style of the partial evaluator λ-*mix* [7].[3] λ-mix is a good illustrative case since it is simple, and one of the few partial evaluators with a rigorous semantic foundation. It has also spawned additional work on the correctness of binding-time analysis [19,28] and specialization [16].

Our results are as follows.

- We give novel specifications of binding-time constraints and specialization as natural-deduction style logics. These specifications simplify meta-theory activities such as proving the correctness of binding-time analysis and specialization.
- We formalize the specifications using LF — a meta-language (a dependently-typed λ-calculus) for defining logics [12]. In LF, judgements (assertions) are represented as types, and deductions are represented as objects. Determining the validity of a deduction is reduced to checking if the representing object is well-typed. Since LF type-checking is decidable, purported deductions can be checked automatically for validity.
- We obtain prototypes directly from the formal specifications using Elf — a logic programming language based on LF [20]. Elf gives an operational interpretation to LF types by treating them as goals. Thus, the LF specifications of the binding-time analysis and specializer are directly executable in Elf.
- We formalize and mechanically verify much of the meta-theory of offline partial evaluation (*e.g.*, correctness of binding-time analysis and soundness of the specializer) *via* meta-programming in Elf. Correctness conditions are formalized as judgements about "lower-level" deductions describing object-language evaluation and transformation. Proofs of correctness are formalized as deductions of the correctness judgements. Elf type-checking mechanically verifies that these deductions (and hence the correctness proofs) are valid.

This methodology of specification/implementation/verification using LF and Elf has been successfully applied in other problem areas [11,17]. In particular, we build on Hannan and Pfenning's work on compiler verification in Elf [11]. They conjectured that their techniques could also be applied to partial evaluation [11, p. 416]. They also identify the verification of transformations based on flow analyses as a "challenging problem, yet to be addressed" [11, p. 415]. The present work addresses both of these points. We confirm their conjecture that LF and Elf can be used for specification/implementation/verification of partial evaluators. Moreover, we give one instance where transformations based on flow analyses can be verified — namely, the specialization of programs based on binding-time analysis.

The rest of the paper is organized as follows. Section 2 summarizes LF and Elf. Section 3 presents the object language and its encoding in Elf. Section 4 presents the specifications of the binding-time analysis and specializer. Section

[3] Our setting differs from that of λ-mix in two ways: (1) we are not concerned with self-applying the partial evaluator, and (2) our object language is typed while λ-mix's is untyped. However, the techniques here apply equally well to the untyped object language of λ-mix (in fact, they are simpler in the untyped case).

5 illustrates how these specifications are executable in Elf. Section 6 shows how meta-theoretic properties of the specifications can be mechanically verified. Section 7 surveys related work, and Section 8 concludes.

2 LF and the Elf Programming Language

2.1 LF — a framework for defining logics

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

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

Family-level constants are denoted by a, and object-level constants by c. $A_1 \to A_2$ abbreviates $\mathit{\Pi} x{:}A_1.\,A_2$ when x does not appear free in A_2 (similarly for $\mathit{\Pi} x{:}A.\,K$). The typing rules for LF can be found in [12]. We take $\beta\eta$-equivalence as the notion of definitional equality in LF [12, 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. 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

The syntax of Elf is as follows (the last column lists the corresponding LF term, and optional components are enclosed in $\langle\cdot\rangle$).

$\mathit{kindexp} ::=$	$\mathtt{type}$	Type
$\mid$	$\{\mathit{id}\langle{:}\mathit{famexp}\rangle\}\ \mathit{kindexp}$	$\mathit{\Pi} x{:}A.\,K$
$\mid$	$\mathit{famexp}\ \texttt{->}\ \mathit{kindexp}$	$A \to K$
$\mathit{famexp} ::=$	id	a
$\mid$	$\{\mathit{id}\langle{:}\mathit{famexp}_1\rangle\}\ \mathit{famexp}_2$	$\mathit{\Pi} x{:}A_1.\,A_2$
$\mid$	$\texttt{[}\mathit{id}\langle{:}\mathit{famexp}_1\rangle\texttt{]}\ \mathit{famexp}_2$	$\lambda x{:}A_1\,.\,A_2$
$\mid$	$\mathit{famexp}\ \mathit{objexp}$	$A\,M$
$\mid$	$\mathit{famexp}_1\ \texttt{->}\ \mathit{famexp}_2$	$A_1 \to A_2$
$\mid$	$\mathit{famexp}_2\ \texttt{<-}\ \mathit{famexp}_1$	$A_1 \to A_2$
$\mathit{objexp} ::=$	id	c
$\mid$	$\texttt{[}\mathit{id}\langle{:}\mathit{famexp}\rangle\texttt{]}\ \mathit{objexp}$	$\lambda x{:}A\,.\,M$
$\mid$	$\mathit{objexp}_1\ \mathit{objexp}_2$	$M_1\,M_2$

Object language:

$$e \in \mathit{Exp} \qquad\qquad \tau \in \mathit{Typ}$$
$$e ::= \mathsf{0} \mid x \mid \mathsf{lam}\, x \,.\, e \mid \mathsf{app}\, e_0\, e_1 \qquad\qquad \tau ::= \mathsf{nat} \mid \tau_1 \rightarrow \tau_2$$

Elf encoding:

```
exp : type.                        typ   : type.

z   : exp.                         nat   : typ.
lam : (exp -> exp) -> exp.         arrow : typ -> typ -> typ.
app : exp -> exp -> exp.
```

Fig. 1. The object language Λ

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) inserts these abstractions as well as appropriate arguments to these abstractions. It also fills in the omitted types in quantifications $\{x\}$ and abstractions `[`x`]` 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.

3 The Object Language

Figure 1 presents the syntax of the object language Λ (a small subset of PCF)[4] and its encoding in Elf.[5] The Elf signature for the object language syntax includes family constants `exp` and `typ`, and object constants for each term and

[4] In the extended version of this paper [13], we treat full PCF, *i.e.*, simply-typed λ-terms with primitive operations (*e.g.*, succ, pred), conditionals, and fixpoint constructs [8]. In the present version, space constraints force us to consider only those constructs which best illustrate principles. Including the other constructs is a straightforward extension of the work here (the encodings are very similar to those given by Michaylov and Pfenning for standard type systems and deductive semantics [17]). At no point in the presentation do we take advantage of the fact that the subset Λ is strongly normalizing and computationally incomplete.

[5] Technically, one must ensure that such encodings are *adequate*, *i.e.*, that there is a compositional bijection between the syntactic entities in the logical system and well-formed LF $\beta\eta$-normal forms under the given signature. An adequate encoding ensures that each entity is encoded uniquely and that no representations of additional entities are introduced. All the encodings we use are adequate. See Harper *et al.*[12]

type constructor. Binding in $\mathsf{lam}\, x \,.\, e$ is represented using binding in the meta-language (*i.e.*, using higher-order abstract syntax [22]). Variables in the object language are identified with variables in the meta-language, so there is no explicit representation of identifiers in the Elf signature for the object language. For example, the expression

$$\mathsf{app}\,(\mathsf{lam}\, x_1 \,.\, \mathsf{lam}\, x_2 \,.\, \mathsf{app}\,(\mathsf{lam}\, x_3 \,.\, x_3)\; x_1)\; \mathsf{0}$$

is encoded as

```
app (lam [x1] lam [x2] app (lam [x3] x3) x1) z.
```

We omit the standard typing rules and call-by-name operational semantics for Λ[6] — the binding-time rules and operational semantics for the program specializer (given in the following section) generalize these.

4 Specifying an Offline Partial Evaluator for Λ

A partial evaluator takes a *source program* p and a subset s of p's input, and produces a *residual program* p_s which is specialized with respect to s. The correctness of the partial evaluator implies that running p_s on p's remaining input d gives the same result as running p on the complete input s and d. The data s and d are often referred to as *static* and *dynamic* data (respectively) since s is fixed at specialization time whereas one may supply various data d during runs of p_s.

The specialized program p_s is obtained from p by evaluating constructs that depend only on s, while rebuilding constructs that may depend on dynamic data. Offline partial evaluation accomplishes this in two phases: (1) a binding-time analysis phase, and (2) a specialization phase.

1. **Binding-time analysis**: Given assumptions about which program inputs are static and dynamic, binding-time analysis assigns each source program construct a *specialization directive* and a *specialization type*. This information is expressed by constructing an annotated version of the source program.
   - **Specialization directives**: A construct is assigned a directive of *eliminable* if it depends only on static data and thus can be completely evaluated during the specialization phase. A construct is assigned a directive of *residual* if it may depend on dynamic data and thus must be reconstructed in the specialization phase.

for a detailed discussion and proofs of adequacy for encodings similar to the ones used here.

[6] These can be found in the extended version of this paper [13]. Michaylov and Pfenning [17] give Elf encodings of typing rules and call-by-value operational semantics for a language similar to Λ.

Annotated object language:

$w \in \mathit{Sexp}$ $\qquad m \in \mathit{Sder}$

$w ::= \mathsf{0}_m \mid y \mid \mathsf{lam}_m\, y \,.\, w \mid \mathsf{app}_m\, w_0\, w_1 \mid \mathsf{lift}\, w$ $\qquad m ::= s \mid d$

Elf encoding:

```
sexp : type.                                sder : type.

bz   : sder -> sexp.                        s    : sder.
blam : sder -> (sexp -> sexp) -> sexp.      d    : sder.
bapp : sder -> sexp -> sexp -> sexp.
lift : sexp -> sexp.
```

Fig. 2. The annotated object language Λ_{bt}

Specialization types:

$\varphi_{\mathsf{nat}} \in \mathit{Styp}[\mathsf{nat}]$

$\varphi_{\mathsf{nat}} ::= \mathsf{sta} \mid \mathsf{dyn}_{\mathsf{nat}}$

$\varphi_{\tau_1 \to \tau_2} \in \mathit{Styp}[\tau_1 \to \tau_2]$

$\varphi_{\tau_1 \to \tau_2} ::= \varphi_{\tau_1} \to \varphi_{\tau_2} \mid \mathsf{dyn}_{\tau_1 \to \tau_2}$

Elf encoding:

```
styp   : typ -> type.

sta    : styp nat.
dyn    : styp T.
barrow : styp T1 -> styp T2
            -> styp (arrow T1 T2).
```

Fig. 3. The specialization types for Λ_{bt}

   – **Specialization types**: The specialization type assigned to a construct describes the directive structure of terms to which it may reduce during specialization. The specialization types are the carriers of information during the analysis phase.

2. **Specialization**: During the specialization phase, the specializer simply follows the directives assigned during binding-time analysis: eliminable constructs are evaluated (and thus eliminated); residual constructs are reconstructed (and thus appear in the residual program).

4.1 Binding-time analysis

A binding-time analysis associates each Λ term with a term in the annotated language Λ_{bt} of Figure 2. An annotated term is indexed by a specialization directive s or d indicating if it is eliminable (*i.e.*, it depends only on static data) or residual (*i.e.*, it may depend on dynamic data). Identifiers are not indexed since the appropriate information can be determined from the environment. A coercion construct lift is added to Λ_{bt} to residualize the result of evaluating an eliminable term. This allows static computation to occur in a residual context. A term $w \in \Lambda_{bt}$ is *completely residual* if it consists of only d-indexed constructs and identifiers. Intuitively, the specializer will output completely residual terms — all eliminable constructs will have been evaluated. The Elf encoding of Λ_{bt} terms follows that of Λ terms (except that directive indexing is captured by supplying an extra argument of type `sder` to a constructor).

Figure 3 presents a τ-indexed family of specialization types for Λ_{bt}. A specialization type φ is *dynamic* if $\varphi = \mathsf{dyn}$; otherwise φ is *static*. We omit type indices on specialization types when they can be inferred from the context. Specialization types are encoded in Elf *via* the type family `styp:typ->type`. For the `dyn` and `barrow` constructors, the type indices are encoded as logic variables `T`, `T1`, and `T2`, which will be instantiated by unification. This formalizes the above convention of allowing type indices to be inferred from the context.

Figure 4 presents a natural-deduction-style logic for deriving binding-time analysis constraints.[7] Parentheses in the hypotheses of the rules *bta_lam_s* and *bta_lam_d* indicate the discharging of zero or more occurrences of assumptions. We write $\Gamma \vdash bta\ e : \tau\ [w : \varphi_\tau]$ when $bta\ e : \tau\ [w : \varphi_\tau]$ is derivable under undischarged assumptions Γ. Intuitively, if $\Gamma \vdash bta\ e : \tau\ [w : \varphi_\tau]$, then given initial binding-time assumptions Γ, a binding-time analysis may map $e \in Terms[\Lambda]$ of type $\tau \in Types[\Lambda]$ to a directive annotated term $w \in Terms[\Lambda_{bt}]$ of specialization type $\varphi_\tau \in Spec\text{-}types[\tau]$. We only consider assumptions involving identifiers (*e.g.*, $bta\ x : \tau\ [y : \varphi_\tau]$) since the logic only allows discharging of assumptions of this form. For $\Gamma = \{bta\ x_1 : \tau_1\ [y_1 : \varphi_{\tau_1}], ..., bta\ x_n : \tau_n\ [y_n : \varphi_{\tau_n}]\}$, the x_i are required to be pairwise distinct (similarly for the y_i). A *static assumption* (resp. *dynamic assumption*) is an assumption $bta\ x : \tau\ [y : \varphi_\tau]$ where φ_τ is static (resp. dynamic). Γ_s denotes a set of only static assumptions; Γ_d denotes a set of only dynamic assumptions.

A simple induction over the structure of deductions for $\Gamma \vdash bta\ e : \tau\ [w : \varphi_\tau]$ shows that the relation between e and w is one-to-many, *i.e.*, there may be many valid annotations of e. One may always obtain a valid annotation of a type correct e by annotating all components as residual. Since the relation is one-to-many, it defines an *annotation forgetting function* from Λ_{bt} to Λ. Intuitively, this function removes directives and lift constructs.

The binding-time judgement is encoded in Elf as follows.

```
bta : exp -> {t : typ} sexp -> styp t -> type.
```

The dependent function type (*i.e.*, `{t : typ}` ...) expresses the dependency of the indexed specialization type on the type of the Λ expression.

Figure 5 presents the Elf encoding of the binding-time logic. The implicit universal quantification of the upper case variables captures the schematic nature of the rules. The rules for binding constructs (which involve hypothetical proofs in the premises) are encoded in Elf as proof constructors that take proofs of hypothetical judgements as arguments. Such proofs are represented as functions mapping proofs of assumption judgments to proofs of consequent

[7] We use the notation of Prawitz [24].

$$\textit{bta\_z\_s}: \quad \textit{bta}\ \mathsf{0} : \mathsf{nat}\ [\mathsf{0}_s : \mathsf{sta}]$$

$$\textit{bta\_lam\_s}: \quad \frac{\begin{array}{c}(\textit{bta}\ x : \tau_1\ [y : \varphi_1])\\ \textit{bta}\ e : \tau_2\ [w : \varphi_2]\end{array}}{\textit{bta}\ \mathsf{lam}\, x\, .\, e : \tau_1 \to \tau_2\ [\mathsf{lam}_s\, y\, .\, w : \varphi_1 \to \varphi_2]}$$

$$\textit{bta\_app\_s}: \quad \frac{\textit{bta}\ e_0 : \tau_1 \to \tau_2\ [w_0 : \varphi_1 \to \varphi_2] \qquad \textit{bta}\ e_1 : \tau_1\ [w_1 : \varphi_1]}{\textit{bta}\ \mathsf{app}\, e_0\, e_1 : \tau_2\ [\mathsf{app}_s\, w_0\, w_1 : \varphi_2]}$$

$$\textit{bta\_z\_d}: \quad \textit{bta}\ \mathsf{0} : \mathsf{nat}\ [\mathsf{0}_d : \mathsf{dyn}]$$

$$\textit{bta\_lam\_d}: \quad \frac{\begin{array}{c}(\textit{bta}\ x : \tau_1\ [y : \mathsf{dyn}])\\ \textit{bta}\ e : \tau_2\ [w : \mathsf{dyn}]\end{array}}{\textit{bta}\ \mathsf{lam}\, x\, .\, e : \tau_1 \to \tau_2\ [\mathsf{lam}_d\, y\, .\, w : \mathsf{dyn}]}$$

$$\textit{bta\_app\_d}: \quad \frac{\textit{bta}\ e_0 : \tau_1 \to \tau_2\ [w_0 : \mathsf{dyn}] \qquad \textit{bta}\ e_1 : \tau_1\ [w_1 : \mathsf{dyn}]}{\textit{bta}\ \mathsf{app}\, e_0\, e_1 : \tau_2\ [\mathsf{app}_d\, w_0\, w_1 : \mathsf{dyn}]}$$

$$\textit{bta\_lift}: \quad \frac{\textit{bta}\ e : \mathsf{nat}\ [w : \mathsf{sta}]}{\textit{bta}\ e : \mathsf{nat}\ [\mathsf{lift}\ w : \mathsf{dyn}]}$$

Fig. 4. The binding-time logic

```
bta_z_s  : bta z nat (bz s) sta.

bta_lam_s  : bta (lam E) (arrow T1 T2) (blam s W) (barrow P1 P2)
           <- {x : exp} {y : sexp} (bta x T1 y P1) -> (bta (E x) T2 (W y) P2).

bta_app_s  : bta (app E0 E1) T2 (bapp s W0 W1) P2
           <- bta E0 (arrow T1 T2) W0 (barrow P1 P2)
           <- bta E1 T1 W1 P1.

bta_z_d  : bta z nat (bz d) dyn.

bta_lam_d  : bta (lam E) (arrow T1 T2) (blam d W) dyn
           <- {x : exp} {y : sexp} (bta x T1 y dyn) -> (bta (E x) T2 (W y) dyn).

bta_app_d  : bta (app E0 E1) T2 (bapp d W0 W1) dyn
           <- bta E0 (arrow T1 T2) W0 dyn
           <- bta E1 T1 W1 dyn.

bta_lift  : bta E nat (lift W) dyn
          <- bta E nat W sta.
```

Fig. 5. The Elf encoding of the binding-time logic

$$\mathsf{0}_s \Downarrow_{spec} \mathsf{0}_s$$

```
spec_z_s  : spec (bz s) (bz s).
```

$$\mathsf{lam}_s\, y\,.\,w \Downarrow_{spec} \mathsf{lam}_s\, y\,.\,w$$

```
spec_lam_s  : spec (blam s W)
                   (blam s W).
```

$$\frac{w_0 \Downarrow_{spec} \mathsf{lam}_s\, y\,.\,w_0' \qquad w_0'[y := w_1] \Downarrow_{spec} a}{\mathsf{app}_s\, w_0\, w_1 \Downarrow_{spec} a}$$

```
spec_app_s  : spec (bapp s W0 W1) A
               <- spec W0 (blam s W0')
               <- spec (W0' W1) A.
```

$$\mathsf{0}_d \Downarrow_{spec} \mathsf{0}_d$$

```
spec_z_d  : spec (bz d) (bz d).
```

$$\frac{\begin{array}{c}(y \Downarrow_{spec} y)\\ w \Downarrow_{spec} a\end{array}}{\mathsf{lam}_d\, y\,.\,w \Downarrow_{spec} \mathsf{lam}_d\, y\,.\,a}$$

```
spec_lam_d  : spec (blam d W) (blam d A)
               <- {y : sexp}
                  (spec y y ->
                   spec (W y) (A y)).
```

$$\frac{w_0 \Downarrow_{spec} a_0 \qquad w_1 \Downarrow_{spec} a_1}{\mathsf{app}_d\, w_0\, w_1 \Downarrow_{spec} \mathsf{app}_d\, a_0\, a_1}$$

```
spec_app_d  : spec (bapp d W0 W1)
                   (bapp d A0 A1)
               <- spec W0 A0
               <- spec W1 A1.
```

$$\frac{w \Downarrow_{spec} \mathsf{0}_s}{\mathsf{lift}\, w \Downarrow_{spec} \mathsf{0}_d}$$

```
spec_lift  : spec (lift W) (bz d)
              <- spec W (bz s).
```

Fig. 6. The specialization logic

judgements. The higher-order syntax representations requires that judgements involving identifiers be represented as schematic judgements (*i.e.*, identifiers are Π-quantified). This is the case in rules `bta_lam_s` and `bta_lam_d` where *e.g.*, the judgement `bta (E x) T2 (W y) P2` expresses that `E` and `W` may be instantiated to representations of terms with free occurrences of x and y respectively [12, Section 3.1].

4.2 Specialization

Figure 6 presents the specialization logic for Λ_{bt} terms.

– The first four rules describe the evaluation of eliminable constructs. These rules correspond to the usual call-by-name "natural" or "deductive" operational semantics for PCF [8, Chapter 4].
– The next four rules describe the reconstruction of residual constructs after subexpressions have been specialized. The rule for lam_d is noteworthy because it introduces operation on open terms (*i.e.*, bodies of lam_d constructs). Following Hannan [9, p. 144], we specialize the body of a lam_d under the assumption that the bound variable evaluates to itself.

– The final rule coerces an eliminable result to a residual expression.

We write $\Sigma \vdash w \Downarrow_{spec} a$ when $w \Downarrow_{spec} a$ is derivable under undischarged assumptions Σ. Intuitively, if $\Sigma \vdash w \Downarrow_{spec} a$, then the specializer maps $w \in \mathit{Terms}[\Lambda_{bt}]$ to answer $a \in \mathit{Terms}[\Lambda_{bt}]$ in context Σ. We only consider assumptions involving identifiers (*e.g.*, $y \Downarrow_{spec} y$) since the logic only allows discharging of assumptions of this form. For $\Sigma = \{y_1 \Downarrow_{spec} y_1, ..., y_n \Downarrow_{spec} y_n\}$ the y_i are required to be pairwise distinct. *Σ is compatible with $\Gamma = \{bta\ x_1 : \tau_1\ [y_1 : \varphi_{\tau_1}], ..., bta\ x_n : \tau_n\ [y_n : \varphi_{\tau_n}]\}$* if $\Sigma = \{y_1 \Downarrow_{spec} y_1, ..., y_n \Downarrow_{spec} y_n\}$. It is easy to check that the relation induced by $\Downarrow_{spec}$ is a partial function (given the constraints on assumptions above).

Figure 6 also gives the Elf encoding of the specialization logic. The encoding techniques are similar to those used in the previous section. In the rule `spec_app_s`, we take advantage of the higher-order abstract syntax representation and use Elf application (*i.e.*, β-reduction) to implement capture-free substitution.

In a conventional deductive semantics for Λ, one has $\mathsf{0}$ and $\mathsf{lam}\, x\,.\, e$ as canonical terms, *i.e.*, these terms are the results of evaluation. Intuitively, a specializer is part evaluator and part compiler. Therefore, the canonical terms of the specializer are $\mathsf{0}_s$ and $\mathsf{lam}_s\, y\,.\, w$ (corresponding to evaluation results), and completely residual terms or *code* (corresponding to compilation results). We will give a mechanically verified proof of this claim in Section 6. In preparation, we formalize the notion of canonical term or *answer* by defining a judgement $ans\ w : \varphi_\tau$. Intuitively, if $ans\ w : \varphi_\tau$ holds, then w is a canonical term of specialization type φ_τ. In particular, if $\vdash ans\ w : \mathsf{dyn}$, then w is completely residual. We omit the direct statement of rules and simply give the following Elf encoding using the type `ans : sexp -> styp T -> type.`

```
                                           ans_lam_d: ans (blam d W) dyn
ans_z_s: ans (bz s) sta.                              <- {y} ans y dyn ->
                                                          ans (W y) dyn.
ans_lam_s: ans (blam s W) (barrow P1 P2).
                                           ans_app_d: ans (bapp d W1 W2) dyn
ans_z_d: ans (bz d) dyn.                              <- ans W1 dyn
                                                      <- ans W2 dyn.
```

We write $A \vdash ans\ w : \varphi_\tau$ when $ans\ w : \varphi_\tau$ is derivable under undischarged assumptions A.

A is compatible with $\Gamma = \{bta\ x_1 : \tau_1\ [y_1 : \varphi_{\tau_1}], ..., bta\ x_n : \tau_n\ [y_n : \varphi_{\tau_n}]\}$ if $A = \{ans\ y_1 : \varphi_{\tau_1}, ..., ans\ y_n : \varphi_{\tau_n}\}$.

4.3 Partial evaluation

We outline how the logics above define offline partial evaluation using the following object term.

$$e \overset{\text{def}}{=} \mathsf{app}\,(\mathsf{lam}\, x_1\,.\, \mathsf{app}\, x_2\, (\mathsf{app}\,(\mathsf{lam}\, x_3\,.\, x_3)\, x_1))\, x_0$$

The free variables x_0, x_2 represent input parameters. The assumptions $\mathit{\Gamma}_s = \{\mathit{bta}\ x_0 : \mathsf{nat}\,[y_0 : \mathsf{sta}]\}$ and $\mathit{\Gamma}_d = \{\mathit{bta}\ x_2 : \mathsf{nat} \rightarrow \mathsf{nat}\,[y_2 : \mathsf{dyn}]\}$ identify x_0 of type nat as *known* and x_2 of type $\mathsf{nat} \rightarrow \mathsf{nat}$ as *unknown*. They also associate x_0 and x_2 with annotated language identifiers y_0 and y_2. The fact that $\mathit{\Gamma}_s \cup \mathit{\Gamma}_d \vdash \mathit{bta}\ e : \mathsf{nat}\,[w : \mathsf{dyn}]$ where

$$w \stackrel{\text{def}}{=} \mathsf{app}_s\,(\mathsf{lam}_s\,y_1\,.\,\mathsf{app}_d\,y_2\,(\mathsf{lift}\,(\mathsf{app}_s\,(\mathsf{lam}_s\,y_3\,.\,y_3)\,y_1)))\,y_0$$

expresses that a binding-time analysis may associate e with w based on assumptions $\mathit{\Gamma}_s \cup \mathit{\Gamma}_d$.

To prepare for specialization, we supply known input *via* substitution.

$$\mathit{\Gamma}_d \vdash \mathit{bta}\ e[x_0 := \mathsf{0}] : \mathsf{nat}\,[w[y_0 := \mathsf{0}_s] : \mathsf{dyn}]$$

Now taking $\mathit{\Sigma} = \{y_2 \Downarrow_{spec} y_2\}$ compatible with $\mathit{\Gamma}_d$ (expressing that the dynamic parameter evaluates to itself), the specialization logic gives

$$\mathit{\Sigma} \vdash w[y_0 := \mathsf{0}_s] \Downarrow_{spec} \mathsf{app}_d\,y_2\,\mathsf{0}_d.$$

Theorem 1 (correctness of binding-time analyis) of Section 6.1 tells us there exists $e' \in \mathit{\Lambda}$ such that $\mathit{\Gamma}_d \vdash \mathit{bta}\ e' : \mathsf{nat}\,[\mathsf{app}_d\,y_2\,\mathsf{0}_d : \mathsf{dyn}]$. As noted in Section 4.1, e' must be $\mathsf{app}\,x_2\,\mathsf{0}$ — the unannotated version of $\mathsf{app}_d\,y_2\,\mathsf{0}_d$. Theorem 2 (soundness of specialization) of Section 6.2 tells us that $e[x_0 := \mathsf{0}]$ is convertible to e' (denoted $e[x_0 := \mathsf{0}] =_{\mathit{\Lambda}} e'$) in the program calculus for $\mathit{\Lambda}$ (defined in Section 6.2). Thus, for all closed inputs $d \in \mathit{\Lambda}$ of type $\mathsf{nat} \rightarrow \mathsf{nat}$,

$$e[x_0 := \mathsf{0}\,,\, x_2 := d] =_{\mathit{\Lambda}} e'[x_2 := d].$$

This reflects the correctness criteria for partial evaluation given at the beginning of Section 4: running e on static input $\mathsf{0}$ and dynamic input d is operationally equivalent to running e' on d.

5 Prototyping an Offline Partial Evaluator for $\mathit{\Lambda}$

5.1 Prototyping a binding-time analysis

The Elf encoding of the binding-time logic (see Figure 5) gives a prototype of the binding-time analysis. The following Elf query returns all the possible annotations that may be assigned to the example term of Section 3 (here we make dyn the specialization type of the entire term).[8]

```
?- bta (app (lam [x1] lam [x2] app (lam [x3] x3) x1) z)
       (arrow nat nat) W dyn.
Solving...

W = bapp s
      (blam s ([y1:sexp]
```

[8] Some of the results of Elf evaluation are α-converted for clarity.

```
                blam d ([y2:sexp]
                        bapp s (blam s ([y3:sexp] y3)) (lift y1))))
      (bz s).

W = bapp s
      (blam s ([y1:sexp]
              blam d ([y2:sexp]
                      lift (bapp s (blam s ([y3:sexp] y3)) y1))))
      (bz s).
```

There are actually twelve correct annotations; we show only the two above.

5.2 Prototyping a specializer

The Elf encoding of the specialization logic (see Figure 6) gives a prototype of the specializer. The following Elf query returns the result `A` of specializing the second annotated term above (there is only one answer since `spec` is a function).

```
?- S : spec (bapp s
              (blam s ([y1:sexp]
                      blam d ([y2:sexp]
                             lift (bapp s (blam s ([y3:sexp] y3))
                                          y1))))
              (bz s)) A.
Solving...

A = blam d ([y2:sexp] bz d),
S = spec_app_s
        (spec_lam_d [y2:sexp] [S:spec y2 y2]
                   spec_lift (spec_app_s spec_z_s spec_lam_s))
         spec_lam_s.
;
no more solutions
```

In this query, we add the variable `S` which binds to the specialization logic deduction used to obtain `A`.[9]

6 Verifying an Offline Partial Evaluator for Λ

6.1 Binding-time analysis

A binding-time analysis is correct if it always produces consistent specialization directives. Directives are consistent if the specializer does not "go wrong". A

[9] Note that the order of arguments to the deduction constructors is the reverse of what one might expect since we use `<-` (instead of `->`) in the encodings of the binding-time and specialization logics.

specializer may go wrong for two reasons: 1) it trusts a part of the program to be eliminable when in fact it is residual, and 2) it trusts a part of the program to be residual when in fact it is eliminable.

The specializer goes wrong for the first reason on $\mathsf{app}_s\,(\mathsf{lam}_d\,x\,.\,x)\,\mathsf{0}_s$. In this case, it attempts evaluation using the rule *spec_app_s* (see Figure 6) but "hangs" (*i.e.*, the result of specialization is undefined) since $\mathsf{lam}_d\,x\,.\,x$ is not an eliminable abstraction. Similar problems may occur with the rule *spec_lift* (*e.g.*, if $w \Downarrow_{spec} \mathsf{0}_d$).

The specializer goes wrong for the second reason on $\mathsf{app}_d\,(\mathsf{lam}_s\,x\,.\,x)\,\mathsf{0}_d$. In this case, it attempts residualization using the rule *spec_app_d* and incorrectly produces an output program that is not completely residual (since $\mathsf{lam}_s\,x\,.\,x$ is eliminable).

The property that the specializer never goes wrong (*i.e.*, the binding-time analysis always yields consistent directives) is a generalization of the type-soundness property for standard type systems. To establish type-soundness, one typically proves a *subject-reduction* result showing that typing is maintained under evaluation. To establish consistency of directives, we prove that specialization typing is maintained under specialization. Furthermore, we show that specialization results are always answers of the appropriate specialization type, *i.e.*, that the *ans* is always satisfied (as promised in Section 4.2).

This ensures that the specializer will not go wrong for the first reason. For example, in the rule *spec_app_s*, the rule *bta_app_s* guarantees that w_0 always has specialization type $\varphi_1 \rightarrow \varphi_2$ and if $w_0 \Downarrow_{spec} a_0$ then $a_0 \equiv \mathsf{lam}_s\,x\,.\,w_0'$ since only appropriate answers are produced.

This also ensures that the specializer will not go wrong for the second reason. For example, in the rule *spec_app_d*, the rule *bta_app_d* guarantees that w_0 always has specialization type dyn and if $w_0 \Downarrow_{spec} a_0$ then a_0 must be completely residual (similarly for a_1).

Theorem 1 Correctness of binding-time analysis. *If $\Sigma \vdash w \Downarrow_{spec} a$ and $\Gamma_d \vdash bta\ e : \tau\ [w : \varphi_\tau]$ and Σ is compatible with Γ_d, then there exists $e' \in \Lambda$ such that $\Gamma_d \vdash bta\ e' : \tau\ [a : \varphi_\tau]$ and $A \vdash ans\ a : \varphi_\tau$ where A is compatible with Γ_d.*

Proof. For notational convenience, we write $\mathcal{D} :: \mathcal{J}$ when $\mathcal{D}$ is a deduction of judgement $\mathcal{J}$ and state deduction rules in a linear format.[10]

For the theorem hypotheses, let $\mathcal{S} :: w \Downarrow_{spec} a$ and $\mathcal{B} :: bta\ e : \tau\ [w : \varphi_\tau]$. We show an effective method for constructing deductions $\mathcal{C} :: bta\ e' : \tau\ [a : \varphi_\tau]$ and $\mathcal{D} :: ans\ a : \varphi_\tau$ where the compatibility constraints on undischarged assumptions are satisfied. The proof proceeds by induction on the pair of deductions $\mathcal{S}$ and $\mathcal{B}$, *i.e.*, the method is primitive recursive. Although this primitive recursive method

[10] For example, a specialization deduction that ends with the rule *spec_app_d* is written *spec_app_d*$(\mathcal{S}_0\,,\,\mathcal{S}_1) :: \mathsf{app}_d\,w_0\,w_1 \Downarrow_{spec} \mathsf{app}_d\,a_0\,a_1$ where $\mathcal{S}_i :: w_i \Downarrow_{spec} a_i$ $(i = 1, 2)$. The notation is somewhat imprecise since we do not use an explicit discharge function for assumptions [12, Section 4.1]. However, the Elf encoding makes matters sufficiently clear.

cannot be represented in Elf as a function (since it is not schematic), it can be represented as a relation *via* the following judgement.

```
t1 : spec W A -> bta E T W P -> bta E' T A P -> ans A P -> type.
```

Each case of the constructive proof is formalized as a rule for the `t1` judgement. Below we show three illustrative cases (each increasing in complexity).

case $\mathcal{S} = \mathit{spec\_z\_s} :: 0_s \Downarrow_{spec} 0_s$ and $\mathcal{B} = \mathit{bta\_z\_s} :: \mathit{bta}\ 0 : \mathsf{nat}\ [0_s : \mathsf{sta}]$:

The required deductions are $\mathit{bta\_z\_s} :: \mathit{bta}\ 0 : \mathsf{nat}\ [0_s : \mathsf{sta}]$ and $\mathit{ans\_z\_s} :: \mathit{ans}\ 0_s : \mathsf{sta}$. This is formalized by the following axiom.

```
t1_z_s : t1 (spec_z_s) (bta_z_s) (bta_z_s) (ans_z_s).
```

case $\mathcal{S} = \mathit{spec\_app\_d}(\mathcal{S}_0\,,\,\mathcal{S}_1) :: \mathsf{app}_d\, w_0\, w_1 \Downarrow_{spec} \mathsf{app}_d\, a_0\, a_1$
$\mathcal{B} = \mathit{bta\_app\_d}(\mathcal{B}_0\,,\,\mathcal{B}_1) :: \mathit{bta}\ \mathsf{app}\, e_0\, e_1 : \tau_2\ [\mathsf{app}_d\, w_0\, w_1 : \mathsf{dyn}]$:

Applying the inductive hypothesis to $\mathcal{S}_0$ and $\mathcal{B}_0$ gives deductions $\mathcal{C}_0 :: \mathit{bta}\ e_0' : \tau_1 \to \tau_2\ [a_0 : \mathsf{dyn}]$ and $\mathcal{D}_0 :: \mathit{ans}\ a_0 : \mathsf{dyn}$. Applying the inductive hypothesis to $\mathcal{S}_1$ and $\mathcal{B}_1$ gives deductions $\mathcal{C}_1 :: \mathit{bta}\ e_1' : \tau_1 \to \tau_2\ [a_1 : \mathsf{dyn}]$ and $\mathcal{D}_1 :: \mathit{ans}\ a_1 : \mathsf{dyn}$. The required deductions are $\mathit{bta\_app\_d}(\mathcal{C}_0\,,\,\mathcal{C}_1)$ and $\mathit{ans\_app\_d}(\mathcal{D}_0\,,\,\mathcal{D}_1)$. This is formalized by the following rule (the arguments to the proof constructors appear in reverse order (*e.g.*, `spec_app_d S1 S0` instead of `spec_app_d S0 S1`) due to the use of `<-` (instead of `->`) in the encodings of binding-time and specialization logic).

```
t1_app_d : t1 (spec_app_d S1 S0) (bta_app_d B1 B0)
              (bta_app_d C1 C0) (ans_app_d D1 D0)
                <- t1 S0 B0 C0 D0
                <- t1 S1 B1 C1 D1.
```

In operational terms, the inductive hypotheses are manifested as recursive calls to the function represented by `t1`.

case $\mathcal{S} = \mathit{spec\_app\_s}(\mathcal{S}_0\,,\,\mathcal{S}_1) :: \mathsf{app}_s\, w_0\, w_1 \Downarrow_{spec} a$
$\mathcal{B} = \mathit{bta\_app\_s}(\mathcal{B}_0\,,\,\mathcal{B}_1) :: \mathit{bta}\ \mathsf{app}\, e_0\, e_1 : \tau_2\ [\mathsf{app}_s\, w_0\, w_1 : \varphi_2]$:

Applying the inductive hypothesis to $\mathcal{S}_0$ and $\mathcal{B}_0$ gives deductions $\mathcal{C}_0' :: \mathit{bta}\ e_0' : \tau_1 \to \tau_2\ [a_0 : \varphi_1 \to \varphi_2]$ and $\mathcal{D}_0 :: \mathit{ans}\ a_0 : \varphi_1 \to \varphi_2$. An examination of the rules of Figure 4 shows that we must have $\mathcal{C}_0' = \mathit{bta\_lam\_s}(\mathcal{C}_0 :: \mathit{bta}\ e_0'' : \tau_2\ [w_0' : \varphi_2]) :: \mathit{bta}\ \mathsf{lam}\, x\,.\, e'' : \tau_1 \to \tau_2\ [\mathsf{lam}_s\, y\,.\, w_0' : \varphi_1 \to \varphi_2]$ where the undischarged assumptions of $\mathcal{C}_0$ include $\mathit{bta}\ x : \tau_1\ [y : \varphi_1]$. A simple substitution lemma (which we omit for lack of space) allows us to replace each of these assumptions in $\mathcal{C}_0$ with the deduction $\mathcal{B}_1 :: \mathit{bta}\ e_1 : \tau_1\ [w_1 : \varphi_1]$ to obtain a deduction $\mathcal{B}_2 :: \mathit{bta}\ e_0''[x := e_1] : \tau_2\ [w_0'[y := w_1] : \varphi_2]$. Applying the inductive hypothesis to $\mathcal{S}_1$ and $\mathcal{B}_2$ gives the required deductions $\mathcal{C} :: \mathit{bta}\ e' : \tau_2\ [a : \varphi_2]$ and $\mathcal{D} :: \mathit{ans}\ a : \varphi_2$. This is formalized as follows.

```
t1_app_s : t1 (spec_app_s S1 S0) (bta_app_s B1 B0) C D
                <- t1 S0 B0 (bta_lam_s C0) D0
                <- t1 S1 (C0 _ _ B1) C D.
```

The "examination of the rules" (that told us deduction $\mathcal{C}_0'$ must have *bta_lam_s* as its last rule) is captured by matching (*i.e.*, `(bta_lam_s C0)`). The required substitution lemma appears for free due to the higher-order abstract syntax representation and the *Transitivity* derived rule of the LF calculus [12, Section 2.3]; it is captured by `(C0 _ _ B1)`. The underscores correspond to the terms e_1 and w_1 above. Using the underscores and letting Elf reconstruct allows us to encode the rule more concisely.

The complete set of rules is given in Appendix A.

The following Elf query illustrates the function induced by the `t1` rules. Given the specialization deduction `S` of Section 5.2 and a binding-time deduction for the term being specialized (*i.e.*, the second annotated term of Section 5.1), `t1` constructs a binding-time deduction `C` for the specialization answer as well as an answer deduction `D` proving that the answer is completely residual.

```
?- t1 (spec_app_s
         (spec_lam_d [y2:sexp] [S:spec y2 y2]
                     spec_lift (spec_app_s spec_z_s spec_lam_s))
         spec_lam_s)
      (bta_app_s bta_z_s
                (bta_lam_s [x1:exp] [y1:sexp]
                    [B1:bta x1 nat y1 sta]
                    bta_lam_d [x2:exp] [y2:sexp]
                       [B2:bta x2 nat y2 dyn]
                       bta_lift
                          (bta_app_s B1
                              (bta_lam_s [x3:exp] [y3:sexp]
                                 [B3:bta x3 nat y3 sta] B3))))
      C D.
Solving...

D = ans_lam_d [y2:sexp] [D2:ans y2 dyn] ans_z_d,
C = bta_lam_d [x2:exp] [y2:sexp] [B2:bta x2 nat y2 dyn] bta_z_d.
```

Elf type-checking mechanically verifies that deductions `C` and `D` are well-formed when they exist. However, verification that such deductions *always* exist (*i.e.*, that `t1` is *total*) cannot be captured in Elf. This phase of verification (called *schema checking*) must be done by hand, although its automation is the subject of current research [23]. Our definition makes `t1` total because we give a rule for each possible pair of *spec* and *bta* rules (giving primitive recursive structure). In addition, the deduction of answer judgements tell us that matching is used (in rules `t1_app_s` and `t1_lift`) only when it will always succeed.

6.2 Soundness of specialization

A specializer is sound if its steps reflect a meaning preserving transformation on the unannotated object program. We prove specializer soundness by appealing to a *program calculus for Λ* (*i.e.*, an equational theory for deducing operational equivalences). Gunter [8, Chapter 4] gives a calculus for PCF, and we may take an appropriate sub-theory of this as the calculus for Λ (this is essentially the traditional $\lambda\beta$-calculus). Let $=_\Lambda$ denote the convertibility relation for the Λ calculus. The following theorem captures the fact that the unannotated object program e is operationally equivalent to the unannotated specialization result e'.

Theorem 2 Soundness of specializer.
If $\Sigma \vdash w \Downarrow_{spec} a$ and $\Gamma_d \vdash bta\ e : \tau\ [w : \varphi_\tau]$ and Σ is compatible with Γ_d, then there exists $e' \in \Lambda$ such that $\Gamma_d \vdash bta\ e' : \tau\ [a : \varphi_\tau]$ and $e =_\Lambda e'$.

Proof. (summary) To formalize the proof, we give an encoding of the Λ-calculus into Elf based on a similar encoding given by Pfenning [21]. Next, we construct a function (*via* a relation as in Theorem 1) which constructs a deduction showing e converts to e'.

```
t2 : spec W A -> bta E T W P -> bta E' T A P -> conv E E' -> type.
```

A strategy similar to the one used in the proof of Theorem 1 gives the rules defining `t2`. The details are given in [13].

7 Related work

Despeyroux first emphasized using deductive systems to define transformations [5]. She specified a compiler, and source and target language semantics using deductive systems. The specifications were executed *via* encodings into Typol. Informal proofs of correctness were given as relations between deductions. However, these proofs could not be formalized in Typol because it does not support the direct manipulation of its own deductions.

Hannan and Pfenning [11] improved upon this by formalizing similar proofs of correctness in Elf. Elf (unlike *e.g.*, Typol, and λ-Prolog) supports the direct manipulation of its own deductions. Furthermore, Elf type checking mechanically verifies that deductions corresponding to correctness proofs are well-formed.

Despeyroux [5, Section 8], and Hannan and Pfenning [11, Section 7], suggested that their methods could be used to specify "mixed computation" and partial evaluation, respectively. Hannan and Miller [10] carried out Despeyroux's suggestion; they use deductive systems encoded in λ-Prolog to obtain executable specifications of mixed computation (in their work, "mixed computation" = non-deterministic on-line partial evaluation).

Our contributions include using deductive systems to specify *program specialization directed by type-based analysis.* Moreover, we adapt the techniques of Hannan and Pfenning [11], and Michaylov and Pfenning [17] to mechanically verify correctness.

In doing so, we obtain verified proofs similar in scope to the ones given by Gomard and Jones [7] for λ-mix. Their denotational meta-language is significantly more complex than our logic-based meta-language and it is unlikely that the proofs there could be formalized or mechanically verified to the extent that we have done here. However, it must be noted that one of their goals was self-application. Relying on the similarity between their meta-language and object-language, they obtain an object-language specification of the partial evaluator (giving self-applicability) by a fairly easy (though informal and unverified) translation from the meta-language specification. In our setting this is more difficult since the character of our meta-language (logical) is quite different from our object language (functional).

Building upon the work of Gomard, Jones, and Mogensen, Palsberg [19] and Wand [28] give detailed presentations of the correctness of binding-time analysis. Palsberg presents a generalization of Gomard and Jones criteria [7] for consistent binding-time annotations. Wand studies Mogensen's self-applicable partial evaluator [18] for the pure λ-calculus. His binding-time analysis is essentially the same as Gomard and Jones's as well the one presented here. Wand's goals with respect to correctness of the analysis and specializer are more ambitious than those here, because his correctness criteria is strong enough to specify the behaviour of the partial evaluator when self-applied. It would be interesting to see to what extent the meta-theory used by Palsberg and Wand could be formalized in Elf.

In recent work, Davies and Pfenning [4] give a type system for expressing staged computation based on the intuitionistic modal logic S4. They have implemented the type system and a portion of the associated correctness proofs in Elf.

Our work focuses on partial evaluation of functional programs, but similar techniques can be applied to imperative languages as well. Blazy and Facon [2] specify a simple specializer for FORTRAN using natural semantics, and derive a prototype from the specification using the Centaur programming environment. However, their correctness proofs are not formalized. Bertot and Fraer [1] show how similar correctness proofs (for a specializer for an imperative language) can be formalized and mechanically checked using Coq.

8 Conclusion

We have specified the main components of an offline partial evaluator (*i.e.*, binding-time constraints and a program specializer) using natural-deduction style logics. These specifications were formalized by encoding them in LF. Prototypes were obtained directly from the formal specifications using Elf. We formalized and mechanically verified a significant portion of the meta-theory of offline partial evaluation (*e.g.*, correctness of binding-time analysis and soundness of the specializer) *via* meta-programming in Elf. A certain degree of synergy is obtained by using the declarative formalism of deductive systems: one may specify, prototype and mechanically verify correctness *via* meta-programming — all within a single framework.

These techniques can be used to verify other forms of partial evaluation for functional programs. For example, if one takes the view that an annotated program is its own generating extension, the techniques here are very close to what one would use to verify the correctness of a *hand-written cogen* [15]. In fact, in a preliminary investigation we have prototyped a higher-order version of the *multi-level cogen* of Glück and Jørgensen [6]. It remains to be seen if these techniques scale up to *(a)* type-based analyses that include conjunctive types, polymorphism, and more general forms of subtyping, and *(b)* more robust forms of partial evaluation that include sophisticated folding strategies (one approach might be to use Sand's calculus for sound fold/unfold transformations [25]).

Acknowledgements

Thanks to Olivier Danvy for his encouragement and comments on various drafts, and to Frank Pfenning for several stimulating conversations and for providing easy access to his materials on Elf. Robert Glück, Kristian Nielsen, David Sands, and other members of the DIKU TOPPS group gave valuable feedback and support. Thanks to the PLILP referees for helpful comments.

References

1. Yves Bertot and Ranan Fraer. Reasoning with executable specifications. In TAPSOFT'95 [27], pages 531–545.
2. Sandrine Blazy and Philippe Facon. Formal specification and prototyping of a program specializer. In TAPSOFT'95 [27], pages 666–680.
3. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
4. Rowan Davies and Frank Pfenning. A modal analysis of staged compuation. In *Proceedings of the Workshop on Types for Program Analysis*, Aarhus, Denmark, 1995.
5. Jöelle Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*, pages 193–205, Cambridge, Massachusetts, 1986. IEEE Computer Society Press.
6. Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions. 1995. To appear in *The Proceedings of the Seventh International Symposium on Programming Languages, Implementations, Logics and Programs.* Utrecht, The Netherlands, September 20-22, 1995.
7. Carsten K. Gomard and Neil Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.
8. Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.
9. John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, 1993.
10. John Hannan and Dale Miller. Deriving mixed evaluation from standard evaluation for a simple functional language. In J. van de Snepscheut, editor, *Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science, pages 239–255, 1989.

11. 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.
12. 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.
13. John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator (extended version). DIKU Report 95/14, University of Copenhagen, Copenhagen, Denmark, 1995.
14. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. DIKU Report 95/15, University of Copenhagen, Copenhagen, Denmark, 1995. Presented at the *Workshop on Logic, Domains, and Programming Languages.* Darmstadt, Germany. May, 1995.
15. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.
16. Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. Technical Report CS-95-178, Computer Science Department, Brandeis University, Waltham, Massachusetts, January 1995. An earlier version appeared in the proceedings of the 1994 ACM Conference on Lisp and Functional Programming.
17. 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.
18. T. Mogensen. Self-applicable partial evaluation for the pure lambda calculus. In Charles Consel, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Research Report 909, Department of Computer Science, Yale University, pages 116–121, San Francisco, California, June 1992.
19. Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, 1993.
20. 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.
21. 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.
22. 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.
23. 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.
24. Dag Prawitz. *Natural Deduction.* Almquist and Wiksell, Uppsala, 1965.
25. David Sands. Total correctness by local improvement in program transformation. In Ron Cytron, editor, *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, pages 221–232, San Francisco, California, January 1995. ACM Press.

26. Morten Heine Sørensen, Robert Glück, and Neil Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *Proceedings of the Fifth European Symposium on Programming*, pages 485–500, Edinburgh, U.K., April 1994.
27. *TAPSOFT '95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, Aarhus, Denmark, May 1995.
28. Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, 1993.

A Correctness of binding-time analysis

```
% definition of t1 judgement

t1 : spec W A -> bta E T W P -> bta E' T A P -> ans A P -> type.


t1_z_s : t1 (spec_z_s) (bta_z_s) (bta_z_s) (ans_z_s).

t1_lam_s : t1 (spec_lam_s) (bta_lam_s B)
              (bta_lam_s B) (ans_lam_s).

t1_app_s : t1 (spec_app_s S1 S0) (bta_app_s B1 B0) C D
                <- t1 S0 B0 (bta_lam_s C0) D0
                <- t1 S1 (C0 _ _ B1) C D.

t1_z_d : t1 (spec_z_d) (bta_z_d) (bta_z_d) (ans_z_d).

t1_app_d : t1 (spec_app_d S1 S0) (bta_app_d B1 B0)
              (bta_app_d C1 C0) (ans_app_d D1 D0)
                <- t1 S0 B0 C0 D0
                <- t1 S1 B1 C1 D1.

t1_lam_d : t1 (spec_lam_d S) (bta_lam_d B)
              (bta_lam_d C) (ans_lam_d D)
        <- {x} {y}
           {S': spec y y} {B' : bta x T1 y dyn} {D': ans y dyn}
               t1 S' B' B' D'
                -> t1 (S y S') (B x y B') (C x y B') (D y D').

t1_lift : t1 (spec_lift S) (bta_lift B) (bta_z_d) (ans_z_d)
                <- t1 S B (bta_z_s) D.
```

This article was processed using the LaTeX macro package with LLNCS style