Mechanically verifying the correctness of an offline partial evaluator *

John Hatcliff †
DIKU
Computer Science Department
Copenhagen University ‡

September 10, 1995

Abstract

We show that using deductive systems to specify an offline partial evaluator allows its correctness to be mechanically verified.

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 evaluation correctness proofs (*e.g.*, the correctness of binding-time analysis) can be coded in Elf and mechanically verified.

This work illustrates the utility of declarative programming and of using deductive systems for defining program specialization systems: by exploiting the logical character of definitions, one may specify, prototype, and mechanically verify correctness *via* meta-programming — all within a single framework.

1 Introduction

As the application of program-specialization techniques increases, it is important to study ways of formally specifying and proving these techniques correct. Unfortunately, relatively few specialization systems have been proven correct, and we are not aware of any that have been mechanically verified. Even when systems have been proven correct, it is unclear from the literature which styles of formalization are preferable.

As an example, consider offline partial evaluation of functional programs [5, 20]. This consists of two phases: a binding-time analysis phase (where information is gathered about what parts of the source program depend on known or unknown data), and a specialization phase (where constructs depending on known data are reduced away). *Denotational semantics* has been the most widely applied formalism for specifying offline partial evaluators. The denotational method allows both phases to be specified in the same framework: the analysis phase is specified as an abstract interpretation [6, 21] and the specialization phase is specified as a function from a source program to a residual program. Moreover, a prototype implementation can often be obtained by a fairly straightforward (though informal) transliteration of the denotational specifications into functional programs.

Recent work specifies the analysis phase using *type systems* while the specialization phase is specified denotationally [10, 22]. Unfortunately, this approach loses a degree of synergy since both phases are not specified in the same framework. For example, one can no longer obtain a prototype *via* a simple translation to functional programs (since the typing rules which specify the analysis cannot be simply translated). Others define the

*DIKU Rapport 95/14. This is an extended version of a paper with the same title which appears in the *Proceedings of the Seventh International Symposium on Programming Languages, Implementations, Logics and Programs.* Utrecht, The Netherlands, September 20-22, 1995.

†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

program specializer using *operational semantics* and thus avoid what Wand has characterized as a "morass of model-theoretic details" [37] associated with the denotational framework.

Note that 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 the logical character of such deductive systems and obtain a uniform framework for specifying, prototyping, and mechanically verifying the correctness of program specialization systems. Specifically, we consider offline partial evaluation in the style of the partial evaluator λ-*mix* [11].[1] 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 and specialization [27, 37].

The results of the present work are as follows.

- We give 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 [19]. 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 [29]. 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.[2]

This methodology of specification/implementation/verification using LF and Elf has been successfully applied in other problem areas [18, 23]. In particular, we build on Hannan and Pfenning's work on compiler verification in Elf [18]. They conjectured that their techniques could also be applied to partial evaluation [18, p. 416]. They also identify the verification of transformations based on flow analyses as a "challenging problem, yet to be addressed" [18, 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. Hannan has recently reported on a second instance: the use of Elf to encode a type system for closure conversion [16].

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 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

This section presents a very brief overview of LF and ELf. The reader is referred to the original work on LF by Harper, Honsell, Plotkin [19] for a detailed presentation of LF. Pfenning [29] gives the technical foundations of

[1] 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).

[2] This paper contains all the required Elf code and is implemented using version 0.3 of Elf. The code is available upon request from the author.

Elf. Papers by Hannan and Pfenning [18] and Michaylov and Pfenning [23] give applications of Elf that provide the main techniques used in the present work.

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.

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

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 [19]. We take $\beta\eta$-equivalence as the notion of definitional equality in LF [19, 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). Optional components are enclosed in $\langle\cdot\rangle$.

kindexp	::=	`type`	Type
	|	{*id*⟨:*famexp*⟩} *kindexp*	$\Pi x\!:\!A.\,K$
	|	*famexp* `->` *kindexp*	$A \to K$
famexp	::=	*id*	a
	|	{*id*⟨:*famexp*$_1$⟩} *famexp*$_2$	$\Pi x\!:\!A_1.\,A_2$
	|	`[`*id*⟨:*famexp*$_1$⟩`]` *famexp*$_2$	$\lambda x\!:\!A_1\,.\,A_2$
	|	*famexp objexp*	$A\,M$
	|	*famexp*$_1$ `->` *famexp*$_2$	$A_1 \to A_2$
	|	*famexp*$_2$ `<-` *famexp*$_1$	$A_1 \to A_2$
objexp	::=	*id*	c
	|	`[`*id*⟨:*famexp*⟩`]` *objexp*	$\lambda x\!:\!A\,.\,M$
	|	*objexp*$_1$ *objexp*$_2$	$M_1\,M_2$

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

3.1 Syntax

Object language:

$e \in Exp$

$e ::= x \mid \mathsf{0} \mid \mathsf{sc}\, e \mid \mathsf{pr}\, e \mid \mathsf{if0}\, e_1\, e_2\, e_3 \mid \mathsf{lam}\, x\, .\, e \mid \mathsf{app}\, e_0\, e_1 \mid \mathsf{fix}\, x.\, e$

$\tau \in Typ$

$\tau ::= \mathsf{nat} \mid \tau_1 \rightarrow \tau_2$

Elf encoding:

```
exp : type.

z : exp.
sc : exp -> exp.
pr : exp -> exp.
if : exp -> exp -> exp -> exp.
lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.
fix : (exp -> exp) -> exp.
```

```
typ : type.

nat : typ.
arrow : typ -> typ -> typ.
```

Figure 1: The object language Λ

Figure 1 presents the syntax of the object language Λ (a PCF-like language) and its encoding in Elf. We identify object-level terms up to α-equivalence (*i.e.*, up to renaming of bound variables). We write $e_1 \equiv e_2$ when e_1 and e_2 are α-equivalent.

The Elf signature for the object language syntax includes family constants `exp` and `typ`, and object constants for each term and type constructor. Variable binding in $\mathsf{lam}\, x\, .\, e$ and $\mathsf{fix}\, x.\, e$ is represented using binding in the meta-language (*i.e.*, using higher-order abstract syntax [31]) — an idea going back to Church [3]. 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

$$\mathtt{app}\, (\mathtt{lam}\ \mathtt{[x1]}\ \mathtt{lam}\ \mathtt{[x2]}\ \mathtt{app}\, (\mathtt{lam}\ \mathtt{[x3]}\ \mathtt{x3})\ \mathtt{x1})\ \mathtt{z}.$$

In this style of encoding, the body of the lam expression

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

is represented as a "higher-order term" that maps expressions to expressions. This allows substitution in the object language to be represented using Elf application and β-reduction. For example,

$$(\mathsf{lam}\, x_2\, .\, \mathsf{app}\, (\mathsf{lam}\, x_3\, .\, x_3)\, x_1)[x_1 := z]$$

is represented in Elf as

$$(\mathtt{[x1]}\ \mathtt{lam}\ \mathtt{[x2]}\ \mathtt{app}\, (\mathtt{lam}\ \mathtt{[x3]}\ \mathtt{x3})\ \mathtt{x1})\ \mathtt{z}.$$

An Elf β-reduction gives the expected result of the substitution.

$$\mathtt{lam}\ \mathtt{[x2]}\ \mathtt{app}\, (\mathtt{lam}\ \mathtt{[x3]}\ \mathtt{x3})\ \mathtt{z}$$

This representation properly implements capture-free object-level substitution since it is expressed *via* Elf substitution (which the Elf implementation guarantees to be capture-free). Note that since variables in the object language are identified with variables in the meta-language, there is no explicit representation of identifiers in the Elf signature for the object language.

Technically, one must ensure that encodings of syntax into Elf are *adequate*, *i.e.*, that there is a compositional bijection between the syntactic entities in the logical system and well-formed LF long $\beta\eta$-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.*[19] for a detailed discussion and proofs of adequacy for encodings similar to the ones used here.

$$tp\_z: \quad tp\ 0 : \mathsf{nat}$$

$$tp\_sc: \quad \frac{tp\ e : \mathsf{nat}}{tp\ \mathsf{sc}\ e : \mathsf{nat}}$$

$$tp\_pr: \quad \frac{tp\ e : \mathsf{nat}}{tp\ \mathsf{pr}\ e : \mathsf{nat}}$$

$$tp\_cnd: \quad \frac{tp\ e_1 : \mathsf{nat} \qquad tp\ e_2 : \tau \qquad tp\ e_3 : \tau}{tp\ \mathsf{if0}\ e_1\ e_2\ e_3 : \tau}$$

$$tp\_lam: \quad \frac{\begin{array}{c}(tp\ x : \tau_1)\\ tp\ e : \tau_2\end{array}}{tp\ \mathsf{lam}\, x\, .\, e : \tau_1 \rightarrow \tau_2}$$

$$tp\_app: \quad \frac{tp\ e_0 : \tau_1 \rightarrow \tau_2 \qquad tp\ e_1 : \tau_1 w_1 \varphi_1}{tp\ \mathsf{app}\ e_0\ e_1 : \tau_2}$$

$$tp\_fix: \quad \frac{\begin{array}{c}(tp\ x : \tau)\\ tp\ e : \tau\end{array}}{tp\ \mathsf{fix}\ x.\ e : \tau}$$

Figure 2: The typing rules for Λ

3.2 Typing

3.2.1 Natural deduction style rules

Figure 3 presents the typing rules for Λ in a natural-deduction style logic (we use the notation of Prawitz [33]).[3] Parentheses in the hypotheses of the rules *tp_lam* and *tp_fix* indicate the discharging of zero or more occurrences of assumptions. We write $T \vdash tp\ e : \tau$ when $tp\ e : \tau$ is derivable under undischarged assumptions T. We shall only be concerned with assumptions involving identifiers (*e.g.*, $tp\ x : \tau$) since the logic only allows discharging of assumptions of this form. For simplicity, we require identifiers x_i in assumption sets $T = \{tp\ x_1 : \tau_1, \ldots, tp\ x_n : \tau_n\}$, to be pairwise distinct.

3.2.2 Elf encoding

Figure 3 presents the Elf encoding of the typing rules. The figure begins with a declaration of the typing judgement `tp` as a relation between source expressions `exp` and types `typ`. The implicit universal quantification of the upper case variables captures the schematic nature of the rules. The rules for binding constructs `lam` and `fix` (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 judgements to proofs of consequent judgements. The higher-order syntax representation requires that judgements involving identifiers be represented as schematic judgements (*i.e.*, identifiers are Π-quantified). This is the case in rules `tp_lam` and `tp_fix` where *e.g.*, the judgement `tp (E x) T2` expresses that `E` may be instantiated to representations of terms with free occurrences of x [19, Section 3.1].

[3] See the textbook of Tennent [36] for extensive examples using natural deduction logics to specify syntax and typing.

```
%%% typing judgement

tp : exp -> typ -> type.

%%% typing axioms and rules

tp_z : tp z nat.

tp_sc : tp (sc E) nat
        <- tp E nat.

tp_pr : tp (pr E) nat
        <- tp E nat.

tp_if : tp (if E1 E2 E3) T
        <- tp E1 nat
        <- tp E2 T
        <- tp E3 T.

tp_lam : tp (lam E) (arrow T1 T2)
         <- {x:exp} tp x T1 -> tp (E x) T2.

tp_app : tp (app E0 E1) T2
         <- tp E0 (arrow T1 T2)
         <- tp E1 T1.

tp_fix : tp (fix E) (arrow T1 T2)
         <- {x:exp} tp x (arrow T1 T2) -> tp (E x) (arrow T1 T2).
```

Figure 3: The Elf encoding of the typing rules

$$\textit{int\_z}: \quad \mathsf{0} \Downarrow_{int} \mathsf{0}$$

$$\textit{int\_sc}: \quad \frac{e \Downarrow_{int} v}{\mathsf{sc}\, e \Downarrow_{int} \mathsf{sc}\, v}$$

$$\textit{int\_prz}: \quad \frac{e \Downarrow_{int} \mathsf{0}}{\mathsf{pr}\, e \Downarrow_{int} \mathsf{0}}$$

$$\textit{int\_prsc}: \quad \frac{e \Downarrow_{int} \mathsf{sc}\, v}{\mathsf{pr}\, e \Downarrow_{int} v}$$

$$\textit{int\_ifz}: \quad \frac{e_1 \Downarrow_{int} \mathsf{0} \qquad e_2 \Downarrow_{int} v}{\mathsf{if0}\, e_1\, e_2\, e_3 \Downarrow_{int} v}$$

$$\textit{int\_ifsc}: \quad \frac{e_1 \Downarrow_{int} \mathsf{sc}\, v_1 \qquad e_3 \Downarrow_{int} v}{\mathsf{if0}\, e_1\, e_2\, e_3 \Downarrow_{int} v}$$

$$\textit{int\_lam}: \quad \mathsf{lam}\, y\, .\, e \Downarrow_{int} \mathsf{lam}\, y\, .\, e$$

$$\textit{int\_app}: \quad \frac{e_0 \Downarrow_{int} \mathsf{lam}\, y\, .\, e_0' \qquad e_0'[y := e_1] \Downarrow_{int} v}{\mathsf{app}\, e_0\, e_1 \Downarrow_{int} v}$$

$$\textit{int\_fix}: \quad \frac{e[y := \mathsf{fix}\, y.\, e] \Downarrow_{int} v}{\mathsf{fix}\, y.\, e \Downarrow_{int} v}$$

Figure 4: The natural semantics rules for Λ (interpretation logic)

3.2.3 Executing the specification

The Elf encoding of the typing rules (see Figure 3) is an executable logic program. For example, type inference for the example expression of Section 3.1 can be performed *via* the following Elf query.

```
?- tp (app (lam [x1] lam [x2] app (lam [x3] x3) x1) z) T.

Solving...

T = arrow T1 nat.
;
no more solutions
```

The solution `T = arrow T1 nat` indicates that the example expression is a function yielding results of type `nat` for any argument type `T1`.

3.3 Operational Semantics

3.3.1 Natural deduction style rules

Figure 4 presents the call-by-name natural semantics rules (*i.e.*, the *interpretation* or *evaluation* logic) for Λ terms. These rules correspond to the usual call-by-name "natural" or "deductive" operational semantics for PCF [14, Chapter 4]. We write $\vdash e \Downarrow_{int} a$ when $e \Downarrow_{int} a$ is derivable, and $e \Uparrow_{int}$ when there does not exist an a such that $\vdash e \Downarrow_{int} a$. It is easy to check that the relation induced by $\Downarrow_{int}$ is a partial function.

Two terms are considered *operationally equivalent* if they cannot be distinguished by any program context [14, Chapter 6].

```
%%% interpreter judgement

int : exp -> exp -> type.

%%% interpreter axioms and rules

int_z : int z z.

int_sc : int (sc E) (sc A)
         <- int E A.

int_prz : int (pr E) z
          <- int E z.

int_prsc : int (pr E) A
           <- int E (sc A).

int_ifz : int (if E1 E2 E3) A
          <- int E1 z
          <- int E2 A.

int_ifsc : int (if E1 E2 E3) A
           <- int E1 (sc A1)
           <- int E3 A.

int_lam : int (lam E) (lam E).

int_app : int (app E0 E1) A
          <- int E0 (lam E0')
          <- int (E0' E1) A.

int_fix : int (fix E) A
          <- int (E (fix E)) A.
```

Figure 5: The Elf encoding of the natural semantics rules (interpreter)

Definition 1 (Operational Equivalence) $e_1, e_2 \in \Lambda$ *are* operationally equivalent *(written* $e_1 \approx e_2$ *if for all contexts* C *such that* $\vdash tp\ e_1 : \mathsf{nat}$ *and* $\vdash tp\ e_2 : \mathsf{nat}$ *one of the following holds:*

1. $e_1 \Uparrow_{int}$ *and* $e_2 \Uparrow_{int}$, *or*
2. $\vdash e_1 \Downarrow_{int} a_1$ *and* $\vdash e_2 \Downarrow_{int} a_2$ *and* $a_1 \equiv a_2$.

Section 6 shows that our partial evaluator produces an output program that is operationally equivalent to the given input program.

3.3.2 Elf encoding

Figure 5 gives the Elf encoding of the natural semantics. The encoding techniques are similar to those used by Michaylov and Pfenning [23]. In the rules `int_app` and `int_fix`, we take advantage of the higher-order abstract syntax representation and use Elf application (*i.e.*, β-reduction) to implement capture-free substitution.

Value rules:

val_z : $val\ \mathsf{0}$

val_sc : $\dfrac{val\ v}{val\ \mathsf{sc}\ v}$

val_lam : $val\ \mathsf{lam}\, x\, .\, e$

Elf encoding:

```
%%% value judgement

val : exp -> type.

%%% value logic

val_z : val z.

val_sc : val (sc E)
         <- val E.

val_lam : val (lam E).
```

Figure 6: The value rules and Elf encoding

3.3.3 Executing the specification

As with the typing rules, the Elf encoding of the natural semantics rules (see Figure 5) is an executable logic program. For example, evaluation of the example expression of Section 3.1 can be performed *via* the following Elf query.

```
?- int (app (lam [x1] lam [x2] app (lam [x3] x3) x1) z) A.

A = lam [x2:exp] app (lam [x3:exp] x3) z.
;
no more solutions
```

3.3.4 Canonical terms of evaluation

An examination of the rules of Figure 4 reveals that the canonical terms (*i.e.*, *values*) of evaluation are either $\mathsf{0}$, $\mathsf{sc}\ e$ where e is also canonical, and $\mathsf{lam}\, x\, .\, e'$. We formalize the notion of canonical term with the judgement *val* of Figure 6. The encoding of the value rules in Elf is straightforward.

The following proposition formalizes the above claim: terms satisfying the value judgement *val* are indeed the results of evaluation.

Proposition 1 *If* $\vdash e \Downarrow_{int} e'$, *then* $\vdash val\ e'$

The proof of this proposition proceeds by induction over derivations of $e \Downarrow_{int} e'$; it can be formalized and mechanically checked in Elf. The reader is referred to the work of Michaylov and Pfenning [23] where a detailed presentation of such a proof for call-by-value evaluation is given.

Finally, note that we have followed convention [14] and defined the value judgement on the syntactic category of expressions while omitting any notion of typing. Thus, the judgement does not enforce well-typedness and admits terms that would not result from the evaluation of a well-typed program. For example, $\vdash val\ \mathsf{sc}\ (\mathsf{lam}\, x\, .\, x)$ but there does not exist a τ such that $\vdash tp\ \mathsf{sc}\ (\mathsf{lam}\, x\, .\, x) : \tau$. Well-typedness of values is expressed by using the typing judgement in conjunction with the value judgement.

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

Annotated object language:

$$
\begin{array}{lcl}
w & \in & \mathit{Sexp} \\
w & ::= & y \mid 0_m \mid \mathsf{sc}_m\, w \mid \mathsf{pr}_m\, w \mid \mathsf{if0}_m\, w_1\, w_2\, w_3 \mid \\
 & & \mathsf{lam}_m\, y\,.\, w \mid \mathsf{app}_m\, w_0\, w_1 \mid \mathsf{fix}_m\, y.\, w \mid \\
 & & \mathsf{lift}\, w
\end{array}
\qquad
\begin{array}{lcl}
m & \in & \mathit{Sder} \\
m & ::= & s \mid d
\end{array}
$$

Elf encoding:

```
sexp : type.

bz : sder -> sexp.
bsc : sder -> sexp -> sexp.
bpr : sder -> sexp -> sexp.
bif : sder -> sexp -> sexp -> sexp -> sexp.
blam : sder -> (sexp -> sexp) -> sexp.
bapp : sder -> sexp -> sexp -> sexp.
bfix : sder -> (sexp -> sexp) -> sexp.
lift : sexp -> sexp.
```

```
sder : type.

s : sder.
d : sder.
```

Figure 7: The annotated object language Λ_{bt}

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 constructs an annotated program where each program construct is annotated with a *specialization directive* and a *specialization type*.

   - **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.
   - **Specialization types**: The specialization types (*a.k.a binding times*) are the carriers of information during the analysis phase. They describe the "knownness" or the "unknownness" of expressions. This information is used to determine the specialization directives assigned to constructs. For example, if the argument of a destructor construct has a specialization type indicating that it is unknown, then that construct cannot be evaluated at specialization time and must be given a *residual* directive.

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).

Several different formalizations of binding-time analysis and specialization are given in the literature (see the textbook of Jones *et al.* [20] for a survey). In the remainder of this section, we give a logical formalization that is suitable for encoding in Elf.

4.1 Binding-time analysis

We first outline how binding-time information is expressed in a program annotated with specialization directives and types. Following this, we give a binding-time logic that determines which annotations are appropriate for a given soure language term.

Specialization types:

$$\begin{array}{lcl} \varphi_{\mathsf{nat}} & \in & \mathit{Styp}[\mathsf{nat}] \\ \varphi_{\mathsf{nat}} & ::= & \mathsf{sta} \mid \mathsf{dyn}_{\mathsf{nat}} \\ \\ \varphi_{\tau_1 \to \tau_2} & \in & \mathit{Styp}[\tau_1 \to \tau_2] \\ \varphi_{\tau_1 \to \tau_2} & ::= & \varphi_{\tau_1} \to \varphi_{\tau_2} \mid \mathsf{dyn}_{\tau_1 \to \tau_2} \end{array}$$

Elf encoding:

```
styp   : typ -> type.

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

Figure 8: The specialization types for Λ_{bt}

4.1.1 Specialization directives

A binding-time analysis for Λ associates each Λ term with a term in the annotated language Λ_{bt} of Figure 7. 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 and thus can be eliminated at specialization time) or residual (*i.e.*, it may depend on *dynamic* data and thus must be placed in the residual program). 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 (an example is given in Section 4.3). A term $w \in \Lambda_{bt}$ is *completely residual* if it consists of only d-indexed constructs and identifiers. For example, the following well-annotated version of the example expression of Section 3.1 is completely residual.

$$\mathsf{app}_d\,(\mathsf{lam}_d\,y_1\,.\,\mathsf{lam}_d\,y_2\,.\,\mathsf{app}_d\,(\mathsf{lam}_d\,y_3\,.\,y_3)\,y_1)\,\mathsf{0}_d$$

Intuitively, the specializer will output completely residual terms — all eliminable constructs will have been evaluated (this will be proven in Section 6.1). 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). For example, the term above is encoded as follows.

```
bapp d (blam d [y1] (blam d [y2] (bapp d (blam d [y3] y3) y1))) (bz d)
```

4.1.2 Specialization types

Figure 8 presents a τ-indexed family of specialization types for Λ_{bt} (we omit type indices on specialization types when they can be inferred from the context). A specialization type φ is *dynamic* if $\varphi = \mathsf{dyn}$; otherwise φ is *static*. The intention is that

- $\mathsf{sta} \in \mathit{Styp}[\mathsf{nat}]$ will tag expressions of type nat that are guaranteed to evaluate to known data (i.e., numerals $\mathsf{0}$, $\mathsf{sc}\,\mathsf{0}$, *etc.*);
- $\mathsf{dyn}_{\mathsf{nat}} \in \mathit{Styp}[\mathsf{nat}]$ will tag expressions of type nat whose evaluation may depend on unknown data and thus cannot be guaranteed to evaluate to numerals;
- $\varphi_{\tau_1} \to \varphi_{\tau_2} \in \mathit{Styp}[\tau_1 \to \tau_2]$ will tag expressions of type $\tau_1 \to \tau_2$ that are guaranteed to evaluate to known data (i.e., abstractions);
- $\mathsf{dyn}_{\tau_1 \to \tau_2} \in \mathit{Styp}[\tau_1 \to \tau_2]$ will tag expressions of type $\tau_1 \to \tau_2$ whose evaluation may depend on unknown data and thus cannot be guaranteed to evaluate to an abstraction.

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.

4.1.3 Binding-time logic

Figure 9 presents a natural-deduction-style logic for deriving judgements of the form $\mathit{bta}\ e : \tau\ [w : \varphi_\tau]$. Parentheses in the hypotheses of the rules for binding constructs (*i.e.*, *bta_lam_s*, *bta_lam_d*, *bta_fix_s*, *bta_fix_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.

Derivable judgements specify constraints that an actual binding-time analysis algorithm must satisfy. Intuitively, if $\Gamma \vdash bta\ e : \tau\ [w : \varphi_\tau]$, then given initial binding-time assumptions Γ, a binding-time analysis algorithm may map $e \in \mathit{Terms}[\Lambda]$ of type $\tau \in \mathit{Types}[\Lambda]$ to a directive annotated term $w \in \mathit{Terms}[\Lambda_{bt}]$ of specialization type $\varphi_\tau \in \mathit{Spec\text{-}types}[\tau]$. Specifying the analysis in this way allows one to reason about correctness of the analysis independently of the actual algorithm — proving correctness with respect to the constraints implies that any algorithm satisfying the constraints will be correct. Furthermore, Section 5.1 shows how the Elf proof search mechanism can be used to extract algorithmic content directly from the logic specifying the constraints.

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 $\Gamma = \{bta\ x_1 : \tau_1\ [y_1 : \varphi_{\tau_1}], \ldots, bta\ x_n : \tau_n\ [y_n : \varphi_{\tau_n}]\}$, 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. As a consequence, a binding-time analysis satisfying these constraints has some flexibility in assigning correct annotations. In practice, one desires an analysis that gives as many eliminable directives (*i.e.*, gives as many s indices) as possible.[4] The constraint system is *complete* in the sense that one may always obtain a valid annotation of a type correct e by annotating all components as residual.

Finally, since the relation induced by *bta* is one-to-many from Λ to Λ_{bt}, it is many-to-one (*i.e.*, a function) from Λ_{bt} to Λ. This is essentially an *annotation forgetting function* that 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 10 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 judgements to proofs of consequent 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`, `bta_lam_d`, `bta_fix_s`, and `bta_fix_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 [19, Section 3.1].

4.2 Specialization

4.2.1 Natural deduction rules

Figures 11 and 12 present the specialization logic for Λ_{bt} terms.

- Figure 11 describes the evaluation of eliminable constructs. These rules correspond to the usual call-by-name "natural" or "deductive" operational semantics for Λ given in Figure 4.
- Figure 12 describes the reconstruction of residual constructs after subexpressions have been specialized. The rules for lam_d and fix_d are noteworthy because they introduce operations on open terms (*e.g.*, bodies of lam_d and fix_d constructs). Following Hannan [15, p. 144], we specialize the body of a lam_d (and fix_d) under the assumption that the bound variable evaluates to itself.

[4] We omit considering whether a term has a "best" annotation according to the logic. See the work on λ-mix [10, 11] for related discussions.

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

$$bta\_sc\_s: \quad \frac{bta\ e : \mathsf{nat}\ [w : \mathsf{sta}]}{bta\ \mathsf{sc}\ e : \mathsf{nat}\ [\mathsf{sc}_s\ w : \mathsf{sta}]}$$

$$bta\_pr\_s: \quad \frac{bta\ e : \mathsf{nat}\ [w : \mathsf{sta}]}{bta\ \mathsf{pr}\ e : \mathsf{nat}\ [\mathsf{pr}_s\ w : \mathsf{sta}]}$$

$$bta\_if\_s: \quad \frac{bta\ e_1 : \mathsf{nat}\ [w_1 : \mathsf{sta}] \qquad bta\ e_2 : \tau\ [w_2 : \varphi] \qquad bta\ e_3 : \tau\ [w_3 : \varphi]}{bta\ \mathsf{if0}\ e_1\ e_2\ e_3 : \tau\ [\mathsf{if0}_s\ w_1\ w_2\ w_3 : \varphi]}$$

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

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

$$bta\_fix\_s: \quad \frac{\begin{array}{c}(bta\ x : \tau\ [y : \varphi]) \\ bta\ e : \tau\ [w : \varphi]\end{array}}{bta\ \mathsf{fix}\ x.\ e : \tau\ [\mathsf{fix}_s\ y.\ w : \varphi]}$$

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

$$bta\_sc\_d: \quad \frac{bta\ e : \mathsf{nat}\ [w : \mathsf{dyn}]}{bta\ \mathsf{sc}\ e : \mathsf{nat}\ [\mathsf{sc}_d\ w : \mathsf{dyn}]}$$

$$bta\_pr\_d: \quad \frac{bta\ e : \mathsf{nat}\ [w : \mathsf{dyn}]}{bta\ \mathsf{pr}\ e : \mathsf{nat}\ [\mathsf{pr}_d\ w : \mathsf{dyn}]}$$

$$bta\_if\_d: \quad \frac{bta\ e_1 : \mathsf{nat}\ [w_1 : \mathsf{dyn}] \qquad bta\ e_2 : \tau\ [w_2 : \mathsf{dyn}] \qquad bta\ e_3 : \tau\ [w_3 : \mathsf{dyn}]}{bta\ \mathsf{if0}\ e_1\ e_2\ e_3 : \tau\ [\mathsf{if0}_d\ w_1\ w_2\ w_3 : \mathsf{dyn}]}$$

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

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

$$bta\_fix\_d: \quad \frac{\begin{array}{c}(bta\ x : \tau\ [y : \mathsf{dyn}]) \\ bta\ e : \tau\ [w : \mathsf{dyn}]\end{array}}{bta\ \mathsf{fix}\ x.\ e : \tau\ [\mathsf{fix}_d\ y.\ w : \mathsf{dyn}]}$$

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

Figure 9: The binding-time logic

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

bta_sc_s : bta (sc E) nat (bsc s W) sta
           <- bta E nat W sta.

bta_pr_s : bta (pr E) nat (bpr s W) sta
           <- bta E nat W sta.

bta_if_s : bta (if E1 E2 E3) T (bif s W1 W2 W3) P
           <- bta E1 nat W1 sta
           <- bta E2 T W2 P
           <- bta E3 T W3 P.

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

bta_z_d : bta z nat (bz d) dyn.

bta_sc_d : bta (sc E) nat (bsc d W) dyn
           <- bta E nat W dyn.

bta_pr_d : bta (pr E) nat (bpr d W) dyn
           <- bta E nat W dyn.

bta_if_d : bta (if E1 E2 E3) T (bif d W1 W2 W3) dyn
           <- bta E1 nat W1 dyn
           <- bta E2 T W2 dyn
           <- bta E3 T W3 dyn.

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

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

Figure 10: The Elf encoding of the binding-time logic

$$\textit{spec\_z\_s}: \quad \mathsf{0}_s \Downarrow_{spec} \mathsf{0}_s$$

$$\textit{spec\_sc\_s}: \quad \frac{w \Downarrow_{spec} a}{\mathsf{sc}_s\ w \Downarrow_{spec} \mathsf{sc}_s\ a}$$

$$\textit{spec\_prz\_s}: \quad \frac{w \Downarrow_{spec} \mathsf{0}_s}{\mathsf{pr}_s\ w \Downarrow_{spec} \mathsf{0}_s}$$

$$\textit{spec\_prsc\_s}: \quad \frac{w \Downarrow_{spec} \mathsf{sc}_s\ a}{\mathsf{pr}_s\ w \Downarrow_{spec} a}$$

$$\textit{spec\_ifz\_s}: \quad \frac{w_1 \Downarrow_{spec} \mathsf{0}_s \qquad w_2 \Downarrow_{spec} a}{\mathsf{if0}_s\ w_1\ w_2\ w_3 \Downarrow_{spec} a}$$

$$\textit{spec\_ifsc\_s}: \quad \frac{w_1 \Downarrow_{spec} \mathsf{sc}_s\ a_1 \qquad w_3 \Downarrow_{spec} a}{\mathsf{if0}_s\ w_1\ w_2\ w_3 \Downarrow_{spec} a}$$

$$\textit{spec\_lam\_s}: \quad \mathsf{lam}_s\ y\ .\ w \Downarrow_{spec} \mathsf{lam}_s\ y\ .\ w$$

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

$$\textit{spec\_fix\_s}: \quad \frac{w[y := \mathsf{fix}_s\ y.\ w] \Downarrow_{spec} a}{\mathsf{fix}_s\ y.\ w \Downarrow_{spec} a}$$

Figure 11: The specialization logic (static constructs)

$$\textit{spec\_z\_d}: \quad \mathsf{0}_d \Downarrow_{spec} \mathsf{0}_d$$

$$\textit{spec\_sc\_d}: \quad \frac{w \Downarrow_{spec} a}{\mathsf{sc}_d\ w \Downarrow_{spec} \mathsf{sc}_d\ a}$$

$$\textit{spec\_pr\_d}: \quad \frac{w \Downarrow_{spec} a}{\mathsf{pr}_d\ w \Downarrow_{spec} \mathsf{pr}_d\ a}$$

$$\textit{spec\_if\_d}: \quad \frac{w_1 \Downarrow_{spec} a_1 \qquad w_2 \Downarrow_{spec} a_2 \qquad w_3 \Downarrow_{spec} a_3}{\mathsf{if0}_d\ w_1\ w_2\ w_3 \Downarrow_{spec} \mathsf{if0}_d\ a_1\ a_2\ a_3}$$

$$\textit{spec\_lam\_d}: \quad \frac{(y \Downarrow_{spec} y) \quad w \Downarrow_{spec} a}{\mathsf{lam}_d\ y\ .\ w \Downarrow_{spec} \mathsf{lam}_d\ y\ .\ a}$$

$$\textit{spec\_app\_d}: \quad \frac{w_0 \Downarrow_{spec} a_0 \qquad w_1 \Downarrow_{spec} a_1}{\mathsf{app}_d\ w_0\ w_1 \Downarrow_{spec} \mathsf{app}_d\ a_0\ a_1}$$

$$\textit{spec\_fix\_d}: \quad \frac{(y \Downarrow_{spec} y) \quad w \Downarrow_{spec} a}{\mathsf{fix}_d\ y.\ w \Downarrow_{spec} \mathsf{fix}_d\ y.\ a}$$

$$\textit{spec\_lift}: \quad \frac{w \Downarrow_{spec} \mathsf{0}_s}{\mathsf{lift}\ w \Downarrow_{spec} \mathsf{0}_d}$$

$$\textit{spec\_lift\_sc}: \quad \frac{w \Downarrow_{spec} \mathsf{sc}_s\ a' \qquad \mathsf{lift}\ a' \Downarrow_{spec} a}{\mathsf{lift}\ w \Downarrow_{spec} \mathsf{sc}_d\ a}$$

Figure 12: The specialization logic (dynamic and coercion constructs)

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

spec_sc_s : spec (bsc s W) (bsc s A)
            <- spec W A.

spec_prz_s : spec (bpr s W) (bz s)
             <- spec W (bz s).

spec_prsc_s : spec (bpr s W) A
              <- spec W (bsc s A).

spec_ifz_s : spec (bif s W1 W2 W3) A
             <- spec W1 (bz s)
             <- spec W2 A.

spec_ifsc_s : spec (bif s W1 W2 W3) A
              <- spec W1 (bsc s A1)
              <- spec W3 A.

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

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

spec_fix_s : spec (bfix s W) A
             <- spec (W (bfix s W)) A.
```

Figure 13: The Elf encoding of the specialization logic (static constructs)

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

spec_sc_d : spec (bsc d W) (bsc d A)
            <- spec W A.

spec_pr_d : spec (bpr d W) (bpr d A)
            <- spec W A.

spec_if_d : spec (bif d W1 W2 W3) (bif d A1 A2 A3)
            <- spec W1 A1
            <- spec W2 A2
            <- spec W3 A3.

spec_lam_d : spec (blam d W) (blam d A)
             <- {x} spec x x -> spec (W x) (A x).

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

spec_fix_d : spec (bfix d W) (bfix d A)
             <- {x} spec x x -> spec (W x) (A x).

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

spec_liftsc : spec (lift W) (bsc d A)
              <- spec W (bsc s A')
              <- spec (lift A') A.
```

Figure 14: The Elf encoding of the specialization logic (dynamic and coercion constructs)

An alternative to introducing local assumptions for bound variables would be to add the axiom $y \Downarrow_{spec} y$ (which states that all variables evaluate to themselves). However, this approach cannot be directly represented in Elf. Under the higher-order abstract syntax representation, the variables y are not objects in the Elf signature. The approach taken in Figure 12 seems the most natural with the higher-order abstract syntax representation.

- The final rules of Figure 12 coerce an eliminable numeral 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 $\Sigma = \{y_1 \Downarrow_{spec} y_1, \ldots, y_n \Downarrow_{spec} y_n\}$ the y_i are required to be pairwise distinct. Σ *is compatible with* $\Gamma = \{\mathit{bta}\ x_1 : \tau_1\ [y_1 : \varphi_{\tau_1}], \ldots, \mathit{bta}\ x_n : \tau_n\ [y_n : \varphi_{\tau_n}]\}$ if $\Sigma = \{y_1 \Downarrow_{spec} y_1, \ldots, y_n \Downarrow_{spec} y_n\}$. It is easy to check that the relation induced by $\Downarrow_{spec}$ is a partial function (given the constraints on assumptions above).

4.2.2 Elf encoding

Figures 13 and 14 give the Elf encoding of the specialization logic. The encoding techniques are similar to those used in the previous section. The specialization judgement specifies a binary relation (which is actually a partial function) on annotated terms *Sexp* and is encoded in Elf as follows.

```
spec : sexp -> sexp -> type.
```

In the rules *spec_app_s* and *spec_fix_s*, we take advantage of the higher-order abstract syntax representation and use Elf application (*i.e.*, β-reduction) to implement capture-free substitution.

4.2.3 Canonical terms of specialization

In the conventional deductive semantics for Λ in Section 3.3, one has numerals (*e.g.*, $\mathsf{0}$, $\mathsf{sc}\ \mathsf{0}$) and $\mathsf{lam}\, x\, .\, e$ as canonical terms, *i.e.*, these terms are the results of evaluation. This was formalized by the *val* judgement.

Intuitively, a specializer is part evaluator and part compiler. Therefore, the canonical terms of the specializer are numerals and $\mathsf{lam}_s\, y\, .\, w$ (corresponding to evaluation results), and completely residual terms or *code* (corresponding to compilation results). In a manner analogous to the formalization of *values* (canonical terms of evaluation), Figure 15 formalizes the notion of *answers* (canonical terms of specialization) by defining a judgement $\mathit{ans}\ w : \varphi_\tau$.

We write $A \vdash \mathit{ans}\ w : \varphi_\tau$ when $\mathit{ans}\ w : \varphi_\tau$ is derivable under undischarged assumptions A. A *is compatible with* $\Gamma = \{\mathit{bta}\ x_1 : \tau_1\ [y_1 : \varphi_{\tau_1}], \ldots, \mathit{bta}\ x_n : \tau_n\ [y_n : \varphi_{\tau_n}]\}$ if $A = \{\mathit{ans}\ y_1 : \varphi_{\tau_1}, \ldots, \mathit{ans}\ y_n : \varphi_{\tau_n}\}$. Intuitively, if $\mathit{ans}\ w : \varphi_\tau$ holds, then w is a canonical term of specialization type φ_τ. In particular, if $\vdash \mathit{ans}\ w : \mathsf{dyn}$, then w is completely residual.

In contrast with the *val* judgement of Section 3.3 which contained no notion of typing, the judgement $\mathit{ans}\ w : \varphi_\tau$ relates a canonical term w to a particular specialization type φ_τ. We will need this additional information when stating the correctness of binding-time analysis in Section 6.1. As with the *val* judgement, the *ans* judgement admits answers that are not well-annotated. For example, $\vdash \mathit{ans}\ \mathsf{sc}\ (\mathsf{lam}\, x\, .\, x) : \mathsf{dyn}_\tau$ but there does not exist e and τ such that $\vdash \mathit{bta}\ e : \tau\ [\mathsf{sc}\ (\mathsf{lam}\, x\, .\, x) : \tau]$. Well-annotatedness of answers is expressed by using the binding-time judgement in conjunction with the answer judgement. Figure 16 gives the Elf encoding of the answer logic.

In Section 3.3, Proposition 1 showed that the value judgement *val* described the results of evaluation. An analogous proposition below shows that the answer judgement describes the results of specialization.

Proposition 2 *If* $\vdash w \Downarrow_{spec} a$ *and* $\vdash \mathit{bta}\ e : \tau\ [w : \varphi_\tau]$, *then* $\vdash \mathit{ans}\ a : \varphi_\tau$.

The proof of this proposition follows as a corrollary of the mechanically verified proof of Theorem 1 in Section 6.

$$ans\_z\_s: \quad ans\ 0_s : \mathsf{sta}$$

$$ans\_sc\_s: \quad \frac{ans\ w : \mathsf{sta}}{ans\ \mathsf{sc}_s\ w : \mathsf{sta}}$$

$$ans\_lam\_s: \quad ans\ \mathsf{lam}_s\ x\ .\ w : \varphi_1 \to \varphi_2$$

$$ans\_z\_d: \quad ans\ 0_d : \mathsf{dyn}$$

$$ans\_sc\_d: \quad \frac{ans\ w : \mathsf{dyn}}{ans\ \mathsf{sc}_d\ w : \mathsf{dyn}}$$

$$ans\_pr\_d: \quad \frac{ans\ w : \mathsf{dyn}}{ans\ \mathsf{pr}_d\ w : \mathsf{dyn}}$$

$$ans\_if\_d: \quad \frac{ans\ w_1 : \mathsf{dyn} \qquad ans\ w_2 : \mathsf{dyn} \qquad ans\ w_3 : \mathsf{dyn}}{ans\ \mathsf{if0}_d\ w_1\ w_2\ w_3 : \mathsf{dyn}}$$

$$ans\_lam\_d: \quad \frac{\begin{array}{c}(ans\ x : \mathsf{dyn}) \\ ans\ w : \mathsf{dyn}\end{array}}{ans\ \mathsf{lam}_d\ x\ .\ w : \mathsf{dyn}}$$

$$ans\_app\_d: \quad \frac{ans\ w_0 : \mathsf{dyn} \qquad ans\ w_1 : \mathsf{dyn}}{ans\ \mathsf{app}_d\ w_0\ w_1 : \mathsf{dyn}}$$

$$ans\_fix\_d: \quad \frac{\begin{array}{c}(ans\ x : \mathsf{dyn}) \\ ans\ w : \mathsf{dyn}\end{array}}{ans\ \mathsf{fix}_d\ x.\ w : \mathsf{dyn}}$$

Figure 15: The answer logic

```
%%% answer judgement

ans : sexp -> styp T -> type.

%%% answer logic

% static answers

ans_z_s : ans (bz s) sta.

ans_sc_s : ans (bsc s W) sta
           <- ans W sta.

ans_lam_s : ans (blam s W) (barrow P1 P2).

% dynamic answers

ans_z_d : ans (bz d) dyn.

ans_sc_d : ans (bsc d W) dyn
           <- ans W dyn.

ans_pr_d : ans (bpr d W) dyn
           <- ans W dyn.

ans_if_d : ans (bif d W1 W2 W3) dyn
           <- ans W1 dyn
           <- ans W2 dyn
           <- ans W3 dyn.

ans_lam_d : ans (blam d W) dyn
            <- {x} ans x dyn -> ans (W x) dyn.

ans_app_d : ans (bapp d W0 W1) dyn
            <- ans W0 dyn
            <- ans W1 dyn.

ans_fix_d : ans (bfix d W) dyn
            <- {x} ans x dyn -> ans (W x) dyn.
```

Figure 16: The Elf encoding of the answer logic

4.3 Partial evaluation

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

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

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

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

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

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

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

Theorem 1 (correctness of binding-time analyis) of Section 6.1 tells us there exists $e' \in \Lambda$ such that $\Gamma_d \vdash bta\ e' : \mathsf{nat}\,[\mathsf{app}_d\,y_2\,\mathsf{0}_d : \mathsf{dyn}]$. 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 3 (soundness of specialization) of Section 6.2 tells us that $e[x_0 := \mathsf{0}]$ is convertible to e' (denoted $e[x_0 := \mathsf{0}] =_\Lambda e'$) in the program calculus for Λ. Thus, for all closed inputs $d \in \Lambda$ of type $\mathsf{nat} \to \mathsf{nat}$,

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

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 Λ

5.1 Prototyping a binding-time analysis

The Elf encoding of the binding-time logic (see Figure 10) 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).[5]

```
?- 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] 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 Figures 13 and 14) 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).

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

```
?- 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 deduction used to obtain `A`.[6]

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 can never "go wrong". As described by Palsberg [27], a 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 11) 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 $\varphi_1 \to \varphi_2$ 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. 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 [19, Section 4.1]. However, the Elf encoding makes matters sufficiently clear.

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 $\mathcal{B}$,

[6] 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.

i.e., the method is primitive recursive. Although this primitive recursive method 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 $\mathcal{S} = \mathit{spec\_z\_s} :: \mathsf{0}_s \Downarrow_{spec} \mathsf{0}_s$ and $\mathcal{B} = \mathit{bta\_z\_s} :: \mathit{bta}\ \mathsf{0} : \mathsf{nat}\ [\mathsf{0}_s : \mathsf{sta}]$:

The required deductions are $\mathit{bta\_z\_s} :: \mathit{bta}\ \mathsf{0} : \mathsf{nat}\ [\mathsf{0}_s : \mathsf{sta}]$ and $\mathit{ans\_z\_s} :: \mathit{ans}\ \mathsf{0}_s : \mathsf{sta}$. 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 *ans* rules (Figure 15) reveals that $a_0 \equiv \mathsf{lam}_s\,y\,.\,w_0'$. Then, an examination of the axiomatization of the *bta* judgement (Figure 9) 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) 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 [19, 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 [32]. 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 describe a meaning preserving transformation on the unannotated object program. Specifically, if *(a)* e_1 is the unannotated version of the program to be specialized (with the static data already supplied), and *(b)* e_2 is the unannotated version of the result of specialization, then a sound specializer should guarantee that e_1 and e_2 are *operationally equivalent* (as defined in Section 3.3).

This ensures that e_1 and e_2 will behave the same under evaluation when dynamic data is supplied. Specifically, for all dynamic data d such that $T \vdash tp\ e_1[x := d] : \mathsf{nat}$ and $T \vdash tp\ e_2[x := d] : \mathsf{nat}$, $e_1[x := d] \approx e_2[x := d]$. The last claim follows since the substitution $[x := d]$ can be represented with the context $\mathsf{app}\,(\mathsf{lam}\,x\,.\,[\cdot])\,d$ and the fact that $\approx$ is easily shown to be a congruence.

We prove the operational equivalence of e_1 and e_2 by appealing to a *program calculus for* Λ (*i.e.*, an equational theory for deducing operational equivalences). Gunter [14, Chapter 4] gives a calculus for PCF, and Figure 17 recasts the calculus in our setting. $e_1 =_\Lambda e_2$ is the judgement expressing convertibility in the calculus. The rules are generally the expected ones for call-by-name. However, note the use of the judgement *val* in the rules *conv_prsc* and *conv_ifsc*. This reflects the strictness of the primitive operation `pr` and the strictness of the conditional `if0` in the test position. Appendix B.1 gives the Elf encoding of the calculus. It is similar to the encoding given by Pfenning for the pure untyped λ-calculus [30].

The following theorem expresses the soundness of the calculus for reasoning about Λ operational equivalences.

Theorem 2 (Soundness of calculus) *For all* $e_1, e_2 \in \Lambda$ *such that* $T \vdash tp\ e_1 : \tau$ *and* $T \vdash tp\ e_2 : \tau$,

$$e_1 =_\Lambda e_2 \ \textit{implies}\ e_1 \approx e_2.$$

Existing techniques for proving the theorem are fairly involved. For example, Gunter [14] appeals to an adequacy result connecting the operational semantics and a model. Other techniques use co-induction principles [12]. Unfortunately, it seems impossible to encode any of these techniques in Elf (Section 8 discusses the encoding of these techniques in other frameworks).

The inability to encode the proof in Elf does not seem detrimental to our approach. Historically, one of the reasons for introducing for a program calculus is to avoid repeated appeals to direct proofs of operational equivalences and reason instead using the simple rules of the calculus. The appropriateness of this approach is underscored by the situation we face here — it is much easier to encode a proof that appeals to the calculus.

The following theorem captures the fact that the unannotated object program e is operationally equivalent to the unannotated specialization result e'.

$$conv\_beta: \quad \mathsf{app}\,(\mathsf{lam}\,x\,.\,e_0)\,e_1 =_\Lambda e_0[x := e_1]$$

$$conv\_fix: \quad \mathsf{fix}\,x.\,e =_\Lambda e[x := \mathsf{fix}\,x.\,e]$$

$$conv\_prz: \quad \mathsf{pr}\ 0 =_\Lambda 0$$

$$conv\_prsc: \quad \frac{val\ e}{\mathsf{pr}\ (\mathsf{sc}\ e) =_\Lambda e}$$

$$conv\_ifz: \quad \mathsf{if0}\ 0\ e_2\ e_3 =_\Lambda e_2$$

$$conv\_ifsc: \quad \frac{val\ e}{\mathsf{if0}\ (\mathsf{sc}\ e)\ e_2\ e_3 =_\Lambda e_3}$$

$$conv\_refl: \quad e =_\Lambda e$$

$$conv\_sym: \quad \frac{e_2 =_\Lambda e_1}{e_1 =_\Lambda e_2}$$

$$conv\_trans: \quad \frac{e_1 =_\Lambda e_2 \qquad e_2 =_\Lambda e_3}{e_1 =_\Lambda e_3}$$

$$conv\_sc\_c: \quad \frac{e =_\Lambda e'}{\mathsf{sc}\ e =_\Lambda \mathsf{sc}\ e'}$$

$$conv\_pr\_c: \quad \frac{e =_\Lambda e'}{\mathsf{pr}\ e =_\Lambda \mathsf{pr}\ e'}$$

$$conv\_if\_c1: \quad \frac{e_1 =_\Lambda e_1'}{\mathsf{if0}\ e_1\ e_2\ e_3 =_\Lambda \mathsf{if0}\ e_1'\ e_2\ e_3}$$

$$conv\_if\_c2: \quad \frac{e_2 =_\Lambda e_2'}{\mathsf{if0}\ e_1\ e_2\ e_3 =_\Lambda \mathsf{if0}\ e_1\ e_2'\ e_3}$$

$$conv\_if\_c3: \quad \frac{e_3 =_\Lambda e_3'}{\mathsf{if0}\ e_1\ e_2\ e_3 =_\Lambda \mathsf{if0}\ e_1\ e_2\ e_3'}$$

$$conv\_lam\_c: \quad \frac{e =_\Lambda e'}{\mathsf{lam}\,x\,.\,e =_\Lambda \mathsf{lam}\,x\,.\,e'}$$

$$conv\_app\_c0: \quad \frac{e_0 =_\Lambda e_0'}{\mathsf{app}\ e_0\ e_1 =_\Lambda \mathsf{app}\ e_0\ e_1}$$

$$conv\_app\_c1: \quad \frac{e_1 =_\Lambda e_1'}{\mathsf{app}\ e_0\ e_1' =_\Lambda \mathsf{app}\ e_0\ e_1'}$$

Figure 17: Program calculus for Λ

Theorem 3 (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: To formalize the proof, we construct a function (*via* a relation as in Theorem 1) which builds a deduction showing e converts to e'.

```
t3 : spec W A -> bta E T W P -> bta E' T A P -> ans A P -> conv E E' -> type.
```

The `t3` judgement expresses a function which takes two proofs as input: a proof of `spec W A` and a proof of `bta E T W P`. The output judgement `bta E' T A P` acts as the unannotating function from Λ_{bt} terms `A` to Λ terms `E'` (see Section 4.1.3).

The output proof of `ans A P` is necessary for extracting value deductions to be used in the *conv_prsc* and *conv_ifsc* rules of the calculus. This requires a small lemma — essentially stating that if the specialization type `P` is static, then dropping the annotations on the answer `A` yields a value. In other words, given a proof `bta E T W P`, a proof `ans W P` that `W` is a value, and a proof that `P` is static, we can construct a proof that the de-annotation of `W` (given by `E`) is a value. This is formalized with the following judgement.

```
ans_to_val : bta E T W P -> ans W P -> static P -> val E -> type.
```

This proof is simple and omitted. The Elf encoding is given in Appendix B.2 along with the encoding of predicates describing static and dynamic specialization types.

Finally, the `conv E E'` judgement expresses the desired property that `E` converts to `E'`. A strategy similar to the one used in the proof of Theorem 1 gives the rules defining `t3` (see Appendix B.3).

Note that by including `bta E' T A P` and `ans A P` as we generate the deduction of `conv E E'`, we are in essence reproving Theorem 1. An alternative approach would be to use Theorem 1 to justify some of the steps in the proof below. This would be manifested in the Elf encoding as "calls" to the goal represented by the `t1` judgement. We choose the formalization below to simplify the presentation. ∎

7 Related work

Despeyroux first emphasized using deductive systems to define transformations [8]. 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 [18] 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. Other work using Elf to encode operational semantics includes Michaylov and Pfenning's encoding of call-by-value natural semantics and associated meta-theory for the language Mini-ML [23], Pfenning's representation of a Church-Rosser proof [30], Hannan's encoding of a type system for closure conversion [16], and Niss and the author's representation of the correctness proofs for the familiar translation of call-by-name to call-by-value using thunks [26].

Despeyroux [8, Section 8], and Hannan and Pfenning [18, Section 7], suggested that their methods could be used to specify "mixed computation" and partial evaluation, respectively. Hannan and Miller [17] 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 [18], and Michaylov and Pfenning [23] to mechanically verify correctness.

In doing so, we obtain verified proofs similar in scope to the ones given by Gomard and Jones [10, 11] 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 [27] and Wand [37] give detailed presentations of the correctness of binding-time analysis. Palsberg presents a generalization of Gomard and Jones criteria [11] for consistent binding-time annotations. Wand studies Mogensen's self-applicable partial evaluator [24] for the pure λ-calculus. His binding-time analysis is essentially the same as Gomard and Jones's, as well as 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 particular, once Wand has established the correctness results for an interpreter and partial evaluator, his proofs for the Futamura projections have a simple calculational style (based on equational reasoning). The proofs follow the same steps as the "Mix equations" [20, Chapter 1] which are often used to reason about the results of self-application. We are currently investigating how these reasoning principles can be formalized (*e.g.*, to obtain a "Mix equational theory") and encoded in LF.

In recent work, Davies and Pfenning [7] 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 since the Centaur environment is not oriented toward this purpose. 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

8.1 Summary

We have specified the main components of a simple 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.

8.2 Assessment

Using deductive systems to specify analyses and transformations: The simplicity of the specifications stems from their declarative nature (which allows inessential algorithmic details and model theory to be avoided). This does not mean that other formalisms such as denotational semantics are not useful. There are certainly situations where the extra reasoning capabilities offered by model theory are beneficial. However, it seems that in many applications — particularly in the area of program specialization — these extra capabilities are never utilized and other styles of formalization such as the ones used here may be more appropriate.

Using LF as a meta-language for defining specifications: Using a formal meta-language such as LF forces one to be precise, and such precision usually lends clarity to presentations. However, the rigidness of LF almost demands that one cast specifications as natural-deduction-style logics and use higher-order abstract syntax. This style of logic may not be appropriate in all situations. In addition, it would be interesting to see how our specifications and proofs would be formalized in other logical frameworks. Recent work suggests that it may be possible to generate program analyses from formal specifications. LF stands as a possible meta-language for formalizing such specifications.

Using Elf to obtain executable specifications: We found Elf particularly useful for prototyping the binding-time analysis. It should be investigated if this scales up to more complicated type-based analyses that use conjunctive types and more general forms of subtyping. More complex specifications may well be prone to looping due to the simplistic depth-first search strategy used by Elf. However, we have successfully used the techniques presented here to prototype a more complex higher-order version of the multi-level binding-time analysis of Glück and Jørgensen [9].

Using Elf to mechanically verify correctness: It seems possible to use these techniques to verify other forms of program specialization. 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* [20]. It remains to be seen if these techniques scale up to 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 [34].

The logic of Elf was not strong enough to encode the calculus soundness proof (see Theorem 2). More generally, the induction principles as required in the proofs of Theorems 1 and 3 could not be completely formalized. This manifested itself in the inabilty to mechanically guarantee the totality of the functions (defined as relations) representing the constructive proofs. Frameworks based on stronger logics (such as Isabelle-HOL and Isabelle-ZF [28], HOL [13], and Coq) can overcome these problems. Nipkow [25] has used Isabelle-HOL to encode a large portion of the theory given in Winskel's semantics textbook [39]. Similar techniques would be used in formalizing the calculus-soundness proof using model theory. Collins [4] has used HOL to encode a theory of bi-similarity for higher-order functional language. This could easily be extended to prove the soundness of the calculus using coinduction techniques. Finally, the forthcoming thesis of Welinder [38] contains a formalization of various principles required for reasoning about specialization of first-order functional programs.

Using stronger logics leads to tradeoffs. Proofs become much more involved (as formalization is more complete). Prototypes are harder to obtain directly from specifications.

Acknowledgements

John Hannan deserves special acknowledgement for his careful reading of earlier drafts of the paper. His detailed comments helped clarify the presentation. Thanks to Frank Pfenning for several enlightening conversations and for making his material on Elf easily accessible. Thanks to Olivier Danvy for his encouragement and comments on various drafts. Robert Glück, Kristian Nielsen, Henning Niss, Dave Savds, and other members of the DIKU TOPPS group gave valuable feedback and support. Thanks to the PLILP 95 referees for helpful comments.

References

[1] Yves Bertot and Ranan Fraer. Reasoning with executable specifications. In TAPSOFT'95 [35], pages 531–545.

[2] Sandrine Blazy and Philippe Facon. Formal specification and prototyping of a program specializer. In TAPSOFT'95 [35], pages 666–680.

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

[4] Graham Collins. A proof tool for reasoning about functional programs. Unpublished report, 1996.

[5] 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.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Ravi Sethi, editor, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[7] Rowan Davies and Frank Pfenning. A modal analysis of staged compuation. In *Proceedings of the Workshop on Types for Program Analysis*, Aarhus, Denmark, 1995.

[8] 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.

[9] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions. 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, Utrecht, The Netherlands, September 1995.

[10] Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[11] Carsten K. Gomard and Neil Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[12] Andrew Gordon. Bisimilarity as a theory of functional programming. In *Proceedings of the Eleventh Conference on the Mathematical Foundations of Programming Semantics*, number 1 in Electronic Notes in Computer Science. Elsevier, 1995.

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

[14] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

[15] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, 1993.

[16] John Hannan. Type systems for closure conversions. In *Proceedings of the Workshop on Types in Program Analysis*, 1995.

[17] 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.

[18] 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.

[19] 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.

[20] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[21] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *The Handbook of Logic in Computer Science.* North-Holland, 1992.

[22] 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.

[23] 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.

[24] 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.

[25] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. Unpublished report, 1996.

[26] Henning Niss and John Hatcliff. Encoding operational semantics in logical frameworks: A case study using LF/Elf. In Bengt Nördstrom, editor, *Proceedings of the 1995 Workshop on Programming Language Theory*, Göteborg, Sweden, November 1995.

[27] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, 1993.

[28] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

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

[36] Robert D. Tennent. *Semantics of Programming Languages.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1990.

[37] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, 1993.

[38] Morten Welinder. *Partial Evaluation and Correctness.* PhD thesis, University of Copenhagen, 1996. Submission expected August 1996.

[39] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 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.

%%% definition of t1 axiom and rules

% static constructs

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

t1_sc_s : t1 (spec_sc_s S) (bta_sc_s B) (bta_sc_s C) (ans_sc_s D)
            <- t1 S B C D.

t1_prz_s : t1 (spec_prz_s S) (bta_pr_s B) (bta_z_s) (ans_z_s)
             <- t1 S B (bta_z_s) D.

t1_prsc_s : t1 (spec_prsc_s S) (bta_pr_s B) C D
              <- t1 S B (bta_sc_s C) (ans_sc_s D).

t1_ifz_s : t1 (spec_ifz_s S2 S1) (bta_if_s B3 B2 B1) C2 D2
             <- t1 S1 B1 (bta_z_s) D1
             <- t1 S2 B2 C2 D2.

t1_ifsc_s : t1 (spec_ifsc_s S3 S1) (bta_if_s B3 B2 B1) C3 D3
              <- t1 S1 B1 (bta_sc_s C1) D1
              <- t1 S3 B3 C3 D3.

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_fix_s : t1 (spec_fix_s S) (bta_fix_s B) C D
             <- t1 S (B (fix E) (bfix s W) (bta_fix_s B)) C D.

% dynamic constructs
```

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

t1_sc_d : t1 (spec_sc_d S) (bta_sc_d B) (bta_sc_d C) (ans_sc_d D)
            <- t1 S B C D.

t1_pr_d : t1 (spec_pr_d S) (bta_pr_d B) (bta_pr_d C) (ans_pr_d D)
            <- t1 S B C D.

t1_if_d : t1 (spec_if_d S3 S2 S1) (bta_if_d B3 B2 B1)
            (bta_if_d C3 C2 C1) (ans_if_d D3 D2 D1)
            <- t1 S1 B1 C1 D1
            <- t1 S2 B2 C2 D2
            <- t1 S3 B3 C3 D3.

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_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_fix_d : t1 (spec_fix_d S) (bta_fix_d B) (bta_fix_d C) (ans_fix_d D)
        <- {x} {y} {S': spec y y}
           {B' : bta x (arrow T1 T2) 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.

t1_liftsc : t1 (spec_liftsc S2 S1) (bta_lift B1) (bta_sc_d C2) (ans_sc_d D2)
              <- t1 S1 B1 (bta_sc_s C1) D1
              <- t1 S2 (bta_lift C1) C2 D2.
```

B Soundness of specializer

B.1 Convertability relation

```
%%% convertability judgement

conv : exp -> exp -> type.

%%% convertability axioms and rules

% axioms

conv_beta : conv (app (lam E0) E1) (E0 E1).

conv_fix : conv (fix E) (E (fix E)).

conv_prz : conv (pr z) z.

conv_prsc : conv (pr (sc E)) E
           <- val E.
```

```
conv_ifz : conv (if z E2 E3) E2.

conv_ifsc : conv (if (sc E1) E2 E3) E3
            <- val E1.

% inference rules

conv_refl : conv E E.

conv_sym : conv E1 E2
           <- conv E2 E1.

conv_trans : conv E1 E3
             <- conv E1 E2
             <- conv E2 E3.

% context rules

conv_sc_c: conv (sc E) (sc E')
           <- conv E E'.

conv_pr_c: conv (pr E) (pr E')
           <- conv E E'.

conv_if_c1: conv (if E1 E2 E3) (if E1' E2 E3)
            <- conv E1 E1'.

conv_if_c2: conv (if E1 E2 E3) (if E1 E2' E3)
            <- conv E2 E2'.

conv_if_c3: conv (if E1 E2 E3) (if E1 E2 E3')
            <- conv E3 E3'.

conv_lam_c: conv (lam E1) (lam E2)
            <- {x} (conv (E1 x) (E2 x)).

conv_fix_c: conv (fix E1) (fix E2)
            <- {x} (conv (E1 x) (E2 x)).

conv_app_c0: conv (app E0 E1) (app E0' E1)
             <- conv E0 E0'.

conv_app_c1: conv (app E0 E1) (app E0 E1')
             <- conv E1 E1'.
```

B.2 Extracting value deductions from answer deductions

B.2.1 Predicates classifying specialization types

```
%%% judgements

static : styp T -> type.
dynamic : styp T -> type.

%%% static rules

static_nat : static sta.

static_arrow : static (barrow P1 Pw).
```

```
%%% dynamic rules

dynamic_dyn : dynamic dyn.
```

B.2.2 Function from static answers to values

```
%%% ans_to_val judgement

ans_to_val : bta E T W P -> ans W P -> static P -> val E -> type.

%%% ans_to_val rules

ans_to_val_z_s : ans_to_val bta_z_s ans_z_s static_nat val_z.

ans_to_val_sc_s : ans_to_val (bta_sc_s B) (ans_sc_s A) static_nat (val_sc V)
                  <- ans_to_val B A S V.

ans_to_val_lam_s : ans_to_val (bta_lam_s B) ans_lam_s static_arrow val_lam.
```

B.3 Proof of Theorem 3

```
%%% definition of t3 judgement

t3 : spec W A -> bta E T W P -> bta E' T A P -> ans A P -> conv E E' -> type.

%%% definition of t3 axiom and rules

% static constructs

t3_z_s : t3 (spec_z_s) (bta_z_s) (bta_z_s) (ans_z_s) (conv_refl).

t3_sc_s : t3 (spec_sc_s S) (bta_sc_s B) (bta_sc_s C) (ans_sc_s D) (conv_sc_c E)
          <- t3 S B C D E.

t3_prz_s : t3 (spec_prz_s S) (bta_pr_s B) (bta_z_s) (ans_z_s)
              (conv_trans (conv_prz) (conv_pr_c E))
              <- t3 S B (bta_z_s) D E.

t3_prsc_s : t3 (spec_prsc_s S) (bta_pr_s B) C D
               (conv_trans (conv_prsc V)
                           (conv_pr_c E))
                <- ans_to_val C D static_nat V
                <- t3 S B (bta_sc_s C) (ans_sc_s D) E.

t3_ifz_s : t3 (spec_ifz_s S2 S1) (bta_if_s B3 B2 B1) C2 D2
              (conv_trans E2 (conv_trans (conv_ifz) (conv_if_c1 E1)))
              <- t3 S1 B1 (bta_z_s) D1 E1
              <- t3 S2 B2 C2 D2 E2.

t3_ifsc_s : t3 (spec_ifsc_s S3 S1) (bta_if_s B3 B2 B1) C3 D3
               (conv_trans E3 (conv_trans (conv_ifsc V) (conv_if_c1 E1)))
               <- ans_to_val C1 D1 static_nat V
               <- t3 S1 B1 (bta_sc_s C1) (ans_sc_s D1) E1
               <- t3 S3 B3 C3 D3 E3.

t3_lam_s : t3 (spec_lam_s) (bta_lam_s B) (bta_lam_s B) (ans_lam_s) (conv_refl).

t3_app_s : t3 (spec_app_s S1 S0) (bta_app_s B1 B0) C D
```

```
              (conv_trans E1 (conv_trans (conv_beta) (conv_app_c0 E0)))
              <- t3 S0 B0 (bta_lam_s C0) D0 E0
              <- t3 S1 (C0 _ _ B1) C D E1.

t3_fix_s : t3 (spec_fix_s S) (bta_fix_s B) C D
              (conv_trans E (conv_fix))
              <- t3 S (B (fix E0) (bfix s W0) (bta_fix_s B)) C D E.

% dynamic constructs

t3_z_d : t3 (spec_z_d) (bta_z_d) (bta_z_d) (ans_z_d) (conv_refl).

t3_sc_d : t3 (spec_sc_d S) (bta_sc_d B) (bta_sc_d C) (ans_sc_d D) (conv_sc_c E)
             <- t3 S B C D E.

t3_pr_d : t3 (spec_pr_d S) (bta_pr_d B) (bta_pr_d C) (ans_pr_d D) (conv_pr_c E)
             <- t3 S B C D E.

t3_if_d : t3 (spec_if_d S3 S2 S1) (bta_if_d B3 B2 B1)
             (bta_if_d C3 C2 C1) (ans_if_d D3 D2 D1)
             (conv_trans (conv_if_c3 E3)
                         (conv_trans (conv_if_c2 E2)
                                     (conv_if_c1 E1)))
             <- t3 S1 B1 C1 D1 E1
             <- t3 S2 B2 C2 D2 E2
             <- t3 S3 B3 C3 D3 E3.

t3_lam_d : t3 (spec_lam_d S) (bta_lam_d B) (bta_lam_d C) (ans_lam_d D)
              (conv_lam_c E)
        <- {x} {y} {S': spec y y} {B' : bta x T3 y dyn} {D': ans y dyn}
                   {E': conv x x}
               t3 S' B' B' D' E'
                -> t3 (S y S') (B x y B') (C x y B') (D y D') (E x).

t3_app_d : t3 (spec_app_d S1 S0) (bta_app_d B1 B0)
              (bta_app_d C1 C0) (ans_app_d D1 D0)
              (conv_trans (conv_app_c1 E1) (conv_app_c0 E0))
              <- t3 S0 B0 C0 D0 E0
              <- t3 S1 B1 C1 D1 E1.

t3_fix_d : t3 (spec_fix_d S) (bta_fix_d B) (bta_fix_d C) (ans_fix_d D)
              (conv_fix_c E)
        <- {x} {y} {S': spec y y}
           {B' : bta x (arrow T1 T3) y dyn} {D': ans y dyn} {E': conv x x}
               t3 S' B' B' D' E'
                -> t3 (S y S') (B x y B') (C x y B') (D y D') (E x).

t3_lift : t3 (spec_lift S) (bta_lift B) (bta_z_d) (ans_z_d) E
             <- t3 S B (bta_z_s) D E.

t3_liftsc : t3 (spec_liftsc S2 S1) (bta_lift B1) (bta_sc_d C2) (ans_sc_d D2)
               (conv_trans (conv_sc_c E2) E1)
               <- t3 S1 B1 (bta_sc_s C1) D1 E1
               <- t3 S2 (bta_lift C1) C2 D2 E2.
```