Efficient compilation of algebraic effects and handlers

Matija Pretnar
Amr Hany Saleh
Axel Faes
Tom Schrijvers

Report CW 708, October 2017

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Efficient compilation of algebraic effects and handlers

Matija Pretnar
Amr Hany Saleh
Axel Faes
Tom Schrijvers

Report CW 708, October 2017

Department of Computer Science, KU Leuven

Abstract

The popularity of algebraic effect handlers as a programming language feature for user-defined computational effects is steadily growing. Yet, even though efficient runtime representations have already been studied, most handler-based programs are still much slower than hand-written code.

In this paper we show that the performance gap can be drastically narrowed (in some cases even closed) by means of type-and-effect directed optimising compilation. Our approach consists of two stages. Firstly, we combine elementary source-to-source transformations with judicious function specialisation in order to aggressively reduce handler applications. Secondly, we show how to elaborate the source language into a handler-less target language in a way that incurs no overhead for pure computations.

This work comes with a practical implementation: an optimizing compiler from Eff, an ML style language with algebraic effect handlers, to OCaml. Experimental evaluation with this implementation demonstrates that in a number of benchmarks, our approach eliminates much of the overhead of handlers and yields competitive performance with hand-written OCaml code.

Efficient Compilation of Algebraic Effects and Handlers

MATIJA PRETNAR, University of Ljubljana
AMR HANY SALEH, KU Leuven
AXEL FAES, KU Leuven
TOM SCHRIJVERS, KU Leuven

The popularity of algebraic effect handlers as a programming language feature for user-defined computational effects is steadily growing. Yet, even though efficient runtime representations have already been studied, most handler-based programs are still much slower than hand-written code.

In this paper we show that the performance gap can be drastically narrowed (in some cases even closed) by means of type-and-effect directed optimising compilation. Our approach consists of two stages. Firstly, we combine elementary source-to-source transformations with judicious function specialisation in order to aggressively reduce handler applications. Secondly, we show how to elaborate the source language into a handler-less target language in a way that incurs no overhead for pure computations.

This work comes with a practical implementation: an optimizing compiler from Eff, an ML style language with algebraic effect handlers, to OCaml. Experimental evaluation with this implementation demonstrates that in a number of benchmarks, our approach eliminates much of the overhead of handlers and yields competitive performance with hand-written OCaml code.

CCS Concepts: •**Software and its engineering → Functional languages; Compilers; Interpreters;**

Additional Key Words and Phrases: algebraic effect handlers, optimising compilation, OCaml

ACM Reference format:
Matija Pretnar, Amr Hany Saleh, Axel Faes, and Tom Schrijvers. 2017. Efficient Compilation of Algebraic Effects and Handlers. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 32 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Algebraic effect handlers (Plotkin and Power 2003; Plotkin and Pretnar 2013) are quickly maturing from a theoretical model of computational effects to a practical (functional) language feature for user-defined side-effects.

Now that a bevy of implementations are available, runtime performance is becoming more and more of a concern. So far, most implementations come in the form of libraries (Brady 2013; Kammar et al. 2013; Kiselyov et al. 2013; Kiselyov and Sivaramakrishnan 2016) and interpreters (Bauer and Pretnar 2015; Hillerström et al. 2016). As a consequence, much of the effort to improve performance has been directed towards improving the runtime representation of computations with handlers and associated operations (Dolan et al. 2015; Hillerström et al. 2016; Kiselyov and Ishii 2015). Yet, we see that in practice effect handlers still incur a significant performance overhead compared to hand-written code and native side-effects.

With an end-to-end overview of a compiler for the Koka language, Leijen (2017) has recently demonstrated that compilation is a valid alternative avenue for implementing algebraic effects and handlers. We believe that *optimising* compilation in particular is interesting because it can further narrow the performance gap with native effects.

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

To substantiate this belief we present our approach to the optimised compilation of Eff, a basic functional language with support for algebraic effects and handlers (Bauer and Pretnar 2015). After giving a short informal overview of the proposed approach in Section 2, we begin the formal development in Section 3, which defines the syntax and semantics of Effy, the core calculus of Eff. The rest of the paper describes our contributions:

- First, we present source-to-source transformations and selective function specialisation with the aim of eliminating explicit uses of handlers and the resulting overhead (Section 4).
- Next, we give a basic monadic elaboration of Effy into OCaml, where effectful computations of Effy are translated into pure computations of OCaml that generate elements of the free monad (Section 5).
- We further refine this translation into one that exploits the purity of computations to generate pure OCaml expressions where possible, resulting in mostly idiomatic OCaml code (Section 6). This stage crucially relies on the information from Effy's type-and-effect system to do its job.
- Finally, we discuss our implementation of the Eff compiler (Section 7), and present an experimental evaluation, which clearly demonstrates the effectiveness of this approach on a number of benchmarks. (Section 8).

Section 9 discusses related work and Section 10 concludes.

2 OVERVIEW

This section motivates the need for optimised compilation on a small example. We explain the relevant concepts of algebraic effects and handlers, though we encourage the reader to look at (Pretnar 2015) for a more detailed introduction. As we are working with both OCaml and Eff, whose syntax closely follows OCaml, we use colors to distinguish between `OCaml code` and `Eff code`.

2.1 Programming with Algebraic Effect Handlers

Our running example is a simple loop that repeatedly increments an implicit integer state. We first declare two effectful operations to manipulate the state and then define a recursive function that increments the state a given number of times.

```
operation Put: int -> unit
operation Get: unit -> int
let rec loop n =
  if n = 0 then () else (Put (Get () + 1); loop (n - 1))
```

Applying `loop` "as is" to any positive integer results in a runtime error similar to one of an uncaught exception, for we have so far given no meaning to `Put` and `Get`. To do so, we use handlers. Like exception handlers, which replace the meaning of raised exceptions, algebraic effect handlers determine how to interpret operations when they appear in a computation (the *operation cases*) and how to interpret the final result of a computation (the *return case*). But in contrast to exceptions, which abort any further computation, operations can have *continuations*, i.e. the remainder of the computation waiting for the result of the operation, and the handler can access them.

A standard way of implementing stateful behaviour is to treat computations as functions from the initial to the final value of the state. This is achieved by the following handler:

```
let state_handler = handler
  | Put s' k -> (fun _ -> k () s')
  | Get () k -> (fun s -> k s s)
  | return _ -> (fun s -> s)
```

Let us first take a look at the `Put` case, which is defined in terms of the parameter `s'` (the new value of the state) and the continuation `k`. We handle `Put` as a function that accepts (though ignores) the initial state, and then resumes the continuation by passing it the expected result `() : unit`. The handlers we present in this paper are *deep*, meaning that the handler implicitly continues handling further operations in the continuation. Thus `k ()` is again a function of the initial state, and we pass it the new state `s'`.

The case for `Get` is similar, except that we pass the initial state `s` to the continuation `k` twice: first as the result of the lookup, and second as the new (unmodified) initial state. Finally, the return case ignores the final result of a computation and instead returns the current value of the state. Note that by modifying the handled computation into a function, the handler changes the computation's type.

To tie everything together, we define the function `main`, which applies the handler to `loop` and provides the initial state `0` to the resulting function:

```
let main n =
  (with state_handler handle loop n) 0
```

2.2 Basic Compilation to OCaml

The basic idea behind compiling algebraic effects to OCaml, which does not support them, is to define a type `'a computation` to represent the free monad of computations that call algebraic effects and return values of type `'a`. We postpone the discussion of the exact implementation of this type to Section 5.

We then build computations using the following constructors: a value embedding `return : 'a -> 'a computation` or basic operations `put : int -> unit computation` and `get : unit -> int computation`. We compose effectful computations using a monadic bind `>>=`, which evaluates an effectful computation of type `'a computation` and passes its result to a continuation of type `'a -> 'b computation`, resulting in a `'b computation`.

It is important to note here that functions can have effectful bodies. Thus an Eff function of a type `'a -> 'b` is translated into a OCaml function of type `'a -> 'b computation`. This applies equally to (curried) multi-argument functions. So an Eff function `f : 'a -> 'b -> 'c` must be translated as `f : 'a -> ('b -> 'c computation) computation`, and an application `f x y` must be translated as `f x >>= fun g -> g y`.

Using the above definitions, the presented compiler translates the `loop` function into the following code (manually reformatted for presentation purposes):

```
let rec loop n =
  equal n >>= fun f ->
  f 0     >>= fun b ->
  if b then return () else
    get ()  >>= fun s  ->
    plus s  >>= fun g  ->
    g 1     >>= fun s' ->
    put s'  >>= fun _  ->
    minus n >>= fun h  ->
    h 1     >>= fun n' ->
    loop n'
```

where `equal`, `plus` and `minus` are translations of Eff's arithmetic operations into predefined OCaml constants of the appropriate function type. For example, we define

```
    let plus = fun x -> return (fun y -> return (x + y))
```

We can translate `state_handler` as

```
let state_handler = handler {
  put_case = (fun s' k -> return (fun _ -> k () >>= fun f -> f s'));
  get_case = (fun () k -> return (fun s -> k s >>= fun f -> f s));
```

```
    return_case = (fun _ -> return (fun s -> return s));
  }
```

where the record specifying handler cases are of a predefined type

```
  type ('a, 'b) handler_cases = {
    put_case : int -> (unit -> 'b computation) -> 'b computation;
    get_case : unit -> (int -> 'b computation) -> 'b computation;
    return_case : 'a -> 'b computation;
  }
```

and `handler : ('a, 'b) handler_cases -> ('a computation -> 'b computation)` is a function that takes handler cases and returns a handler, represented as a function between computations. Finally, the `main` function may be translated as

```
  let main n =
    state_handler (loop n) >>= (fun f -> f 0)
```

2.3 Purity Aware Compilation

Obviously, the continual composition and decomposition of computations incurs a substantial overhead. By identifying pure computations and generating regular OCaml code for them, as proposed by Leijen (2017), we can avoid some of that overhead. For instance, since the arithmetic operators used in `loop` are pure, we can translate them directly into OCaml's arithmetic operations and bind their result with `let` rather than with the more expensive `>>=`.

```
  let rec loop n =
    let f = (=) n in
    let b = f 0 in
    if b then return () else
      get () >>= fun s  ->
      let g = plus s in
      let s' = g 1 in
      put s' >>= fun _  ->
      let h = minus n in
      let n' = h 1 in
      loop n'
```

Nevertheless, the backbone of the computation still makes use of `>>=` to sequence the effectful `get` and `put` operations. Hence, the overall impact of this optimisation on this example is limited.

2.4 Optimising Compilation

However, with the help of more aggressively optimised compilation, which replaces operation calls in a handled operation with their corresponding operation cases, we can obtain altogether much tighter code:

```
  let main n = (
    let rec state_handler_loop m =
      if m = 0 then (fun s -> s) else (fun s -> state_handler_loop (m - 1) (s + 1))
    in
    state_handler_loop n) 0
```

Here, all of the handler, the explicit operations `get` and `put`, and their explicit sequencing with `>>=` have been eliminated. The recursive function `loop` has been locally specialised for the particular interpretation of `state_handler`. The resulting code is very close to the hand-written equivalent, the only difference being that a human programmer would hoist the abstractions out of the two branches of the conditional expression.

3 THE CALCULUS

This section presents Effy, a basic functional calculus with support for algebraic effect handlers, which forms the core language of our optimising compiler. We describe the relevant concepts, but refer the reader to Pretnar's (2015) tutorial, which explains essentially the same calculus in more detail.

3.1 Syntax

Terms. There are two main kinds of terms, given in Figure 1: (pure) values v and (dirty) computations c, which may call effectful operations. We assume a given set of *constants* k, such as `true`, `false`, integer literals, arithmetic functions $+, -, =, <, >$, or similar, and of *operations* Op, such as `Get` or `Put`.

$$\begin{array}{rcll}
\text{value } v & ::= & x & \text{variable} \\
& \mid & \mathsf{k} & \text{constant} \\
& \mid & \mathtt{fun}\ x \mapsto c & \text{function} \\
& \mid & \{ & \text{handler} \\
& & \quad \mathtt{return}\ x \mapsto c_r, & \quad \text{return case} \\
& & \quad \mathsf{Op}_1\ x\ k \mapsto c_{\mathsf{Op}_1}, \ldots, \mathsf{Op}_n\ x\ k \mapsto c_{\mathsf{Op}_n} & \quad \text{operation cases} \\
& & \} & \\
\text{computation } c & ::= & v_1\ v_2 & \text{application} \\
& \mid & \mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2 & \text{recursive definition} \\
& \mid & \mathtt{return}\ v & \text{returned value} \\
& \mid & \mathsf{Op}\ v & \text{operation call} \\
& \mid & \mathtt{do}\ x \leftarrow c_1\ ;\ c_2 & \text{sequencing} \\
& \mid & \mathtt{handle}\ c\ \mathtt{with}\ v & \text{handling}
\end{array}$$

Fig. 1. Terms of Effy

We often abbreviate $\mathsf{Op}_1\ x\ k \mapsto c_{\mathsf{Op}_1}, \ldots, \mathsf{Op}_n\ x\ k \mapsto c_{\mathsf{Op}_n}$ as $[\mathsf{Op}\ x\ k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}$, and write O to denote the set of handled operations $\{\mathsf{Op}_1, \ldots, \mathsf{Op}_n\}$.

Types. Similarly, we distinguish between two kinds of types: the pure types A, B of values and the dirty types $\underline{C}, \underline{D}$ of computations, described in Figure 2. Type $A \to \underline{C}$ is given to functions that take an argument of type A and perform a computation of type $\underline{C}$, while the type $\underline{C} \Rightarrow \underline{D}$ is given to handlers that turn a computation of type $\underline{C}$ into one of type $\underline{D}$. The dirty type $A \mathbin{!} \Delta$ is assigned to a computation returning values of type A and potentially calling operations from the set Δ.

3.2 Type System

3.2.1 *Subtyping.*

The dirty type $A \mathbin{!} \Delta$ is assigned to a computation returning values of type A and potentially calling operations from the set Δ. This set Δ is always an over-approximation of the actually called operations, and may safely be increased, inducing a natural subtyping judgement $A \mathbin{!} \Delta \leq A \mathbin{!} \Delta'$ on dirty types. As dirty types can occur inside pure types, we also get a derived subtyping judgement on pure types. Both judgements

$$
\begin{array}{rcll}
\text{(pure) type } A, B & ::= & \texttt{bool} \mid \texttt{int} & \text{basic types} \\
& \mid & A \to \underline{C} & \text{function type} \\
& \mid & \underline{C} \Rightarrow \underline{D} & \text{handler type} \\
\text{dirty type } \underline{C}, \underline{D} & ::= & A \,!\, \Delta & \\
\text{dirt } \Delta & ::= & \{\mathtt{Op}_1, \ldots, \mathtt{Op}_n\} &
\end{array}
$$

Fig. 2. Types of Effy

Subtyping

$$
\textsc{Sub-bool}\ \frac{}{\texttt{bool} \leqslant \texttt{bool}}
\qquad
\textsc{Sub-int}\ \frac{}{\texttt{int} \leqslant \texttt{int}}
\qquad
\textsc{Sub-}\!\to\ \frac{A' \leqslant A \qquad \underline{C} \leqslant \underline{C}'}{A \to \underline{C} \leqslant A' \to \underline{C}'}
\qquad
\textsc{Sub-}\!\Rightarrow\ \frac{\underline{C}' \leqslant \underline{C} \qquad \underline{D} \leqslant \underline{D}'}{\underline{C} \Rightarrow \underline{D} \leqslant \underline{C}' \Rightarrow \underline{D}'}
\qquad
\textsc{Sub-!}\ \frac{A \leqslant A' \qquad \Delta \subseteq \Delta'}{A \,!\, \Delta \leqslant A' \,!\, \Delta'}
$$

Fig. 3. Subtyping for pure and dirty types

are defined in Figure 3. Observe that, as usual, subtyping is contravariant in the argument types of functions and handlers, and covariant in their return types.

3.2.2 Typing. Figure 4 defines the typing judgements for values and computations with respect to a standard typing context Γ.

Values. The rules for subtyping, variables, and functions are entirely standard. For constants we assume a signature Σ that assigns a type A to each constant k, which we write as $(\mathsf{k} : A) \in \Sigma$.

A handler expression has type $A \,!\, \Delta \cup O \Rightarrow B \,!\, \Delta$ iff all branches (both the operation cases and the return case) have dirty type $B \,!\, \Delta$ and the operation cases cover the set of operations O. Note that the intersection $\Delta \cap O$ is not necessarily empty. The handler deals with the operations O, but in the process may re-issue some of them (i.e., $\Delta \cap O$).

When typing operation cases, the given signature for the operation $(\mathtt{Op} : A_{\mathtt{Op}} \to B_{\mathtt{Op}}) \in \Sigma$ determines the type $A_{\mathtt{Op}}$ of the parameter x and the domain $B_{\mathtt{Op}}$ of the continuation k. As our handlers are deep, the codomain of k should be the same as the type $B \,!\, \Delta$ of the cases.

Computations. With the following exceptions, the typing judgement $\Gamma \vdash c : \underline{C}$ has a straightforward definition. The `return` construct renders a value v as a pure computation, i.e., with empty dirt. An operation invocation $\mathtt{Op}\,v$ is typed according to the operation's signature, with the operation itself as its only operation. Finally, rule With shows that a handler with type $\underline{C} \Rightarrow \underline{D}$ transforms a computation with type $\underline{C}$ into a computation with type $\underline{D}$.

3.3 Big-Step Operational Semantics

Figure 5 defines the big-step operational semantics of Effy. The judgement $c \Downarrow r$ states that computation c reduces to result r. A result is either a returned value, `return` v, or an unhandled operation, $\mathtt{Op}\,v\,(y.c)$, where v is the operation's parameter and $y.c$ is its continuation.

The rules Eval-App and Eval-LetRec are straightforward. Next, the value result is generated by the `return` v computation (Eval-Ret), while the unhandled operation (with trivial continuation $y.$`return` y) is generated by the $\mathtt{Op}\,v$ computation (Eval-Op). If the intermediate result of a sequential do is a value (Eval-Do-Ret), it is substituted into the second computation. If it is an unhandled operation (Eval-Do-Op), the second computation

$$\text{typing contexts } \Gamma \ ::= \ \epsilon \mid \Gamma, x : A$$

Expressions

$$\textsc{SubVal}\ \frac{\Gamma \vdash v : A \qquad A \leqslant A'}{\Gamma \vdash v : A'} \qquad \textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Const}\ \frac{(\mathsf{k} : A) \in \Sigma}{\Gamma \vdash \mathsf{k} : A} \qquad \textsc{Fun}\ \frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \mathtt{fun}\ x \mapsto c : A \to \underline{C}}$$

$$\textsc{Hand}\ \frac{\Gamma, x : A \vdash c_r : B \,!\, \Delta \qquad \Big[(\mathsf{Op} : A_{\mathsf{Op}} \to B_{\mathsf{Op}}) \in \Sigma \qquad \Gamma, x : A_{\mathsf{Op}}, k : B_{\mathsf{Op}} \to B \,!\, \Delta \vdash c_{\mathsf{Op}} : B \,!\, \Delta\Big]_{\mathsf{Op} \in O}}{\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r, [\mathsf{Op}\, x\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\} : A \,!\, \Delta \cup O \Rightarrow B \,!\, \Delta}$$

Computations

$$\textsc{SubComp}\ \frac{\Gamma \vdash c : \underline{C} \qquad \underline{C} \leqslant \underline{C}'}{\Gamma \vdash c : \underline{C}'} \qquad \textsc{App}\ \frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\, v_2 : \underline{C}}$$

$$\textsc{LetRec}\ \frac{\Gamma, f : A \to \underline{C}, x : A \vdash c_1 : \underline{C} \qquad \Gamma, f : A \to \underline{C} \vdash c_2 : \underline{D}}{\Gamma \vdash \mathtt{let\ rec}\ f\, x = c_1\ \mathtt{in}\ c_2 : \underline{D}} \qquad \textsc{Ret}\ \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathtt{return}\ v : A \,!\, \emptyset}$$

$$\textsc{Op}\ \frac{(\mathsf{Op} : A \to B) \in \Sigma \qquad \Gamma \vdash v : A}{\Gamma \vdash \mathsf{Op}\, v : B \,!\, \{\mathsf{Op}\}} \qquad \textsc{Do}\ \frac{\Gamma \vdash c_1 : A \,!\, \Delta \qquad \Gamma, x : A \vdash c_2 : B \,!\, \Delta}{\Gamma \vdash \mathsf{do}\ x \leftarrow c_1 \,;\, c_2 : B \,!\, \Delta} \qquad \textsc{With}\ \frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathtt{handle}\ c\ \mathtt{with}\ v : \underline{D}}$$

Fig. 4. Typing

is appended to its continuation. When a handled computation evaluates to a value (Eval-With-Ret), this value is substituted into the handler's return case. Finally, an unhandled operation is passed to the appropriate operation case, if there is one (Eval-With-Handled-Op), or propagated further, if there is not (Eval-With-Unhandled-Op). In either case, the continuation $y.c'$ is handled by the same handler.

3.4 Reasoning

To facilitate equational reasoning about programs with algebraic effect handlers, Bauer & Pretnar (2014) establish several observational equivalences, which we adapt to Effy in Figure 6 (we omit structural equivalences that make $\equiv$ a congruence). The meta-variables v, c stand for arbitrary values and computations that make the left- and right-hand sides of the equivalences well-typed with the same type.

Moreover, algebraic effects validate an induction principle (Bauer and Pretnar 2014; Plotkin and Pretnar 2008) (see (Pretnar 2015) for a more detailed explanation): to show that a property ϕ holds for all computations $\Gamma \vdash c : A \,!\, \Delta$, it suffices to show that 1) $\phi(\mathtt{return}\ v)$ holds for all values v, and 2) that $\phi(\mathsf{do}\ x \leftarrow \mathsf{Op}\, v \,;\, c')$ holds for all operations $\mathsf{Op} \in \Delta$, values v and computations c', given that $\phi(c')$ holds.

These equivalences and induction principle will turn out to be useful in the next section to prove the soundness of the optimisations.

$$\text{result } r \ ::= \ \mathsf{return}\ v \ \mid \ \mathsf{Op}\, v\, (y.\, c)$$

Evaluation

Eval-App
$$\frac{c[v/x] \Downarrow r}{(\mathsf{fun}\ x \mapsto c)\ v \Downarrow r}$$

Eval-LetRec
$$\frac{c_2[(\mathsf{fun}\ x \mapsto \mathsf{let\ rec}\ f\ x = c_1\ \mathsf{in}\ c_1)/f] \Downarrow r}{\mathsf{let\ rec}\ f\ x = c_1\ \mathsf{in}\ c_2 \Downarrow r}$$

Eval-Ret
$$\frac{}{\mathsf{return}\ v \Downarrow \mathsf{return}\ v}$$

Eval-Op
$$\frac{}{\mathsf{Op}\, v \Downarrow \mathsf{Op}\, v\, (y.\, \mathsf{return}\ y)}$$

Eval-Do-Ret
$$\frac{c_1 \Downarrow \mathsf{return}\ v \qquad c_2[v/x] \Downarrow r}{\mathsf{do}\ x \leftarrow c_1\ ;\ c_2 \Downarrow r}$$

Eval-Do-Op
$$\frac{c_1 \Downarrow \mathsf{Op}\, v\, (y.\, c_1')}{\mathsf{do}\ x \leftarrow c_1\ ;\ c_2 \Downarrow \mathsf{Op}\, v\, (y.\, \mathsf{do}\ x \leftarrow c_1'\ ;\ c_2)}$$

Eval-With-Ret
$$\frac{c \Downarrow \mathsf{return}\ v \qquad c_r[v/x] \Downarrow r}{\mathsf{handle}\ c\ \mathsf{with}\ h \Downarrow r}$$

Eval-With-Handled-Op
$$\frac{c \Downarrow \mathsf{Op}\, v\, (y.\, c') \qquad c_{\mathsf{Op}}[v/x, (\mathsf{fun}\ y \mapsto \mathsf{handle}\ c'\ \mathsf{with}\ h)/k] \Downarrow r}{\mathsf{handle}\ c\ \mathsf{with}\ h \Downarrow r}$$

Eval-With-Unhandled-Op
$$\frac{c \Downarrow \mathsf{Op}'\, v\, (y.\, c') \qquad \mathsf{Op}' \notin O}{\mathsf{handle}\ c\ \mathsf{with}\ h \Downarrow \mathsf{Op}'\, v\, (y.\, \mathsf{handle}\ c'\ \mathsf{with}\ h)}$$

Fig. 5. Operational semantics (in the last three rules, $h = \{\mathsf{return}\ x \mapsto c_r, [\mathsf{Op}\, x\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\}$)

$$\begin{array}{rcll}
(\mathsf{fun}\ x \mapsto\ c)\ v & \equiv & c[v/x] & (1)\\
\mathsf{fun}\ x \mapsto\ v\, x & \equiv & v & (2)\\
\mathsf{let\ rec}\ f\ x = c_1\ \mathsf{in}\ c_2 & \equiv & c_2[(\mathsf{fun}\ x \mapsto\ \mathsf{let\ rec}\ f\ x = c_1\ \mathsf{in}\ c_1)/f] & (3)\\
\mathsf{do}\ x \leftarrow \mathsf{return}\ v\ ;\ c & \equiv & c[v/x] & (4)\\
\mathsf{do}\ x \leftarrow c\ ;\ \mathsf{return}\ x & \equiv & c & (5)\\
\mathsf{do}\ x_2 \leftarrow (\mathsf{do}\ x_1 \leftarrow c_1\ ;\ c_2)\ ;\ c_3 & \equiv & \mathsf{do}\ x_1 \leftarrow c_1\ ;\ (\mathsf{do}\ x_2 \leftarrow c_2\ ;\ c_3) & (6)\\
\mathsf{handle}\ c_1\ \mathsf{with}\ \{\mathsf{return}\ x \mapsto c_2\} & \equiv & \mathsf{do}\ x \leftarrow c_1\ ;\ c_2 & (7)\\
\mathsf{handle}\ (\mathsf{return}\ v)\ \mathsf{with}\ h & \equiv & c_r[v/x] & (8)\\
\mathsf{handle}\ (\mathsf{do}\ y \leftarrow \mathsf{Op}\, v\ ;\ c)\ \mathsf{with}\ h & \equiv & c_{\mathsf{Op}}[v/x, (\mathsf{fun}\ y \mapsto\ \mathsf{handle}\ c\ \mathsf{with}\ h)/k] & (9)\\
\mathsf{handle}\ (\mathsf{do}\ y \leftarrow \mathsf{Op}'\, v\ ;\ c)\ \mathsf{with}\ h & \equiv & \mathsf{do}\ y \leftarrow \mathsf{Op}'\, v\ ;\ \mathsf{handle}\ c\ \mathsf{with}\ h \qquad (\mathsf{Op}' \notin O) & (10)
\end{array}$$

Fig. 6. Basic Equivalences (in the last three rules, $h = \{\mathsf{return}\ x \mapsto c_r, [\mathsf{Op}\, x\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\}$)

4 SOURCE-LEVEL OPTIMISATIONS

This section discusses the heart of our optimising compiler, which consists of source-to-source transformations that aim to improve the runtime performance of the program.

4.1 Term Rewriting Rules

We use the information of the type and effect system and the syntactic structure of the terms to perform a number of optimisations. We mainly target optimisations that (help) remove handlers. We denote these

Simplification

$$\textsc{App-Fun}\quad \frac{}{(\mathsf{fun}\ x \mapsto c)\ v \rightsquigarrow c[v/x]} \qquad \textsc{Do-Ret}\quad \frac{}{\mathsf{do}\ x \leftarrow \mathtt{return}\ v\ ;\ c \rightsquigarrow c[v/x]}$$

$$\textsc{Do-Op}\quad \frac{}{\mathsf{do}\ x \leftarrow (\mathsf{do}\ y \leftarrow \mathsf{Op}\,v\ ;\ c_1)\ ;\ c_2 \quad \rightsquigarrow \quad \mathsf{do}\ y \leftarrow \mathsf{Op}\,v\ ;\ (\mathsf{do}\ x \leftarrow c_1\ ;\ c_2)}$$

Handler Reduction

$$\textsc{With-LetRec}\quad \frac{}{\mathtt{handle}\ (\mathtt{let\ rec}\ f\,x = c_1\ \mathtt{in}\ c_2)\ \mathtt{with}\ v \rightsquigarrow \mathtt{let\ rec}\ f\,x = c_1\ \mathtt{in}\ (\mathtt{handle}\ c_2\ \mathtt{with}\ v)}$$

$$\textsc{With-Ret}\quad \frac{h = \{\mathtt{return}\ x \mapsto c_r, [\mathsf{Op}\,x\,k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op}\in O}\}}{\mathtt{handle}\ (\mathtt{return}\ v)\ \mathtt{with}\ h \rightsquigarrow c_r[v/x]} \qquad \textsc{With-Handled-Op}\quad \frac{h = \{\mathtt{return}\ x \mapsto c_r, [\mathsf{Op}\,x\,k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op}\in O}\}}{\mathtt{handle}\ (\mathsf{Op}\,v)\ \mathtt{with}\ h \rightsquigarrow c_{\mathsf{Op}}[v/x, (\mathsf{fun}\ x \mapsto c_r)/k]}$$

$$\textsc{With-Pure}\quad \frac{h = \{\mathtt{return}\ x \mapsto c_r, [\mathsf{Op}\,x\,k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op}\in O}\} \qquad \Gamma \vdash c : A\ !\ \Delta \qquad \Delta \cap O = \emptyset}{\mathtt{handle}\ c\ \mathtt{with}\ h \rightsquigarrow \mathsf{do}\ x \leftarrow c\ ;\ c_r}$$

$$\textsc{With-Do}\quad \frac{\begin{array}{c} h = \{\mathtt{return}\ x \mapsto c_r, [\mathsf{Op}\,x\,k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op}\in O}\} \\ h' = \{\mathtt{return}\ y \mapsto (\mathtt{handle}\ c_2\ \mathtt{with}\ h), [\mathsf{Op}\,x\,k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op}\in O}\}\end{array}}{\mathtt{handle}\ (\mathsf{do}\ y \leftarrow c_1\ ;\ c_2)\ \mathtt{with}\ h \rightsquigarrow \mathtt{handle}\ c_1\ \mathtt{with}\ h'}$$

Fig. 7. Term Rewriting Rules

optimisations in terms of semantics-preserving rewrite rules of the form $c_1 \rightsquigarrow c_2$, listed in Figure 7. The rewriting rules are divided into two groups:

Simplification. These three rules simplify the structure of the program in the hope of exposing opportunities to eliminate handlers. The first two, App-Fun and Do-Ret, are static counterparts of the semantic rules Eval-App and Eval-Do-Ret. The last rule, Do-Op, exploits the associativity of do to bring an operation to the front, where it can potentially be reduced by a handler.

Handler Reduction. These rules reduce terms of the form (`handle` c `with` h) for different shapes of computation c and are the heart of our optimization. Rule With-LetRec moves the handler into the main subcomputation. Rules With-Ret and With-Handled-Op are the static counterparts of the semantic rules Eval-With-Ret and Eval-With-Handled-Op.

Next, rule With-Pure applies when the computation c is pure relative to the handler h. This is the case when the intersection of the operations that may be called by c with the operations handled by h is empty. In this case, only the handler's return case is relevant. Hence, we insert it at the end of c.

Finally, the most unusual rule is With-Do which reduces the handling of a sequence of two computations to a form where the two computations are handled separately. The validity of this transformation becomes more obvious when we split it into two steps:

(1) We replace the sequential do with a handler:

$$\mathsf{do}\ y \leftarrow c_1\ ;\ c_2 \quad \rightsquigarrow \quad \mathsf{handle}\ c_1\ \mathtt{with}\ \{\mathsf{return}\ \ y \mapsto c_2\}$$

The intuition is that both forms express that the value returned by c_1 gets bound to y in c_2.

(2) We change the association of the handlers:

$$\mathsf{handle}\ (\mathsf{handle}\ c_1\ \mathtt{with}\ \{\mathsf{return}\ \ y \mapsto c_2\})\ \mathtt{with}\ \{\mathsf{return}\ x \mapsto c_r, [\mathsf{Op}\, x\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op}\in O}\}$$
$$\rightsquigarrow$$
$$\mathsf{handle}\ c_1\ \mathtt{with}\ \{\mathsf{return}\ \ y \mapsto (\mathsf{handle}\ c_2\ \mathtt{with}\ \{\mathsf{return}\ x \mapsto c_r, [\mathsf{Op}\, x\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op}\in O}\}), [\mathsf{Op}\, x\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op}\in O}\}$$

The intuition is that any operations generated by c_1 are forwarded anyway from the inner handler to the outer handler. However, any return case is first handled by the inner handler, and the computation that results is further processed by the outer handler. The rewritten form accomplishes the same workflow with a single handler around c_1. The hope is that the handler around c_2 can be specialised independently from specialising the handler around c_1.

Soundness. The rewrite rules preserve observational equivalence.

Theorem 4.1.

$$\forall c, c', \Gamma, A, \Delta :\ \Gamma \vdash c : A\,!\,\Delta\ \wedge\ c \rightsquigarrow c'\ \Rightarrow\ c \equiv c'$$

The proof of this theorem, given in Appendix A, makes use of the equivalences and induction principle presented in the previous section.

4.2 Function Specialisation

The rewrite rules above deal with most computations of the form (handle c with h) where h is a handler expression, either dropping the handler altogether or pushing it down in the subcomputations. However, one important case is not dealt with: the case where c is of the form $f\ v$ with f the name of a user-defined recursive function.[1]

Consider this small example of the above situation:

```
let rec go n = go (Next n) in
handle (go 0) with
| return x -> x
| Next n k -> if n > 100 then n else k (n * n + 1)
```

The non-terminating recursive function go seems to diverge. Yet, with the provided handler, its argument steadily increases and evaluation eventually terminates when the argument exceeds 100.

In order to optimise this situation, we create a specialised copy of the function that has the handler pushed into its body. In other words, for any recursive definition let rec $f\ x = c_f$ in c, we perform the following general rewrite inside c:

$$\mathsf{handle}\ f\ v\ \mathsf{with}\ h \quad \rightsquigarrow \quad \mathsf{let\ rec}\ f'\ x = \mathsf{handle}\ c_f\ \mathsf{with}\ h\ \mathsf{in}\ f'\ v$$

The expectation is that, by exposing the handler to the body of the function (c_f), further optimizations succeed in eliminating the explicit handler. A critical step involved in the post-processing is to "tie the knot": after several rewrite steps in c_f, the handler is applied to the (original) recursive call, so we have a term of the form handle $f\ v'$ with h, which we can replace by $f'\ v'$. This eliminates the handler entirely and turns the original example into

[1]If f is a function parameter of a higher-order function, we don't do anything.

```
let rec go n = ... in
let rec go' n = if n > 100 then n else go' (n * n + 1) in
go' 0
```

Generalization to Non-Tail Recursion. The above basic specialisation strategy only works when the recursive call has a tail position. Yet, that is often not the case. Take for instance the following example.

```
let rec range n =
  match n with
  | 0 -> []
  | _ -> Fetch () :: range (n - 1)
in
handle (range 5) with
| return x  -> x
| Fetch _ k -> k 42
```

The function range creates a list of given length, filling it with elements obtained by the Fetch operation. To keep the example small we use a handler that always yields the value 42.

With the basic specialisation strategy, further optimisation does not succeed in tying the knot. Instead, we obtain this partially optimised form:

```
let rec range n = ... in
let rec range' n =
  match n with
  | 0 -> []
  | _ -> handle (range (n - 1)) with
         | return x  -> 42 :: x
         | Fetch _ k -> k 42
in
range' 5
```

In the tail position, the rewrite rule With-Do has kicked in to pull the call's continuation into the return case of the handler. This has turned the call into a tail call, but the return case of the new handler around this call differs from the original handler's return case. This prevents us from tying the knot.

We could create a second specialised function definition for this new handler, but the same problem would arise at its recursive call and so on, yielding an infinite sequence of specialised functions. Instead, we use generalisation to break out of this diverging process. Instead of specialising the function for one specific handler in this diverging sequence, we specialise it for what they all have in common (the operation cases) and parametrise it in what is different (the return case).

This yields the following general rewrite rule: for any recursive definition $\texttt{let rec}\ f\ x = c_f\ \texttt{in}\ c$, we perform the following general rewrite inside c:

$$\texttt{handle}\ f\ v\ \texttt{with}\ \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}$$

$$\leadsto$$

$$\texttt{let rec}\ f'\ (x,k) = \texttt{handle}\ c_f\ \texttt{with}\ \{\texttt{return}\ x \mapsto k\ x, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}\ \texttt{in}\ f'\ (v, \texttt{fun}\ x \mapsto c_r)$$

and replace each handled recursive call $\texttt{handle}\ f\ v'\ \texttt{with}\ \{\texttt{return}\ x \mapsto c_r', [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in\mathcal{O}}\}$ with $f'\ (v', c_r')$.

This strategy enables us to tie the knot in the range example and obtain this form

```
let rec range' (n, k) =
  match n with
  | 0 -> k []
  | _ -> range' (n - 1, fun x -> k (42 :: x))
in
range' (5, fun x -> x)
```

Note that in effect this approach selectively CPS-transforms recursive functions to specialise them for a particular handler.

Termination. If left unchecked, function specialisation can diverge. This is illustrated by the following small example program:

```
let rec go n =
  if n = 0 then Fail
    else if Decide then go (n-1) else go (n-2)
in handle (go m) with
    | Decide _ k ->
        handle k true with
        | Fail _ _ -> k false
```

After specialisation for the top-level handler, we obtain

```
let rec go n =
  if n = 0 then Fail
    else if Decide then go (n-1) else go (n-2)
in let rec go1 n1 =
      if n1 = 0 then Fail
        else handle go1 (n1-1) with
              | Fail -> go1 (n1-2)
     in go1 m
```

Note that the specialised function go1 still contains the second handler, which is now applied to a recursive call. Hence, we can continue by specialising this handled call to obtain

```
let rec go n =
  if n = 0 then Fail
    else if Decide then go (n-1) else go (n-2)
in let rec go1 n1 =
      if n1 = 0 then Fail
        else let rec go2 n2 =
               if n2 = 0 then go1 (n1-2)
                 else handle (handle go1 (n2-1) with
                              | Fail -> go1 (n2-2)) with
                      | Fail -> go1 (n1-2)
             in go2 (n1-1)
     in go1 m
```

Now the resulting code contains two nested handlers around a recursive call. However, the inner of those two handlers is distinct from any of the previous handlers because it refers to the new variable n2. Hence, we can specialise again and again without end. This is non-termination is obviously undesirable and so we currently enforce termination by not re-specialising any already specialised function. We leave more sophisticated solutions, e.g., abstracting over the variation among the specialised handlers, to future work.

5 BASIC TRANSLATION TO OCAML

5.1 Translating Types

As a concrete target for translating Effy, we pick a small subset of OCaml that includes standard constructs such as booleans, integers, functions and local definitions (both non-recursive and recursive). Its types are given in Figure 8, and in addition to the standard ones, they include a predefined type T `computation`, which represents computations returning values of type T, and a type (T_1, T_2) `handler_cases`, which lists all cases of a handler that takes computations of type T_1 `computation` into T_2.

$$\text{type } T \;::=\; \texttt{bool} \mid \texttt{int} \mid T_1 \to T_2 \mid T\ \texttt{computation} \mid (T_1, T_2)\ \texttt{handler\_cases}$$

Fig. 8. Types of (a subset of) OCaml

$$\begin{aligned} [\![\texttt{bool}]\!] &= \texttt{bool} \\ [\![\texttt{int}]\!] &= \texttt{int} \\ [\![A \to \underline{C}]\!] &= [\![A]\!] \to [\![\underline{C}]\!] \\ [\![\underline{C} \Rightarrow \underline{D}]\!] &= [\![\underline{C}]\!] \to [\![\underline{D}]\!] \\ \\ [\![A \,!\, \Delta]\!] &= [\![A]\!]\ \texttt{computation} \end{aligned}$$

Fig. 9. Compilation of Effy types to OCaml

We translate types of Effy into OCaml by means of the compilation function $[\![\cdot]\!]$ listed in Figure 9. Primitive types and function types are straightforwardly mapped onto their OCaml counterparts. The handler type is translated to a function type that turns one type of computation into another.

Computation types are mapped to the predefined `computation` type, defined by default as a datatype representation of a free monad (where $\mathsf{Op}_i : A_i \to B_i$ ranges over the signature of all Effy operations):

```
type 'a computation =
| Return: 'a -> 'a computation
| Op₁: ⟦A₁⟧ * (⟦B₁⟧ -> 'a computation) -> 'a computation
| ...
| Opₙ: ⟦Aₙ⟧ * (⟦Bₙ⟧ -> 'a computation) -> 'a computation
```

Here `Return x` represents a value `x` as a (pure) computation, and `Op x k` denotes an impure computation that calls operation Op with argument `x` and continuation `k`. We can interchange the implementation of `'a computation` to obtain different runtime representations, though in this paper, we fix the free monad representation, as it is both simple and efficient enough for our purposes.

Note that the translation erases the dirt Δ from computation types $A \,!\, \Delta$, for lack of a convenient way to represent it in OCaml. The algebraic effect handlers implementation of Multicore OCaml (Dolan et al. 2015) has made a similar choice not to reflect the set of possible operations in the type.

$$
\begin{array}{rcl}
\text{expression } E & ::= & x \\
& | & \mathtt{k} \\
& | & \mathtt{fun}\ x \mapsto E \\
& | & E_1\, E_2 \\
& | & \mathtt{let}\ x = E_1\ \mathtt{in}\ E_2 \\
& | & \mathtt{let\ rec}\ f\, x = E_1\ \mathtt{in}\ E_2 \\
& | & \{\mathtt{return} = E; \mathtt{op}_1 = E_1; \dots; \mathtt{op}_n = E_n\} \\
& | & \mathtt{return} \\
& | & \mathtt{op}_1 \mid \cdots \mid \mathtt{op}_n \\
& | & \mathtt{handler} \\
& | & \gg=
\end{array}
$$

Fig. 10. Terms of (a subset of) OCaml

Finally, the type of all handler cases is defined to be the record type:

```
type ('a, 'b) handler_cases = {
  return: 'a -> 'b;
  op₁: ⟦A₁⟧ -> (⟦B₁⟧ -> 'b) -> 'b;
  ...
  opₙ: ⟦Aₙ⟧ -> (⟦Bₙ⟧ -> 'b) -> 'b
}
```

Here the operation cases are represented by a function that takes the argument and the continuation of the operation and then performs the operation's corresponding behaviour. (Note that the domain 'b is not necessarily of the form _ computation. We exploit this fact in Section 6 to allow handlers that handle all computations into a pure value.)

5.2 Translating Terms

OCaml terms, given in Figure 10, include the standard ones: variables, constants (corresponding to ones in Effy), function abstractions & applications, and both non-recursive & recursive local definitions. Next, we include records that list handler cases and a number of predefined functions for value embedding, operations, handler definitions and sequencing. Both Effy values and computations are translated to OCaml expressions as described in Figure 11. With several notable exceptions, most forms have a direct counterpart in OCaml.

A handler value is translated to an application of the handler function to a record value that gathers the return and operation cases. For the default free monad representation, handler is defined as follows:

```
let rec handler (h : ('a, 'b) handler_cases) : ('a computation -> 'b) =
  function
  | Return x -> h.return x
  | Op₁ (x, k) -> h.op₁ x (fun y -> handler h (k y))
  ...
  | Opₙ (x, k) -> h.opₙ x (fun y -> handler h (k y))
```

In case a handler does not provide a handling case for an operation Op_i, we fill it in with a default case that propagates it outwards, in which case 'b needs to be of the form _ computation. Note that this is always the case with the basic translation presented in this section.

$$
\begin{aligned}
[\![x]\!] &= x \\
[\![\mathsf{k}]\!] &= \mathsf{k} \\
[\![\mathsf{fun}\ x \mapsto c]\!] &= \mathsf{fun}\ x \mapsto [\![c]\!] \\
[\![\{\mathsf{return}\ x \mapsto c_r, [\mathsf{Op}\, x\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\}]\!] &= \mathsf{handler}\ \{\mathsf{return} = \mathsf{fun}\ x \mapsto [\![c_r]\!]; \mathsf{op}_1 = E_1; \ldots; \mathsf{op}_n = E_n\} \\
\text{where } E_i &= \begin{cases} \mathsf{fun}\ x\, k \mapsto [\![c_{\mathsf{Op}_i}]\!] & \mathsf{Op}_i \in O \\ \mathsf{fun}\ x\, k \mapsto \mathsf{op}_i\ x \mathbin{>\!\!>\!=} k & \text{otherwise} \end{cases} \\
\\
[\![v_1\ v_2]\!] &= [\![v_1]\!]\ [\![v_2]\!] \\
[\![\mathsf{let\ rec}\ f\ x = c_1\ \mathsf{in}\ c_2]\!] &= \mathsf{let\ rec}\ f\ x = [\![c_1]\!]\ \mathsf{in}\ [\![c_2]\!] \\
[\![\mathsf{return}\ v]\!] &= \mathsf{return}\ [\![v]\!] \\
[\![\mathsf{Op}_i\ v]\!] &= \mathsf{op}_i\ [\![v]\!] \\
[\![\mathsf{do}\ x \leftarrow c_1\ ;\ c_2]\!] &= [\![c_1]\!] \mathbin{>\!\!>\!=} (\mathsf{fun}\ x \mapsto [\![c_2]\!]) \\
[\![\mathsf{handle}\ c\ \mathsf{with}\ v]\!] &= [\![v]\!]\ [\![c]\!]
\end{aligned}
$$

Fig. 11. Compilation of Effy terms to OCaml

Building a computation from a value, from an operation or by binding two computations together all happens in terms of the corresponding operations on the underlying free monad representation.

```
let return (x : 'a) : 'a computation = Return x

let op₁ (x : ⟦A₁⟧) (k : ⟦B₁⟧ -> 'a computation) : 'a computation = Op₁ (x, k)
...
let opₙ (x : ⟦Aₙ⟧) (k : ⟦Bₙ⟧ -> 'a computation) : 'a computation = Opₙ (x, k)

let rec (>>=) (c : 'a computation) (f : 'a -> 'b computation) : 'b computation =
  match c with
  | Return x -> f x
  | Op₁ (x, k) -> Op₁ (x, (fun y -> (k y) >>= f))
  ...
  | Opₙ (x, k) -> Opₙ (x, (fun y -> (k y) >>= f))
```

Finally, handling a computation with a handler is simply translated as applying the handler to the computation. This translation is type-preserving.

Theorem 5.1.

$$\Gamma \vdash c : \underline{C} \quad \Longrightarrow \quad [\![\Gamma]\!] \vdash [\![c]\!] : [\![\underline{C}]\!] \qquad\qquad (\forall c, t, \Gamma)$$

The proof of this theorem is given in Appendix B; it proceeds by induction on the typing derivation.

$$\frac{}{(\mathtt{bool} \leqslant \mathtt{bool}) \leadsto (\mathtt{fun}\ x \mapsto\ x)}\ \textsc{Sub-bool} \qquad \frac{}{(\mathtt{int} \leqslant \mathtt{int}) \leadsto (\mathtt{fun}\ x \mapsto\ x)}\ \textsc{Sub-int}$$

$$\frac{(A' \leqslant A) \leadsto E_1 \qquad (\underline{C} \leqslant \underline{C}') \leadsto E_2}{(A \to \underline{C} \leqslant A' \to \underline{C}') \leadsto (\mathtt{fun}\ f\ x \mapsto\ E_2\ (f\ (E_1\ x)))}\ \textsc{Sub-}{\to}$$

$$\frac{(\underline{C}' \leqslant \underline{C}) \leadsto E_1 \qquad (\underline{D} \leqslant \underline{D}') \leadsto E_2}{(\underline{C} \Rightarrow \underline{D} \leqslant \underline{C}' \Rightarrow \underline{D}') \leadsto (\mathtt{fun}\ h\ x \mapsto\ E_2\ (h\ (E_1\ x)))}\ \textsc{Sub-}{\Rightarrow} \qquad \frac{(A \leqslant A') \leadsto E}{(A\ !\ \emptyset \leqslant A'\ !\ \emptyset) \leadsto E}\ \textsc{Sub-!-Pure}$$

$$\frac{(A \leqslant A') \leadsto E \qquad \Delta' \neq \emptyset}{(A\ !\ \emptyset \leqslant A'\ !\ \Delta') \leadsto (\mathtt{fun}\ x \mapsto\ \mathtt{return}\ (E\ x))}\ \textsc{Sub-!-PureImpure} \qquad \frac{(A \leqslant A') \leadsto E \qquad \Delta \subseteq \Delta' \qquad \Delta \neq \emptyset}{(A\ !\ \Delta \leqslant A'\ !\ \Delta') \leadsto (\mathtt{fmap}\ E)}\ \textsc{Sub-!-Impure}$$

Fig. 12. Subtyping induced coercions

6 PURITY-AWARE TRANSLATION TO OCAML

The basic compilation scheme's free monad representation introduces a substantial performance overhead for pure computations. This section presents an extended compilation scheme that avoids this overhead for pure computations.

The main aim of our extended compilation scheme is to differentiate between pure and impure computations. This is nicely summarised in the more nuanced compilation of computation types:

$$[\![A\ !\ \Delta]\!] = \begin{cases} [\![A]\!] & \Delta = \emptyset \\ [\![A]\!]\ \mathtt{computation} & \Delta \neq \emptyset \end{cases}$$

At the term level we express the extended compilation by means of type-and-effect-directed elaboration judgements that extend the declarative type system with a target OCaml expression. The crucial ingredient are judgements that elaborate subtyping of value types $A \leq B$ or computation types $\underline{C} \leq \underline{D}$ into functions that coerce between the two types (Figure 12).

The reflexive cases for bool and int yield an identity coercion, while function and handler types give rise to pre- and post-composition of the coercions. We distinguish three different cases for the coercion between computation types. If both computations are pure, the coercion is just that between the values. A pure computation is coerced to an impure one by composing the value coercion with the return embedding. Finally, if the first computation is impure, so is the other one, and we map the coercion over the free monad with the predefined function

```
let rec fmap (f : 'a -> 'b) : ('a computation -> 'b computation) =
  function
  | Return x -> f x
  | Op₁ (x, k) -> Op₁ (x, (fun y -> fmap f (k y)))
  ...
  | Opₙ (x, k) -> Opₙ (x, (fun y -> fmap f (k y)))
```

Lemma 6.1. *For all pure types A and B, and for all computation types $\underline{C}$ and $\underline{D}$ we have that:*

$$(A \leqslant B) \rightsquigarrow E \implies \vdash E : [\![A]\!] \to [\![B]\!]$$

and

$$(\underline{C} \leqslant \underline{D}) \rightsquigarrow E \implies \vdash E : [\![\underline{C}]\!] \to [\![\underline{D}]\!]$$

The elaboration judgement $(\Gamma \vdash v : A) \rightsquigarrow E$ yields a corresponding OCaml expression E for the value v (see Figure 13). There are two noteworthy cases. Firstly, SubVal applies the subtyping coercion to the elaborated term. Secondly, HandPure singles out the case where the handler maps pure expressions to pure expressions, which is only possible when there are no operation cases. In this case, we elaborate the handler into its elaborated value case. The impure case, HandImpure, works similar to the basic translation. For each operation $\mathtt{Op}_i$, we need to provide a case E_i. If an operation is handled by a handler, we take the corresponding elaboration. If the operation is not handled, but is still listed in the outgoing dirt Δ, we propagate it as before. Finally, the case when the operation is neither handled nor listed in Δ can never occur at runtime, so we may safely raise an (OCaml) exception. Such treatment ensures that a handler with empty Δ and non-empty O is translated with a pure co-domain.

The elaboration judgement $(\Gamma \vdash c : \underline{C}) \rightsquigarrow E$ yields a corresponding OCaml expression E for the computation c. There are three differences with the basic compilation scheme. Firstly, a pure $\mathtt{return}\ v$ computation is translated just like the value v, i.e., without the `return` wrapper. Secondly, we distinguish between pure and impure do computations. The latter are translated in terms of the auxiliary >>= operator like before, but the former can be simplified to a more efficient OCaml **`let`** expression. Finally, computation subtyping is handled in the same way as expression subtyping, by applying the coercion function.

Just like the basic compilation scheme, this more advanced elaboration is type-preserving.

Theorem 6.2.

$$\Gamma \vdash c : \underline{C} \rightsquigarrow E \quad \implies \quad [\![\Gamma]\!] \vdash E : [\![\underline{C}]\!] \qquad (\forall \Gamma, c, \underline{C}, E)$$

The proof of this theorem is given in Appendix C.

7 IMPLEMENTATION IN EFF

To test the presented ideas in practice, we have implemented an optimising compiler for Eff, a prototype functional programming language with algebraic effects and handlers. This section describes the practical aspects of transforming Eff source code into efficient OCaml code, and points out the main differences between the idealised representation and the actual implementation.

7.1 Converting Source to Core Syntax

The actual source syntax of Eff is based on OCaml's and features only a single syntactic sort of terms, which lumps together values and computations. This source syntax is desugared into the core syntax, which is very close to Effy, in a straightforward manner (Bauer and Pretnar 2015). For example, Eff program **`if`** `f x` **`then`** `0` **`else`** `g 5 x` gets elaborated into

$$\mathtt{do}\ b \leftarrow f\,x\ ;\ \mathtt{if}\ b\ \mathtt{then}\ (\mathtt{return}\ 0)\ \mathtt{else}\ (\mathtt{do}\ h \leftarrow g\,5\ ;\ h\,x)$$

Our implementation supports standard features such as datatype declarations and control structures (like the conditional above), which we have omitted from Effy to avoid the clutter.

7.2 Type Inference

In this paper we have assumed that the type-and-effect information is available for all (sub)terms. Moreover, the purity-aware code generation of Section 6 assumes that the typing derivations themselves are also given.

Values

SubVal
$$\frac{(\Gamma \vdash v : A) \rightsquigarrow E_1 \qquad (A \leqslant A') \rightsquigarrow E_2}{(\Gamma \vdash v : A') \rightsquigarrow (E_2\, E_1)}$$

Var
$$\frac{(x : A) \in \Gamma}{(\Gamma \vdash x : A) \rightsquigarrow x}$$

Const
$$\frac{(\mathsf{k} : A) \in \Sigma}{(\Gamma \vdash \mathsf{k} : A) \rightsquigarrow \mathsf{k}}$$

Fun
$$\frac{(\Gamma, x : A \vdash c : \underline{C}) \rightsquigarrow E}{(\Gamma \vdash \mathsf{fun}\ x \mapsto c : A \to \underline{C}) \rightsquigarrow (\mathtt{fun}\ x \mapsto\ E)}$$

HandPure
$$\frac{(\Gamma, x : A \vdash c_r : B \,!\, \emptyset) \rightsquigarrow E_r}{(\Gamma \vdash \{\mathsf{return}\ x \mapsto c_r\} : A \,!\, \emptyset \Rightarrow B \,!\, \emptyset) \rightsquigarrow (\mathtt{fun}\ x \mapsto\ E_r)}$$

HandImpure
$$\frac{\begin{array}{c}(\Gamma, x : A \vdash c_r : B \,!\, \Delta) \rightsquigarrow E_r \\ \left[(\mathsf{Op} : A_{\mathsf{Op}} \to B_{\mathsf{Op}}) \in \Sigma \qquad (\Gamma, x : A_{\mathsf{Op}}, k : B_{\mathsf{Op}} \to B \,!\, \Delta \vdash c_{\mathsf{Op}} : B \,!\, \Delta) \rightsquigarrow E_{\mathsf{Op}}\right]_{\mathsf{Op} \in O} \\ E_i = \begin{cases} \mathtt{fun}\ x\, k \mapsto [\![c_{\mathsf{Op}_i}]\!] & \mathsf{Op}_i \in O \\ \mathtt{fun}\ x\, k \mapsto \mathtt{op}_i x \mathtt{>>=} k & \mathsf{Op}_i \in \Delta - O \\ \mathtt{fun}\ x\, k \mapsto \mathtt{assert\ false} & \text{otherwise} \end{cases}\end{array}}{\begin{array}{c}(\Gamma \vdash \{\mathsf{return}\ x \mapsto c_r, [\mathsf{Op}\, x\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\} : A \,!\, \Delta \cup O \Rightarrow B \,!\, \Delta) \\ \rightsquigarrow \mathtt{handler}\ \{\mathtt{return} = \mathtt{fun}\ x \mapsto E_r; \mathtt{op}_1 = E_1; \ldots; \mathtt{op}_n = E_n\}\}\end{array}}$$

Computations

SubComp
$$\frac{(\Gamma \vdash c : \underline{C}) \rightsquigarrow E_1 \qquad (\underline{C} \leqslant \underline{C}') \rightsquigarrow E_2}{(\Gamma \vdash c : \underline{C}') \rightsquigarrow (E_2\, E_1)}$$

App
$$\frac{(\Gamma \vdash v_1 : A \to \underline{C}) \rightsquigarrow E_1 \qquad (\Gamma \vdash v_2 : A) \rightsquigarrow E_2}{(\Gamma \vdash v_1\, v_2 : \underline{C}) \rightsquigarrow (E_1\, E_2)}$$

LetRec
$$\frac{(\Gamma, f : A \to \underline{C}, x : A \vdash c_1 : \underline{C}) \rightsquigarrow E_1 \qquad (\Gamma, f : A \to \underline{C} \vdash c_2 : \underline{D}) \rightsquigarrow E_2}{(\Gamma \vdash \mathsf{let\ rec}\ f\, x = c_1\ \mathsf{in}\ c_2 : \underline{D}) \rightsquigarrow (\mathtt{let\ rec}\ f\, x = E_1\ \mathtt{in}\ E_2)}$$

Ret
$$\frac{(\Gamma \vdash v : A) \rightsquigarrow E}{(\Gamma \vdash \mathsf{return}\ v : A \,!\, \emptyset) \rightsquigarrow E}$$

Op
$$\frac{(\mathsf{Op} : A \to B) \in \Sigma \qquad (\Gamma \vdash v : A) \rightsquigarrow E}{(\Gamma \vdash \mathsf{Op}\, v : B \,!\, \{\mathsf{Op}\}) \rightsquigarrow (\mathtt{op}\, E)}$$

DoPure
$$\frac{(\Gamma \vdash c_1 : A \,!\, \emptyset) \rightsquigarrow E_1 \qquad (\Gamma, x : A \vdash c_2 : B \,!\, \emptyset) \rightsquigarrow E_2}{(\Gamma \vdash \mathsf{do}\ x \leftarrow c_1\ ;\ c_2 : B \,!\, \emptyset) \rightsquigarrow (\mathtt{let}\ x = E_1\ \mathtt{in}\ E_2)}$$

DoImpure
$$\frac{(\Gamma \vdash c_1 : A \,!\, \Delta) \rightsquigarrow E_1 \qquad (\Gamma, x : A \vdash c_2 : B \,!\, \Delta) \rightsquigarrow E_2 \qquad \Delta \neq \emptyset}{(\Gamma \vdash \mathsf{do}\ x \leftarrow c_1\ ;\ c_2 : B \,!\, \Delta) \rightsquigarrow (E_1 \mathtt{>>=fun}\ x \mapsto E_2)}$$

With
$$\frac{(\Gamma \vdash v : \underline{C} \Rightarrow \underline{D}) \rightsquigarrow E_1 \qquad (\Gamma \vdash c : \underline{C}) \rightsquigarrow E_2}{(\Gamma \vdash \mathsf{handle}\ c\ \mathsf{with}\ v : \underline{D}) \rightsquigarrow (E_1\, E_2)}$$

Fig. 13. Type-&-effect-directed compilation

Eff computes the necessary type information through an inference algorithm (Pretnar 2014). For obvious practical reasons, the effect system of Eff is polymorphic, and assigns universally quantified *type schemes* rather than plain monomorphic types to terms. Furthermore, to account for subtyping, the inference algorithm is constraint-based, and a type scheme consists of a type together with a set of constraints between its parameters. For example, the type of the identity function $\mathtt{fun}\ x \mapsto \mathtt{return}\ x$ is:

$$\forall \alpha_1, \alpha_2, \delta . \alpha_1 \to \alpha_2 \mathbin{!} \delta \mid \alpha_1 \leqslant \alpha_2 \tag{11}$$

For easier constraint resolution, dirt is represented by dirt parameters δ, and we can recognize a pure computation if its dirt has no lower bounds.

The existing algorithm of Eff computes the most general type scheme of a given term in a bottom-up fashion, where constraints of subterms are joined together with additional constraints determined by the shape of the term. For example, take the term $(\mathtt{fun}\ x \mapsto \mathtt{return}\ x)\ 1$. The algorithm infers that the type of $\mathtt{fun}\ x \mapsto x$ is $\forall \alpha, \beta, \delta . \alpha \to \beta \mathbin{!} \delta \mid \alpha \leqslant \beta$, and since the type of 1 is $\mathtt{int} \mid \emptyset$, the whole computation has the type

$$\forall \alpha_1, \alpha_2, \delta . \alpha_2 \mathbin{!} \delta \mid \alpha_1 \leqslant \alpha_2, \mathtt{int} \leqslant \alpha_1$$

which can then be simplified to

$$\forall \delta . \mathtt{int} \mathbin{!} \delta \mid \emptyset .$$

Since our translations need access to the full types of all subterms, we extended the algorithm with an additional phase that propagates the constraints and annotates the whole syntax tree with appropriate types. For example, the subterm $\mathtt{fun}\ x \mapsto \mathtt{return}\ x$ is given the more appropriate scheme $\forall \delta . \mathtt{int} \to \mathtt{int} \mathbin{!} \delta$. In this example, chosen for simplicity, the additional phase does not impact the translation, but it does so in a majority of effectful computations.

To reduce the number of bugs, we implemented construction of typed terms through the use of "smart" constructors, which take already typed subterms as arguments, and contain the necessary logic to return the appropriately annotated term. The inference algorithm then simply traverses the untyped term and applies the corresponding smart constructors.

7.3 Translating Higher-Order Functions

One of the crucial properties of OCaml, which Eff honours, is that higher-order functions can accept both pure and impure functions as arguments. But as described in Section 6, these two kinds of functions are translated differently, so higher-order functions need to be translated in a way that accepts both. One possible approach is to compile multiple versions of each higher-order function, and select the appropriate one depending on the purity of its arguments. We opted for a simpler approach and labelled all higher-order functions as accepting impure arguments. If such a function is then applied to a pure argument, we use the coercions described in Section 6.

7.4 Embedding Pure Computations Into Values

Recall that in Effy, the two subterms of an application are values, whereas the application itself is a computation. This implies that a nested application `f x y` must be translated into `f x >>= fun g -> g y`. With the purity-aware translation, we can do a bit better when `f x` is pure (the common case for curried functions), and translate it as `let g = f x in g y`. However, this still incurs a significant overhead in comparison to `f x y`. To remedy that, we extend the core syntax of Eff with a coercion from computations into values, which behaves as a retraction of the value embedding.

In the basic translation to OCaml, we translate this coercion using a function `run : 'a computation -> 'a`, defined as

```
let run (Return x) = x
```

and the above application is translated as `(run (f x)) y`. Note that this function is partial and causes a runtime error if applied to an operation call. To avoid that, we make sure we apply coercions only to computations guaranteed to be pure by the effect system. The second thing to note is that this translation is no better than `f x >>= fun g -> g y`, as both variants need to extract the value, returned by `f x`.

In the purity-aware translation, we make the coercion invisible, just like the value embedding, and we translate the nested application simply as `f x y`, resulting in zero overhead.

7.5 Extensible Set of Operations

In Section 5 we assumed a fixed set of operations. However, users may want declare their own operations, and Eff enables them to do so through declarations such as:

```
operation Decide : unit -> bool
```

To support this extensibility in our translation, we make use of OCaml's extensible (GADT) variant type feature to define an initially empty type of operations, indexed by their argument and result type:

```
type ('arg, 'res) operation = ..
```

Then, an operation declaration like the one above can be translated to an extension of the `operation` type:

```
type (_, _) operation += Decide : (unit, bool) operation
```

Next, the free monad representation is adapted to:

```
type 'a computation =
| Return: 'a -> 'a computation
| Call: ('arg, 'res) operation * 'arg * ('res -> 'a computation) -> 'a computation
```

where instead of multiple constructors, we have a single one that takes the called operation, its argument and its continuation.

Handler cases are described with two fields: a return case as before, and a function that takes an operation to its appropriate case:

```
type ('a, 'b) handler_cases = {
  return: 'a -> 'b computation;
  operations: 'arg 'res. ('arg, 'res) operation ->
    'arg -> ('res -> 'b computation) -> 'b computation
}
```

A handler $\{\mathsf{return}\ x \mapsto c_r, \mathsf{Op}_1\ x\ k \mapsto c_{\mathsf{Op}_1}, \ldots, \mathsf{Op}_n\ x\ k \mapsto c_{\mathsf{Op}_n}\}$ is translated as:

```
handler {
  return = (fun x -> ⟦c_r⟧);
  operations = (function
    | Op_1 -> (fun x k -> ⟦c_Op_1⟧)
    ...
    | Op_n -> (fun x k -> ⟦c_Op_n⟧)
    | op' -> (fun x k -> Call (op', x, k))
    )
  }
```

where the last case of the `operations` function reinvokes any operation `op'` that is not captured by the handler.

The function `handler` and computation sequencing are redefined analogously:

```
let rec handler (h : ('a, 'b) handler_cases) : ('a computation -> 'b computation) =
  function
  | Return x -> h.return x
  | Call (op, x, k) -> (h.operations op) x (fun y -> handler h (k y))

let rec (>>=) (c : 'a computation) (f : 'a -> 'b computation) : 'b computation =
  match c with
  | Return x -> f x
  | Call (op, x, k) -> Call (op, x, (fun y -> (k y) >>= f))
```

8 EVALUATION

We evaluate the effectiveness of our optimizing compiler for Eff on a number of benchmarks. First, we compare our different compilation schemes with hand-written OCaml code. Then, we measure our compiler's performance against other OCaml-based implementations of algebraic effects and handlers. All benchmarks were run on a MacBook Pro with an 2.5 GHz Intel Core I7 processor and 16 GB 1600 MHz DDR3 RAM running Mac OS 10.12.3.

8.1 Eff versus OCaml

Our first evaluation, in Fig. 14, considers four different variations on the looping program from Section 2: 1) `Pure` is version without side-effects, 2) `Latent` contains an operation that is never called during the execution of the benchmark, 3) `Incr` calls a single increment operation that increments an implicit state, 4) is the version of Section 2 that increments the implicit state with the `Get` and `Put` operations. We compile these programs in four different ways: 1) basic compilation mode without any optimization (Basic), 2) purity-aware compilation (Pure), 3) source-to-source optimizations (Opt), 4) the combination of the previous two. Finally, we compare these different versions against hand-written (Native) OCaml code: 1) a pure loop, 2) a latent OCaml exception, 3) a reference cell increment, and 4) a reference cell read followed by a write. The programs were compiled with version 4.02.2 of the OCaml compiler.

Figure 14 shows the time relative to the Basic version for running each of the 20 programs for 10,000 iterations. The results show a substantial gap between the basic compilation scheme and the hand-written OCaml, in the range of 25×–50×. The source-to-source transformations and purity-aware code generation each have individually varying success in reducing the gap to a smaller, but still significant level. It is only when the two optimizations are combined that we obtain performance that is competitive with the hand-written versions (1×-1.5×). In particular, the combined optimizations succeed in eliminating all trace of the handlers and free monad from the generated OCaml code.

8.2 Eff versus Other Systems

Our second evaluation features two different versions of the well-known N-Queens problem. Both versions use the same underlying program to explore the combinatorial space with the operations `Decide : unit -> bool` and `Fail : unit -> void` for non-determinism. The two versions only differ in which handler they use to interpret the operations.

The first handler computes *all solutions* and returns them in a list and the second computes only the *first solution*:

```
let all_solutions = handler
  | return x    -> [x]
  | Decide _ k -> k true @ k false
  | Fail _ _    -> []
```

```
let first_solution = handler
  | return x     -> Some x
  | #Decide _ k -> match k true with
                       | Some x -> Some x
```

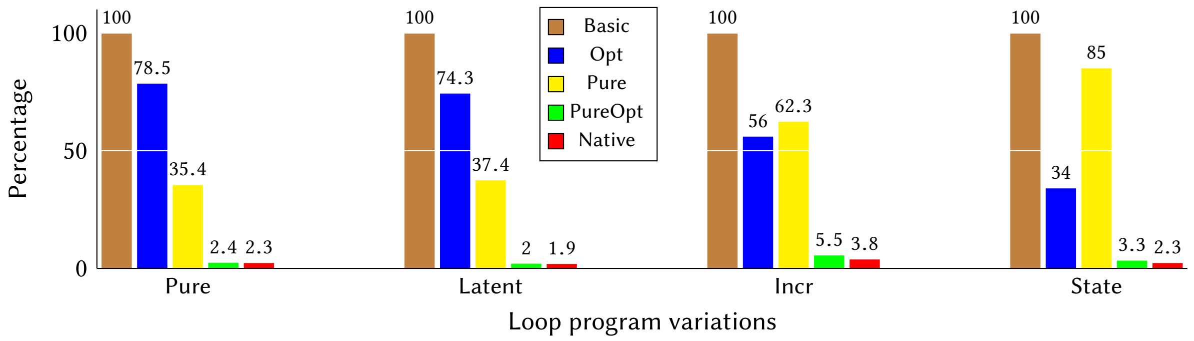

Fig. 14. Relative run-times of Loops example

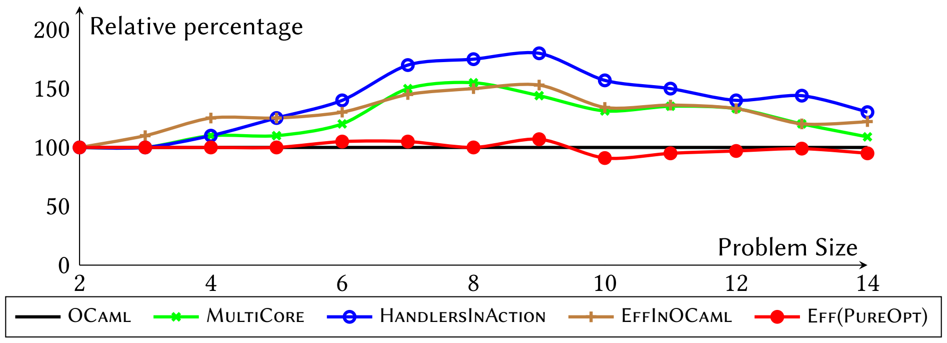

Fig. 15. Results of running N-Queens for all solutions on multiple systems

```
                                          | None -> k false
                                          | #Fail _ _   -> None
```

We compare the fully optimized Eff version against similar programs in three other OCaml-based systems and use the hand-written OCaml program as a baseline. The three systems are Multicore OCaml (Dolan et al. 2015) (Multicore), which provides native lightweight threads for running continuations, but requires expensive copying of the continuation for the non-linear use of the continuation in the above two handlers. The second and third system are the *Handlers in Action* (Kammar et al. 2013) OCaml implementation (MultiCore) and the *Eff Directly in OCaml* implementation (Kiselyov and Sivaramakrishnan 2016) (EffInOCaml); both are based on OCaml's DelimCC libray for delimited control (Kiselyov 2012). Because this library does not support native compilation, we compile all benchmarks to bytecode and run that instead. We omit a fourth system, the OCaml backend of Links (Cooper et al. 2006; Hillerström et al. 2016), from the results, because it is 20 times slower than the hand-written OCaml code and thus would dwarf all other runtimes if we included it.

Figure 15 shows the `all_solutions` runtimes of the different systems for different problem sizes, each time relative to the hand-written OCaml code. The results clearly show that Eff is consistently the fastest and competitive with hand-written OCaml code. We see a similar, even more uniform picture in Figure 16, where Eff is consistently 25-30% faster than the closes competitor Multicore OCaml.

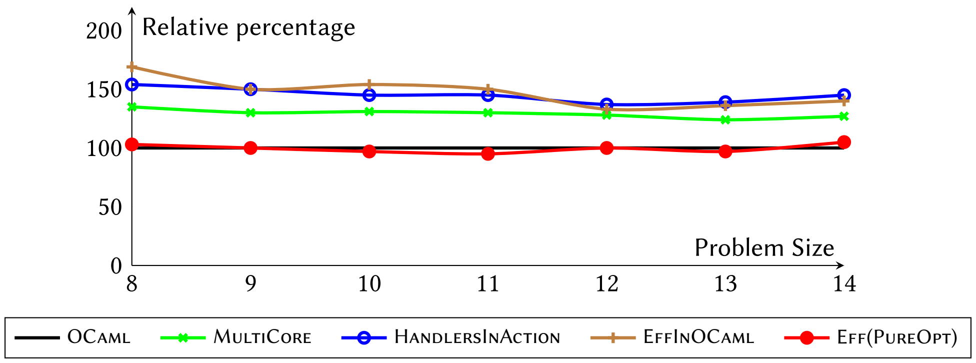

Fig. 16. Results of running N-Queens for one solution on multiple systems

9 RELATED WORK

Leijen (2017) presents a type-directed compilation approach for the Koka language. His setting differs from ours in several ways: Firstly, Koka features row-typing rather than effect subtyping. Secondly, Koka's compiler directly targets a CPS backend, rather than an intermediate language. The only featured optimisation is that of selective CPS: pure computations are not translated to direct-style expressions. Unfortunately, no experimental evaluation is provided to establish the significance of this optimisation. We believe that our source-to-source transformations can be easily inserted as an additional stage in the Koka compiler and adapted to draw on the row-typing information.

The Multicore OCaml backend (Dolan et al. 2015) provides supports for algebraic effects in terms of the multicore *fibers* to efficiently represent delimited continuations at runtime. These come both in a cheaper one-shot and more expensive multi-shot form. Several works (Hillerström et al. 2016; Kiselyov and Sivaramakrishnan 2016) have shown that this provides an effective compilation target for algebraic effects. Yet, as far as we know, no existing works performs optimising compilation in this setting.

Kammar et al. (2013) compare the performance of a number of different encodings of effect handlers in Haskell. Inspired by this comparison, Wu and Schrijvers (2015) show how effect handlers can be fused and inlined when programs are represented with the codensity monad. They explain that, with a careful setup based on type classes, the GHC Haskell compiler automatically carries out this optimisation as part of (constrained) polymorphic function specialisation; benchmarks illustrate the effectiveness of this approach. While the result of this approach is similar to our function specialisation, the type-class approach does not readily carry over to other compilers and languages.

Kiselyov and others (2015; 2013; 2014) investigate a number of different implementations of the free monad that exhibit good runtime performance and/or algorithmic time complexity. There are several differences between their setting and ours. They consider a library in the lazily evaluated language Haskell, while we compile to eagerly evaluated OCaml. They support so-called *shallow handlers* and explicit manipulation of the free monad structure in the source language, while we do not. Hence, we did not adopt their dequeue-based implementation of the free monad's bind. Nevertheless we share the same functor construction based on the co-Yoneda lemma and our free monad is specialised in the same way for this construction.

Saleh and Schrijvers (2016) present a term-rewriting-based approach for optimising an embedding of algebraic effects and handlers in Prolog. They obtain good speed-ups, but their setting is simpler and less general. Firstly, Prolog is essentially a procedure-oriented rather than expression-oriented language. Hence, the rewrites related to returning values are not relevant in their setting. Moreover, they do not support higher-order programming and feature only a crude effect system. Also, their handlers may only contain lexically visible calls to the continuation. In addition, they do not perform selective CPS to deal with non-tail recursion. Finally, they do not perform purity-aware code generation but instead require native support of (Prolog-style) delimited control (Schrijvers et al. 2013).

Kammar and Plotkin (2012) develop a theory of optimisations valid for effectful programs that satisfy a certain algebraic theory. For example, if we assume symmetry of non-deterministic choice, we may safely exchange the order in which non-deterministic computations are executed. Bauer and Pretnar (2014) consider a subset of these optimisations for computations in an absolutely free algebra, i.e., with a trivial equational theory, but under particular handlers. Some of their work is already subsumed by our rewriting optimisations, but there is still ample untapped potential for exploiting effect information in specialised optimisations through sophisticated reasoning.

10 CONCLUSION

This paper has presented a two-pronged approach for the optimised compilation of algebraic effects and handlers. First we perform a number of source-to-source transformations, including the specialisation of recursive function definitions for particular handlers. Then we generate target code in a purity-aware fashion. Our experimental combination shows that the synergy between these two approaches is effective at eliminating handlers from a number of benchmarks and obtaining performance that is competitive with hand-written code and better than that of existing non-optimising implementations of algebraic effects and handlers.

In future work we would like to investigate how to best optimise complex patterns like nested handlers. Also on our agenda is investigating alternative type-and-effect systems that are more practical to use for Effy's typed intermediate representation. One option is to maintain subtyping, but move to a unification based algorithm (Dolan and Mycroft 2017). Another option is to consider a simpler effect system, for example one based on row-polymorphism (Leijen 2014, 2017) or one with implicit effect polymorphism (Lindley et al. 2017). As the only point of contact between the effect system and translations is determining which computations are considered pure, we expect our work to migrate relatively smoothly.

ACKNOWLEDGMENTS

We are grateful to Nicolas Wu, Mauro Jaskelioff, Ruben Pieters, Alexander Vandenbroucke and Klara Mardirosian for their helpful comments. Part of is work is funded by the Flemish Fund for Scientific Research (FWO).

REFERENCES

Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10, 4 (2014).

Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *Journal of Logic and Algebraic Programming* 84, 1 (2015), 108–123.

Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, 133–144.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects (Lecture Notes in Computer Science)*, Vol. 4709. Springer, 266–296.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 60–72.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective Concurrency through Algebraic Effects. In *OCaml Workshop*.

Daniel Hillerström, Sam Lindley, and KC Sivaramakrishnan. 2016. Compiling Links Effect Handlers to the OCaml Backend. In *OCaml Workshop*.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming (ICFP '14)*. ACM, 145–158.

Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 349–360.

Oleg Kiselyov. 2012. Delimited control in OCaml, abstractly and concretely. *Theorical Computer Science* 435 (2012), 56–76.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 94–105.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, Chung-chieh Shan (Ed.). 59–70.

Oleg Kiselyov and KC Sivaramakrishnan. 2016. Eff Directly in OCaml. In *OCaml Workshop*.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014. (EPTCS)*, Paul Levy and Neel Krishnaswami (Eds.), Vol. 153. 100–126.

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 486–499.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 500–514.

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94.

Gordon D. Plotkin and Matija Pretnar. 2008. A Logic for Algebraic Effects. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 118–129.

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).

Matija Pretnar. 2014. Inferring Algebraic Effects. *Logical Methods in Computer Science* 10, 3 (2014).

Matija Pretnar. 2015. An introduction to algebraic effects and handlers, invited tutorial. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35.

Amr Hany Saleh and Tom Schrijvers. 2016. Efficient algebraic effect handlers for Prolog. *Theory and Practice of Logic Programming* 16, 5-6 (2016), 884–898.

Tom Schrijvers, Bart Demoen, Benoit Desouter, and Jan Wielemaker. 2013. Delimited continuations for Prolog. *Theory and Practice of Logic Programming* 13, 4-5 (2013), 533–546.

Atze van der Ploeg and Oleg Kiselyov. 2014. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 133–144.

Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free - Efficient Algebraic Effect Handlers. In *Mathematics of Program Construction (LNCS)*, Vol. 9129. Springer, 302–322. DOI: http://dx.doi.org/10.1007/978-3-319-19797-5_15

A SOUNDNESS OF REWRITING

Proof of Theorem 4.1. We prove the theorem for each rewrite rule:

- App-Fun: This follows directly from Equation 1.
- Do-Ret: This follows directly from Equation 4.
- Do-Op: This follows directly from Equation 6.
- With-LetRec:

$$\begin{array}{ll} & \texttt{handle}\ (\texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2)\ \texttt{with}\ v \\ \equiv & (\text{Eq. } 3) \\ & \texttt{handle}\ (c_2[(\texttt{fun}\ x \mapsto\ \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_1)/f])\ \texttt{with}\ v \\ \equiv & (f \notin v) \\ & (\texttt{handle}\ c_2\ \texttt{with}\ v)[(\texttt{fun}\ x \mapsto\ \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_1)/f] \\ \equiv & (\text{Eq. } 3) \\ & \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ (\texttt{handle}\ c_2\ \texttt{with}\ v) \end{array}$$

- With-Ret: This follows directly from Equation 8.
- With-Handled-Op:

$$\begin{array}{ll} & \texttt{handle}\ (\texttt{Op}\,v)\ \texttt{with}\ h \\ \equiv & (\text{Eq. } 5) \\ & \texttt{handle}\ (\texttt{do}\ x \leftarrow \texttt{Op}\,v\ ;\ \texttt{return}\ x)\ \texttt{with}\ h \\ \equiv & (\text{Eq. } 9) \\ & c_{\texttt{Op}}[v/x, (\texttt{fun}\ x \mapsto\ \texttt{handle return}\ x\ \texttt{with}\ h)/k] \\ \equiv & (\text{Eq. } 8) \\ & c_{\texttt{Op}}[v/x, (\texttt{fun}\ x \mapsto\ c_r)/k] \end{array}$$

- With-Pure: We prove this property by induction on c.
  - Case $c = \texttt{return}\ v$:

$$\begin{array}{ll} & \texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ h \\ \equiv & (\text{Eq. } 8) \\ & c_r[v/x] \\ \equiv & (\text{Eq. } 4) \\ & \texttt{do}\ x \leftarrow \texttt{return}\ v\ ;\ c_r \end{array}$$

  - Case $c = \texttt{do}\ y \leftarrow \texttt{Op}\,v\ ;\ c'$ with $\texttt{Op} \in \mathcal{O}$:
    This cannot happen since $\Delta \cap \mathcal{O} = \emptyset$.
  - Case $c = \texttt{do}\ y \leftarrow \texttt{Op}\,v\ ;\ c'$ with $\texttt{Op} \notin \mathcal{O}$:

$$\begin{array}{ll} & \texttt{handle}\ (\texttt{do}\ y \leftarrow \texttt{Op}\,v\ ;\ c')\ \texttt{with}\ h \\ \equiv & (\text{Eq. } 10) \\ & \texttt{do}\ y \leftarrow \texttt{Op}\,v\ ;\ \texttt{handle}\ c'\ \texttt{with}\ h \\ \equiv & (\text{Induction hypothesis}) \\ & \texttt{do}\ y \leftarrow \texttt{Op}\,v\ ;\ (\texttt{do}\ x \leftarrow c'\ ;\ c_r) \\ \equiv & (\text{Eq. } 6) \\ & \texttt{do}\ x \leftarrow (\texttt{do}\ y \leftarrow \texttt{Op}\,v\ ;\ c')\ ;\ c_r \end{array}$$

- With-Do: We prove this property by induction on c_1.

– Case $c_1 = \texttt{return}\ v$:

$$\begin{array}{rl} & \texttt{handle}\ (\texttt{do}\ y \leftarrow \texttt{return}\ v\ ;\ c_2)\ \texttt{with}\ h \\ \equiv & (\text{Eq. } 4) \\ & \texttt{handle}\ (c_2[v/y])\ \texttt{with}\ h \\ \equiv & (y \notin h) \\ & (\texttt{handle}\ c_2\ \texttt{with}\ h)[v/y] \\ \equiv & (\text{Eq. } 8) \\ & \texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ h' \end{array}$$

– Case $c_1 = \texttt{do}\ z \leftarrow \texttt{Op}\,v\ ;\ c_1'$ with $\texttt{Op} \in O$:

$$\begin{array}{rl} & \texttt{handle}\ (\texttt{do}\ y \leftarrow (\texttt{do}\ z \leftarrow \texttt{Op}\,v\ ;\ c_1')\ ;\ c_2)\ \texttt{with}\ h \\ \equiv & (\text{Eq. } 6) \\ & \texttt{handle}\ (\texttt{do}\ z \leftarrow \texttt{Op}\,v\ ;\ (\texttt{do}\ y \leftarrow c_1'\ ;\ c_2))\ \texttt{with}\ h \\ \equiv & (\text{Eq. } 9) \\ & c_{\texttt{Op}}[v/x, (\texttt{fun}\ z \mapsto \texttt{handle}\ (\texttt{do}\ y \leftarrow c_1'\ ;\ c_2)\ \texttt{with}\ h)/k] \\ \equiv & (\text{Induction hypothesis}) \\ & c_{\texttt{Op}}[v/x, (\texttt{fun}\ z \mapsto \texttt{handle}\ c_1'\ \texttt{with}\ h')/k] \\ \equiv & (\text{Eq. } 9) \\ & \texttt{handle}\ (\texttt{do}\ z \leftarrow \texttt{Op}\,v\ ;\ c_1')\ \texttt{with}\ h' \end{array}$$

– Case $c_1 = \texttt{do}\ z \leftarrow \texttt{Op}\,v\ ;\ c_1'$ with $\texttt{Op} \notin O$:

$$\begin{array}{rl} & \texttt{handle}\ (\texttt{do}\ y \leftarrow (\texttt{do}\ z \leftarrow \texttt{Op}\,v\ ;\ c_1')\ ;\ c_2)\ \texttt{with}\ h \\ \equiv & (\text{Eq. } 6) \\ & \texttt{handle}\ (\texttt{do}\ z \leftarrow \texttt{Op}\,v\ ;\ (\texttt{do}\ y \leftarrow c_1'\ ;\ c_2))\ \texttt{with}\ h \\ \equiv & (\text{Eq. } 10) \\ & \texttt{do}\ z \leftarrow \texttt{Op}\,v\ ;\ \texttt{handle}\ (\texttt{do}\ y \leftarrow c_1'\ ;\ c_2)\ \texttt{with}\ h \\ \equiv & (\text{Induction hypothesis}) \\ & \texttt{do}\ z \leftarrow \texttt{Op}\,v\ ;\ \texttt{handle}\ c_1'\ \texttt{with}\ h' \\ \equiv & (\text{Eq. } 10) \\ & \texttt{handle}\ (\texttt{do}\ z \leftarrow \texttt{Op}\,v\ ;\ c_1')\ \texttt{with}\ h' \end{array}$$

□

B TYPE PRESERVATION OF BASIC COMPILATION

To support the proof, we give the type system for the targeted subset of OCaml in Figure 17. To simplify the proof, and omit unnecessary details, the type system contains a number of "derived rules" for the OCaml functions used in the elaboration. This way we can also avoid the additional complexity of object-level polymorphism.

Before we prove the main lemma, we prove a lemma about subtyping.

Lemma B.1. *For all pure types A and B, and for all computation types $\underline{C}$ and $\underline{D}$ we have that:*

$$A \leqslant B \Rightarrow [\![A]\!] = [\![B]\!]$$

and

$$\underline{C} \leqslant \underline{D} \Rightarrow [\![\underline{C}]\!] = [\![\underline{D}]\!]$$

Proof. The proof proceeds by mutual induction on the derivation of subtyping for pure and dirty types.

Sub-bool: $\texttt{bool} \leqslant \texttt{bool}$

In this case the lemma holds trivially.

$$\textsc{O-Var}\quad\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}\qquad \textsc{O-Const}\quad\frac{(\mathsf{k} : A) \in \Sigma}{\Gamma \vdash \mathsf{k} : \llbracket A \rrbracket}\qquad \textsc{O-Fun}\quad\frac{\Gamma, x : T_1 \vdash E : T_2}{\Gamma \vdash \mathsf{fun}\ x \mapsto E : T_1 \to T_2}\qquad \textsc{O-App}\quad\frac{\Gamma \vdash E_1 : T_1 \to T_2 \qquad \Gamma \vdash E_2 : T_1}{\Gamma \vdash E_1\, E_2 : T_2}$$

$$\textsc{O-Let}\quad\frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma, x : T_1 \vdash E_2 : T_2}{\Gamma \vdash \mathtt{let}\ x = E_1\ \mathtt{in}\ E_2 : T_2}\qquad \textsc{O-LetRec}\quad\frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash E_1 : T_2 \qquad \Gamma, f : T_1 \to T_2 \vdash E_2 : T_3}{\Gamma \vdash \mathtt{let\ rec}\ f\ x = E_1\ \mathtt{in}\ E_2 : T_3}$$

$$\textsc{O-HandlerCases}\quad\frac{\Gamma \vdash E : T_1 \to T_2 \qquad \left[\Gamma \vdash E_i : \llbracket A_{\mathsf{Op}} \rrbracket \to (\llbracket B_{\mathsf{Op}} \rrbracket \to T_2) \to T_2\right]_{(\mathsf{Op}_i : A_i \to B_i) \in \Sigma}}{\Gamma \vdash \{\mathtt{return} = E; \mathtt{op}_1 = E_1; \dots; \mathtt{op}_n = E_n\} : (T_1, T_2)\ \mathtt{handler\_cases}}$$

$$\textsc{O-Ret}\quad\frac{}{\Gamma \vdash \mathtt{return} : T \to T\ \mathtt{computation}}\qquad \textsc{O-Operation}\quad\frac{(\mathsf{Op} : A \to B) \in \Sigma}{\Gamma \vdash \mathtt{op} : \llbracket A \rrbracket \to \llbracket B \rrbracket\ \mathtt{computation}}$$

$$\textsc{O-Handler}\quad\frac{}{\Gamma \vdash \mathtt{handler} : (T_1, T_2)\ \mathtt{handler\_cases} \to (T_1\ \mathtt{computation} \to T_2)}$$

$$\textsc{O-Bind}\quad\frac{}{\Gamma \vdash (\mathtt{>>=}) : T_1\ \mathtt{computation} \to (T_1 \to T_2\ \mathtt{computation}) \to T_2\ \mathtt{computation}}$$

$$\textsc{O-FMap}\quad\frac{}{\Gamma \vdash \mathtt{fmap} : (T_1 \to T_2) \to (T_1\ \mathtt{computation} \to T_2\ \mathtt{computation})}$$

Fig. 17. Typing of (a subset of) OCaml

Sub-int: $\mathtt{int} \leqslant \mathtt{int}$
In this case the lemma holds trivially.

Sub-$\to$: $A \to \underline{C} \leqslant A' \to \underline{C}'$.
From the rule's first hypothesis we have that $A' \leqslant A$. Thus by the induction hypothesis we have that $\llbracket A' \rrbracket = \llbracket A \rrbracket$. From the rule's second hypothesis we have that $\underline{C} \leqslant \underline{C}'$. Thus by the induction hypothesis we have that $\llbracket \underline{C} \rrbracket = \llbracket \underline{C}' \rrbracket$. Hence, we have that $\llbracket A \rrbracket \to \llbracket \underline{C} \rrbracket = \llbracket A' \rrbracket \to \llbracket \underline{C}' \rrbracket$. As $\llbracket A \to \underline{C} \rrbracket = \llbracket A \rrbracket \to \llbracket \underline{C} \rrbracket$ and $\llbracket A' \to \underline{C}' \rrbracket = \llbracket A' \rrbracket \to \llbracket \underline{C}' \rrbracket$, we thus conclude that $\llbracket A \to \underline{C} \rrbracket = \llbracket A' \to \underline{C}' \rrbracket$.

Sub-$\Rightarrow$: $\underline{C} \Rightarrow \underline{D} \leqslant \underline{C}' \Rightarrow \underline{D}'$.
From the rule's first hypothesis we have that $\underline{C}' \leqslant \underline{C}$. Thus by the induction hypothesis we have that $\llbracket \underline{C}' \rrbracket = \llbracket \underline{C} \rrbracket$. From the rule's second hypothesis we have that $\underline{D} \leqslant \underline{D}'$. Thus by the induction hypothesis we have that $\llbracket \underline{D} \rrbracket = \llbracket \underline{D}' \rrbracket$. Hence, we have that $\llbracket \underline{C} \rrbracket \to \llbracket \underline{D} \rrbracket = \llbracket \underline{C}' \rrbracket \to \llbracket \underline{D}' \rrbracket$. As $\llbracket \underline{C} \Rightarrow \underline{D} \rrbracket = \llbracket \underline{C} \rrbracket \to \llbracket \underline{D} \rrbracket$ and $\llbracket \underline{C}' \Rightarrow \underline{D}' \rrbracket = \llbracket \underline{C}' \rrbracket \to \llbracket \underline{D}' \rrbracket$, we thus conclude that $\llbracket \underline{C} \Rightarrow \underline{D} \rrbracket = \llbracket \underline{C}' \Rightarrow \underline{D}' \rrbracket$.

Sub-!: $A \,!\, \Delta \leqslant A' \,!\, \Delta'$.
The rule's first hypothesis is $A \leqslant A'$. Thus by the induction hypothesis we have $\llbracket A \rrbracket = \llbracket A' \rrbracket$. Moreover,

we have that $[\![A \,!\, \Delta]\!] = [\![A]\!]$ `computation` and $[\![A' \,!\, \Delta']\!] = [\![A']\!]$ `computation`. Hence, we conclude $[\![A \,!\, \Delta]\!] = [\![A' \,!\, \Delta']\!]$.

□

Now follows the proof of the main theorem.

Proof of Theorem 5.1. We prove the preservation of typing for the basic elaboration by mutual induction on the typing derivations for values and computations.

(SubVal): $\Gamma \vdash v : A'$

The rule's first assumption is that $\Gamma \vdash v : A$. By the induction hypothesis we thus have that $[\![\Gamma]\!] \vdash [\![v]\!] : [\![A]\!]$. The rule's second assumption is that $A \leqslant A'$. From Lemma B.1 we then have that $[\![A]\!] = [\![A']\!]$. Thus we conclude $[\![\Gamma]\!] \vdash [\![v]\!] : [\![A']\!]$.

(Var): $\Gamma \vdash x : A$

From the hypothesis of the rule, we have that $(x : A) \in \Gamma$. It follows that $(x : [\![A]\!]) \in [\![\Gamma]\!]$. Hence, by rule Var we have $[\![\Gamma]\!] \vdash x : [\![A]\!]$. Because $[\![x]\!] = x$, we conclude $[\![\Gamma]\!] \vdash [\![x]\!] : [\![A]\!]$.

(Const): $\Gamma \vdash \mathsf{k} : A$

We have that $(k : A) \in \Sigma$. Hence, by rule O-Const we have $[\![\Gamma]\!] \vdash k : [\![A]\!]$. Because $[\![k]\!] = k$ we conclude $[\![\Gamma]\!] \vdash [\![k]\!] : [\![A]\!]$.

(Fun): $\Gamma \vdash \mathsf{fun}\; x \mapsto\; c : A \to \underline{C}$

From the rule it follows that $\Gamma, x : A \vdash c : \underline{C}$. By the induction hypothesis, we thus have $[\![\Gamma, x : A]\!] \vdash [\![c]\!] : [\![\underline{C}]\!]$. Because $[\![\Gamma, x : A]\!] = [\![\Gamma]\!], x : [\![A]\!]$ we thus have from rule O-Fun that $[\![\Gamma]\!] \vdash \mathtt{fun}\; x \mapsto\; [\![c]\!] : [\![A]\!] \to [\![\underline{C}]\!]$. As $[\![A \to \underline{C}]\!] = [\![A]\!] \to [\![\underline{C}]\!]$ and $[\![\mathsf{fun}\; x \mapsto\; c]\!] = \mathtt{fun}\; x \mapsto\; [\![c]\!]$, we conclude $[\![\Gamma]\!] \vdash [\![\mathsf{fun}\; x \mapsto\; c]\!] : [\![A \to \underline{C}]\!]$.

(Hand): $\Gamma \vdash \{\mathsf{return}\; x \mapsto c_r, [\mathsf{Op}\, x\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\} : A \,!\, \Delta \cup O \Rrightarrow B \,!\, \Delta$

From the first hypothesis of the rule and the induction hypothesis we have that $[\![\Gamma, x : A]\!] \vdash [\![c_r]\!] : [\![B \,!\, \Delta]\!]$. We can simplify this to $[\![\Gamma]\!], x : [\![A]\!] \vdash [\![c_r]\!] : [\![B]\!]$ `computation`.

From the second hypothesis and the induction hypothesis we have that $[\![\Gamma, x : A_{\mathsf{Op}}, k : B_{\mathsf{Op}} \to B \,!\, \Delta]\!] \vdash [\![c_{\mathsf{Op}}]\!] : [\![B \,!\, \Delta]\!]$ for each $\mathsf{Op} \in O$. This simplifies to $[\![\Gamma]\!], x : [\![A_{\mathsf{Op}}]\!], k : [\![B_{\mathsf{Op}}]\!] \to [\![B]\!]$ `computation` $\vdash [\![c_{\mathsf{Op}}]\!] : [\![B]\!]$ `computation`.

For any $\mathsf{Op} \notin O$, we have that $[\![\Gamma]\!], x : [\![A_{\mathsf{Op}}]\!], k : [\![B_{\mathsf{Op}}]\!] \to [\![B]\!]$ `computation` $\vdash (\mathtt{op} x \mathtt{>>=} k) : [\![B]\!]$ `computation` by O-Operation, O-Bind and O-Apply.

Hence, by rule O-Fun and O-HandlerCases, we may conclude that $[\![\Gamma]\!] \vdash \{\mathtt{return} = \mathtt{fun}\; x \mapsto [\![c_r]\!]; \mathtt{op}_1 = E_1; \ldots; \mathtt{op}_n = E_n\} : ([\![A]\!], [\![B]\!]\; \mathtt{computation})\; \mathtt{handler\_cases}$. Finally, by O-Handler and O-Apply we have that $[\![\Gamma]\!] \vdash \mathtt{handler}\; \{\mathtt{return} = \mathtt{fun}\; x \mapsto [\![c_r]\!]; \mathtt{op}_1 = E_1; \ldots; \mathtt{op}_n = E_n\} : [\![A]\!]\; \mathtt{computation} \to [\![B]\!]\; \mathtt{computation}$, implying $[\![\Gamma]\!] \vdash [\![\{\mathsf{return}\; x \mapsto c_r, [\mathsf{Op}\, x\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\}]\!] : [\![A \,!\, \Delta \cup O \Rrightarrow B \,!\, \Delta]\!]$.

(SubComp): $\Gamma \vdash c : \underline{C}'$

The proof proceeds analogously to one for rule SubVal.

(App): $\Gamma \vdash e_1\, e_2 : \underline{C}$

From the rule's two hypotheses and the induction hypotheses, we have that $[\![\Gamma]\!] \vdash [\![e_1]\!] : [\![A \to \underline{C}]\!]$ and $[\![\Gamma]\!] \vdash [\![e_2]\!] : [\![A]\!]$. Because $[\![A \to \underline{C}]\!] = [\![A]\!] \to [\![\underline{C}]\!]$ we have by rule O-App that $[\![\Gamma]\!] \vdash [\![e_1]\!]\, [\![e_2]\!] : [\![\underline{C}]\!]$. Because $[\![e_1\, e_2]\!] = [\![e_1]\!]\, [\![e_2]\!]$, we conclude $[\![\Gamma]\!] \vdash [\![e_1\, e_2]\!] : [\![\underline{C}]\!]$.

(LetRec): $\Gamma \vdash \mathsf{let\ rec}\; f\, x = c_1\; \mathsf{in}\; c_2 : \underline{D}$

From the rule's first hypothesis and the induction hypothesis we have $[\![\Gamma, f : A \to \underline{C}, x : A]\!] \vdash [\![c_1]\!] : [\![\underline{C}]\!]$. We can simplify this to $[\![\Gamma]\!], f : [\![A]\!] \to [\![\underline{C}]\!], x : [\![A]\!] \vdash [\![c_1]\!] : [\![\underline{C}]\!]$. From the rule's second hypothesis and the induction hypothesis we have $[\![\Gamma, f : A \to \underline{C}]\!] \vdash [\![c_2]\!] : [\![\underline{D}]\!]$. We can simplify this to $[\![\Gamma]\!], f : [\![A]\!] \to [\![\underline{C}]\!] \vdash [\![c_2]\!] : [\![\underline{D}]\!]$. By rule O-LetRec we then have $[\![\Gamma]\!] \vdash \mathtt{let\ rec}\; f\, x = [\![c_1]\!]\; \mathtt{in}\; [\![c_2]\!] : [\![\underline{D}]\!]$. Since $[\![\mathsf{let\ rec}\; f\, x = c_1\; \mathsf{in}\; c_2]\!] = \mathtt{let\ rec}\; f\, x = [\![c_1]\!]\; \mathtt{in}\; [\![c_2]\!]$, we conclude $[\![\Gamma]\!] \vdash [\![\mathsf{let\ rec}\; f\, x = c_1\; \mathsf{in}\; c_2]\!] : [\![\underline{D}]\!]$.

(Ret): $\Gamma \vdash \mathsf{return}\, v : A \,!\, \emptyset$

From the rule's assumption and the induction hypothesis we have that $[\![\Gamma]\!] \vdash [\![v]\!] : [\![A]\!]$. Hence, by means of rules O-Ret and O-App we have $[\![\Gamma]\!] \vdash \mathtt{return}\ [\![v]\!] : [\![A]\!]\ \mathtt{computation}$. As $[\![\mathtt{return}\ v]\!] = \mathtt{return}\ [\![v]\!]$ and $[\![A \,!\, \emptyset]\!] = [\![A]\!]\ \mathtt{computation}$, we conclude $[\![\Gamma]\!] \vdash [\![\mathtt{return}\ v]\!] : [\![A \,!\, \emptyset]\!]$.

(Op): $\Gamma \vdash \mathsf{Op}\, v : B \,!\, \{\mathsf{Op}\}$

From the first hypothesis of the rule, we have $(\mathsf{Op} : A \to B) \in \Sigma$. From the second hypothesis of the rule and the induction hypothesis, we have that $[\![\Gamma]\!] \vdash [\![v]\!] : [\![A]\!]$. By rules O-Operation and O-App we then have $[\![\Gamma]\!] \vdash \mathsf{op}\, [\![v]\!] : [\![B]\!]\ \mathtt{computation}$. As $[\![\mathsf{Op}\, v]\!] = \mathsf{op}\, [\![v]\!]$ and $[\![B \,!\, \{\mathsf{Op}\}]\!] = [\![B]\!]\ \mathtt{computation}$, we conclude $[\![\Gamma]\!] \vdash [\![\mathsf{op}\, v]\!] : [\![B \,!\, \{\mathsf{Op}\}]\!]$.

(Do): $\Gamma \vdash \mathsf{do}\ x \leftarrow c_1 \,;\, c_2 : B \,!\, \Delta$

From the rule's first hypothesis and the induction hypothesis we have $[\![\Gamma]\!] \vdash [\![c_1]\!] : [\![A \,!\, \Delta]\!]$. We can simplify this to $[\![\Gamma]\!] \vdash [\![c_1]\!] : [\![A]\!]\ \mathtt{computation}$. From the rule's second hypothesis and the induction hypothesis we have $[\![\Gamma, x : A]\!] \vdash [\![c_2]\!] : [\![B \,!\, \Delta]\!]$. We can simplify this to $[\![\Gamma]\!], x : [\![A]\!] \vdash [\![c_2]\!] : [\![B]\!]\ \mathtt{computation}$. Using rule O-Fun we have $[\![\Gamma]\!] \vdash \mathtt{fun}\ x \mapsto\ [\![c_2]\!] : [\![A]\!] \to [\![B]\!]\ \mathtt{computation}$. Using rules O-Bind and O-App we then have $[\![\Gamma]\!] \vdash [\![c_1]\!] \mathtt{>>=} (\mathtt{fun}\ x \mapsto\ [\![c_2]\!]) : [\![B]\!]\ \mathtt{computation}$. As $[\![\mathsf{do}\ x \leftarrow c_1 \,;\, c_2]\!] = [\![c_1]\!] \mathtt{>>=} (\mathtt{fun}\ x \mapsto [\![c_2]\!])$, we conclude $[\![\Gamma]\!] \vdash [\![\mathsf{do}\ x \leftarrow c_1 \,;\, c_2]\!] : [\![B]\!]\ \mathtt{computation}$.

(With): $\Gamma \vdash \mathsf{handle}\ c\ \mathsf{with}\ v : \underline{D}$

From the rule's first assumption and the induction hyptothesis we have that $[\![\Gamma]\!] \vdash [\![v]\!] : [\![\underline{C} \Rightarrow \underline{D}]\!]$. Because $[\![\underline{C} \Rightarrow \underline{D}]\!] = [\![\underline{C}]\!] \to [\![\underline{D}]\!]$ we thus have $[\![\Gamma]\!] \vdash [\![v]\!] : [\![\underline{C}]\!] \to [\![\underline{D}]\!]$. From the rule's second assumption and the induction hypothesis we also have $[\![\Gamma]\!] \vdash [\![c]\!] : [\![\underline{C}]\!]$. By rule O-App we then have $[\![\Gamma]\!] \vdash [\![v]\!]\ [\![c]\!] : [\![\underline{D}]\!]$. Because $[\![\mathsf{handle}\ c\ \mathtt{with}\ v]\!] = [\![v]\!]\ [\![c]\!]$ we thus conclude $[\![\Gamma]\!] \vdash [\![\mathsf{handle}\ c\ \mathtt{with}\ v]\!] : [\![\underline{D}]\!]$.

□

C TYPE PRESERVATION OF PURITY-AWARE COMPILATION

We first prove an variant of the earlier subtyping lemma.

Proof of Lemma . The proof proceeds by mutual induction on the derivation for pure and dirty types.

Sub-bool: $(\mathtt{bool} \leqslant \mathtt{bool}) \rightsquigarrow (\mathtt{fun}\ x \mapsto\ x)$

In this case the lemma holds trivially.

Sub-int: $(\mathtt{int} \leqslant \mathtt{int}) \rightsquigarrow (\mathtt{fun}\ x \mapsto\ x)$

In this case the lemma holds trivially.

Sub-$\to$: $(A \to \underline{C} \leqslant A' \to \underline{C}') \rightsquigarrow (\mathtt{fun}\ f\, x \mapsto\ E_2\, (f\, (E_1\, x)))$.

From the rule's first hypothesis we have that $(A' \leqslant A) \rightsquigarrow E_1$. Thus by the induction hypothesis we have that $\vdash E_1 : [\![A']\!] \to [\![A]\!]$. From the rule's second hypothesis we have that $(\underline{C} \leqslant \underline{C}') \rightsquigarrow E_2$. Thus by the induction hypothesis we have that $\vdash E_2 : [\![\underline{C}]\!] \to [\![\underline{C}']\!]$. Hence, we have that $f : [\![A]\!] \to [\![\underline{C}]\!], x : [\![A']\!] \vdash E_2\, (f'\, (E_1\, x)) : [\![\underline{C}']\!]$ from rules O-Var and O-App. Hence, by rule O-Fun we have that $\vdash \mathtt{fun}\ f\, x \mapsto\ E_2\, (f'\, (E_1\, x)) : [\![\underline{C}']\!] : ([\![A]\!] \to [\![\underline{C}]\!]) \to ([\![A']\!] \to [\![\underline{C}']\!])$. Using the definition of $[\![\cdot]\!]$ we conclude $\vdash \mathtt{fun}\ f\, x \mapsto\ E_2\, (f'\, (E_1\, x)) : [\![\underline{C}']\!] : [\![A \to \underline{C}]\!] \to [\![A' \to \underline{C}']\!]$.

Sub-$\Rightarrow$: $(\underline{C} \Rightarrow \underline{D} \leqslant \underline{C}' \Rightarrow \underline{D}') \rightsquigarrow (\mathtt{fun}\ h\, x \mapsto\ E_2\, (h\, (E_1\, x)))$.

From the rule's first hypothesis we have that $\underline{C}' \leqslant \underline{C} \rightsquigarrow E_1$. Thus by the induction hypothesis we have that $\vdash E_1 : [\![\underline{C}']\!] \to [\![\underline{C}]\!]$. From the rule's second hypothesis we have that $\underline{D} \leqslant \underline{D}' \rightsquigarrow E_2$. Thus by the induction hypothesis we have that $\vdash E_2 : [\![\underline{D}]\!] \to [\![\underline{D}']\!]$. Hence, we have that $h : [\![\underline{C}]\!] \to [\![\underline{D}]\!], x : [\![\underline{C}']\!] \vdash E_2\, (h'\, (E_1\, x)) : [\![\underline{D}']\!]$ from rules O-Var and O-App. Hence, by rule O-Fun we have that $\vdash \mathtt{fun}\ h\, x \mapsto\ E_2\, (h'\, (E_1\, x)) : ([\![\underline{C}]\!] \to [\![\underline{D}]\!]) \to ([\![\underline{C}']\!] \to [\![\underline{D}']\!])$. Using the definition of $[\![\cdot]\!]$ we conclude $\vdash \mathtt{fun}\ h\, x \mapsto\ E_2\, (h'\, (E_1\, x)) : [\![\underline{C} \Rightarrow \underline{D}]\!] \to [\![\underline{C}' \Rightarrow \underline{D}']\!]$.

Sub-!-Pure: $(A \mathbin{!} \emptyset \leqslant A' \mathbin{!} \emptyset) \rightsquigarrow E$.
From the rule's hypothesis and the induction hypothesis we have that $\vdash E : [\![A]\!] \to [\![A']\!]$. Because $[\![A \mathbin{!} \emptyset]\!] = [\![A]\!]$ and $[\![A' \mathbin{!} \emptyset]\!] = [\![A']\!]$ we conclude $\vdash E : [\![A \mathbin{!} \emptyset]\!] \to [\![A' \mathbin{!} \emptyset]\!]$.

Sub-!-PureImpure: $(A \mathbin{!} \emptyset \leqslant A' \mathbin{!} \Delta') \rightsquigarrow (\mathtt{fun}\ x \mapsto \mathtt{return}\,(E\,x))$.
From the rule's hypothesis and the induction hypothesis we have that $\vdash E : [\![A]\!] \to [\![A']\!]$. From rules O-Retue, O-App and O-Var we have $x : [\![A]\!] \vdash \mathtt{return}\,(E\,x) : [\![A']\!]\ \mathtt{computation}$. From rule O-Fun we have $\vdash \mathtt{fun}\ x \mapsto \mathtt{return}\,(E\,x) : [\![A]\!] \to [\![A']\!]\ \mathtt{computation}$. Because $[\![A \mathbin{!} \emptyset]\!] = [\![A]\!]$ and $[\![A' \mathbin{!} \Delta']\!] = [\![A']\!]\ \mathtt{computation}$ (as $\Delta' \neq \emptyset$ from the rule's second hypothesis) we conclude that $\vdash \mathtt{fun}\ x \mapsto \mathtt{return}\,(E\,x) : [\![A \mathbin{!} \emptyset]\!] \to [\![A' \mathbin{!} \Delta']\!]$.

Sub-!-Impure: $(A \mathbin{!} \Delta \leqslant A' \mathbin{!} \Delta') \rightsquigarrow (\mathtt{fmap}\,E)$.
From the rule's hypothesis and the induction hypothesis we have that $\vdash E : [\![A]\!] \to [\![A']\!]$. From rules O-FMap and O-App we have that $\vdash \mathtt{fmap}\,E : [\![A]\!]\ \mathtt{computation} \to [\![A']\!]\ \mathtt{computation}$. As $\Delta \neq \emptyset$ and $\Delta' \neq \emptyset$, we have that $[\![A \mathbin{!} \Delta]\!] = [\![A]\!]\ \mathtt{computation}$ and $[\![A' \mathbin{!} \Delta']\!] = [\![A']\!]\ \mathtt{computation}$. Hence we conclude $\vdash \mathtt{fmap}\,E : [\![A \mathbin{!} \Delta]\!] \to [\![A' \mathbin{!} \Delta']\!]$.

□

Now follows the proof of the main theorem.

Proof of Theorem 6.2. We prove the preservation of typing for the basic elaboration by mutual induction on the elaboration derivations for values and computations.

(SubVal): $(\Gamma \vdash v : A') \rightsquigarrow E_2\,E_1$
The rule's first assumption is that $(\Gamma \vdash v : A) \rightsquigarrow E_1$. By the induction hypothesis we thus have that $[\![\Gamma]\!] \vdash E_1 : [\![A]\!]$. The rule's second assumption is that $(A \leqslant A') \rightsquigarrow E_2$. From Lemma C we then have that $\vdash E_2 : [\![A]\!] \to [\![A']\!]$. Thus we conclude by rule O-App $[\![\Gamma]\!] \vdash E_2\,E_1 : [\![A']\!]$.

(Var): $(\Gamma \vdash x : A) \rightsquigarrow x$
The proof proceeds analogously to one in Theorem 5.1.

(Const): $(\Gamma \vdash \mathsf{k} : A) \rightsquigarrow k$
The proof proceeds analogously to one in Theorem 5.1.

(Fun): $(\Gamma \vdash \mathtt{fun}\ x \mapsto c : A \to \underline{C}) \rightsquigarrow (\mathtt{fun}\ x \mapsto E)$
The proof proceeds analogously to one in Theorem 5.1.

(HandPure): $(\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r\} : A \mathbin{!} \emptyset \Rightarrow B \mathbin{!} \emptyset) \rightsquigarrow (\mathtt{fun}\ x \mapsto E_r)$
From the first hypothesis of the rule and the induction hypothesis we have that $[\![\Gamma, x : A]\!] \vdash E_r : [\![B \mathbin{!} \emptyset]\!]$. This simplifies to $[\![\Gamma]\!], x : [\![A]\!] \vdash E_r : [\![B \mathbin{!} \emptyset]\!]$. Hence, by rule O-Fun we have $[\![\Gamma]\!] \vdash \mathtt{fun}\ x \mapsto E_r : [\![A]\!] \to [\![B \mathbin{!} \emptyset]\!]$. As $[\![A \mathbin{!} \emptyset]\!] = [\![A]\!]$ we conclude $[\![\Gamma]\!] \vdash \mathtt{fun}\ x \mapsto E_r : [\![A \mathbin{!} \emptyset]\!] \to [\![B \mathbin{!} \emptyset]\!]$.

(HandImpure): $(\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r, [\mathsf{Op}\,x\,k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\} : A \mathbin{!} \Delta \cup O \Rightarrow B \mathbin{!} \Delta) \rightsquigarrow \mathtt{handler}\ \{\mathtt{return} = \mathtt{fun}\ x \mapsto E_r; \mathsf{op}_1 = E_1; \ldots; \mathsf{op}_n = E_n\}\}$
From the first hypothesis of the rule and the induction hypothesis we have that $[\![\Gamma, x : A]\!] \vdash [\![c_r]\!] : [\![B \mathbin{!} \Delta]\!]$. We can simplify this to $[\![\Gamma]\!], x : [\![A]\!] \vdash [\![c_r]\!] : [\![B \mathbin{!} \Delta]\!]$.

Next, we consider three possible cases for each E_i.

- First, if $\mathsf{Op}_i \in O$, we get $[\![\Gamma]\!], x : [\![A_{\mathsf{Op}_i}]\!], k : ([\![B_{\mathsf{Op}_i}]\!] \to [\![B \mathbin{!} \Delta]\!]) \vdash [\![E_{\mathsf{Op}_i}]\!] : [\![B \mathbin{!} \Delta]\!]$. Hence, by rule O-Fun we have that $[\![\Gamma]\!] \vdash E_i : [\![A_{\mathsf{Op}_i}]\!] \to ([\![B_{\mathsf{Op}_i}]\!] \to [\![B \mathbin{!} \Delta]\!]) \to [\![B \mathbin{!} \Delta]\!]$.
- Next, if $\mathsf{Op}_i \in \Delta - O$, we need to consider two options. If $\Delta = \emptyset$, the case is vacuous. If not, we have $[\![B \mathbin{!} \Delta]\!] = [\![B]\!]\ \mathtt{computation}$, and it is easy to see by rules O-Operation, O-Bind, O-App and O-Var that $[\![\Gamma]\!], x : [\![A_{\mathsf{Op}_i}]\!], k : ([\![B_{\mathsf{Op}_i}]\!] \to [\![B \mathbin{!} \Delta]\!]) \vdash (\mathsf{op}_i\,x \mathbin{>\!\!>\!\!=} k) : [\![B \mathbin{!} \Delta]\!]$. Hence, by rule O-Fun we have that $[\![\Gamma]\!] \vdash E_i : [\![A_{\mathsf{Op}_i}]\!] \to ([\![B_{\mathsf{Op}_i}]\!] \to [\![B \mathbin{!} \Delta]\!]) \to [\![B \mathbin{!} \Delta]\!]$.
- Finally, in the last case, we have that `assert false` has an arbitrary type due to polymorphism, thus again $[\![\Gamma]\!] \vdash E_i : [\![A_{\mathsf{Op}_i}]\!] \to ([\![B_{\mathsf{Op}_i}]\!] \to [\![B \mathbin{!} \Delta]\!]) \to [\![B \mathbin{!} \Delta]\!]$.

Analogously to the proof in Theorem 5.1, we conclude that $[\![\Gamma]\!] \vdash \{\texttt{return} = \texttt{fun}\ x \mapsto [\![c_r]\!]; \texttt{op}_1 = E_1; \ldots; \texttt{op}_n = E_n\} : ([\![A]\!], [\![B \mathbin{!} \Delta]\!])\ \texttt{handler\_cases}$ and $[\![\Gamma]\!] \vdash \texttt{handler}\ \{\texttt{return} = \texttt{fun}\ x \mapsto [\![c_r]\!]; \texttt{op}_1 = E_1; \ldots; \texttt{op}_n = E_n\} : [\![A]\!]\ \texttt{computation} \to [\![B \mathbin{!} \Delta]\!]$, implying $[\![\Gamma]\!] \vdash [\![\{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\, x\, k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in O}\}]\!] : [\![A \mathbin{!} \Delta \cup O \Rightarrow B \mathbin{!} \Delta]\!]$.

(SubComp): $(\Gamma \vdash c : \underline{C}') \rightsquigarrow E_2\, E_1$

The proof proceeds analogously to one for rule SubVal.

(App): $(\Gamma \vdash e_1\, e_2 : \underline{C}) \rightsquigarrow E_1\, E_2$

The proof proceeds analogously to one in Theorem 5.1.

(LetRec): $(\Gamma \vdash \texttt{let rec}\ f\, x = c_1\ \texttt{in}\ c_2 : \underline{D}) \rightsquigarrow \texttt{let rec}\ f\, x = E_1\ \texttt{in}\ E_2$

The proof proceeds analogously to one in Theorem 5.1.

(Ret): $\Gamma \vdash \texttt{return}\, v : A \mathbin{!} \emptyset \rightsquigarrow E$

The proof proceeds analogously to one in Theorem 5.1.

(Op): $\Gamma \vdash \texttt{Op}\, v : B \mathbin{!} \{\texttt{Op}\} \rightsquigarrow \textit{operation}\, \texttt{Op}\, E$

The proof proceeds analogously to one in Theorem 5.1.

(DoPure): $\Gamma \vdash \texttt{do}\ x \leftarrow c_1\ ; c_2 : B \mathbin{!} \emptyset$

From the rule's first hypothesis and the induction hypothesis we have $[\![\Gamma]\!] \vdash E_1 : [\![A \mathbin{!} \emptyset]\!]$. We can simplify this to $[\![\Gamma]\!] \vdash E_1 : [\![A]\!]$. From the rule's second hypothesis and the induction hypothesis we have $[\![\Gamma, x : A]\!] \vdash E_2 : [\![B \mathbin{!} \emptyset]\!]$. We can simplify this to $[\![\Gamma]\!], x : [\![A]\!] \vdash E_2 : [\![B]\!]$. Using rule O-Let we conclude $[\![\Gamma]\!] \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 : [\![B]\!]$.

(DoImpure): $\Gamma \vdash \texttt{do}\ x \leftarrow c_1\ ; c_2 : B \mathbin{!} \Delta \rightsquigarrow E_1 \gg\!= \texttt{fun}\ x \mapsto E_2$

The proof proceeds analogously to one for Do in Theorem 5.1.

(With): $\Gamma \vdash \texttt{handle}\ c\ \texttt{with}\ v : \underline{D} \rightsquigarrow E_1\, E_2$

The proof proceeds analogously to one in Theorem 5.1.

□