J. Functional Programming **9** (5): 483–525, September 1999. Printed in the United Kingdom

Combinators for program generation

PETER THIEMANN

Institut für Informatik, Universität Freiburg,
Universitätsgelände Flugplatz, D-79110 Freiburg, Germany
(*e-mail:* thiemann@informatik.uni-freiburg.de)

Abstract

We present a general method to transform a compositional specification of a specializer for a functional programming language into a set of combinators that can be used to perform the same specialization more efficiently. The main transformation steps are the transition to higher-order abstract syntax and untagging. All transformation steps are proved correct. The resulting combinators can be implemented in any functional language, typed or untyped, pure or impure. They may also be considered as forming a domain-specific language for meta-programming. We demonstrate the generality of the method by applying it to several specializers of increasing strength. We demonstrate its efficiency by comparing it with a traditional specialization system based on self-application.

Capsule Review

The first partial evaluators (specializers) worked *interpretively*: given a program `p` and partial input `s`, the specializer executed those of `p`'s operations depending only on `s`, and generated residual code for the remainder. A recent trend is the *compilative* approach, building a *generating extension* `p-gen` from `p`. (The concept first arose from specializer self-application.) The generating extension, when run on `s` as input, yields the result of specializing `p` to known input `s`.

Surprise: building the generating extension, instead of writing an interpretive specializer, yields both faster specialization and better residual code quality, when specializing strongly typed programs. Their construction, however, remained something of a black art, and questions of correctness and reasons for these advantages were for some time obscure.

This paper gives insight into the nature of generating extensions, shows how systematically to derive one in combinator form from a compositional specializer, contains a number of correctness proofs, and is the basis of an implementation, whose empirical behavior is also described.

1 Introduction

A specializer for a functional programming language is often presented in a denotational style, that is, as a compositional interpreter of a two-level version of the language, where the interpreter itself is written in a functional programming language. In a two-level functional language (Nielson and Nielson, 1992; Jones *et al.*, 1993), every syntactic construct comes in two versions, one which is executed by the specializer "at specialization-time" and one for which the specializer generates

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

code. The former constructs are called *static* and the latter ones *dynamic*, they are often indicated by underlining or by annotating the standard construct with `D`.

Typically, we do not want to program in a two-level language. Instead, we leave the task of transforming an ordinary program of the source language into a two-level program to a *binding-time analysis* that inserts the annotations for us, given a classification of the free variables according to their binding time, i.e. static if their value is available at specialization-time and otherwise dynamic. This approach is called *offline partial evaluation* (Jones *et al.*, 1993; Consel and Danvy, 1993).

Using essentially an interpreter to perform specialization has a number of weaknesses that affect the efficiency of specialization:

- The interpreter performs syntax dispatch: it examines each expression to decide which construct it is and whether it is static or dynamic. Based on the outcome the interpreter chooses one particular branch to process the construct.
- The interpreter manipulates an explicit environment to implement the binding mechanism of the source language.
- If the interpreter is written in a statically typed language then it cannot manipulate values of the source language directly. Instead it must employ an encoding via a universal type (Launchbury, 1991).

The first two of these weaknesses can be avoided if the source language is also the implementation language of the specializer. In this case, we can specialize the specializer with respect to the source program and obtain a *generating extension* for the source program (Futamura, 1971). The generating extension takes the values classified as static as parameters and returns the source program specialized with respect to these values. Specializing with generating extensions is usually faster than specializing directly, typically by a factor of three to four in current practice (Bondorf, 1991). The cause of the speedup is the removal of the syntax dispatch and the resolution of some environment manipulations due to specialization.

We can also repeat the step and speed up the construction of the generating extension by creating a generating extension for the specializer first, i.e. by specializing the specializer with respect to itself (Turchin, 1979). The product is called *cogen* for historical reasons. It maps a two-level source program to its generating extension.

While specializing with a specialized specializer addresses the first two efficiency issues and works well in untyped languages, it is less satisfactory for typed languages. Each specialization step adds one level of encoding (in the universal type) to the program and the values. Therefore, a cogen takes a doubly encoded source program as an argument and the specialized programs work on encoded values, too, which is clearly undesirable (Launchbury, 1991).

Launchbury and Holst (1991) discovered a way out of this problem. Instead of writing a specializer, they propose to program the transformation from two-level source program to generating extension – a cogen – directly from scratch. A generating extension constructed thus does not require encoding of the values but it can use them directly. Hence, this approach addresses all three problems satisfactorily and it has been exercised successfully later on for a variety of languages (Birkedal

and Welinder, 1993;Glück and Jøergensen, 1995; Draves, 1996; Jøergensen and Leuschel, 1996; Andersen, 1994).

However, the construction of these "hand-written cogens" is *ad hoc*, and it is not clear why this approach solves the problems and what is the exact relation to the specializer.

1.1 Contribution

The contribution of this paper is the clarification of the advantages of writing a cogen by hand and a formal explanation of why it resolves the above-named weaknesses of the interpreter-based approach to specialization. To do so we exhibit a provably correct step-by-step method to *derive* the building bricks for generating extensions from a denotational specification of the specializer in a statically typed language. Each brick is actually a combinator that implements the specialization-time action of a particular construct of the two-level source language. As a consequence, the hand-written cogen becomes trivial: it just replaces each syntax constructor by the combinator that implements its specialization-time action. In other words, we re-interpret a two-level program as its own generating extension.

The proposed method is generally applicable to functional languages, no matter whether they are typed or untyped, pure or impure. It results in a highly efficient way of doing offline partial evaluation. The derived combinators are of independent interest because they can be used for meta-programming in a typed language. The method also provides guidance as how to come up with an efficient prototype implementation of a domain-specific language which is defined in a compositional style. For example, parser combinators (Hutton, 1992) could be derived from a parsing function using the same conceptual steps.

After introducing some notational preliminaries and fixing a programming language for our investigation (section 2), we present a simple specializer for the lambda calculus in section 3. The first transformation step removes the syntax dispatch (section 4). Even the correctness of this simple step requires some care, because different runs of the same specialization give only α-convertible specialized programs in general. The second transformation step in section 5 removes the explicit environment by introducing higher-order abstract syntax. Again, this transformation step involves a non-trivial correctness proof involving the semantics of the implementation language. In addition, we observe that higher-order syntax is also advantageous to use in the specialized program because it improves modularity and eases the implementation in pure languages. The third transformation step in section 6 removes the need for the universal type. The correctness of this step depends upon the binding-time analysis that was used to construct the two-level program. We show that the type checker of the implementation language rejects generating extensions with a binding-time mismatch and we formally relate the results computed before and after the transformation. Thus our combinators can be used for meta-programming in a typed language.

Although we introduce the transformation steps with a very simple specializer, section 7 demonstrates that the same methods apply to more powerful and practically

```
type ident = string
type ('a,'b) env = ('a * 'b) list
let rec lookup ((y,v)::r) x = if x = y then v else lookup r x
let upd r x y = (x,y)::r
let gensym_counter = ref 0
let gensym_reset () = gensym_counter := 0
let gensym () =
  begin
    incr gensym_counter;
    "x"^string_of_int (!gensym_counter)
  end
```

Fig. 1. Some library functions.

relevant specializers. Section 8 shows how to extend the combinators to encompass all syntactic constructs of a small functional language. It also discusses the issue of program point specialization. Section 9 shows the connection between the derived combinators, the recursive formulation, and the fold functionals associated with the abstract syntax of the two-level language. Section 10 documents that specialization using generating extensions constructed from the derived combinators is about three times faster than specialization using generating extensions constructed with a traditional cogen, even in the untyped language Scheme. Finally, we discuss related work in section 11, and conclude in section 12.

This paper grew out of earlier work on program generation for untyped languages (Thiemann, 1996a).

2 Preliminaries

We are using OCaml notation (Leroy, 1997) for all code in this paper. It is, however, straightforward to rewrite it for an untyped language like Scheme (Kelsey *et al.*, 1998) or for a pure language like Haskell (Haskell, 1997). The code has been type-checked with the OCaml system and all of it – except the code in section 7.2 – has been tested using OCaml. In the example runs, the system's prompt is `#` and all subsequent text up to the next `;;` comprises the input. Responses of the system have the form

`- :` ⟨*inferred type*⟩ `=` ⟨*value*⟩

The use of `typewriter font` marks concrete syntax in the text. Figure 1 defines some types and functions that are shared by all programs in the rest of the paper. The type `ident` is the type of identifiers, represented by strings. The type `('a, 'b) env` is the type of finite functions from `'a` to `'b`, which we will use as environments. The functions `lookup` and `upd` implement environment lookup and update. The `gensym` group of definitions implements a facility to generate unique strings. The main function is `gensym : unit -> string` which generates a new string to be used as a fresh identifier.

We write `e[f := g]` for the substitution of `g` for `f` in `e` where `e`, `f`, and `g` are syntactic objects. For functions g, $\mathrm{dom}(g)$ is the domain of g and $g[x \mapsto y]$ is defined by $g[x \mapsto y](x) = y$ and $g[x \mapsto y](z) = g(z)$ if $x \neq z$.

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
type exp =
    Var of ident
  | Lam of ident * exp
  | App of exp * exp
  | LamD of ident * exp
  | AppD of exp * exp

type value =
    Fun of (value -> value)
  | Code of exp

(* spec : exp -> (ident, value) env -> value *)
let rec spec e r =
  match e with
    Var x ->
      lookup r x
  | Lam (x, e) ->
      Fun (fun y -> spec e (upd r x y))
  | App (f, a) ->
      let Fun ff = spec f r in
      ff (spec a r)
  | LamD (x, e) ->
      let xx = gensym () in
      let Code body = spec e (upd r x (Code (Var xx))) in
      Code (Lam (xx, body))
  | AppD (f, a) ->
      let Code ff = spec f r in
      let Code aa = spec a r in
      Code (App (ff, aa))
```

Fig. 2. A fragment of Lambdamix.

3 A simple specializer

The starting point of our investigation is a simple specializer for the pure lambda calculus, in the spirit of Lambdamix (Gomard, 1992). Figure 2 shows its definition in denotational style, i.e. as an interpreter for two-level expressions in a functional programming language.

The datatype `exp` defines the abstract syntax of expressions in a two-level lambda calculus (Nielson and Nielson, 1992): there are variables `Var`, lambda abstractions `Lam`, applications `App`, and in addition, there are dynamic lambda abstractions `LamD` and applications `AppD`. FV(`e`) is the set of free variables of `e : exp` with the cases for `LamD` and `AppD` defined as for `Lam` and `App`.

The datatype `value` defines the domain of semantic values. A semantic value is either `Fun` f where f is a function mapping `value` to `value` or it is `Code` e where e is a piece of generated (specialized) code of type `exp` using only the `Var`, `Lam`, and `App` constructors.

The specializer `spec` is straightforward. It is a function that maps a two-level expression `e : exp` and an environment `r : (ident, value) env` to a value. The interpretation of `Var`, `Lam`, and `App` follows the standard semantics of lambda cal-

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

culus. The interpretation of a dynamic lambda generates a new identifier, then it specializes the body of the lambda, and finally it puts both together and returns a lambda abstraction. The case for the dynamic application specializes the subexpressions and puts them together to an application expression. Some care must be taken to properly inject into and project out of the `value` type. Here are some examples.

```
# spec (App (Lam ("x", (Var "x")), LamD ("x", (Var "x")))) [] ;;
- : value = Code (Lam ("x1", Var "x1"))
# spec (LamD ("x", LamD ("y", App (Lam ("x", (Var "x")), (Var "y"))))) [] ;;
- : value = Code (Lam ("x5", Lam ("x6", Var "x6")))
```

The environment provides static or dynamic values for free variables. First, we bind `"i"` to the static identity function, which works nicely, but then we bind it to a dynamic variable, which results in an error due to a binding-time mismatch: `"i"` is used as a function and the case for `App` tries to match the `Fun` constructor against `Code (Var "id")`.

```
# spec (LamD ("x", LamD ("y", App (Var "i", (Var "y")))))
  [("i", Fun (fun z -> z))] ;;
- : value = Code (Lam ("x1", Lam ("x2", Var "x2")))
# spec (LamD ("x", LamD ("y", App (Var "i", (Var "y")))))
  [("i", Code (Var "id"))] ;;
Uncaught exception: Match_failure("", 150, 156)
```

Although we could model this error explicitly by adding another alternative to the type `value`, we refrain from doing so to avoid further cluttering of the code.

Starting from this simple specializer we develop our library of combinators.

4 Combinators for specialization

Our declared aim is to re-interpret a two-level expression as its own generating extension. In the first step, we define for each syntax constructor a combinator that performs the associated specialization-time action. For example, the combinator for the constructor `Lam` is the right side of the definition of `spec (Lam (x, e))` after moving the parameters to the right side in the obvious way, i.e. `fun (x, e) -> fun r -> Fun (fun y -> e (upd r x y))`. If we name this function `lam` and proceed analogously for the remaining syntax constructors then we only have to replace the syntax constructors by these functions to obtain a generating extension. Figure 3 shows the resulting library of combinators. Here are the examples, rewritten to use the combinators:

```
# app (lam ("x", (var "x")), lamD ("x", (var "x"))) [] ;;
- : value = Code (Lam ("x8", Var "x8"))
# lamD ("x", lamD ("y", app (lam ("x", (var "x")), (var "y")))) [] ;;
- : value = Code (Lam ("x9", Lam ("x10", Var "x10")))
```

Just as with `spec`, the environment provides input and we can provoke errors by confusing the binding times.

```
# lamD ("x", lamD ("y", app (var "i", (var "y"))))
  [("i", Fun (fun z -> z))] ;;
- : value = Code (Lam ("x5", Lam ("x6", Var "x6")))
```

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
(*
   type envcomp = (ident, value) env -> value
   var  : ident             -> envcomp
   lam  : ident   * envcomp -> envcomp
   app  : envcomp * envcomp -> envcomp
   lamD : envcomp * envcomp -> envcomp
   appD : envcomp * envcomp -> envcomp
*)
let var x =
  fun r ->
    lookup r x
let lam (x, e) =
  fun r ->
    Fun (fun y -> e (upd r x y))
let app (f, a) =
  fun r ->
    let Fun ff = f r in
    ff (a r)
let lamD (x, e) =
  fun r ->
    let xx = gensym () in
    let Code body = e (upd r x (Code (Var xx))) in
    Code (Lam (xx, body))
let appD (f, a) =
  fun r ->
    let Code ff = f r in
    let Code aa = a r in
    Code (App (ff, aa))
```

Fig. 3. Simple combinators.

```
# lamD ("x", lamD ("y", app (var "i", (var "y"))))
  [("i", Code (Var "id"))] ;;
Uncaught exception: Match_failure("", 36, 42)
```

4.1 Correctness

Establishing the correctness means to show that for all e : `exp`, `spec` $e = e'$, where e' is obtained from e by replacing `Var` by `var`, `Lam` by `lam`, `App` by `app`, and so forth. Henceforth, we write this kind of replacement as e[`Var` := `var`,...].

The symbol = stands for equality of denotations in an arbitrary monadic model. That is, either both sides are undefined, or both sides are defined and have the same value. The notion of a monadic model specifically includes models with store effects as the implementation of `gensym` requires. The computational lambda calculus (Moggi, 1988) provides a suitable theory for transforming expressions in the presence of such monadic effects. We recall some axioms of the calculus, as far as we need them in the proofs.

- Extensionality: if e has function type then $e =$ `fun x -> e x`.
- βV: if v is a syntactic value then `(fun` x `->` e`)` $v = e[x := v]$.
- let$V$: if v is a syntactic value then `let` x `=` v `in` $e = e[x := v]$.

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

Informally, syntactic values are variables (since OCaml is call-by-value), constants, lambda abstractions, and data constructors applied to syntactic values. Using these equations, it is straightforward to transform `spec e` into `e'`. By extensionality, it suffices to prove the following lemma.

Lemma
Suppose

1. e : `exp`;
2. $e' = e$[`Var` := `var`,...];
3. `r : (ident, value) env`.

Then `spec` $e\ r = e'\ r$.

Proof
Straightforward induction on e. We show two representative cases.

Case `Var` x.

$$
\begin{array}{ll}
 & e'\ r \\
= & \\
 & \texttt{var}\ x\ r \\
= & \text{definition, } \beta V \\
 & \texttt{lookup}\ r\ x \\
= & \text{definition, } \beta V \\
 & \texttt{spec (Var}\ x\texttt{)}\ r
\end{array}
$$

Case `LamD (`x`, `e`)`.

```
    lamD (x, e') r
=   definition, βV
     let xx = gensym () in
     let Code body = e' (upd r x (Code (Var xx))) in
     Code (Lam (xx, body))
=   letV
     let xx = gensym () in
     let r' = upd r x (Code (Var xx)) in
     let Code body = e' r' in
     Code (Lam (xx, body))
=   by induction
     let xx = gensym () in
     let r' = upd r x (Code (Var xx)) in
     let Code body = spec e r' in
     Code (Lam (xx, body))
=   letV
     let xx = gensym () in
     let Code body = spec e (upd r x (Code (Var xx))) in
     Code (Lam (xx, body))
=   definition, βV
    spec (LamD (x, e)) r
```

□

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

4.2 Discussion

What have we gained in this step? We have got rid of the syntax dispatch during specialization: at each expression, `spec` has to determine the top-most constructor of its argument `e`, which selects the right side with which it has to proceed. Using the combinators eliminates this test-and-select part completely.

Preparing a generating extension in traditional style by specializing `spec` with respect to `e` would also remove the syntax dispatch. In addition, it would resolve the environment lookups because it would unfold the definitions of the combinators – we address this issue in the next section. However, the combinators evade the encoding of the program due to self-application.

5 Introducing higher-order abstract syntax

When we look back at the examples of the previous section, we find that the generating extensions are combinator terms with constants. However, they implement the binding constructs `Lam` and `LamD` and they do so by explicitly manipulating an environment `r`. It seems rather wasteful to do so in a functional programming language, the implementation of which is optimized towards handling bindings efficiently.

Higher-order abstract syntax (Pfenning and Elliott, 1988) is a technique conceived exactly for the purpose of moving the handling of binding operations to the meta-language (which is in our case the underlying functional language implementation). The idea is to represent binding constructs by syntax constructors that have arguments with functional type. For example, a higher-order syntax version of the `exp` datatype looks like this:

```
type hexp =
  HVar of ident
| HLam of (ident -> hexp)
| HApp of hexp * hexp
| HLamD of (ident -> hexp)
| HAppD of hexp * hexp
```

The two binding constructs each have a function as an argument that maps an `ident` to an `hexp`, that is, applied to an identifier, the argument function returns the body of the `HLam` or `HLamD` with the identifier substituted as appropriate. Here is the encoding of `Lam ("x", Var "x")` in higher-order abstract syntax:

```
HLam (fun x -> HVar x)
```

Figure 5 defines a function Φ that maps an `e : exp` into its higher-order representation and its inverse Ψ. Despite giving their definition in OCaml we will use them to express syntactic transformations in formal proofs. It is not hard to modify the specializer `spec` to process expressions in `hexp` instead of in `exp` (Thiemann, 1996a) and then convert the modified specializer to combinators as we did in the first step. We won't do that here, because it would divert from the straight course. Instead, we

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
(*
   hvar  : value            -> value
   hlam  : (value -> value) -> value
   happ  : value * value    -> value
   hlamD : (value -> value) -> value
   happD : value * value    -> value
*)
let hvar x =
  x
let hlam x_e =
  Fun (fun y -> x_e y)
let happ (f, a) =
  let Fun ff = f in
  ff a
let hlamD x_e =
  let xx = gensym () in
  let Code body = x_e (Code (Var xx)) in
  Code (Lam (xx, body))
let happD (f, a) =
  let Code ff = f in
  let Code aa = a in
  Code (App (ff, aa))
```

Fig. 4. Higher-order abstract syntax for source programs.

apply the idea directly to the combinators `var`, `lam`, etc., that we already have (see figure 3).

Figure 4 shows the result. The most important point is the disappearance of the environment `r`. In the interpretation of a variable, the binding operation in the implementation language has already replaced the variable by its *value*, so `hvar` is the identity function at type `value -> value`. The type of `hlam` is `(value -> value) -> value`, so the functional argument maps the value of a variable to the value of the body: it is itself the function. In principle, we could eta-reduce `fun y -> x_e y` to `x_e` but we leave it as it is to enable comparison with the specializers in section 7. `hlamD` works similarly to `hlam` and `happ` and `happD` are completely unsurprising. The types given for the combinators in figure 4 are not the most general ones, but they are the types at which we use the combinators in a generating extension.

Let's have a look at our examples:

```
# happ (hlam (fun x -> hvar x), hlamD (fun x -> hvar x)) ;;
- : value = Code (Lam ("x1", Var "x1"))
# hlamD (fun x -> hlamD (fun y -> happ (hlam (fun x -> hvar x), hvar y))) ;;
- : value = Code (Lam ("x2", Lam ("x3", Var "x3")))
```

Everything seems to work as before, we do not even have to supply an environment. But how do we specify static or dynamic input? Since there is no simulated environment, we have to bind the input values to "real" variables:

```
# let i = Fun (fun z -> z) in
  hlamD (fun x -> hlamD (fun y -> happ (hvar i, hvar y))) ;;
- : value = Code (Lam ("x4", Lam ("x5", Var "x5")))
```

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
let rec Φ e δ =
  match e with
    Var x       -> HVar (lookup δ x)
  | Lam (x, e)  -> HLam (fun y -> Φ e (upd δ x y))
  | App (f, a)  -> HApp (Φ f δ, Φ a δ)
  | LamD (x, e) -> HLamD (fun y -> Φ e (upd δ x y))
  | AppD (f, a) -> HAppD (Φ f δ, Φ a δ)

let rec Ψ h =
  match h with
    HVar x -> Var x
  | HLam x_e ->
      let xx = gensym () in
      Lam (xx, Ψ (x_e xx))
  | HApp (f, a) ->
      App (Ψ f, Ψ a)
  | HLamD x_e ->
      let xx = gensym () in
      LamD (xx, Ψ (x_e xx))
  | HAppD (f, a) ->
      AppD (Ψ f, Ψ a)
```

Fig. 5. Mapping first-order syntax into higher-order syntax and back.

$$[\![\cdot]\!] \ : \texttt{mlexpr} \to (\texttt{ident} \to \texttt{mlexpr}) \to \texttt{mlexpr}$$

$$\begin{array}{lcll}
[\![x]\!]\ \rho & = & \rho(x) & \\
[\![c]\!]\ \rho & = & c & \\
[\![\texttt{fun}\ x\ \texttt{->}\ e]\!]\ \rho & = & \texttt{fun x ->}\ [\![e]\!]\ \rho[x \mapsto \texttt{x}] & \text{fresh}(\rho, \texttt{x}) \\
[\![f\ \ a]\!]\ \rho & = & [\![f]\!]\ \rho\ ([\![a]\!]\ \rho) & \\
[\![\texttt{let}\ x\ \texttt{=}\ h\ \texttt{in}\ e]\!]\ \rho & = & \texttt{let x =}\ [\![h]\!]\ \rho\ \texttt{in}\ [\![e]\!]\ \rho[x \mapsto \texttt{x}] & \text{fresh}(\rho, \texttt{x}) \\
[\![\texttt{let}\ C\ x\ \texttt{=}\ h\ \texttt{in}\ e]\!]\ \rho & = & \texttt{let}\ C\ \texttt{x =}\ [\![h]\!]\ \rho\ \texttt{in}\ [\![e]\!]\ \rho[x \mapsto \texttt{x}] & \text{fresh}(\rho, \texttt{x})
\end{array}$$

`mlexpr` is the type of OCaml abstract syntax.
$\text{fresh}(\rho, x) = \forall y \in \text{dom}(\rho).F\rho(y \neq x)$

Fig. 6. Translation to environment-passing style.

```
# let i = Code (Var "id") in
  hlamD (fun x -> hlamD (fun y -> happ (hvar i, hvar y))) ;;
Uncaught exception: Match_failure("", 24, 30)
```

5.1 Correctness

Since the idea seems to work well in practice, we set out to develop a correctness proof for this step. To state the correctness requires us to relate the explicit environment `r` that is passed around by the `var`, `lam`, ... combinators with the implicit environment that the underlying implementation passes around for the `hvar`, `hlam`, ... combinators. We need to appeal to the semantics of the implementation language to establish such a relation.

Fortunately, for our purposes it is sufficient to define a translation from the im-

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

plementation language into itself that makes the environment manipulation explicit while leaving the other aspects (e.g. store manipulation) implicit. Figure 6 defines the part of this transformation that we need to state and prove the correctness result. As it stands, it essentially performs α-conversion. The cases cover variables x, constants c including constructors, lambda abstraction `fun` x `->` e, application $f\ a$, let expression, and let expression with pattern matching `let` $C\ x$ `=` h `in` e where C must be a data constructor. The translation is purely syntactic. For r `: (ident, value) env`, we write dom(r) for the set of all x `: ident` such that `lookup r x` is defined.

Theorem 1
Suppose

1. e `: exp` is a metavariable standing for an expression;
2. r `: (ident, value) env` with dom(`r`) $\supseteq FV$(`e`);
3. δ `: ident` $\to$ `ident` is an injective syntactic mapping;
4. ρ `: ident` $\to$ `mlexpr` is a syntactic environment;
5. $e' = e$[`Var` := `var`, ...];
6. $e'' = (\Phi\ e\ \delta)$[`HVar` := `hvar`, `HLam` := `hlam`, ...];
7. $\delta(\mathrm{dom}(r)) = \mathrm{dom}(\rho)$;
8. for all $x \in \mathrm{dom}(r)$, `lookup` $r\ x = \rho(\delta(x))$.

Then $e'\ r = [\![e'']\!]\ \rho$.

Proof
By induction on e. In the proof, we regard Φ, δ, ρ, and $[\![\cdot]\!]$ as compile-time entities that operate on syntax. In addition, we apply equations valid in the computational lambda calculus. Finally, we use `lookup (upd` $r\ x\ y$`)` $x = y$ and `lookup (upd` $r\ x\ y$`)` $z =$ `lookup` $r\ z$ if $x \neq z$.

We abbreviate $(\Phi\ e\ \delta)$[`HVar` := `hvar`, `HLam` := `hlam`, ...] by $\Phi^{\texttt{hvar}}\ e\ \delta$.

Case `Var` x. The left side transforms into the right side as follows:

$$
\begin{aligned}
& \texttt{var}\ x\ r \\
= \quad & \\
& (\texttt{fun r -> lookup r}\ x)\ r \\
= \quad & \\
& \texttt{lookup}\ r\ x \\
= \quad & \\
& \rho(\delta(x)) \\
= \quad & \\
& [\![\delta(x)]\!]\ \rho \\
= \quad & \\
& [\![\texttt{hvar}\ (\delta(x))]\!]\ \rho \\
= \quad & \\
& [\![\Phi^{\texttt{hvar}}\ (\texttt{Var}\ x)\ \delta]\!]\ \rho
\end{aligned}
$$

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

Case `Lam (`x`, `e`)`. The left side transforms into the right side as follows:

```
    lam (x, e) r
=
    (fun r -> Fun (fun y -> e (upd r x y))) r
=
    Fun (fun y -> e (upd r x y))
=     using the axioms for lookup and induction
    Fun (fun y -> [[Φ^hvar e δ[x ↦ y]]] ρ[y ↦ y])
=
    [[Fun (fun y -> (fun z -> Φ^hvar e δ[x ↦ z]) y)]] ρ
=
    [[hlam (fun z -> Φ^hvar e δ[x ↦ z])]] ρ
=
    [[Φ^hvar (Lam (x, e)) δ]] ρ
```

The inductive step requires that for all $w \in \mathrm{dom}(\texttt{upd}\ r\,x\,\texttt{y})$, `lookup (upd` $r\,x\,$`y)` $w =$ $\rho[\texttt{y} \mapsto \texttt{y}](\delta[x \mapsto \texttt{y}](w))$. For $w = x$, the left side is `lookup (upd` r x `y)` $x = \texttt{y}$ and the right side is $\rho[\texttt{y} \mapsto \texttt{y}](\delta[x \mapsto \texttt{y}](x)) = \rho[\texttt{y} \mapsto \texttt{y}](\texttt{y}) = \texttt{y}$. For $w \neq x$, the assumption for r, ρ, and δ applies.

Case `App (`f`, `a`)`. The left side transforms into the right side as follows:

```
    app (f, a) r
=
    (fun r -> let Fun ff = f r in ff (a r)) r
=
    let Fun ff = f r in ff (a r)
=     induction applied to a and f
    let Fun ff = [[Φ^hvar f δ]] ρ in ff ([[Φ^hvar a δ]] ρ[ff ↦ ff])
=
    [[let Fun ff = Φ^hvar f δ in ff Φ^hvar a δ]] ρ
=
    [[happ (Φ^hvar f δ, Φ^hvar a δ)]] ρ
=
    [[Φ^hvar (App (f, a)) δ]] ρ
```

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
let rec value2hvalue val d =
  match val with
    Fun f -> HFun (fun hv -> value2hvalue (f (hvalue2value hv d)) d)
  | Code e -> Φ e d
and hvalue2value hval d =
  match hval with
    HFun f -> Fun (fun v -> hvalue2value (f (value2hvalue v d)) d)
  | HCode e -> Ψ e
```

Fig. 7. Conversion from `hvalue` to `value` and back.

Case `LamD (`x`, `e`)`. The left side transforms as follows:

```
    lamD (x, e) r
=
                  (fun r ->
                  let xx = gensym () in
                  let Code body = e (upd r x (Code (Var xx))) in
                  Code (Lam (xx, body))) r
=
                  let xx = gensym () in
                  let Code body = e (upd r x (Code (Var xx))) in
                  Code (Lam (xx, body))
=    induction
           let xx = gensym () in
           let Code body = ⟦Φ^hvar e δ[x ↦ y]⟧ ρ[xx ↦ xx, y ↦ Code (Var xx)] in
           Code (Lam (xx, body))
=
     let xx = gensym () in
     let Code body =          (fun y -> ⟦Φ^hvar e δ[x ↦ y]⟧ ρ[xx ↦ xx, y ↦ y])
                              (Code (Var xx)) in
     Code (Lam (xx, body))
=
           let xx = gensym () in
           let Code body =
               ⟦(fun y -> Φ^hvar e δ[x ↦ y]) (Code (Var xx))⟧ ρ[xx ↦ xx] in
           Code (Lam (xx, body))
=
   ⟦hlamD (fun y -> Φ^hvar e δ[x ↦ y])⟧ ρ
=
   ⟦Φ^hvar (LamD (x, e)) δ⟧ ρ
```

The inductive step requires a calculation similar to that in Case `Lam (`x`, `e`)`.

Case `AppD (`f`, `a`)`. Identical to case `App (`f`, `a`)`. □

5.2 Generated code

It is also possible to use higher-order syntax for the generated code. Indeed, this leads to further simplification because it allows us to avoid the issue of generating new identifiers in the specializer. First, we have to redefine the set of semantic values to

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
(*
   hhvar  : hvalue             -> hvalue
   hhlam  : (hvalue -> hvalue) -> hvalue
   hhapp  : hvalue * hvalue    -> hvalue
   hhlamD : (hvalue -> hvalue) -> hvalue
   hhappD : hvalue * hvalue    -> hvalue
*)
let hhvar x =
  x
let hhlam x_e =
  HFun (fun y -> x_e y)
let hhapp (f, a) =
  let HFun ff = f in
  ff a
let hhlamD x_e =
  HCode (HLam (fun xx ->
    let HCode body = x_e (HCode (HVar xx)) in
    body))
let hhappD (f, a) =
  let HCode ff = f in
  let HCode aa = a in
  HCode (HApp (ff, aa))
```

Fig. 8. Higher-order syntax for specialized code.

```
type hvalue =
    HFun of (hvalue -> hvalue)
  | HCode of hexp
```

Next, we have to change the combinators. Except for the new constructor names, only the combinator for `LamD` really changes with respect to the `hvar` set. For reference, figure 8 shows all combinators in this style.

This form of output might be advantageous to perform subsequent program transformations on the generated code. However, if we do not have a clever compiler that deals directly with input in terms of `hexp` or if we want to print the generated code then we need to convert it to a first-order representation at some point. For these purposes we use the Ψ function from figure 5, and it is routine to show that the `hvar`-style combinators behave in the same way as the `hhvar`-style combinators composed with a conversion from `hvalue` to `value`.

Proposition 1
Suppose `e : exp` and

1. δ : `ident` $\rightarrow$ `ident` with $\mathrm{dom}(\delta) = FV(\mathtt{e})$ and $\delta(\mathtt{x}) = \mathtt{x}$ for all $\mathtt{x} \in FV(\mathtt{e})$;
2. `e'` $= (\Phi\ \mathtt{e}\ \delta)[\mathtt{HVar} := \mathtt{hvar}, \ldots]$;
3. `e''` $= (\Phi\ \mathtt{e}\ \delta)[\mathtt{HVar} := \mathtt{hhvar}, \ldots]$.

Then `e'` = `hvalue2value e''` δ where `hvalue2value` is defined in figure 7.

For the rest of the article we stick to this higher-order formulation, which is more concise.

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
(*
   huvar  : 'a              -> 'a
   hulam : ('a -> 'b)      -> ('a -> 'b)
   huapp  : ('a -> 'b) * 'a -> 'b
   hulamD : (hexp -> hexp)  -> hexp
   huappD : hexp * hexp     -> hexp
*)
let huvar x =
  x
let hulam x_e =
  fun y -> x_e y
let huapp (f, a) =
  f a
let hulamD x_e =
  HLam (fun xx -> x_e (HVar xx))
let huappD (f, a) =
  HApp (f, a)
```

Fig. 9. Combinators in higher-order syntax without tagging.

5.3 Discussion

The technique of this section, the transformation to higher-order abstract syntax, provides the means to replace the simulated (or interpreted) environment by the compiled binding mechanism of the underlying implementation. This step is a significant improvement over a cogen generated by self-application. Applied to the generated code, higher-order abstract syntax serves to separate the issues of code generation and name generation. This separation simplifies the implementation of the combinators in a pure language like Haskell, where name generation requires passing around a state argument (either explicitly or implicitly using a monad). In addition, it simplifies the correctness proof of the next transformation step.

6 Removing tags

The last of the alleged advantages of the handwritten-cogen approach is the fact that generating extensions produced by a handwritten cogen do not require a universal type. However, our last revision of the combinators still requires encoded input and it still performs tag manipulation. In this step, we remove the tags – again by lifting them to the meta-level – and discuss the implications.

It turns out that if we want to get rid of the universal type, we need to know something about the type of the expression that we specialize. Up to now, every combinator returns objects of type `value` and there is no restriction on the combination of the combinators. For example, the types of `happ` and `happD` are identical, as are the types of `hlam` and `hlamD`. It is easy to confuse them as the following expression shows:

```
happ (hlamD (fun x -> hvar x), hlamD (fun x -> hvar x))
```

This is an expression of type `value` which raises an exception when executed: `happ`

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

$$\text{(Var)}\quad \frac{\tau(\mathtt{x}) = t}{\tau \vdash \mathtt{Var\ x} : t}$$

$$\text{(Abstr)}\quad \frac{\tau[\mathtt{x} \mapsto t_2] \vdash \mathtt{e} : t_1}{\tau \vdash \mathtt{Lam\ (x,\ e)} : t_2 \to t_1}$$

$$\text{(Apply)}\quad \frac{\tau \vdash \mathtt{f} : t_2 \to t_1 \qquad \tau \vdash \mathtt{a} : t_2}{\tau \vdash \mathtt{App\ (f,\ a)} : t_1}$$

$$\text{(Abstr-dyn)}\quad \frac{\tau[\mathtt{x} \mapsto D] \vdash \mathtt{e} : D}{\tau \vdash \mathtt{LamD\ (x,\ e)} : D}$$

$$\text{(Apply-dyn)}\quad \frac{\tau \vdash \mathtt{f} : D \qquad \tau \vdash \mathtt{a} : D}{\tau \vdash \mathtt{App\ (f,\ a)} : D}$$

Fig. 10. Well-formed two-level expressions section 8.3.

expects its first argument to be `Fun ff` whereas `hlamD` returns `Code e`, for some expression `e`.

Removing the tagging from the `hvar` set of combinators leads to the combinators in figure 9. With this set of combinators, the type checker rejects the transcription of the above expression.

```
# huapp (hulamD (fun x -> huvar x), hulamD (fun x -> huvar x));;
Characters 7-59:
This expression has type hexp * hexp but is here used with
         type ('a -> 'b) * 'a
```

In fact, the expression which we tried to specialize has a binding-time mismatch: the outermost application is static, but its function argument is dynamic. Our observation of the type error leads to the proposition that a thus constructed generating extension is type correct if the underlying two-level expression obeys some well-formedness criterion. This proposition is indeed true and we turn to formalizing it.

6.1 Typability

First, we define when a two-level expression is well-formed. We adopt the criterion of Gomard (1992), who has formalized the well-formedness of two-level expressions using a partial type system (Gomard, 1990). In his system (extracted from (Jones *et al.*, 1993, section 8.3) and restricted to the present setting), the type language is

$$t \quad ::= \quad D \mid t \to t$$

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

and a type environment τ is a mapping from program variables to types. The type D is the type of an arbitrary dynamic expression, and $t_2 \to t_1$ is the type of a static function that maps values of type t_2 to values of type t_1.

Figure 10 defines the typing rules for the judgement $\tau \vdash \texttt{e} : t$ (in type environment τ, `e` is well-formed with type t). The rules for variables (Var), lambda abstraction (Abstr), and application (Apply) are standard for a simply typed lambda calculus. The rule (Abstr-dyn) requests that the variable and the body of a dynamic abstraction must be dynamic. The rule (Apply-dyn) states that both the function part `f` and the argument part `a` of a dynamic application must be dynamic. Gomard (1990) has shown that every lambda expression has a type in this system, the untypable parts receiving type D. Gomard (1992), Palsberg (1993) and Moggi (1998) address the semantic correctness of the type system with respect to `spec`, i.e. `spec` does not confuse `Code` with `Fun` on well-formed expressions.

For example, the term `LamD ("x", Var "x")` has type D, so the term `App (LamD ("x", Var "x"), LamD ("x", Var "x"))` is not typable in our system because the (Apply) rule expects the first expression to be of type $t_2 \to t_1$ whereas it has type D.

However, the term `App (Lam ("x", Var "x"), LamD ("x", Var "x"))` is typable with type D since $\tau \vdash$ `Lam ("x", Var "x")` $: D \to D$ is derivable.

With the above type system in place, we are left with defining a translation of the partial types t to ML types, before we can state the connection precisely.

Definition 1
The translation $\mathscr{T}$ maps partial types to ML types.

$$\begin{array}{lcl} \mathscr{T}\,[\![D]\!] & = & \texttt{hexp} \\ \mathscr{T}\,[\![t_2 \to t_1]\!] & = & \mathscr{T}\,[\![t_2]\!]\ \texttt{->}\ \mathscr{T}\,[\![t_1]\!] \end{array}$$

We will not state the typing rules of the implementation language, they are just the standard ML typing rules that prove judgements of the form $A \vdash_{\mathrm{ML}} \texttt{e} : \texttt{t}$ where `t` is a type defined by

$$\texttt{t} \ ::= \ \texttt{'a} \mid \texttt{int} \mid \texttt{hexp} \mid \texttt{t * t} \mid \texttt{t -> t}$$

We also use `'b` as an ML type variable.

Now we can state the connection between well-formedness of a two-level expression and typability of the corresponding generating extension.

Theorem 2
Suppose `e : exp` and

1. δ : `ident` $\to$ `ident` injective with $\mathrm{dom}(\delta) = FV(\texttt{e})$;
2. $\tau \vdash \texttt{e} : t$;
3. $\texttt{e'} = \Phi^{\texttt{huvar}}\ \texttt{e}\ \delta = (\Phi\ \texttt{e}\ \delta)[\texttt{HVar} := \texttt{huvar}, \ldots]$;
4. the type environment A is defined by $\mathrm{dom}(A) = \delta(\mathrm{dom}(\tau))$ and $A(\delta(\texttt{x})) = \mathscr{T}\,[\![\tau(\texttt{x})]\!]$ for all $\texttt{x} \in \mathrm{dom}(\tau)$.

Then $A \vdash_{\mathrm{ML}} \texttt{e'} : \mathscr{T}\,[\![t]\!]$.

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

Proof
By induction on `e`. The typing rules of the implementation language are the syntax-directed ML typing rules (Clément *et al.*, 1986). In the proof, we call the variable rule (ML-var), the application rule (ML-app), the abstraction rule (ML-abs), and the pair introduction rule (ML-pair).

Case `Var x`. If $\tau \vdash$ `Var x` $: t$ it must be due to the (Var) rule. Hence $\tau(\mathtt{x}) = t$. Since `e'` $= \delta(\mathtt{x})$ and $A(\delta(\mathtt{x})) = \mathscr{T}[\![t]\!]$ we conclude by the (ML-var) rule that $A \vdash_{\mathrm{ML}} \delta(\mathtt{x}) : \mathscr{T}[\![t]\!]$. With (ML-var) applied to `huvar : ∀ 'a . 'a -> 'a` and (ML-app) we finally get $A \vdash_{\mathrm{ML}}$ `huvar` $\delta(\mathtt{x}) : \mathscr{T}[\![t]\!]$.

Case `Lam (x, e)`. The last rule in the derivation of $\tau \vdash$ `Lam (x, e)` $: t$ must have been (Abstr):

$$\frac{\tau[\mathtt{x} \mapsto t_2] \vdash \mathtt{e} : t_1}{\tau \vdash \mathtt{Lam\ (x,\ e)} : t_2 \to t_1}$$

so $t = t_2 \to t_1$. Furthermore,

$$\begin{aligned} & \Phi^{\mathtt{huvar}}(\mathtt{Lam\ (x,\ e)})\delta \\ = \ & \mathtt{hulam\ (fun\ y\ ->}\ \Phi^{\mathtt{huvar}}\ \mathtt{e}\ \delta[\mathtt{x} \mapsto \mathtt{y}]\mathtt{)} \end{aligned}$$

Since $\delta[\mathtt{x} \mapsto \mathtt{y}](\mathtt{x}) = \mathtt{y}$, we can apply induction to get

$$A[\mathtt{y} \mapsto \mathscr{T}[\![t_2]\!]] \vdash_{\mathrm{ML}} \Phi^{\mathtt{huvar}}\ \mathtt{e}\ \delta[\mathtt{x} \mapsto \mathtt{y}] : \mathscr{T}[\![t_1]\!]$$

Applying the (ML-abs) rule results in

$$A \vdash_{\mathrm{ML}} \mathtt{fun\ y\ ->}\ \Phi^{\mathtt{huvar}}\ \mathtt{e}\ \delta[\mathtt{x} \mapsto \mathtt{y}] : \mathscr{T}[\![t_2]\!]\ \mathtt{->}\ \mathscr{T}[\![t_1]\!]$$

Using `hulam : ∀ 'a . ∀ 'b . ('a -> 'b) -> ('a -> 'b)`, the (ML-var) rule and the (ML-app) rule, we have that

$$A \vdash_{\mathrm{ML}} \mathtt{hulam\ (fun\ y\ ->}\ \Phi^{\mathtt{huvar}}\ \mathtt{e}\ \delta[\mathtt{x} \mapsto \mathtt{y}]\mathtt{)} : \mathscr{T}[\![t_2]\!]\ \mathtt{->}\ \mathscr{T}[\![t_1]\!]$$

We conclude by observing $\mathscr{T}[\![t_2 \to t_1]\!] = \mathscr{T}[\![t_2]\!]$ `->` $\mathscr{T}[\![t_1]\!]$.

Case `App (f, a)`. The last rule in the derivation of $\tau \vdash$ `App (f, a)` $: t$ must have been (Apply):

$$\frac{\tau \vdash \mathtt{f} : t_2 \to t \qquad \tau \vdash \mathtt{a} : t_2}{\tau \vdash \mathtt{App\ (f,\ a)} : t}$$

By induction,

$$A \vdash_{\mathrm{ML}} \Phi^{\mathtt{huvar}}\ \mathtt{f}\ \delta : \mathscr{T}[\![t_2 \to t]\!]$$

and

$$A \vdash_{\mathrm{ML}} \Phi^{\mathtt{huvar}}\ \mathtt{a}\ \delta : \mathscr{T}[\![t_2]\!].$$

Using $\mathscr{T}[\![t_2 \to t]\!] = \mathscr{T}[\![t_2]\!]$ `->` $\mathscr{T}[\![t]\!]$ and the (ML-pair) rule gets us

$$A \vdash_{\mathrm{ML}} (\Phi^{\mathtt{huvar}}\ \mathtt{f}\ \delta,\ \Phi^{\mathtt{huvar}}\ \mathtt{a}\ \delta) : (\mathscr{T}[\![t_2]\!]\ \mathtt{->}\ \mathscr{T}[\![t]\!])\ \mathtt{*}\ \mathscr{T}[\![t_2]\!]$$

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

Finally, apply (ML-var) using `huapp : ∀ 'a . ∀ 'b . ('a -> 'b) * 'a -> 'b` and the (ML-app) rule to get

$$A \vdash_{\mathrm{ML}} \texttt{huapp (} \Phi^{\texttt{huvar}}\ \texttt{f}\ \delta\texttt{, } \Phi^{\texttt{huvar}}\ \texttt{a}\ \delta\texttt{)} : \mathscr{T}\,[\![t]\!]$$

which concludes this case.

Case `LamD (x, e)`. The last rule in the derivation must have been (Abstr-dyn)

$$\frac{\tau[\texttt{x} \mapsto D] \vdash \texttt{e} : D}{\tau \vdash \texttt{LamD (x, e)} : D}$$

Furthermore,

$$\begin{aligned} & \Phi^{\texttt{huvar}}\ \texttt{(LamD (x, e))}\ \delta \\ = \quad & \texttt{hulamD (fun y -> } \Phi^{\texttt{huvar}}\ \texttt{e}\ \delta[\texttt{x} \mapsto \texttt{y}]\texttt{)} \end{aligned}$$

Now, by induction and since $\delta[\texttt{x} \mapsto \texttt{y}](\texttt{x}) = \texttt{y}$ and $\mathscr{T}\,[\![D]\!] = \texttt{hexp}$

$$A[\texttt{y} \mapsto \texttt{hexp}] \vdash_{\mathrm{ML}} \Phi^{\texttt{huvar}}\ \texttt{e}\ \delta[\texttt{x} \mapsto \texttt{y}] : \texttt{hexp}$$

Applying the (ML-abs) rule results in

$$A \vdash_{\mathrm{ML}} \texttt{fun y -> } \Phi^{\texttt{huvar}}\ \texttt{e}\ \delta[\texttt{x} \mapsto \texttt{y}] : \texttt{hexp -> hexp}$$

and using `hulamD : (hexp -> hexp) -> hexp` and the (ML-app) rule, we have that

$$A \vdash_{\mathrm{ML}} \texttt{hulamD (fun y -> } \Phi^{\texttt{huvar}}\ \texttt{e}\ \delta[\texttt{x} \mapsto \texttt{y}]\texttt{)} : \texttt{hexp}$$

We conclude by observing again that $\mathscr{T}\,[\![D]\!] = \texttt{hexp}$.

Case `AppD (f, a)`. (Similar to case `App (f, a)`) The last rule in the derivation of $\tau \vdash \texttt{App (f, a)} : D$ must have been (Apply-dyn):

$$\frac{\tau \vdash \texttt{f} : D \qquad \tau \vdash \texttt{a} : D}{\tau \vdash \texttt{AppD (f, a)} : D}$$

By induction,

$$A \vdash_{\mathrm{ML}} \Phi^{\texttt{huvar}}\ \texttt{f}\ \delta : \texttt{hexp}$$

and

$$A \vdash_{\mathrm{ML}} \Phi^{\texttt{huvar}}\ \texttt{a}\ \delta : \texttt{hexp}.$$

Since `huappD : hexp * hexp -> hexp` the (ML-pair) rule and the (ML-app) rule are applicable leading to

$$A \vdash_{\mathrm{ML}} \texttt{huappD (} \Phi^{\texttt{huvar}}\ \texttt{f}\ \delta\texttt{, } \Phi^{\texttt{huvar}}\ \texttt{a}\ \delta\texttt{)} : \texttt{hexp}$$

which concludes this case. □

The proof demonstrates that the implementation language must be polymorphic. Otherwise, there would be restrictions on the use of `huvar`, `hulam`, and `huapp` that would render them useless.

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

6.2 Correctness

It remains to show that the `hhvar` and the `huvar` combinators indeed compute "the same" answer. Since their types are different, we have to look for a suitable relation between the results of using them. It turns out that we can use techniques developed for proving representation independence to establish such a relation. To this end, we define a relation $\sim$ which is indexed by partial types. It is reminiscent of a logical relation (Mitchell, 1996, chapter 8).

To define the relation, let H `: hvalue`, k, h `: hexp`, F `: hvalue`, f `: hvalue -> hvalue`, and $g : \mathscr{T}[\![t_2 \to t_1]\!]$.

- $H \sim_D h$ iff $H =$ `HCode` k and $k = h$ using equality in the computational lambda calculus, and
- $F \sim_{t_2 \to t_1} g$ iff $F =$ `HFun` f and, for all x `: hvalue`, $y : \mathscr{T}[\![t_2]\!]$, $x \sim_{t_2} y$ implies $fx \sim_{t_1} gy$.

If we show that a generating extension constructed with the `hhvar` combinators is related by $\sim_D$ to a `huvar`-generating extension for the same two-level program then we know that both yield the same specialized program.

Theorem 3
Suppose

1. $\tau \vdash$ `e` $: t$;
2. $\delta_1, \delta_2 :$ `ident` $\to$ `ident`;
3. $FV($`e`$) \subseteq \mathrm{dom}(\delta_1)$;
4. $FV($`e`$) \subseteq \mathrm{dom}(\delta_2)$;
5. for all $x \in FV($`e`$)$ with $\tau(x) = t'$, $\delta_1(x) \sim_{t'} \delta_2(x)$.

Then $\Phi^{\mathtt{hhvar}}$ `e` $\delta_1 \sim_t \Phi^{\mathtt{huvar}}$ `e` δ_2.

Proof
We use induction on `e`.

Case `Var` x. In this case, $\tau(x) = t$ by rule (Var), that is $t' = t$.

$$
\begin{array}{ll}
 & \Phi^{\mathtt{hhvar}}\ \mathtt{(Var}\ x\mathtt{)}\ \delta_1 \\
= & \\
 & \mathtt{hhvar}\ (\delta_1(x)) \\
= & \\
 & \delta_1(x) \\
\sim_t & \quad \text{by assumption} \\
 & \delta_2(x) \\
= & \\
 & \mathtt{huvar}\ (\delta_2(x)) \\
= & \\
 & \Phi^{\mathtt{huvar}}\ \mathtt{e}\ \delta_2
\end{array}
$$

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

Case `Lam (x, e)`. In this case, $t = t_2 \to t_1$ and $\tau[x \mapsto t_2] \vdash e : t_1$ by rule (Abstr).

$$\begin{aligned}
&\Phi^{\mathtt{hhvar}}\ (\mathtt{Lam}\ (x,\ e))\ \delta_1\\
=\\
&\mathtt{hhlam}\ (\mathtt{fun\ x\ ->}\ \Phi^{\mathtt{hhvar}}\ e\ \delta_1[x \mapsto \mathtt{x}])\\
=\\
&\mathtt{HFun}\ (\mathtt{fun\ x\ ->}\ (\mathtt{fun\ x\ ->}\ \Phi^{\mathtt{hhvar}}\ e\ \delta_1[x \mapsto \mathtt{x}])\ \mathtt{x})\\
=\\
&\mathtt{HFun}\ (\mathtt{fun\ x\ ->}\ \Phi^{\mathtt{hhvar}}\ e\ \delta_1[x \mapsto \mathtt{x}])
\end{aligned}$$

Supposing that $\mathtt{x} \sim_{t_2} \mathtt{y}$, induction yields that

$$\Phi^{\mathtt{hhvar}}\ e\ \delta_1[x \mapsto \mathtt{x}] \sim_{t_1} \Phi^{\mathtt{huvar}}\ e\ \delta_2[x \mapsto \mathtt{y}].$$

Therefore, appealing to βV yields

$$(\mathtt{fun\ x\ ->}\ \Phi^{\mathtt{hhvar}}\ e\ \delta_1[x \mapsto \mathtt{x}])\ \mathtt{x} \sim_{t_1} (\mathtt{fun\ y\ ->}\ \Phi^{\mathtt{huvar}}\ e\ \delta_2[x \mapsto \mathtt{y}])\ \mathtt{y}$$

and the proof proceeds as follows by definition of $\sim_{t_2 \to t_1}$:

$$\begin{aligned}
&\mathtt{HFun}\ (\mathtt{fun\ x\ ->}\ \Phi^{\mathtt{hhvar}}\ e\ \delta_1[x \mapsto \mathtt{x}])\\
\sim_{t_2 \to t_1}\\
&\mathtt{fun\ y\ ->}\ \Phi^{\mathtt{huvar}}\ e\ \delta_2[x \mapsto \mathtt{y}]\\
=\\
&\mathtt{hulam}\ (\mathtt{fun\ y\ ->}\ \Phi^{\mathtt{huvar}}\ e\ \delta_2[x \mapsto \mathtt{y}])\\
=\\
&\Phi^{\mathtt{huvar}}\ (\mathtt{Lam}\ (x,\ e))\ \delta_2
\end{aligned}$$

Case `App (f, a)`. In this case, $t = t_1$ and rule (Apply) yields that $\tau \vdash f : t_2 \to t_1$ and $\tau \vdash a : t_2$.

$$\begin{aligned}
&\Phi^{\mathtt{hhvar}}\ (\mathtt{App}\ (f,\ a))\ \delta_1\\
=\\
&\mathtt{hhapp}\ (\Phi^{\mathtt{hhvar}}\ f\ \delta_1,\ \Phi^{\mathtt{hhvar}}\ a\ \delta_1)\\
=\\
&\mathtt{let\ HFun\ ff\ =}\ \Phi^{\mathtt{hhvar}}\ f\ \delta_1\ \mathtt{in\ ff}\ (\Phi^{\mathtt{hhvar}}\ a\ \delta_1)
\end{aligned}$$

By induction, $\Phi^{\mathtt{hhvar}}\ f\ \delta_1 \sim_{t_2 \to t_1} \Phi^{\mathtt{huvar}}\ f\ \delta_2$ and $\Phi^{\mathtt{hhvar}}\ a\ \delta_1 \sim_{t_2} \Phi^{\mathtt{huvar}}\ a\ \delta_2$. Therefore, by definition of $\sim_{t_2 \to t_1}$, if $\Phi^{\mathtt{hhvar}}\ f\ \delta_1 = \mathtt{HFun\ ff}$ then $\mathtt{ff}\ (\Phi^{\mathtt{hhvar}}\ a\ \delta_1) \sim_{t_1} \Phi^{\mathtt{huvar}}\ f\ \delta_2(\Phi^{\mathtt{huvar}}\ a\ \delta_2)$. That is,

$$\begin{aligned}
&\mathtt{let\ HFun\ ff\ =}\ \Phi^{\mathtt{hhvar}}\ f\ \delta_1\ \mathtt{in\ ff}\ (\Phi^{\mathtt{hhvar}}\ a\ \delta_1)\\
\sim_{t_1}\\
&\Phi^{\mathtt{huvar}}\ f\ \delta_2(\Phi^{\mathtt{huvar}}\ a\ \delta_2)\\
=\\
&\mathtt{huapp}\ (\Phi^{\mathtt{huvar}}\ f\ \delta_2,\ \Phi^{\mathtt{huvar}}\ a\ \delta_2)\\
=\\
&\Phi^{\mathtt{huvar}}\ (\mathtt{App}\ (f,\ a))\ \delta_2
\end{aligned}$$

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

Case `LamD (x, e)`. In this case, $t = D$ and rule (Abstr-dyn) yields that $\tau[x \mapsto D] \vdash e : D$.

```
    Φ^hhvar (LamD (x, e)) δ1
=
    hhlamD (fun x -> Φ^hhvar e δ1[x ↦ x])
=
          HCode (HLam (fun xx ->
            let HCode body = (fun x -> Φ^hhvar e δ1[x ↦ x]) (HCode (HVar xx))
            in body))
```

Given `x` $\sim_D$ `y`, induction yields that

$$\Phi^{\mathtt{hhvar}}\ e\ \delta_1[x \mapsto \mathtt{x}] \sim_D \Phi^{\mathtt{huvar}}\ e\ \delta_1[x \mapsto \mathtt{y}].$$

Since `HCode (HVar xx)` $\sim_D$ `HVar xx`, this means

```
                (fun x -> Φ^hhvar e δ1[x ↦ x]) (HCode (HVar xx))
          ∼D
                (fun y -> Φ^huvar e δ2[x ↦ y]) (HVar xx)
```

If `(fun x -> Φ^hhvar e δ1[x ↦ x]) (HCode (HVar xx))` = `HCode body` then

```
HCode body ∼D (fun y -> Φ^huvar e δ2[x ↦ y]) (HVar xx)
```

Hence,

```
            HCode (HLam (fun xx ->
              let HCode body = (fun x -> Φ^hhvar e δ1[x ↦ x]) (HCode (HVar xx))
              in body))
∼D
    HLam (fun xx -> body)
=
    HLam (fun xx -> (fun y -> Φ^huvar e δ2[x ↦ y]) (HVar xx))
=
    hulamD (fun y -> Φ^huvar e δ2[x ↦ y])
=
    Φ^huvar (LamD (x, e)) δ2
```

Case `AppD (f, a)`. Similar. □

6.3 Discussion

We have shown that removing the tagging operations from the generating extension does not affect its typability. We found that polymorphic type-checking of the generating extension is an additional test for the well-formedness of the underlying two-level expression. In consequence, it is safe to compile generating extensions with type checking turned off if the well-formedness has been established.

As an additional benefit, the mapping from annotated expressions to a combinator expression may be improved to drop `huvar`, `hulam`, and `huapp` and directly use variables, lambda abstractions, and applications of the implementation language. In this case, a simply typed implementation language is sufficient.

Alternatively, if we adopt the improved mapping, polymorphic type-checking

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

of the translated two-level expression gives rise to an improved well-formedness criterion. Actually, there are combinator terms composed from the `huvar` combinators which are ML-typable despite being rejected by Gomard's type system for well-formedness. An example of such a term is

```
# let id = fun x -> x in
  (id (fun x -> x)) (id (hulamD (fun x -> x)));;
- : exp = Lam ("x1", Var "x1")
```

Therefore, we might *define* well-formedness of a two-level expression via ML-typability of its improved translation to the `huvar` combinators.

Finally, we have reaped some benefit from the higher-order representation of the output. If the combinators had involved name generation then establishing a relation between the two formulations would have been substantially harder.

7 Generalizing

In the past three sections, we have witnessed a transformation from a simple specializer into equally powerful, but more efficient combinators for program generation. Using these combinators, we can eliminate the syntax dispatch in the specializer, the cost of an explicit environment, and finally the cost of tagging and untagging operations.

But we started from a very simple specializer! Does the same technique work for other styles of offline partial evaluation? Indeed, it works for all partial evaluators that have a compositional specification. Since Lambdamix-style specialization is of little practical use, the rest of this section discusses the transformation for two examples that extend the capabilities of the simple specializer: continuation-based partial evaluation (Bondorf, 1992) and continuation-based partial evaluation in direct style (Lawall and Danvy, 1994). Continuation-based partial evaluators and generalizations thereof are the method of choice for specializing pure and impure functional programs (Dussart and Thiemann1, 1997; Thiemann, 1998).

7.1 Continuation-based partial evaluation

Suppose we add a `Let` expression to the syntax of two-level expressions and extend the Lambdamix specializer (Fig. 2) so that `Let (x, h, e)` is equivalent to `App (Lam (x, e), h)` and `LetD (x, h, e)` is equivalent to `AppD (LamD (x, e), h)`, the difference being that `LetD (x, h, e)` generates a `Let` expression instead of a β-redex:

```
let rec spec e r =
  match e with
  ...
  | Let (x, h, e) ->
      spec e (upd r x (spec h r))
  | LetD (x, h, e) ->
```

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
    let xx = gensym () in
    let Code hh = spec h r in
    let Code ee = spec e (upd r x (Code (Var xx))) in
    Code (Let (xx, hh, ee))
```

Let's try to specialize something:

```
# spec (LamD ("q",
              App (LetD ("id", AppD (Var "q", Var "q"),
                         Lam ("z", Var "z")),
                   Var "q"))) [];;
Uncaught exception: Match_failure("", 578, 585)
```

Evidently, this term has a binding-time mismatch! The intervening `LetD` expression stops us from reducing `App (Lam ("z", Var "z"), Var "q")`. However, the unannotated expressions `App (Let ("id", App (Var "q", Var "q"), Lam ("z", Var "z")), Var "q")` and `Let ("id", AppD (Var "q", Var "q"), App (Lam ("z", Var "z"), Var "q"))` cannot be distinguished in the call-by-value lambda calculus[1] and the two-level version of the latter specializes successfully:

```
# spec (LamD ("q",
             LetD ("id", AppD (Var "q", Var "q"),
                   App (Lam ("z", Var "z"),
                        Var "q")))) [];;
- : value =
Code (Lam ("x13", Let ("x14", App (Var "x13", Var "x13"), Var "x13")))
```

Continuation-based specialization grew out of the desire to treat expressions like the two considered above in the same way, by somehow dragging the specialization of the context of a `LetD` expression into its body. Continuations are a means to achieve this feat and Bondorf (Bondorf, 1992) defines a specializer using continuations that treats our two expressions in the same way.

Figure 11 shows the essential part of Bondorf's specializer. The domain of semantic values `cvalue` changes as expected for a continuation semantics (Schmidt, 1986; Mosses, 1990). The specializer `cspec` now maps an expression `e : exp`, an environment `r : (ident, cvalue) env`, and a continuation `c : (cvalue -> cvalue)` to a result of type `cvalue`.

The interpretation of `Var`, `Lam`, and `App` is just like in a continuation semantics for a call-by-value lambda calculus (omitting the trivial `Let`) (Schmidt, 1986; Mosses, 1990). The interpretation of the dynamic constructs is slightly different: `LamD` specializes its body using the identity continuation `(fun z -> z)` and passes the constructed `Lam` to its continuation, `AppD` is treated like a primitive operation in a continuation semantics, and `LetD` first specializes the header `h` then it constructs the `Let` expression and specializes its body `e` using its own continuation. Since the continuation `c` performs the specialization of the context of the `LetD`, this specializer essentially generates the specialized code for the context *inside* the body of the generated `Let` expression.

[1] They are observationally equivalent, but cannot be proved equal in the call-by-value lambda calculus. However, they can be proved equal in the computational lambda calculus (Sabry and Felleisen, 1993).

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
type cvalue =
    CFun of (cvalue -> (cvalue -> cvalue) -> cvalue)
  | CCode of exp

(* cspec : exp -> (ident, cvalue) env -> (cvalue -> cvalue) -> cvalue *)
let rec cspec e r c =
  match e with
    Var x ->
      c (lookup r x)
  | Lam (x, e) ->
      c (CFun (fun y -> cspec e (upd r x y)))
  | App (f, a) ->
      cspec f r (fun (CFun ff) ->
        cspec a r (fun aa ->
          ff aa c))
  | LamD (x, e) ->
      let xx = gensym () in
      let CCode body = cspec e (upd r x (CCode (Var xx))) (fun z -> z) in
      c (CCode (Lam (xx, body)))
  | AppD (f, a) ->
      cspec f r (fun (CCode ff) ->
        cspec a r (fun (CCode aa) ->
          c (CCode (App (ff, aa)))))
  | LetD (x, h, e) ->
      let xx = gensym () in
      cspec h r (fun (CCode hh) ->
        let CCode ee = cspec e (upd r x (CCode (Var xx))) c in
        CCode (Let (xx, hh, ee)))
```

Fig. 11. Bondorf's specializer using continuations (Bondorf, 1992, figures 4–6).

This devious treatment of the `LetD` cures exactly the problem that we had before. With this specializer, our example works out fine:

```
# cspec (LamD ("q", App (LetD ("id", AppD (Var "q", Var "q"),
                                Lam ("z", Var "z")),
                         Var "q"))) [] (fun x -> x);;
- : cvalue =
CCode (Lam ("x1", Let ("x2", App (Var "x1", Var "x1"), Var "x1")))
```

The specializer `cspec` looks considerably more involved than the simple Lambdamix specializer from Fig. 2. However, all three transformation steps work just the same as before. Figure 12 shows the result. Most combinators are by now familiar, except the `hucletD` combinator: it takes a pair corresponding to the header expression `h` and a function `x_e` that maps the binding for the variable to the specialization function for the body. The transcription to the new combinators of our example shows the use of `hucletD` and it demonstrates that the transformation has preserved the special behavior of `cspec` (`outHOAS` is the OCaml name for Ψ):

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
(*
   type 'a comp = ('a -> hexp) -> hexp
   hucvar  : 'a                              -> 'a comp
   huclam  : ('a -> 'b comp)                 -> ('a -> 'b comp) comp
   hucapp  : ('a -> 'b comp) comp * 'a comp -> 'b comp
   huclamD : (hexp -> hexp comp)             -> hexp comp
   hucappD : hexp comp * hexp comp           -> hexp comp
   hucletD : hexp comp * (hexp -> 'a comp)   -> 'a comp
*)
let hucvar x =
  fun c ->
    c x
let huclam x_e =
  fun c ->
    c (fun y -> x_e y)
let hucapp (f, a) =
  fun c ->
    f (fun ff ->
      a (fun aa ->
        ff aa c))
let huclamD x_e =
  fun c ->
    c (HLam (fun xx -> x_e (HVar xx) (fun z -> z)))
let hucappD (f, a) =
  fun c ->
    f (fun ff ->
      a (fun aa ->
        c (HApp (ff, aa))))
let hucletD (h, x_e) =
  fun c ->
    h (fun hh ->
      HLet (hh, fun xx -> x_e (HVar xx) c))
```

Fig. 12. Combinators from Bondorf's specializer.

$$\text{(Let-dyn)}\quad \frac{\tau \vdash' \texttt{h} : D \qquad \tau[\texttt{x} \mapsto D] \vdash' \texttt{e} : t}{\tau \vdash' \texttt{LetD (x, h, e)} : t}$$

Fig. 13. Well-formedness of `LetD` for `cspec`.

```
# outHOAS
  (huclamD (fun q ->
     hucapp (hucletD (hucappD (hucvar q, hucvar q),
                      fun id -> huclam (fun z -> hucvar z)),
             hucvar q)) (fun x -> x));;
- : exp = Lam ("x3", Let ("x4", App (Var "x3", Var "x3"), Var "x3"))
```

Following the transformations, we can prove adaptions of Theorem 5.1 (the transformation to higher-order abstract syntax) and Theorem 6.1 (the connection between well-formedness and typability of the generating extension).

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

While statement and proof of Theorem 5.1 only require routine changes (and therefore, we do not state it here), we need some new definitions to state the adapted Theorem 6.1.

Figure 13 defines a typing rule for the `LetD` which is suitable for `cspec` (). The remaining rules for the judgement $\tau \vdash' \texttt{e} : t$ are the ones from figure 10 with $\vdash$ replaced by $\vdash'$.

In addition, the translation from partial types to ML types needs to take continuation-passing into account. This is a standard transformation on types (Hatcliff and Danvy, 1994).

Definition 2
The translations $\mathscr{V}$ and $\mathscr{C}$ map partial types into ML types assuming a Plotkin-style transformation to call-by-value continuation-passing style (CPS).

$$\begin{array}{lcl} \mathscr{V}[\![D]\!] & = & \texttt{hexp} \\ \mathscr{V}[\![t_2 \to t_1]\!] & = & \mathscr{V}[\![t_2]\!] \texttt{ -> } \mathscr{C}[\![t_1]\!] \\ \\ \mathscr{C}[\![t]\!] & = & \texttt{(}\mathscr{V}[\![t]\!] \texttt{ -> hexp) -> hexp} \quad = \quad \mathscr{V}[\![t]\!] \texttt{ comp} \end{array}$$

$\mathscr{V}$ translates the types of values whereas $\mathscr{C}$ translates the types of computations that accept a continuation and pass their value to it. The intuition of using `hexp` as the type of the answers in $\mathscr{C}$ is that the final answer of a specializer is specialized code of type `hexp`.

Now we can state the adaption of Theorem 6.1 for continuation-based specialization.

Theorem 4
Suppose `e : exp` and

1. δ : `ident` $\to$ `ident` injective with $\mathrm{dom}(\delta) = FV(\texttt{e})$;
2. $\tau \vdash' \texttt{e} : t$;
3. $\texttt{e'} = \Phi^{\texttt{hucvar}}\ \texttt{e}\ \delta$;
4. the type environment A is defined by $\mathrm{dom}(A) = \delta(\mathrm{dom}(\tau))$ and $A(\delta(\texttt{x})) = \mathscr{V}[\![\tau(\texttt{x})]\!]$ for all $\texttt{x} \in \mathrm{dom}(\tau)$.

Then $A \vdash_{\mathrm{ML}} \texttt{e'} : \mathscr{C}[\![t]\!]$.

Proof
By induction on the derivation of $\tau \vdash' \texttt{e} : t$. □

The `huvar` set of combinators for the simple specializer (figure 9) has the intriguing property that a clever translation from two-level expressions to combinators can eliminate the `huvar`, `hulam`, and `huapp` combinators. While there seems to be no chance to eliminate the `huclam` and `hucapp` combinators, the `hucvar` combinator can be eliminated at the price of modifying the combinators once again. The idea goes back to Reynolds's CPS translation for a call-by-value language. Instead of having

```
let hucvar x = fun c -> c x
```

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
(* dspec : exp -> (ident, value) env -> value *)
let rec dspec e r =
  match e with
  ...
  | LamD (x, e) ->
      let xx = gensym () in
      let Code body =
        reset (fun () -> dspec e (upd r x (Code (Var xx)))) in
      Code (Lam (xx, body))
  | AppD (f, a) ->
      let Code ff = dspec f r in
      let Code aa = dspec a r in
      Code (App (ff, aa))
  | LetD (x, h, e) ->
      let xx = gensym () in
      shift (fun k ->
        let Code hh = dspec h r in
        let Code body =
          reset (fun () -> k (dspec e (upd r x (Code (Var xx))))) in
        Code (Let (xx, hh, body)))
```

Fig. 14. Lawall and Danvy's (1994) specializer.

where the variable transforms itself into a computation that accepts a continuation `c`, we bind variables to computations in the first place, that is, in the cases for `Lam`, `LamD`, and `LetD`. We only show the case for `Lam` and leave the remaining cases as an exercise.

```
let hurcvar x = x
let hurclam x_e =
  fun c ->
    c (fun y -> x_e (fun c -> c y))
```

By the way, instead of changing the `Lam` case, we could also change the `App` case to achieve the same goal, but with slightly different typing properties.

Instead of adapting and proving Theorem 7.1 for this modification (which amounts to redefining A to be $A(\delta(\mathtt{x})) = \mathscr{C}[\![\tau(\mathtt{x})]\!]$) we conclude with running our example using the last set of combinators:

```
# outHOAS
  (hurclamD (fun q ->
      hurcapp (hurcletD (hurcappD (q, q),
                         fun id -> hurclam (fun z -> z)),
               q))
  (fun x -> x));;
- : exp = Lam ("x7", Let ("x8", App (Var "x7", Var "x7"), Var "x7"))
```

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
(*
   hudlamD : (hexp -> hexp) -> hexp
   hudappD : hexp * hexp -> hexp
   hudletD : hexp * (hexp -> 'a) -> 'a
*)
let hudlamD x_e =
  HLam (fun xx -> reset (fun () -> x_e (HVar xx)))
let hudappD (f, a) =
  HApp (f, a)
let hudletD (h, x_e) =
  shift (fun k ->
    HLet (h, fun xx -> reset (fun () -> k (x_e (HVar xx)))))
```

Fig. 15. Combinators for continuation-based specialization in direct style.

7.2 Continuation-based specialization in direct style

Lawall and Danvy (1994) show how to achieve the beneficial effect of Bondorf's specializer without using continuations. The conceptual price is high, since they make use of control operators in the implementation. However, the investment pays back in terms of improved performance of the specializer. Therefore, let's see whether their style of specialization is amenable to combinator-based program generation, too.

First, we put their main tools on the table: the control operators `shift` and `reset` (Danvy and Filinski, 1990, 1992). They have the following effect: `reset` expects a parameterless function `thunk`, it runs `thunk` and "sets a marker" to delimit its context. `shift` expects a function, say `f`, as its argument and passes to `f` another function that corresponds to the context of `shift f` up to the next enclosing `reset`. At the same time, `shift` removes this context. A few examples serve to clarify their operation.

```
# 1 + reset (fun () -> 10 * shift (fun k -> 41))
- : int = 42
```

`k` is bound to the captured context `fun z -> 10 * z`. However, this example discards `k`, so the multiplication by `10` never happens. The next examples show that we can use `k` like any other function:

```
# 1 + reset (fun () -> 10 * shift (fun k -> k 41));;
- : int = 411
# 1 + reset (fun () -> 10 * shift (fun k -> k (k 41)));;
- : int = 4101
```

Exactly this mechanism can relocate the specialization of the context of a `LetD` into its body. Figure 14 shows how (Lawall and Danvy, 1994). It omits the cases for `Var`, `Lam`, and `App`: they are identical to figure 2. In the case for `LamD` the `reset` delimits the specialization of the body. The type of `reset` here is `(unit -> value) -> value`. The case for `AppD` remains unchanged. In the case for `LetD` the `shift` abstracts the specialization of the context of the `LetD` and binds it to `k`. After specializing the header `h` and the original body `e`, the application of `k` specializes

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

the context of `e` *inside* the body of the generated `Let`. The type of `shift` here is `((value -> value) -> value) -> value`.

The types assumed for `shift` and `reset` are not the most general ones. However, a discussion of their typing properties is not in the scope of this paper. We refer to Gunter *et al.* (1995) for a discussion of the typing of control operators.

From figure 14 we derive – just as before – the combinator representation. Figure 15 shows the result for the `LamD`, `AppD`, and `LetD` cases. There are no surprises, except that the typings of the control operators are now `reset : (unit -> hexp) -> hexp` and `shift : (('a -> hexp) -> hexp) -> 'a`, because we have stripped away the tags.

Lawall and Danvy (1994) show that `cspec` and `dspec` are connected via a CPS translation. Just the same is true for the `hucvar` and `hudvar` combinators: The `hucvar` combinators are the images of the `hudvar` combinators under Danvy and Filinski's (1990, 1992) extended CPS translation.

OCaml type-checks these combinators but it is not possible to run them properly. The implementation neither supports `shift/reset` nor `call/cc`, which can be used to implement them (Filinski, 1994). However, we are using their transcription to Scheme successfully in our specialization system (see section 10).

8 Extending

In this section, we discuss various issues involved in extending the concepts introduced in the last sections to a specializer for a full language. Our main concern is coverage: Can we provide combinators for all constructs of the language?

Another concern would be support for program point specialization and memoization. Since memoization involves comparison of static data, we have to find for each datatype a representation that can be compared for equality. The main problem here is with functions. However, once we have such a representation, we simply replace the `Lam` combinator with the constructor of the representation and enhance the `App` combinator with a destructor of the representation. Thiemann (1996) presents such a representation for typed and untyped functional programming languages. We will not discuss it here.

Back to coverage of language constructs: Our guideline here is the work on Lambdamix (Gomard, 1992; Jones *et al.*, 1993), and we simply exhibit combinator implementations for constants, lifting, fixpoint, conditional, and primitive operations. We consider two of the variants, namely for the `huvar` set (simple combinators with higher-order syntax, untagged, figure 9) and for the `hucvar` set (combinators with higher-order syntax, untagged, for continuation-based specialization, figure 12). For conciseness we use

```
type 'a comp = ('a -> hexp) -> hexp
```

in the type signatures for the `hucvar` combinators. None of the constructs is a binding construct, so the syntax representation of the output does not matter. It is straightforward to extend our technical results to the full language.

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

Constants. These are straightforward to integrate. We extend the syntax and give the two implementations. The `huccon` combinator works like the constant case in a continuation semantics.

```
type exp = ... | Con of int
(*
   hucon  : int -> int
   huccon : int -> int comp
*)
let hucon i = i
let huccon i =
  fun c -> c i
```

Lifting. This means to propagate a value at specialization time to its syntactic representation in the specialized program. Again, it is straightforward to implement. The `huclift` combinator works like a unary primitive in a continuation semantics.

```
type exp = ... | Lift of exp
(*
   hulift  : int -> hexp
   huclift : int comp -> hexp comp
*)
let hulift e =
  Con e
let huclift e =
  fun c -> e (fun i -> c (Con i))
```

Fixpoint. Here we have to restrict the argument of the fixpoint operator to have functional type because our implementation language is call-by-value. If our implementation were for a non-strict language like Haskell, this restriction would not be necessary. Otherwise, we just extend the syntax and give the implementation.

```
type exp = ... | Fix of exp | FixD of exp
(*
   fixv  : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
   hufix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
*)
let rec fixv f = f (fun x -> fixv f x)
let hufix e = fixv e
```

The continuation semantics requires a fixpoint operator in continuation-passing style which will then serve as the implementation for `hucfix`.

```
(*
  fixvc  : (('a -> 'b comp) -> ('a -> 'b comp) comp) -> ('a -> 'b comp) comp
  hucfix : (('a -> 'b comp) -> ('a -> 'b comp) comp) -> ('a -> 'b comp) comp
*)
let rec fixvc f = fun c -> f (fun x c -> fixvc f (fun g -> g x c)) c
let hucfix e = fixvc e
```

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

The dynamic versions of the fixpoint operator are much simpler, they just build a `Fix` expression.

```
(*
   hufixD  : hexp -> hexp
   hucfixD : hexp comp -> hexp comp
*)
let hufixD e =
  Fix e
let hucfixD e =
  fun c -> e (fun ee -> c (Fix ee))
```

Conditional. The syntax extends as follows:

```
type exp = ... | If of exp * exp * exp | IfD of exp * exp * exp
```

Here, we encounter the first small problem in the `huvar` combinators. The problem is that every function that replaces `If` evaluates both branches of the conditional, which breaks the semantics. There are two ways out of the problem:

1. The translation encapsulates the branches in thunks (i.e. functions of type `unit -> 'a`) and uses

   ```
   (* huif : int * (unit -> 'a) * (unit -> 'a) -> 'a *)
   let huif (i, t', e') =
     if i!=0 then t' () else e' ()
   ```

   or
2. the translation uses the implementation language's `if` directly.

The latter is the simplest solution, so we stick to that. The dynamic case just builds an `If` expression.

```
let huifD (i, t, e) =
  If (i, t, e)
```

However, this combinator is not suitable for use with the `hudvar` set of combinators because Lawall and Danvy's (1994) specializer places `reset`s on top of the branches of the conditional. The solution is to have the translation insert thunks on the branches of the dynamic conditional:

```
(* hudifD : exp * (unit -> exp) * (unit -> exp) -> exp *)
let hudifD (i, t', e') =
  If (i, reset t', reset e')
```

In continuation-passing style, the problem with the non-strictness of the `If` disappears since the arguments to the `hucif` function are themselves functions that expect a continuation.

```
(*
   int comp * ('a comp) * ('a comp) -> 'a comp
```

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

```
*)
let hucif (i, t, e) =
  fun c -> i (fun ii ->
    if ii != 0 then t c else e c)
```

The dynamic case is handled like in Bondorf's (1992) specializer: the combinator specializes the condition and constructs the specializations of the branches using the identity continuation.

```
let hucifD (i, t, e) =
  fun c -> i (fun ii ->
    c (If (ii, t (fun z -> z), e (fun z -> z))))
```

Primitive operations. For simplicity, we restrict the discussion here to the case of unary primitives. The generalization is straightforward.

The syntactic representation of a primitive includes its name as a `string`, the function itself of type `int -> int`, and the argument expression. The string only serves to please human readers, the combinators never inspect it.

```
type exp = ... | Prim  of string * (int -> int) * exp
               | PrimD of string * (int -> int) * exp
```

Both cases are simple.

```
(*
   huprim  : string * (int -> int) * int -> int
   hucprim : string * (int -> int) * int comp -> int comp
*)
let huprim (_, p, i) =
  p i
let hucprim (_, p, e) =
  fun c -> e (fun i -> c (p i))
```

Functions with a differing number of arguments may be handled using

```
Prim' of string * (int list -> int) * exp list
```

The implementation of `huprim` does not change with this signature. The implementation of `hucprim` gets a little bit tedious, but it is still straightforward.

In the dynamic case, the simple specializer just rebuilds the expression and the continuation-based specializer performs a primitive operation.

```
(*
   huprimD  : string * (int -> int) * hexp -> hexp
   hucprimD : string * (int -> int) * hexp comp -> hexp comp
*)
let huprimD (s, p, e) =
  Prim (s, p, e)
let hucprimD (s, p, e) =
  fun c -> e (fun ee -> c (Prim (s, p, ee)))
```

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

9 Folding

The astute reader might have expected a connection between the combinators that we derived and the recursive `spec` functions in terms of a fold functional. Indeed, we could rephrase the result of Lemma 4.1 in Sec. 4 as

$$\texttt{spec} = \texttt{fold}_{\texttt{exp}}(\texttt{var}, \texttt{lam}, \ldots)$$

where $\texttt{fold}_{\texttt{exp}}$ is the fold functional for the `exp` datatype (Sheard and Fegaras, 1993). Unfortunately, we cannot apply the standard theorems about fold functionals because `spec` and the `var` combinators involve state manipulation due to the use of `gensym`.

The `hvar` set of combinators also has a connection to a fold functional. However, it is not the fold functional associated to the `hexp` type because the types do not work out. To be usable for folding over `hexp` only occurrences of `hexp` in the types of the constructors should have been replaced by `value`, but in fact the occurrences of `ident` have been replaced by `value`, too. The right type for folding with the `hvar` combinators is

```
type 'a phexp =
    PHPlace of 'a
  | PHLam of ('a phexp -> 'a phexp)
  | PHApp of 'a phexp * 'a phexp
  | PHLamD of ('a phexp -> 'a phexp)
  | PHAppD of 'a phexp * 'a phexp
```

which is parameterized over the domain `'a` of semantic values. The difference is that variables have no explicit representation, i.e. the encoding of `Lam ("x", Var "x")` is `PHLam (fun x -> x)`. Fegaras and Sheard (1996) explain how to construct a fold over `'a phexp` where `'a phexp` can occur negatively in the argument type of a constructor. They also explain the rôle of the `PHPlace` constructor. However, the connection to the fold functional is less gratifying this time, because it does not yield the correctness proof of the transformation step for free.

10 Performance

In this section we report comparative measurements between four methods for doing offline partial evaluation:

- specialization directly with Similix (Bondorf, 1993), a partial evaluator for Scheme;
- specialization with a generating extension produced by a Similix-generated cogen (using double self-application);
- specialization with a generating extension composed of `hucvar`-style combinators;
- specialization with a generating extension composed of `hudvar`-style combinators.

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

program	size	description
app	67	list append
ctors	167	partially-static constructors
lambda	133	partially-static functions
cps-lr	727	functional LR-parser using continuations (Sperber and Thiemann, 1995)
direct-lr	1581	functional LR-parser in direct style (Sperber and Thiemann, 1995)
scheme1	6625	interpreter for higher-order rec. equations (Thiemann, 1995)

Fig. 16. Benchmark programs (size = number of cons cells + number of atoms).

program	Similix gen	Similix comp	Similix spec	Similix $\frac{\text{spec}}{\text{comp}}$	PGG CPS	PGG DS	ratios $\frac{\text{Sim}}{\text{CPS}}$	ratios $\frac{\text{Sim}}{\text{DS}}$	ratios $\frac{\text{CPS}}{\text{DS}}$
app	177.6	6.7	13.3	1.98	1.4	1.0	4.78	6.70	1.40
ctors	368.6	12.3	27.4	2.22	3.1	5.3	3.96	2.32	0.58
lambda	476.5	20.4	32.2	1.57	5.4	6.5	3.77	3.13	0.83
direct-lr	6690	13090	10840	0.82	3970	3910	3.29	3.34	1.01
cps-lr	3770	10810	9870	0.91	3360	3390	3.21	3.18	0.99
scheme1	26360	560	2370	4.23	390	180	1.43	3.11	2.16

Fig. 17. Runtimes for the benchmarks (in ms).

All specializers have comparable features, i.e. all perform continuation-based specialization, they perform polyvariant program point specialization, and handle partially static data. There is no postprocessing (we have turned it off in Similix, since it contributes massively to specialization time). To get a sensible comparison, we have implemented all of the combinators in Scheme (Kelsey *et al.*, 1998) as part of the PGG system (Thiemann, 1999). Our implementation of the `hudvar` combinators uses Filinski's (1994) implementation of `shift` and `reset`.

Figure 16 describes the benchmark programs. Three of them (**app**, **ctors**, and **lambda**) are artificial programs that test specific features. The remaining three are realistic examples: we have used **cps-lr** and **direct-lr** in previous work on parser generation (Sperber and Thiemann, 1995) and **scheme1** to generate higher-order online specializers (Thiemann and Glueck, 1995). The times are measured on an IBM 43P with 64MB main memory running AIX 4.1.4 using the `,time` command of the Scheme 48 system version 0.44 (Kelsey and Rees, 1995) with `-h 4000000` heap space.

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

Figure 17 displays the timing data. All times are given in milliseconds. They include garbage collection if any, but exclude loading and compiling programs. They do not include preprocessing and binding-time analysis, which is about the same for all systems. There is one row for each program. The first column gives the name of the benchmark. The next four columns give timings for Similix as follows:

gen time to generate the generating extension.
comp time to run the generating extension on a representative input.[2]
spec time to specialize directly with Similix on the same input.
$\frac{\textbf{spec}}{\textbf{comp}}$ ratio of time spec divided by time comp.

The next two columns give timings for the PGG system.

CPS indicates a specialization run using the `hucvar`-style combinators.
DS indicates a specialization run using the combinators in `hudvar`-style.

The final three columns contain ratios between Similix times and PGG times.

$\frac{\textbf{Sim}}{\textbf{CPS}}$ Similix time for comp divided by PGG time for CPS.
$\frac{\textbf{Sim}}{\textbf{DS}}$ Similix time for comp divided by PGG time for DS.
$\frac{\textbf{CPS}}{\textbf{DS}}$ PGG time for CPS divided by PGG time for DS.

The PGG system does not have to create a generating extension from the binding-time annotated program; it simply outputs the annotated program at the end of the binding-time analysis. Hence, the **generate** column is only applicable to Similix. The PGG-constructed generating extensions in `hudvar` style are faster than Similix' generating extensions by a factor between 2.3 and 6.7, with an average of 3.63. The generating extensions using the `hucvar`-style combinators are faster than Similix' by a factor between 1.4 and 4.8, the average being about 3.4. In the comparison of the `hucvar`-style combinators with the `hudvar`-style combinators, the `hucvar` versions vary between being twice as fast and half as fast as the corresponding `hudvar` versions with an average of 1.16. This makes a strong case for the `hudvar`-style combinators, although the figures are positively influenced by the fact that Scheme 48 has a fast implementation of `call/cc` (which is used to implement `shift` and `reset` (Filinski, 1994)). Other implementations could give different results, i.e. less favorable for the `hudvar`-style combinators.

Interestingly, the speedup of **specialize** vs. **compile** for Similix ranges between 0.82 (specializing with Similix' generating extension is slower than specializing directly) and 4.23 with the slow cases being the parsers.

11 Related work

The construction of combinators for generating extensions relies on a compositional specification of the specializer. This facilitates moving the syntax dispatch to the

[2] The Similix-generated generating extensions that we used to measure specialization times were of course produced with postprocessing. Otherwise they would compile significantly slower.

https://doi.org/10.1017/S0956796899003469 Published online by Cambridge University Press

meta-level by re-interpreting the syntax constructors as functions that perform specialization. Holst (1989) is the first to construct generating extensions by re-interpreting the syntax constructors of a two-level program using Scheme macros. The development of these macros is ad-hoc and does not consider typing issues.

The next key point is to exploit the binding mechanism of the implementation language using higher-order abstract syntax. The idea of higher-order abstract syntax can be traced back to Church (1940). It was popularized by Pfenning and Elliott (1988). Mogensen (1992 *a*, 1992 *b*, 1995) has used higher-order abstract syntax to develop efficient self-interpreters and specializers for the pure lambda calculus. The present work is yet another indication of the value of higher-order abstract syntax in program transformation.

Hand-written program-generator generators are not new. Beckman and others (Beckman *et al*., 1976) already describe a program generator for online specialization. Their motivation was the lack of self-applicable specializers at that time. Romanenko (1988) gives some recipes to construct compilers from binding-time annotated interpreters. These recipes bear some similarity to the combinator definitions that we derived from the Lambdamix specializer. Romanenko discovered them by inspection of compilers produced by a self-applicable specializer, not by derivation. Hence, he does not present any correctness arguments. Since then a number of papers report hand-written program-generator generators for a variety of functional languages (Launchbury and Holst, 1991; Birkedal and Welinder,1993; Glueck and Jørgensen, 1995; Draves, 1996), all of them using ad-hoc methods. Of them, Launchbury and Holst (1991) were the first to realize the potential for efficiently specializing programs in statically typed languages. Bondorf and Dussart (1994) construct a hand-written continuation-based cogen and prove its equivalence to Bondorf's specializer (Bondorf,1992), Lawall and Danvy (1995) present equivalent continuation-based cogens in the style of `cspec` and `dspec`. Both works have tedious equivalence proofs with respect to a continuation-based specializer. In contrast, our combinators have a simple and direct derivation from the specializer.

Glück and Jørgensen (1995) consider a cogen for multi-level specialization and multi-level binding-time analysis (1996). However, their work is for an untyped language, they only cover the simple `spec`-style specialization, and they do not state a formal connection to a specializer. It is straightforward to extend our work to multi-level specialization (see Thiemann (1996)).

Thiemann and Sperber (1997) demonstrate another approach to program generation in a language with an extension of Haskell-style overloading. They have only one combinator for each syntax constructor, but its type is overloaded so that it can act for the static case as well as for the dynamic case, depending on the context of use. They argue that this approach can eliminate the need for a binding-time analysis.

The general approach of this work in also transferable to other programming paradigms: Leuschel and Jørgensen (1996) demonstrate the feasibility of hand-written program-generator generators for logic programming. The C-mix specialization system (Andersen, 1994) constructs generating extensions directly from two-level programs. The Tempo system (Consel and Noël, 1996) goes one step further

in constructing generating extensions for run-time code generation from two-level programs. A similar approach is used by Sperber and Thiemann (1997).

Wand (1982) has introduced combinator-based compilation. He expresses a continuation semantics of a programming language in terms of combinators. The compiled program is obtained by re-interpreting the semantic combinators as instructions of a stack machine. He performs optimizations by rewriting the combinator term using rules derived from the original semantics. The similarities of his method to our approach lie in the re-interpretation of the combinators, which is possible due to the compositionality of the semantics and which yields the correctness-by-derivation property. In both cases, the compiled (specialized) programs are expressed in terms of the combinators. However, we are concerned with performing a source-to-source program transformation, not with transforming a program into a form suitable for efficient execution on a stack machine. Hence, we abstract entire right sides of the definition of the specializer into combinators.

Finally, we could view the combinators as a domain-specific programming language for writing program generators. The polymorphic type-checking of the implementation language would test the well-formedness as outlined in section 6, thus removing the need for an extra binding-time analysis. Compared to a dedicated language for meta-programming (like MetaML – Taha and Sheard, 1997), our approach has the disadvantage that it does not enforce the generated program to be well-typed. However, we prefer having the binding-time analysis figure out the static and dynamic parts for us rather than constructing the program generator by hand.

12 Conclusion

We have presented a simple and general methodology for constructing combinators for specialization from a denotational specification of a specializer. The method relies on the compositionality of the specification, higher-order abstract syntax, and untagging. We exhibit three corresponding transformation steps and prove their correctness. The resulting set of combinators serves to re-interpret the syntax constructors of a two-level program, thus turning it into its own generating extension.

We have applied the methodology to derive combinators for standard specialization and for continuation-based specialization (once using continuations and once using control operators).

The implementation of our combinators in other functional programming languages, typed or untyped, pure or impure, is straightforward. This is mainly due to our decoupling of program generation and name generation. For example, in a Haskell implementation only the final printing part of the system has to be written in monadic style to perform name generation. All other parts of the specialization are pure.

We have shown that specialization combinators facilitate efficient program specialization. Our full-blown specialization system for Scheme outperforms a self-applicable specializer by a factor of three for realistic examples.

We believe that it is simpler to construct combinators for specialization than to write a self-applicable specializer. First, the programmer does not have to be

wary of binding times while constructing the underlying specializer. Secondly, the underlying specializer can freely exploit all features of the language, it is not restricted to "specializable" features. Thirdly, specialization using the combinators is more efficient than using generating extensions from a self-applicable specializer as we have demonstrated. And finally, the correctness proof with respect to the specializer reduces to the correctness of the transformation steps that we have done once and for all in this work.

Acknowledgements

Thanks to Olivier Danvy, Dirk Dussart, Robert Glück, Jesper Jørgensen and Julia Lawall for discussions and suggestions about the precursor to this work (Thiemann, 1996). Thanks also to the reviewers for their comments that helped to improve this work.

References

Andersen, L. O. (1994) *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen. (DIKU report 94/19.)

Beckman, L., Haraldsson, A., Oskarsson, Ö. and Sandewall, E. (1976) A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, **7**(4), 319–357.

Birkedal, L. and Welinder, M. (1993) Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen.

Bondorf, A. (1991) Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, **17**, 3–34.

Bondorf, A. (1992) Improving binding times without explicit CPS-conversion. *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pp. 1–10. San Francisco, CA.

Bondorf, A. (1993) *Similix 5.0 Manual*. DIKU, University of Copenhagen.

Bondorf, A. and Dussart, D. (1994) Improving CPS-based partial evaluation: Writing cogen by hand. In: P. Sestoft and H. Søndergaard, editors, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, pp. 1–10. Orlando, FL. (University of Melbourne, Australia. Technical Report 94/9, Department of Computer Science.)

Church, A. (1940) A formulation of the simple theory of types. *J. Symbolic Logic*, **5**, 56–68.

Clément, D, Despeyroux, J., Despeyroux, T. and Kahn, G. (1986) A simple applicative language: Mini-ML. *ACM Conference on LISP and Functional Programming*, pp. 13–27.

Consel, C. and Danvy, O. (1993) Tutorial notes on partial evaluation. *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pp. 493–501. Charleston, SC. ACM Press.

Consel, C. and Noël, F. (1996) A general approach for run-time specialization and its application to C. *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 145–156. St. Petersburg, FL. ACM Press.

Danvy, O. and Filinski, A. (1990) Abstracting control. *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160. Nice, France. ACM Press.

Danvy, O. and Filinski, A. (1992) Representing control: A study of the CPS transformation. *Mathematical Structures in Comput. Sci.*, **2**, 361–391.

Danvy, O., Glück, R. and Thiemann, P. editors (1996) *Dagstuhl Seminar on Partial Evaluation 1996: Lecture Notes in Computer Science 1110*, Schloß Dagstuhl, Germany. Springer-Verlag.

Draves, S. (1996) Compiler generation for interactive graphics using intermediate code. In Danvy *et al.*, editors, *Dagstuhl Seminar on Partial Evaluation 1996: Lecture Notes in Computer Science 1110*, pp. 95–114. Schloß Dagstuhl, Germany. Springer-Verlag.

Fegaras, L. and Sheard, T. (1996) Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 284–294. St. Petersburg, FL. ACM Press.

Filinski, A. (1994) Representing monads. *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 446–457. Portland, OG. ACM Press.

Futamura, Y. (1971) Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, **2**(5), 45–50.

Glück, R., and Jørgensen, J. (1995) Efficient multi-level generating extensions for program specialization. In: D. Swierstra and M. Hermenegildo, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95): Lecture Notes in Computer Science 982*, pp. 259–278. Utrecht, The Netherlands. Springer-Verlag.

Glück, R., and Jörgensen, J. (1996) Fast multi-level binding-time analysis for multiple program specialization. *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics: Lecture Notes in Computer Science 1181*, Novosibirsk, Russia. Springer-Verlag.

Gomard, C. K. (1990) Partial type inference for untyped functional programs. *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pp. 282–287. Nice, France. ACM Press.

Gomard, C. K. (1992) A self-applicable partial evaluator for the lambda-calculus. *ACM Trans. Programming Languages and Systems*, **14**(2), 147–172.

Gunter, C. A., Rémy, D. and Riecke, J. G. (1995) A generalization of exceptions and control in ML-like languages. In: S. Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, pp. 12–23. La Jolla, CA. ACM Press.

Haskell 1.4, a non-strict, purely functional language (April 1997) `http://haskell.systemsz.cs.yale.edu/onlinereport/`.

Hatcliff, J. and Danvy, O. (1994) A generic account of continuation-passing styles. *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 458–471. Portland, OG. ACM Press.

Holstm C. K. (1989) Syntactic currying. Student report, DIKU, University of Copenhagen.

Hutton, G. (1992) Higher-order functions for parsing. *J. Functional Programming*, **2**(3), 323–344.

Jones, N. D., Gomard, C. K. and Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

Jørgensen, J. and Leuschel, M. (1996) Efficiently generating efficient generating extensions in Prolog. In Danvy *et al.*, editors, *Dagstuhl Seminar on Partial Evaluation 1996: Lecture Notes in Computer Science 1110*, pp. 238–262. Schloß Dagstuhl, Germany. Springer-Verlag.

Kelsey, R., Clinger, W. and Rees, J. (1998) Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, **33**(9), 26–76.

Kelsey, R. A. and Rees, J. A. (1995) A tractable Scheme implementation. *Lisp and Symbolic Computation*, **7**(4), 315–335.

Launchbury, J. (1991) A strongly-typed self-applicable partial evaluator. In: J. Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991: Lecture Notes in Computer Science 523*, pp. 145–164. Cambridge, MA. Springer-Verlag.

Launchbury, J. and Holst, C. K. (1991) Handwriting cogen to avoid problems with static typing. *Draft Proceedings, 4th Annual Glasgow Workshop on Functional Programming*, pp. 210–218, Skye, Scotland.

Lawall, J. L. and Danvy, O. (1994) Continuation-based partial evaluation. *Proc. 1994 ACM Conference on Lisp and Functional Programming*, pp. 227–238, Orlando, FL. ACM Press.

Lawall, J. L. and Danvy, O. (1995) Continuation-based partial evaluation. Technical Report Technical Report CS-95-178, Brandeis University, Waltham, Massachusetts. (Extended version from `ftp://ftp.brics.dk/pub/danvy/Papers/lawall-danvy-lfp94-extended.ps.gz`.)

Leroy, X. (1997) *The Objective Caml system release 1.07, Documentation and user's manual.* INRIA, France. From `http://pauillac.inria.fr/caml`.

Proc. 1990 ACM Conference on Lisp and Functional Programming, Nice, France. ACM Press.

Mitchell, J. C. (1996) *Foundations for Programming Languages.* MIT Press.

Mogensen, T. Æ. (1992*a*) Efficient self-interpretation in lambda calculus. *J. Functional Programming*, **2**(3), 345–364.

Mogensen, T. Æ. (1992*b*) Self-applicable partial evaluation for pure lambda calculus. In: C. Consel, editor, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '92*, pp. 116–121. San Francisco, CA. Yale University.

Mogensen, T. Æ. (1995) Self-applicable online partial evaluation of pure lambda calculus. In: W. Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pp. 39–44. La Jolla, CA. ACM Press.

Moggi, E. (1988) Computational lambda-calculus and monads. *Technical Report ECS-LFCS-88-86*, University of Edinburgh.

Moggi, E. (1998) Functor categories and two-level languages. In: M. Nivat and A. Arnold, editors, *Foundations of Software Science and Computation Structures, FoSSaCS'98: Lecture Notes in Computer Science*, Lisbon, Portugal.

Mosses, P. D. (1990) *Denotational Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11. Elsevier.

Nielson, F. and Nielson, H. R. (1992) *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press.

Palsberg, J. (1993) Correctness of binding-time analysis. *J. Functional Programming*, **3**(3), 347–364.

Pfenning, F. and Elliott, C. (1988) Higher-order abstract syntax. *Proc. Conference on Programming Language Design and Implementation '88*, pp. 199–208, Atlanta, GA. ACM Press.

Romanenko, S. A. (1988) A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In: D. Bjorner, A. P. Ershov and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pp. 445–463. North-Holland.

Sabry, A. and Felleisen, M. (1993) Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, **6**(3/4), 289–360.

Schmidt, D. A. (1986) *Denotational Semantics, A Methodology for Software Development*. Allyn and Bacon.

Sheard, T. and Fegaras, L. (1993) A fold for all seasons. In: Arvind, editor, *Proc. Functional Programming Languages and Computer Architecture 1993*, pp. 233–242. Copenhagen, Denmark. ACM Press.

Sperber, M. and Thiemann, P. (1995) The essence of LR parsing. In Scherlis, W., editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pp. 146–155. La Jolla, CA. ACM Press.

Sperber, M. and Thiemann, P. (1997) Two for the price of one: Composing partial evaluation and compilation. *Proc. of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pp. 215–225. Las Vegas, NV. ACM Press.

Taha, W. and Sheard, T. (1997) Multi-stage programming with explicit annotations. In:

C. Consel, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97*, pp. 203–217. Amsterdam, The Netherlands. ACM Press.

Thiemann, P. (1996*a*) Cogen in six lines. In: R. K. Dybvig, editor, *Proc. International Conference on Functional Programming 1996*, pp. 180–189. Philadelphia, PA. ACM Press.

Thiemann, P. (1996*b*) Implementing memoization for partial evaluation. In: H. Kuchen and D. Swierstra, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96): Lecture Notes in Computer Science 1140*, pp. 198–212. Aachen, Germany. Springer-Verlag.

Thiemann, P. (1998) A generic framework for specialization. In: C. Hankin, editor, *Proc. 7th European Symposium on Programming: Lecture Notes in Computer Science 1381*, pp. 267–281. Lissabon, Portugal. Springer-Verlag.

Thiemann, P. (1999) *The PGG System – User Manual.* Universität Freiburg, Germany. (Available from `http://www.informatik.uni-freiburg.de/proglang/software/pgg/`.

Thiemann, P. and Dussart, D. (1996) Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universität Tübingen.

Thiemann, P. and Glück, R. (1995) The generation of a higher-order online partial evaluator. In: M. Takeichi and T. Ida, editors, *Fuji International Workshop on Functional and Logic Programming*, pp. 239–253. World Scientific.

Thiemann, P. and Sperber, M. (1997) Program generation with class. In: M. Jarke, K. Pasedach and K. Pohl, editors, *Proceedings Informatik'97*, Reihe Informatik aktuell. Springer-Verlag.

Turchin, W. F. (1979) A supercompiler system based on the language Refal. *SIGPLAN Notices*, **14**(2), 46–54.

Wand, M. (1982) Deriving target code as a representation of continuation semantics. *ACM Trans. Programming Languages and Systems*, **4**(3), 496–517.