CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Towards Normalization by Evaluation for Erlang

From Moggi's computational λ-calculus to Erlang

Master's thesis in Computer science and engineering

CARL AGRELL
HAOHAN YANG

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Master's thesis 2023

Towards Normalization by Evaluation for Erlang

From Moggi's computational λ-calculus to Erlang

CARL AGRELL
HAOHAN YANG

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Towards Normalization by Evaluation for Erlang
From Moggi's computational λ-calculus to Erlang
CARL AGRELL
HAOHAN YANG

Supervisor: Nachiappan Valliappan, CSE
Examiner: Andreas Abel, CSE

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Towards Normalization by Evaluation for Erlang
From Moggi's computational λ-calculus to Erlang
CARL AGRELL
HAOHAN YANG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Normalization by Evaluation (NbE) is a technique to extract the normal form of a λ-calculus term. Originally targeting pure λ-calculus, the algorithm can be extended and adjusted to work on more practical call-by-value programming languages. We implement NbE for a fragment of Erlang, starting at the Moggi's semantics framework of computational lambda calculus, and largely inspired by Filinski's research. The result shows that our normalizer can be applied on programs with rich semantics, and can potentially be extended to perform partial evaluation.

Keywords: Erlang, Lambda calculus, Normalization, Call-by-value, Normalization by Evaluation, Partial Evaluation.

Acknowledgements

We would like to express our gratitude to our supervisor Nachiappan Valliappan for his advice and guidance, both regarding the research and the administrative parts of this thesis. We would also like to thank Professor John Hughes for valuable feedback on the project.

Carl Agrell and Haohan Yang, Gothenburg, 2023-06-19

Contents

1

Introduction

The last decade has witnessed the success of Erlang and its increasing applications in industries. Erlang features immutable data, a lightweight processes-based concurrency model, asynchronous message passing, and special error handling. With these traits, Erlang shines in building large-scale, fault-tolerant distributed systems. Some of the noticeable applications include CouchDB, a distributed NoSQL database system, and RabbitMQ, a message broker. Erlang is a departure from statically-typed functional programming languages such as Standard ML and Haskell. Although dynamic typing offers more flexibility in practice, Erlang somewhat lacks some advantages provided by type constraints. For example, type errors are captured only at runtime.

A substantial amount of work has benefited Erlang by applying type-related theories. For example, several type checkers of Erlang have been developed to statically diagnose type errors. In programming language semantics, *Normalization by Evaluation* (NbE) is an important research topic on obtaining the normal form of a lambda term from an evaluation result. NbE was initially applied to simple pure lambda calculus, so a challenge is to adjust this theoretical framework on lambda calculus to incorporate more practical programming languages. We aim to apply the algorithm of typed NbE to a fragment of Erlang and develop a normalizer that returns the normal form of an input Erlang program. NbE usually produces a simplified program with possibly better execution efficiency, eliminating reducible static expressions. The results can contribute to both practical applications of the NbE algorithm and Erlang programming.

NbE on Erlang also brings better insights into the semantics of Erlang programs. The convertibility, or equivalence, of two Erlang programs, can be decided by comparing their normal forms. This provides very helpful assistance in terms of code refactoring. Further, *type-directed partial evaluation* (TDPE), an application of NbE, can be applied to Erlang if the static inputs are known, resulting in a specialized program that can be executed with improved performance.

1.1 Normalization by Evaluation

Normalization by Evaluation (NbE) is a method to obtain the normal form of a term of λ-calculus. *Normalization* is used to simplify the proofs in logical systems where a term can perform step-wise reductions into a $\beta\eta$-long normal term with no more

possible reductions [8]. For example, a simple lambda calculus term $\lambda x.\,(\lambda y.\,y)x$ is not in its normal form but can be β-reduced to the normal form $\lambda x.\,x$, which is essentially an identity function. The NbE algorithm usually consists of two steps: A term is first *evaluated* into a denotational, non-standard semantics, and then *reified* into a normal representation[8]. Thus the *normalization function* `norm` on a term `E` can be expressed as the following:

```
norm(E) = reify(eval(E))
```

There has been a significant amount of research on this topic. NbE was first presented by Martin-Löf[15], with a normalization proof where a normalization function is proved to actually return a normal form[8]. He regarded it as a model of normalization by intuitionistic construction in which he defined "definitional equality" to be driven by the evaluation of an intuitionistic meta-language[14, 8]. Berger and Schwichtenberg[3] developed NbE for typed λ-calculus with β and η conversion[8]. Danvy[5] presented *type-directed partial evaluation* based on NbE for closed and monomorphically typable lambda calculus. Filinski[11] extended NbE to function on a call-by-value language based on Moggi's computational λ-calculus[16]. A typeful NbE utilizing *Generalized Algebraic Data Types* (GADTs) and *Continuation-Passing Style* (CPS) was implemented in OCaml[7].

One striking property of NbE is that the convertibility between two terms can be decided by computing their normal forms, given that the normalization function is computable[8]:

$$\vdash E \equiv E' \text{ iff } \operatorname{norm} E = \operatorname{norm} E'$$

1.2 Normalizing Erlang

Initially developed as a programming language aiming for better performance and reliability of telecommunications applications[1], Erlang has been used extensively as a practical programming language in building robust, reliable, fault-tolerant distributed systems. The NbE on Erlang is likely to be beneficial in both research and practical applications.

However, the actual implementation of NbE for Erlang poses several challenges. The nature of dynamical typing makes it difficult to benefit Erlang from type specifications. Additionally, given the rather restrictive original setting of NbE in λ-calculus, several adaptations of the original NbE algorithm are necessary. Under a formally defined normal-order setting, the algorithm needs to be adjusted to work in a call-by-value language that has computational side effects[5, 9]. Filinski[11] extended NbE algorithm for an ML-like call-by-value language whose semantics is defined in Moggi's framework of computational lambda calculus[16]. Erlang, similarly, should be a potential candidate for the algorithm, although more adaptations are required.

This thesis provides a concrete implementation of NbE for a fragment of Erlang, based on Filinski *et al.*'s[11, 8] theoretical background on call-by-value languages based on Moggi's computational lambda calculus[16]. Our work bridges the gap between Moggi's theoretical framework and its concrete application in Erlang. The

supported language is a subset of Erlang, including integer arithmetic, string, atoms, tuples, conditionals, `case` clauses, certain built-in functions, IO, function definitions, and type specifications. Given an input source file with proper type specifications, our normalizer can recognize both local and remote functions, and output a new source file containing normalized functions. We also explored partial evaluation using NbE, inspired by *type-directed partial evaluation*[5]. Our work shows the versatility of NbE in a more practical application and its potential to function on full-fledged functional programming languages. Although we only present part of Erlang, the work can be extended to subsume more Erlang syntax.

In the following chapters, we will first introduce a simple call-by-name NbE, move on to the computational lambda calculus and monads, and advance towards a concrete implementation of the algorithm on Erlang. Following this, we present our results as a number of examples of normalized programs. In the final chapter, we will summarize our work.

The finished product can be found online on GitHub[1].

[1] `https://github.com/haohanyang/nbe_erlang`

2

Theory

2.1 Normalization by Evaluation

In λ calculus, a term is in β-η normal form if neither β nor η reductions can be performed. *Normalization by evaluation* (NbE) is a method of computing such a normal form from an arbitrary term. A term is first interpreted into a suitable, quasi-syntactic denotational model of the conversion relation [11, 4], after which this semantic form is converted into a syntactic normal form.

Important contributions to this area include Martin-Löf's work on normalization proofs and intuitionistic type theory[14, 15, 8], Berger and Schwichtenberg's work on β-η conversion of typed λ-calculus[3, 8], Danvy's work on *type-directed partial evaluation*[5] and Filinksi's work on computational λ-calculus[11].

2.1.1 A simple call-by-name language

We first present NbE in the setting of a simple call-by-name language[8]. Simple types τ include the base types b, function types, and pair types, defined by the rules:

$$\frac{}{\vdash b \text{ type}} \qquad \frac{\vdash \tau_1 \text{ type} \quad \vdash \tau_2 \text{ type}}{\vdash \tau_1 \to \tau_2 \text{ type}} \qquad \frac{\vdash \tau_1 \text{ type} \quad \vdash \tau_2 \text{ type}}{\vdash \tau_1 \times \tau_2 \text{ type}}$$

Type inference in a context Δ are defined inductively by the following rules:

$$\frac{\Delta(x) = \tau}{\Delta \vdash x : \tau} \qquad \frac{\Delta, x : \tau_1 \vdash E : \tau_2}{\Delta \vdash \lambda x.\, E : \tau_1 \to \tau_2} \qquad \frac{\Delta \vdash E_1 : \tau_1 \quad \Delta \vdash E_2 : \tau_2}{\Delta \vdash \{E_1, E_2\} : \tau_1 \times \tau_2}$$

$$\frac{\Delta \vdash E_1 : \tau_1 \to \tau_2 \quad \Delta \vdash E_2 : \tau_1}{\Delta \vdash E_1\, E_2 : \tau_2} \qquad \frac{\Delta, x : \tau_1 \vdash E : \tau_2}{\Delta \vdash \lambda x.\, E : \tau_1 \to \tau_2}$$

$$\frac{\Delta \vdash E : \tau_1 \times \tau_2}{\Delta \vdash \mathrm{fst}\, E : \tau_1} \qquad \frac{\Delta \vdash E : \tau_1 \times \tau_2}{\Delta \vdash \mathrm{snd}\, E : \tau_2}$$

The $\beta\eta$-convertibility between two terms E and E', denoted as $\vdash E =_{\beta\eta} E'$, is generated by the following rules[1]:

$$\frac{\vdash E_1 =_{\beta\eta} E_1' \quad \vdash E_2 =_{\beta\eta} E_2'}{\vdash E_1\, E_2 =_{\beta\eta} E_1'\, E_2'} \qquad \frac{\vdash E =_{\beta\eta} E'}{\vdash \lambda x.\, E =_{\beta\eta} \lambda x.\, E'}$$

[1] Plus the usual reflexivity, symmetry, and transitivity.

$$\frac{}{\vdash (\lambda x.E_1)\, E_2 =_{\beta\eta} E_1[x := E_2]} \qquad \frac{}{\vdash \lambda x.\, E\, x =_{\beta\eta} E}(x \notin E)$$

$$\frac{\vdash A =_{\beta\eta} A' \quad \vdash B =_{\beta\eta} B'}{\vdash \{A, B\} =_{\beta\eta} \{A', B'\}} \qquad \frac{}{\vdash \{\mathrm{fst}\, E, \mathrm{snd}\, E\} =_{\beta\eta} E}$$

$$\frac{}{\vdash \mathrm{fst}\{A, B\} =_{\beta\eta} A} \qquad \frac{}{\vdash \mathrm{snd}\{A, B\} =_{\beta\eta} B}$$

The $\beta\eta$-long normal forms are usually expressed using two mutually recursive judgements enumerating terms in normal and *atomic* (also known as *neutral*) forms:

$$\frac{\Delta \vdash^{at} E_1 : \tau_1 \to \tau_2 \quad \Delta \vdash^{nf} E_2 : \tau_1}{\Delta \vdash^{at} E_1\, E_2 : \tau_2} \qquad \frac{\Delta(x) = \tau}{\Delta \vdash^{at} x : \tau}$$

$$\frac{\Delta \vdash^{at} E : b}{\Delta \vdash^{nf} E : b} \qquad \frac{\Delta, x : \tau_1 \vdash^{nf} E : \tau_2}{\Delta \vdash^{nf} \lambda x.\, E : \tau_1 \to \tau_2}$$

$$\frac{\Delta \vdash^{nf} E_1 : \tau_1 \quad \Delta \vdash^{nf} E_2 : \tau_2}{\Delta \vdash^{nf} \{E_1, E_2\} : \tau_1 \times \tau_2} \qquad \frac{\Delta \vdash^{nf} E : \tau_1 \times \tau_2}{\Delta \vdash^{nf} \mathrm{fst}\, E : \tau_1} \qquad \frac{\Delta \vdash^{nf} E : \tau_1 \times \tau_2}{\Delta \vdash^{nf} \mathrm{snd}\, E : \tau_2}$$

Assume a simple language consisting of the syntax constructors `LAM`, `VAR`, `APP`, `PAIR`, `FST`, and `SND`. `PAIR` constructs a tuple of two elements. `FST` and `SND` extract the first and second elements of a pair, respectively. The abstract syntax and types for this language are as follows[2]:

```
-type syn() ::
    {'LAM', atom(), syn()}
    | {'APP', syn(), syn()}
    | {'PAIR', syn(), syn()}
    | {'FST', syn()}
    | {'SND', syn()}
    | {'VAR', atom()}

-type typ() ::
    {ty_fun, typ(), typ()}
    | {ty_pair, typ(), typ()}
    | ty_base.
```

We introduce the semantics a term is interpreted into. `sem_lam` contains a directly callable Erlang function. Elements of `sem_pair` can be extracted using Erlang's built-in function `element/2`. The static reduction is performed by Erlang's function application and call of `element/2`. `sem_syn` wraps up a syntax fragment that is supposed to be an atomic term.

```
-type sem() ::
    {sem_lam, fun((sem()) -> sem())}
    | {sem_pair, {sem(), sem()}}
    | {sem_syn, syn()}.
```

[2]In this section, we use capital symbols to denote syntax, and lowercase symbols for semantics. In Erlang, uppercase symbols need to be surrounded by single quotes, which is possibly part of why Erlang's own syntax (section 2.4) does not do this.

The evaluation function has an environment `Ctx` that maps variables to semantics and produces the denotational semantics.

```
eval({'LAM', Para, Body}, Ctx) ->
    {sem_lam, fun(X) -> eval(Body, maps:put(Para, X, Ctx)) end};
eval({'APP', FUN, X}, Ctx) ->
    {sem_lam, Fun} = eval(FUN, Ctx),
    Fun(eval(X, Ctx));
eval({'PAIR', T1, T2}, Ctx) ->
    {sem_pair, {eval(T1, Ctx), eval(T2, Ctx)}};
eval({'FST', T}, Ctx) ->
    {sem_pair, {V, _}} = eval(T, Ctx),
    V;
eval({'SND', T}, Ctx) ->
    {sem_pair, {_, V}} = eval(T, Ctx),
    V;
eval({'VAR', At}, Ctx) ->
    maps:get(At, Ctx).
```

We define two mutually recursive functions `reify` and `reflect`. Reification, denoted as $\downarrow_\tau$ over type τ, converts a semantic value to its syntactic normal form. Reflection, denoted as $\uparrow_\tau$ over type τ, wraps up a fragment of syntax (usually atomic term) by interpreting its type[8]. We have the following algorithm of reification and reflection.

$$
\begin{aligned}
\downarrow_\tau &:: \texttt{sem()} \to \texttt{syn()} \\
\downarrow_{\underline{b}} &= \lambda t.\, t \\
\downarrow_{\tau_1 \times \tau_2} &= \lambda p.\, \mathrm{PAIR}(\downarrow_{\tau_1}(\mathrm{fst}\, p), \downarrow_{\tau_2}(\mathrm{snd}\, p)) \\
\downarrow_{\tau_1 \to \tau_2} &= \lambda f.\, \mathrm{LAM}(\lambda v.\, \downarrow_{\tau_2}(f(\uparrow_{\tau_1}(\mathrm{VAR}\, v)))) \\
\uparrow_\tau &:: \texttt{syn()} \to \texttt{sem()} \\
\uparrow_{\underline{b}} &= \lambda e.\, e \\
\uparrow_{\tau_1 \times \tau_2} &= \lambda p.\, \{\uparrow_{\tau_1}(\mathrm{FST}\, p), \uparrow_{\tau_2}(\mathrm{SND}\, p)\} \\
\uparrow_{\tau_1 \to \tau_2} &= \lambda e.\, \lambda a.\, \uparrow_{\tau_2}(\mathrm{APP}(e, \downarrow_{\tau_1} a))
\end{aligned}
$$

The reification of functions requires a fresh name generation. Here, we use a parallel process for this. In our full implementation, we instead use a state monad[3].

```
-spec reify(sem(), typ()) -> syn().
reify({sem_lam, Fun}, {ty_fun, A, B}) ->
    Var = aux_server:new_var(),
    {'LAM', Var, reify(Fun(reflect({'VAR', Var}, A)), B)};
```

[3]The main difference between the approaches is that the state monad is deterministic in the case of concurrent invocations. If the normalizer runs multiple times in parallel, such as when running our test suite, the name generation process would interleave in an unpredictable way.

```
reify({sem_pair, {A, B}}, {ty_pair, T1, T2}) ->
    {'PAIR', reify(A, T1), reify(B, T2)};
reify({sem_syn, Syn}, ty_base) ->
    Syn.

-spec reflect(term(), typ()) -> sem().
reflect(Term, {ty_fun, A, B}) ->
    {sem_lam, fun(X) -> reflect({'APP', Term, (reify(X, A))}, B) end};
reflect(Term, {ty_pair, {T1, T2}}) ->
    {sem_pair, {reflect({'FST', Term}, T1), reflect({'SND', Term}, T2)}};
reflect(Term, ty_base) ->
    {sem_syn, Term}.

norm(Term, Type) ->
    reify(eval(Term, maps:new()), Type)
```

As an example, we define the SK combinators[4] using the aforementioned syntax.

```
k() -> {'LAM', x, {'LAM', y, {'VAR', x}}}.
s() ->
    {'LAM', x,
        {'LAM', y,
            {'LAM', z,
                {'APP',
                    {'APP', {'VAR', x}, {'VAR', z}},
                    {'APP', {'VAR', y}, {'VAR', z}}
                }
            }
        }
    }.
skk() -> {'APP', {'APP', s(), k()}, k()}.
```

The normalization result of SKK is simply an identity function.

```
{'LAM', 'V0', {'VAR', 'V0'}}
```

2.1.2 Towards Call-by-Value

The previous section summarizes NbE on a simple call-by-name language, which is far from a full-fledged programming language. It is preferable to work on a call-by-value language for practical reasons, considering several drawbacks of the call-by-name setting.

Suppose the language additionally has a string constructor STRING and a side-effecting function PRINT. In this setting, a program such as $(\lambda x.\,(x, x))$ (print *"Hello"*) would be normalized to (print *"Hello"*, print *"Hello"*), while $(\lambda x.\,\lambda y.\,y)$ (print *"Hello"*) would be normalized to $\lambda y.\,y$. The duplication and elimination of side effects is con-

[4] $K = \lambda a.\,\lambda x.\,a$; $S = \lambda a.\,\lambda b.\,\lambda x.\,ax(bx)$

tradicting the semantics of a call-by-value programming language, as a result of the application rule in call-by-name λ-calculus:

$$(\lambda x.E_1)\, E_2 \Rightarrow E_1[x := E_2].$$

This rule inserts the function argument syntactically for each occurrence of the parameter, which works well in the call-by-name λ-calculus. But this is not viable in terms of side effects. Instead, we need the call-by-value lambda application rule[5]:

$$(\lambda x.E_1)\, E_2 \Rightarrow \textbf{let}\, x = E_2\, \textbf{in}\, E_1.$$

The rule ensures not only that side effects are neither duplicated nor discarded, but also that they occur in the correct order. This comes with much richer semantics to deal with: our language now has the notions of variable bindings[6] and sequencing.

We can see that NbE has a rather restrictive setting for pure λ-calculus, and the algorithm subsequently has to be adjusted to function on a call-by-value language with effects[5, 10, 9]. Thus, the NbE algorithm in the setting of Moggi's computational λ-calculus is needed. We will see that the use of monads in terms of modeling computational effects plays a vital role in this setting.

2.2 Continuations

A *continuation* represents the rest or future of computation, which is traditionally handled explicitly in *Continuation-Passing Style* (CPS) —- a style of programming where the result is not returned directly, but instead passed to the next function as an argument. CPS consists of a chain of several continuous computations where the result is passed through. The result is usually extracted by passing an identity function, or the `return` function of a suitable monad.

The following program of CPS passes the initial value `2` through two CPS computations `plus_one` and `times_two`, yielding the result `6` in the end.

```
plus_one(X) ->
    fun(F) -> F(X + 1) end.

times_two(X) ->
    fun(F) -> F(2 * X) end.

comp() ->
    (plus_one(2))
```

[5]Other rules such as pairs also need adaptation for the call-by-value semantics, but there is little value in continuing the discussion of this here. Instead, we invite readers to our implementation of Erlang semantics in section 3.

[6]Our implementation of `eval` above already has a map of variable bindings, but this is merely an implementation detail.

```
(fun(Res1) ->
    (times_two(Res1))
        (fun(Res2) -> Res2 end) end).
```

2.2.1 Continuation monad

However, writing programs in this way is cumbersome and error-prone, with the need to carry these continuation arguments around. In some languages, including Scheme, Ruby, and Standard ML of New Jersey, continuations are natively supported as first-class continuations. Erlang doesn't have first-class continuations. However, it happens to be the case that CPS can be expressed in a monadic style, which alleviates the need to manually pass the continuations.

But what about the cases where we do not want to blithely pass the continuation forward? Being able to manipulate the continuation is, after all, what makes CPS so powerful. For this, there are a small set of commonly implemented *control operators*[2].

Perhaps the most famous control operator is `call/CC` (*call-with-current-continuation*), which captures the entire continuation of the program. However, it is more useful in practice to manipulate a part of the program via *delimited continuation* operators including `shift` and `reset` operators[2]. Whereas `call/CC` captures the entire continuation, `shift` captures the continuation up to the nearest enclosing `reset`.

2.2.2 Delimited continuations

In this section, we introduce control operators *shift* and *reset* [6] that will be utilized in our NbE algorithm. In his original paper, Danvy[6], etc described that "*shift* abstracts the current context as an ordinary, composable procedure and *reset* delimits the scope of such a context".

Suppose a program is trying to compute an expression $\text{print}(1 + \underline{2 \times 3} + 4)$, where 2×3 is the sub-expression currently being computed. In this case, the *continuation* is $\text{print}(1+\bullet+4)$, where to $\bullet$ needs to be filled in with a value before the computation can proceed. If we were to capture the continuation with *shift* and invoke it with the value 6, the program would print out 11, and the continuation would return a unit value. However, unit values aren't all that useful: we would rather get 11 as a value. This is what the *reset* operator is used for, and why they are called *delimited* continuations: if we insert a *reset* in the appropriate place, we instead get the continuation $\text{print}([1 + \bullet + 4])$. When we invoke the captured continuation, the part within the [] is executed and we get the value 11.

The following example demonstrates how the combination of delimited operators can be used to trace different possible values by plugging multiple values into the continuation k while preserving the original structure. Suppose we extend the language from section 2.1.1 with `IF`, `TRUE`, and `FALSE`[7].

[7]The examples below are written in an ML-like pseudocode, as Erlang's syntax is a bit noisy.

```
reset (if (shift (fn k -> IF (VAR "v", k true, k false)))
       then TRUE
       else FALSE)
```

We first observe that the whole delimited continuation will be replaced by `IF (VAR "V", k true, k false)`. The delimited continuation is `if • then TRUE else FALSE`, where the hole is to be filled with a boolean value. The captured continuation `k` is invoked in both branches in the `IF` block: with a true value in the true branch, and a false value in the false branch. In effect, this code is equivalent to:

```
IF (VAR "v",
    if true (* k true *)
       then TRUE
       else FALSE,
    if false (* k false *)
       then TRUE
       else FALSE)
```

While this example is a bit mundane, if we did away with the `reset` call and only had the `shift (fn k -> IF (VAR "v", k true, k false))`, there would be no possible such closed form. Instead, the `true` and `false` would bubble upwards, capturing whatever context the function is called in and placing it inside a conditional statement. This way, we can implement a wide number of control flow statements without having to know anything about how they are used. Our normalizer uses this property to implement reflection of boolean values and sum types, in section 3.3.

Delimited continuations can also be used to switch the execution order of the surrounding context and the local computation[8]. To demonstrate this, we further extend our language with a `LET` binding, integers `INT`, and an arithmetic operation `PlusOne`.

```
reset (APP (VAR "s",
            shift (fn -> LET ("v", INT 1, k (PlusOne (VAR "v"))))
      ))
```

As usual, the continuation will be replaced by a let binding `LET ("v", INT 1, k (PlusOne (VAR "v"))`. The remaining computation `APP` should be ideally placed in the body of the binding. Notice that the current continuation is `APP (VAR "s", •)`. The hole is filled with the term `PlusOne (VAR "v")`. We can directly use a new variable in the sub-term since the term is already surrounded by the binding context. Finally, the computation yields the following result.

```
LET ("v", INT 1, APP (VAR "s", PlusOne (VAR "v")))
```

This is very useful in a call-by-value language setting with effects. The let insertion also solves the problem of duplicating side effects, since the binding value can be used in later computations.

2.2.3 Implementation in Erlang

`shift/reset` can be emulated using `call/CC` and reference cells[12, 2]. Since Erlang doesn't have first-class continuations, we use the continuation monad `m(T)`, where T is the result type.

$$
\begin{aligned}
m\,a &= (a \to e) \to e \\
\texttt{return} &:: a \to m\,a \\
\texttt{return}\ a &= \lambda\kappa.\,\kappa\ a \\
\texttt{>>=} &:: ma \to (a \to m\,b) \to m\,b \\
t\ \texttt{>>=}\ f &= \lambda\kappa.\,t\,(\lambda a.\,f\,a\,\kappa)
\end{aligned}
$$

We can then define `shift` and `reset`

$$
\begin{aligned}
\texttt{reset} &:: m\,e \to m\,e \\
\texttt{reset}\ t &= \texttt{return}\,(t\,\lambda x.x) \\
\texttt{shift} &:: ((a \to m\,e) \to m\,e) \to m\,a \\
\texttt{shift}\ h &= \lambda\kappa.\,(h\,(\lambda a\,\kappa'.\,\kappa'(\kappa\,a)))\,(\lambda x.\,x)
\end{aligned}
$$

For example, "reset$(2 + \text{shift}(\lambda k.\,k\,3) + 1)$" is expressed as

$$\text{reset}\,(\text{shift}(\lambda k.\,k\,3)\ \texttt{>>=}\ \lambda c.\,\text{return}(2 + c + 1))$$

One way to implement the continuation monad in Erlang is as follows[8]:

```
-type m(A) :: {cont, fun((fun((A) -> syn())) -> syn())}.

-spec app(m(A), fun((A) -> syn())) -> syn().
app({cont, M}, V) -> M(V).

-spec run(m(A)) -> A.
run(M) ->
    app(M, fun(A) -> A end).

-spec return(A) -> m(A).
return(A) ->
    {cont, fun(K) -> K(A) end}.

-spec bind(m(A), fun((A) -> B)) -> m(B).
bind(T = {cont, _}, F) ->
    {cont, fun(K) -> app(T, fun(A) -> app(F(A), K) end) end}.

-spec reset(m(A)) -> m(A).
reset(T) ->
```

[8]The matching of `T = {cont, _}` in `bind` is technically redundant, but it helps to catch type errors earlier.

```
    return(run(T)).

-spec shift(fun((fun((A) -> m(syn()))) -> m(syn()))) -> m(A).
shift(H) ->
    {cont, fun(K) -> run(H(fun(A) -> return(K(A)) end)) end}.
```

Suppose we want to refactor a nested function application by introducing extra `let`-bindings. This kind of normalization result is known as A-normal form[2]. For example, in the S combinator, we want to replace the internal computation inside the function argument with an immediate variable.

```
{app,
    {app, {var, x}, {var, z}},
    {app, {var, y}, {var, z}}
}

% A-normal form
{let, 'M', {app, {var, x}, {var, z}},
{let, 'N', {app, {var, y}, {var, z}},
{app, 'M', 'N'}
}}
```

We can write a recursive normalization function utilizing `shift` and `reset`[2]. `reset` delimit the continuation up to `LAM`'s body, while `shift` wraps up a `let`-binding around the remaining computation.

```
-type syn() ::
    {'LAM', atom(), syn()}
    | {'APP', syn(), syn()}
    | {'VAR', atom()}
    | {'LET', atom(), syn(), syn()}

norm({'VAR', Var}) ->
    return({'VAR', Var});
norm({'LAM', Var, Body}) ->
    reset(norm(Body))
    >>= fun(T) -> return({'LAM', Var, T}) end;
norm({'APP', E1, E2}) ->
    shift(fun(K) ->
        Var = {'VAR', aux_server:new_var()},
        sequence([norm(E1), norm(E2), K(Var)])
        >>= fun([E1_, E2_, E_]) ->
            return({'LET', Var, {'APP', E1_, E2_}, E_})
        end
    end).
```

As a result, the A-normalized S combinator introduces more `let`-bindings.

```
{'LAM', x,
{'LAM', y,
{'LAM', z,
    {'LET', {'VAR', 'V2'}, {'app', {'VAR', x}, {'VAR', z}},
    {'LET', {'VAR', 'V3'}, {'app', {'VAR', y}, {'VAR', z}},
    {'LET', {'VAR', 'V1'}, {'app', {'VAR', 'V2'}, {'VAR', 'V3'}},
    {'VAR', 'V1'}
    }}}
}}}
```

2.3 State monad

The state monad enables state management in a setting without mutable variables, without having to explicitly thread variables around as function arguments. The state monad is essentially a function that takes a state (context) and returns a tuple carrying the next state and a result value:

```
-type m(A) :: {state, fun((ctx()) -> inner:m({ctx(), A}))}.
```

In our scenario, the state monad is actually a state monad transformer over another monad (continuation monad). We wrap a type with another monad using *lift* and extract the inner type with the *run* operator. The monad transformer combines several monads together where each monad's ability can be utilized.

2.4 Abstract syntax

Erlang has a standard representation for its abstract syntax tree, known as the *abstract format*. This syntax tree is used both by the canonical interpreter and a variety of third-party tools. Consequently, there are built-in modules for converting between the source code string and abstract syntax: tokenizing with `erl_scan`, parsing with `erl_parse`, and pretty-printing with `erl_pp`.

Our normalizer primarily works with expressions, known in Erlang as `abstract_expr()`. This represents a variety of expressions such as literals, anonymous functions, function calls, variable assignments, and conditionals.[9]

Each node has an annotation indicating the line number of the corresponding input syntax. These annotations have no bearing on functionality, so we disregard them.

The fragment of Erlang that we support can be summarized as:

```
-type expr() ::
    {integer, _, integer()}
    | {string, _, string()}
```

[9]It does not include top-level definitions such as function declarations or imports. These instead fall under `abstract_form()`, which we do not interact with much.

```
    | {atom, _, atom()}
    | {tuple, _, [expr()]}
    | {var, _, atom()} % Variable
    | {match, _, expr(), expr()} % `=` operator
    | {op, _, '+'|'-'|'*', expr(), expr()} % Other operators
    | {'if', _, [clause()]}
    | {'case', _, expr(), [clause()]}
    | {'fun', _, {clauses, [clause()]}}
    | {call, _, expr(), [expr()]}.

-type clause() ::
    {clause,
     _,
     [expr()], % Match pattern - case and function only
     [expr()], % Guards - if only
     [expr()] % Body
    }.
```

As an example, `B = A + 1` is represented as:

```
{match, _,
    {var, _, 'B'},
    {op, _, '+',
        {var, _, 'A'},
        {integer, _, 1}
    }
}
```

3

Methods

The normalization process consists of several steps. We start by parsing the input Erlang program into an *abstract syntax tree* representation (section 2.4) using built-in functions. Next, we *evaluate* this abstract syntax into a custom *semantic domain* (section 3.1), after which we *reify* this semantic value back into a new abstract syntax tree[1] (section 3.2). Finally, this new syntax tree is pretty-printed to a readable Erlang program, again via built-in functions.

3.1 Evaluation and Semantics

The evaluator interprets an Erlang term into a suitable *semantics*. The main function involved is

```
-type sem() ::
    {sem_var, var()}
    | {sem_int, integer()}
    | {sem_str, string()}
    | {sem_atom, atom()}
    | {sem_fun, fun(([sem()]) -> rere:m(sem()))}
    | {sem_tuple, [sem()]}.

-spec eval(abstract_expr()) -> m(sem()).
```

`sem()` represents values, or *semantics*, for our chosen Erlang fragment. `rere:m(T)` is a monad [2], used to carry a variety of state needed for the evaluation process. Both components will be discussed in tandem with the `eval()` function in the remainder of this section.

Literals are simple: their semantics are plain values, and evaluating the syntactic literal produces the corresponding semantic value.

```
eval({integer, _, I}) -> return({sem_int, I});
eval({string, _, S}) -> return({sem_str, S});
```

[1] These two steps are not quite performed sequentially, but are interleaved with the help of monads. More on this will be discussed later.

[2] The evaluator does, in principle, not care about the underlying monad. However, for implementation reasons, it is fixed to the monad used for reflection and reification (hence the `rere`), in section 3.2.

```
eval({atom, _, A}) -> return({sem_atom, A});
```

Tuples hold a sequence of values, and correspondingly, evaluating a tuple consists of evaluating each item in turn and collecting them into a semantic tuple. `eval_list/1` evaluates a list of expressions, returning a list of semantic values: in Haskell parlance, it would be equivalent to `mapM eval`.

```
eval({tuple, _, Exprs}) ->
    eval_list(Exprs)
    >>= fun(Sems) -> return({sem_tuple, Sems}) end;
```

It should be noted that, for brevity, these code examples use an inline operator `>>=` to represent the function `bind/2`.

Variables are where things start to get interesting. Astute readers may have noted the `sem_var` in the semantics, but this is in fact largely unrelated — `sem_var` represents an opaque value that the evaluator has no knowledge about, and therefore can do very little with. It will be discussed in greater detail in section 3.3.

Instead, variable assignments are tracked in the state monad transformer. The state being tracked is as follows:

```
-record(ctx, {
    scope : #{atom() => sem()},
    remote :: #{{atom(), atom(), integer()} => type()},
    local :: #{{atom(), integer()} => expr()}
}).
```

Reading a variable is done by looking it up in the `scope` dictionary. The fields `remote` and `local` are used for function calls, which are elaborated on below.

Reading a variable loads the variable from the state monad with the `lookup/1` function, while assignments perform pattern matching and variable binding with the `match/2` function. These functions do not do anything particularly noteworthy, so their definitions are omitted.

```
eval({var, _, Name}) ->
    lookup(Name)
    >>= fun({ok, Sem}) -> return(Sem) end;
```

Variable assignments, which are represented as `match` nodes, are evaluated as infallible pattern matches.

```
eval({match, _, Pat, Expr}) ->
    eval(Expr)
    >>= fun(Sem) ->
        match(Sem, Pat)
        >>= fun(true) -> return(Sem) end
    end;
```

Pattern matching is fallible, and failed matches should not affect the variable scope. For this reason, the actual matching is done by the `match_/2` function, while

`match/2` is a wrapper around this that reverts the state if it fails. The `state/1` function is used to directly manipulate the state, also providing a value to return.

```
-spec match(sem(), expr()) -> m(boolean()).
match(Sem, Pat) ->
    get_state() >>= fun(S) ->
        match_(Sem, Pat) >>= fun
            (true) -> return(true);
            (false) -> state(fun(_) -> {S, false} end)
        end)
    end.
```

The `match_/2` function performs pattern matching. It directly matches the value of base types and recursively matches elements a tuple term.

```
match_({sem_atom, A}, {atom, _, A}) ->
    return(true);
match_({sem_int, A}, {integer, _, A}) ->
    return(true);
match_({sem_tuple, Ss}, {tuple, _, Ps}) when length(Ss) == length(Ps) ->
    all([match_(S, P) || {S, P} <- lists:zip(Ss, Ps)]);
```

Pattern matching against variables is different from most other languages: if the variable is already in scope, its value is matched, rather than being re-bound. If the variable is not in scope, it is appended to the scope.

```
match_(Sem, {var, _, V}) ->
    lookup(V) >>=, fun
        ({ok, Val}) -> return(Sem == Val);
        (error) -> state(fun(S) ->
            {S#ctx{scope = maps:put(V, Sem, S#ctx.scope)}, true}
        end)
    end;
```

Finally, any value not matched by the above rules is judged to not be a match, which is insufficient for handling unknown values, `sem_var`. We have not found any way around this limitation.

```
match_(_, _) ->
    return(false).
```

Shifting our focus back to the evaluation, `if` and `case` statements also use this pattern-matching infrastructure. `find_clause/2` scans a list of clauses — which is a common format shared for both `if`, `case`, and functions — returning the body of the first clause first whose patterns match. `eval_clause` evaluates this body, which is a list of expressions, returning only the last value.

```
eval({'if', _, Clauses}) ->
    find_clause([], Clauses)
    >>= fun(Clause) -> eval_clause(Clause) end;
eval({'case', _, Expr, Clauses}) ->
```

```
    eval(Expr)
    >>= fun(Sem) -> find_clause([Sem], Clauses) end
    >>= fun(Clause) -> eval_clause(Clause) end;
```

This is where the real `fun` begins: it is time to deal with functions. The semantics for functions is `{sem_fun, fun([sem()]) -> rere:m(sem())}`, where `rere:m()` is the underlying monad used for reification and reflection.

Most of the complexity in dealing with lambda expressions comes from handling closures. In a perfect world, all we would have to do is grab a copy of the current variable scope and have the function run inside this scope, discarding any changes once the function returns:

```
eval({'fun', _, {clauses, Clauses}}) ->
    get_state() >>= fun(S) ->
        return({sem_fun, fun(Args) ->
            run(
                S#ctx{scope = #{}},
                find_clause(Args, Clauses)
                >>= fun(Clause) -> eval_clause(Clause) end
            )
        end})
    end;
```

However, this is not the world we live in, and lambda functions in Erlang have the quirk that variables bound in the argument list shadow closure variables instead of matching against them. This means that `X = 1, X = 2` fails to match because it attempts to match `2` against the existing `1`, but `X = 1, (fun(X) -> ... end)(2)` succeeds and instead shadows the previous `X`.

The code above thus does not accurately model Erlang's semantics, and instead, we need to perform the pattern matching in an empty variable scope and merge these bindings into the closure when evaluating the function body[3]:

```
eval({'fun', _, {clauses, Clauses}}) ->
    get_state() >>= fun(S) ->
        return({sem_fun, fun(Args) ->
            run(
                S#ctx{scope = #{}},
                find_clause(Args, Clauses)
                >>= fun(Clause) ->
                    state(fun(S2) -> { S2#ctx{
                        scope = maps:merge(S#ctx.scope, S2#ctx.scope)
                    }, ok } end)
                    >>= fun(ok) -> eval_clause(Clause) end
                end
            )
```

[3]Recall that values in Erlang are immutable; the `S#ctxscope = #` produces a new value, rather than mutating it.

```
        end})
    end;
```

Calling functions is comparatively simple: all we have to do is evaluate all involved parties and then lift the resulting `rere:m()` into our state monad transformer.

```
eval({call, _, Expr, Args}) ->
    eval_list([Expr | Args]),
    >>= fun([{sem_fun, Fun} | Sems]) -> lift(Fun(Sems)) end;
```

However, calling function *expressions* is only one of several kinds of function calls in Erlang. In our evaluator, functions defined in the current module are evaluated inline, with the definitions provided as a different field of the state monad[4]. This works similarly to above, except the function is evaluated in an empty scope to prevent local variables from leaking.

```
eval({'fun', _, {function, Name, Arity}}) ->
    get_state() >>= fun(S) ->
        Fun = maps:get({Name, length(Args)}, S#ctx.local),
        eval_list(Args) >>= fun(Sems) ->
            lift(run(
                S#ctx{scope = #{}},
                eval(Fun) >>= fun(F) -> call(F, Sems) end
            ))
        end
    end;
```

Last, but definitely not least, we have remote functions. Similar to local functions, we look them up in a monad-provided dictionary, only this time, instead of recursively calling `run` and `eval`, we instead signal to the reflector that we are doing a function call.

```
eval({call, _, {remote, _, {atom, _, Module}, {atom, _, Name}}, Args}) ->
    get_state() >>= fun(S) ->
        Type = maps:get({Mod, Name, length(Args)}, S#ctx.remote),
        eval_list(Args) >>= fun(Sems) ->
            lift(rere:call(Type, {Mod, Name}, Sems))
        end
    end.
```

This `rere:call` function produces a semantic value representing a remote function call. But our semantics does not include any clause for function calls, so how does it do this? The answer lies in reflection; however, for this to make sense, we first need to introduce reification.

[4]This field is immutable and should therefore be handled via a reader monad, but we chose to go for practicality over theoretical correctness.

3.2 Reification

Once we have obtained our semantic value, as well as in certain steps when producing it, we need a way to turn these semantic values back into syntax, since the output of the normalizer is an equivalent Erlang program. This process is called *reification* — the act of making something real. It is, in a sense, the opposite of evaluation.

Our implementation, based on *type-driven partial evaluation* (TDPE), uses type specifications to provide much-needed information about what kinds of values are in variables; in particular, in function arguments and return values, since those cannot be deduced from syntax alone.

The main function for reification is, naturally, `reify`. Its purpose is, again, to convert semantic values back into a standard form that is useful outside of our normalizer. Given a type and a semantic value, it will return a sequence of expressions that would evaluate to this value.

```
-spec reify(type(), sem()) -> m([expr()]).
```

Reflection produces a list of expressions, to be able to handle sequential function calls.

Just like for evaluation, a number of simple types can be reified directly into syntax: if we know the value of an integer, for example, we insert that exact value into the abstract syntax. This includes singleton atom types, which are reified as that exact atom. In addition, we also convert unknown semantics into the corresponding variable name, regardless of type.

```
reify({atom, _, At}, {sem_atom, At}) ->
    return([ast_utils:make_at(At)]);
reify(_, {sem_var, Var}) ->
    return([Var]);
reify({type, _, integer, []}, {sem_int, Int}) ->
    return([ast_utils:make_int(Int)]);
reify({type, _, string, []}, {sem_str, Str}) ->
    return([ast_utils:make_str(Str)]);
reify({type, _, atom, []}, {sem_atom, At}) ->
    return([ast_utils:make_at(At)]);
reify({type, _, boolean, []}, {sem_atom, At}) ->
    return([ast_utils:make_at(At)]);
```

Tuples are, just like for evaluation, refied sequentially and then stored as a tuple.

```
reify({type, _, tuple, Types}, {sem_tuple, Sems}) ->
    reify_seq(Types, Sems),
    >>= fun(Exprs) -> return([ast_utils:make_tuple(Exprs)]) end;
```

There is a catch, however: `reify/2` returns a list of expressions, but tuples only take a single expression per item. For this, we have the `reify_seq/2` function, which uses the continuation monad to move any extra expressions to before the tuple. This is the first we see of the continuation monad, but we will be seeing it plenty more.

```
-spec reify_seq([tp()], [sem()]) -> m([expr()], [expr()]).
reify_seq(Ts, As) ->
    sequence(lists:zipwith(fun reify_one/2, Ts, As)).

-spec reify_one(tp(), sem()) -> m(expr(), [expr()]).
reify_one(T, A) ->
    reify(T, A) >>= fun(X) ->
        shift(fun(K) ->
            K(lists:last(X))
            >>= fun(V) -> return(lists:droplast(X) ++ V) end
        end)
    end.
```

Next, we have union types. We do not support the full versatility of union types, but we do support a sizeable subset that aligns well with conventional usage in Erlang: unions of singleton atoms (tags) and tuples with tags as their first element, in a manner that closely approximates sum types. Reifying these sum types is done by checking each variant in turn, and if the tag matches, we reify them as the type corresponding to that branch.

```
reify(
    {type, _, union, [T0 = {atom, _, Tag} | _]},
    V0 = {sem_atom, Tag}
) ->
    reify(T0, V0);
reify(
    {type, _, union, [T0 = {type, _, tuple, [{atom, _, Tag} | _]} | _]},
    V0 = {sem_tuple, [{sem_atom, Tag} | _]}
) ->
    reify(T0, V0);
reify({type, ANNO, union, [_ | Rest]}, V) ->
    reify({type, ANNO, union, Rest}, V).
```

Finally, we have functions. The goal is, of course, to turn a `sem_fun` value into a `fun(A) -> B end` expression. But unfortunately, here we have to interact with the real world — when this function is called, the arguments will be real Erlang values, not our semantic values. Thus we need to, for each incoming argument, produce a semantic value that reflects this real Erlang value.

```
reify({type, _, 'fun', [{type, _, product, Ts}, T2]}, {sem_fun, F}) ->
    reflect_seq(Ts, fun(X) ->
        F(X) >>= fun(B) -> reify(T2, B) end
    end)
    >>= fun({Vars, E}) -> return([ast_utils:make_fun(Vars, E)]) end;
```

3.3 Reflection

Accompanying `reify`, there is also a second function, `reflect`, which is used to interpret values from the outside world — that is, incoming function arguments, as well as return values from remote functions. Given a type and a variable name, it will produce a semantic value that corresponds to this variable name.

```
-spec reflect(type(), var()) -> m(sem()).
```

This type signature is somewhat unsatisfactory and limiting: we would have liked to have it for example return a pattern match expression, which would allow significantly more sophisticated pattern matching including nested tuples and sums. Alas, despite our best efforts, we were unable to produce such a function, and so we will leave this problem to the future.

Just like for evaluation and reification, there are a number of types that are trivial to reflect: we record that they are a value stored in a variable, but we don't know which.

```
reflect({atom, _, At}, _) ->
    return({sem_atom, At});
reflect({type, _, integer, []}, At) ->
    return({sem_var, At});
reflect({type, _, string, []}, At) ->
    return({sem_var, At});
reflect({type, _, atom, []}, At) ->
    return({sem_var, At});
```

Boolean values are different, however. We do not store boolean variables as `sem_var`, but instead, we use the continuation monad to evaluate the function twice; once where we assume that the argument was true, and once where it was false. If both cases produce the same result, this means that the function is independent of that argument, and we simply return that result; otherwise, we produce an if/else statement containing both cases. Thus, a function `fun(B) -> Body end` will be normalized to `fun(B) -> if B -> Body`$_B$`; true -> Body`$_{not\ B}$ `end end`[5].

```
reflect({type, _, boolean, []}, At) ->
    shift(fun(K) ->
        sequence([
            K({sem_atom, true}),
            K({sem_atom, false})
        ]) >>= fun
            ([E, E]) -> return(E);
            ([E1, E2]) -> return([ast_utils:make_if_else(At, E1, E2)])
        end
    end);
```

Tuples also have a few things worth pointing out. Due to the aforementioned unsat-

[5] To `B` or `not B`, that is the question.

isfactory function signature, we cannot have the function return a pattern match, so instead we prepend a destructuring assignment to the function body: instead of `fun({A, B}) -> Body end`, we write `fun(V1) -> {A, B} = V1, Body end`. The `prepend/2` function uses `shift_seq/1` instead of the regular `shift/1`; this is a slightly different variant that interacts differently with the state monad transformer used for variable name generation.

```
reflect({type, _, tuple, Types}, At) ->
    prepend(At, fun(K) ->
        reflect_seq(Types, fun(Sems) ->
            K({sem_tuple, Sems})
        end) >>= fun({Vars, B}) ->
            return({ast_utils:make_tuple(Vars), B})
        end
    end);

prepend(Val, F) ->
    shift_seq(fun(K) ->
        F(K) >>= fun({Pat, Rest}) ->
            return([ast_utils:make_match(Pat, Val) | Rest])
        end
    end).
```

Unions are very similar to booleans, only a little bit more powerful. If we had the desired function signature we could simply use `reflect()` recursively to produce the patterns to match against, but alas, this is not possible with the current signature. Instead, we construct the required patterns here, leading to a somewhat unnecessarily bloated function. This function is the reason why we support sums rather than unions: since we reflect these as a pattern match, we need to be able to construct a pattern that matches against each branch, and this is much easier for sums than for unions.

```
reflect({type, _, union, Types}, At) ->
    shift(fun(K) ->
        sequence(
            lists:map(
                fun ({atom, _, Tag}) ->
                        K({sem_atom, Tag}),
                        >>= fun(E) ->
                            return({
                                ast_utils:make_at(Tag),
                                E
                            })
                        end;
                    ({type, _, tuple, [{atom, _, Tag} | Types]}) ->
                        reflect_seq(Types, fun(Sems) ->
                            K({sem_tuple, [{sem_atom, Tag} | Sems]})
                        end) >>= fun({Vars, E}) ->
```

```
                                return({
                                    ast_utils:make_tuple([
                                        ast_utils:make_at(Tag)
                                        | Vars
                                    ]),
                                    E
                                })
                            end
                    end,
                    Types
                )
            ) >>= fun(Clauses) ->
                return([ast_utils:make_case(At, Clauses)])
            end
        end);
```

And now we have once again arrived at functions. This one is very short since most of the functionality is abstracted away inside a different function.

```
reflect(Type = {type, _, 'fun', _}, At) ->
    return({sem_fun, fun(Sems) -> call(Type, At, Sems) end});
```

This `call()` function happens to be the very same as the one encountered at the end of section 3.1. We saw how reifying a function requires reflecting the arguments since they come from the outside world; in this case, since we are exposing values to the outside world, we need to reify them and reflect the return value back into semantics. Since function calls can have side effects, we cannot discard or duplicate them regardless of how their return values are used, so we instead bind the values to variables, similarly to how we do for reflecting tuples.

```
-spec call(tp(), atom(), [sem()]) -> m(sem(), [expr()]).
call({type, _, 'fun', [{type, _, product, Ts}, T2]}, Name, Sems) ->
    reify_seq(Ts, Sems)
    >>= fun(Vs) ->
        prepend(
            ast_utils:make_call(Name, Vs),
            fun(K) ->
                new_var()
                >>= fun(Var) ->
                    reset(reflect(T2, Var) >>= K)
                    >>= fun(V) -> return({Var, V}) end
                end
            end
        )
    end.
```

3.4 Normalization

At long last, we have all the components we need to normalize a function: it's only a matter of chaining them together in the right order. Specifically, we run the evaluator with an initial state containing all local and remote functions, which gets us a `rere:m(sem())`. We then reify this semantic value and execute the continuation monad, getting us a list of expressions. We know that reifying a function will always give only one expression, so we take that single expression and return it.

```
-spec norm(tp(), expr(), eval:ctx()) -> expr().
norm(Type, Expr, Context) ->
    [A] = rere:run(
        eval:run(Context, eval:eval(Expr)),
        >>= fun(Sem) -> rere:reify(Type, Sem) end
    ),
    A.
```

Normalizing a function is theoretically interesting in itself, but it is not very useful to end users. To be more practically useful, we also include a function for normalizing a whole Erlang module in one go. First, we collect all function declarations and `-spec` attributes, in order to build the evaluation context. The abstract syntax for function declarations is slightly different than for function *expressions*, so rather than making the evaluator handle function declarations, we convert each function into an equivalent `fun` expression.

```
collect_module([], C) ->
    C;
collect_module([{attribute, _, spec, {{Name, Arity}, [Ft]}} | Forms], C) ->
    V = maps:put({Name, Arity}, Ft,
                 C#collect.local_type),
    collect_module(Forms, C#collect{local_type = V});
collect_module([{attribute, _, spec, {{Mod, Name, Arity}, [Ft]}} | Forms], C) ->
    V = maps:put({Mod, Name, Arity}, Ft,
                 C#collect.remote_type),
    collect_module(Forms, C#collect{remote_type = V});
collect_module([{function, _, Name, Arity, _Clauses} = F | Forms], C) ->
    V = maps:put({Name, Arity}, ast_utils:function2fun(F),
                 C#collect.local_body),
    collect_module(Forms, C#collect{local_body = V});
collect_module([_ | Forms], C) ->
    collect_module(Forms, C).
```

After this, it traverses the list of forms a second time: functions with accompanying type specifications are normalized, those without are assumed to be internal implementation details and are removed, and all other forms are kept unchanged.

```
-spec norm_module([erl_parse:abstract_form()]) -> [erl_parse:abstract_form()].
norm_module(Forms) ->
    C = collect_module(Forms, #collect{}),
```

```
    Ctx = eval:ctx(C#collect.local_body, C#collect.remote_type),
    lists:filtermap(
        fun ({function, _, Name, Arity, _} = F) ->
                case maps:find({Name, Arity}, C#collect.local_type) of
                    {ok, Ty} ->
                        Norm = nbe:norm(Ty, ast_utils:function2fun(F), Ctx),
                        {true, ast_utils:fun2function(Name, Arity, Norm)};
                    error ->
                        false
                end;
            (_) ->
                true
        end,
        Forms
    ).
```

4

Results

We have developed a normalizer covering a sizeable fragment of Erlang, which can be found online on GitHub[1]. In this section, we present the normalization results of some non-trivial programs as examples. Since our normalizer assumes the source program is well-annotated, it is required that type signatures are correctly specified in the top definitions of the source code.

We will also introduce the notion of *partial evaluation* and *type-directed partial evaluation* in the last example, where our normalizer deals with a recursive function and *partially* evaluates the `pow` function over a known exponent.

4.1 Example: Conditionals

We start with a simple example where a function `f` has a function and a boolean parameter.

```
-spec f(fun((integer()) -> integer()), boolean()) -> integer().
f(F, B) ->
    V = if  B -> 8;
           true -> 2
        end,
    F(V).
```

As described in section 3.3, the normalized function creates a branch on the boolean parameter.

```
f(V1, V2) ->
    if  V2 ->
            V3 = V1(8),
            V3;
        true ->
            V4 = V1(2),
            V4
    end.
```

[1] https://github.com/haohanyang/nbe_erlang

4.2 Example: Monadic refactoring

In the next example, we want to sequentially call a number of fallible remote functions and collect the results when all of them have succeeded. If any of them fails, the return value should indicate the error. Our first attempt is to nest `case` clauses.

```
-spec a:get() -> {ok, integer()} | err.
-spec b:get() -> {ok, integer()} | err.
-spec c:get() -> {ok, integer()} | err.

-spec main() -> {ok, {integer(), integer(), integer()}} | err.
main() ->
    case a:get() of
        {ok, A} ->
            case b:get() of
                {ok, B} ->
                    case c:get() of
                        {ok, C} -> {ok, {A, B, C}};
                        err -> err
                    end;
                err -> err
            end;
        err -> err
    end.
```

The program's structure becomes verbose as the nesting goes deeper. A natural simplication would be to refactor into a monadic style.

```
main_monadic() ->
    bind(a:get(), fun(A) ->
        bind(b:get(), fun(B) ->
            bind(c:get(), fun(C) ->
                return({A, B, C})
            end)
        end)
    end).

return(V) -> {ok, V}.

bind({ok, V}, F) -> F(V);
bind(err, _) -> err.
```

Such refactorings can easily introduce subtle bugs. However, the validity of such refactoring can be tested by comparing the normal forms of the original and refactored program. In this case, the normalized forms of `main` and `main_monadic` are observably equivalent, meaning that the refactoring does not affect any semantics.

```
main() ->
    V1 = a:get(),
```

```
    case V1 of
        {ok, V2} ->
            V3 = b:get(),
            case V3 of
                {ok, V4} ->
                    V5 = c:get(),
                    case V5 of
                        {ok, V6} -> {ok, {V2, V4, V6}};
                        err -> err
                    end;
                err -> err
            end;
        err -> err
    end.
```

4.3 Example: Side effects

The next example shows how our normalizer takes side effects into account. In addition, it shows how lambda expressions, including higher-order ones, are optimized away.

```
-spec io:format(string()) -> ok.

-spec printing(string()) -> integer().
printing(S) ->
    F = fun() -> io:format(S) end,
    G = fun(X) -> {X("C"), X("D")} end,
    H = fun(A, B) -> B end,
    io:format("A"),
    F(),
    io:format("B"),
    G(fun(X) -> io:format(X), F() end),
    H(io:format("E"), 4).
```

This example is a bit convoluted, but it shows that side effects are neither duplicated, eliminated, or reordered. The return values are all assigned to variables, which is somewhat unnecessary, but doing so avoids having to test if the variable is actually used, simplifying the implementation.

```
printing(V1) ->
    V2 = io:format("A"),
    V3 = io:format(V1),
    V4 = io:format("B"),
    V5 = io:format("C"),
    V6 = io:format(V1),
    V7 = io:format("D"),
    V8 = io:format(V1),
```

```
    V9 = io:format("E"),
    4.
```

4.4 Example: Recursion

Our normalizer also supports limited forms of recursion, here shown in the form of the Fibonnaci function. We cannot yet normalize recursive functions directly, but we can normalize functions that *call* recursive functions.

```
-spec fib_example() -> integer().
fib_example() -> fib(9).

fib(1) -> 1;
fib(2) -> 1;
fib(X) -> fib(X - 2) + fib(X - 1).
```

This will, of course, be evaluated to the number 34. Note that the actual recursive function was removed: this is because it does not have a type signature, and is thus deemed auxiliary.

```
-spec fib_example() -> integer().
fib_example() -> 34.
```

4.5 Partial evaluation

Partial evaluation, sometimes called *program specialization* or *program residualization*[2], is a program optimization technique to eliminate or pre-compute deductible computations in a program. The chief motivation of partial evaluation is speed[13]. Given a program of two inputs $P : \tau_1 \times \tau_2 \to \tau_3$ and a static input argument $p_1 : \tau_1$, a *partial evaluator* PE yields a more efficient residual program $PE(P, p_1) = \|P\|_1 : \tau_2 \to \tau_3$, if the static input p_1 is known. Calling the specialized program with the remaining input p_2 produces the same result as calling the original program P with both inputs p_1 and p_2, i.e., $P(p_1, p_2) = \|P\|_1(p_2)$. The partial evaluator simplifies the program by executing operations that only depend on known inputs.

A classic example is the partial evaluation of *pow* function, as defined below. If the exponent `N` is known to be 4, the program can be partially evaluated into `pow_4(X) -> X * X * X * X`, eliminating the overhead of recursion in further execution. If only the base parameter `X` is known, not many simplifications can be performed, since the function only branches on `N`.

Here, we show again that our normalizer has a limited capability to deal with recursive functions. The result is a partial evaluation of the following `pow` function with a known `N`, which is just the normalization of `pow_example`.

[2]Other related concepts are *constant propagation* and *inlining*; both of which can be seen as special cases of partial evaluation.

```
-spec pow_example(integer()) -> integer().
pow_example(A) -> pow(A, 4).

pow(X, 0) -> 1;
pow(X, N) -> X * pow(X, N - 1).

-spec pow_example_norm(integer()) -> integer().
pow_example_norm(V1) ->
    V2 = V1 * V1,
    V3 = V1 * V2,
    V4 = V1 * V3,
    V4.
```

As expected, the function is normalized to a number of multiplications, which is equivalent to `X * X * X * X`. The program's structure is similar to the `io:format` example, where multiplications are split into several separate clauses. This is because multiplication can technically have side effects, in terms of errors on invalid inputs[3].

Note that the normalization will introduce the issue of non-termination if the given input is `X` instead, although this problem could be addressed using fix-point operators[5, 8]. Furthermore, our normalization differs from traditional *type-directed partial evaluation*, which normally involves *binding-time annotations*[5] that we do not have. We will leave the relevant work for the future.

[3]This does not matter for multiplication since we assume that inputs are type-correct, but for division, reordering operations could make certain functions more or less prone to errors on division by zero.

5

Conclusion

We have implemented a type-directed normalizer that handles a sizable fragment of Erlang syntax. We thus are able to reason the convertibility between two Erlang programs by comparing their corresponding normal forms. This result also contributes to the partial evaluation of Erlang in a type-driven manner. It is somewhat surprising that NbE is not limited to its original setting for pure lambda calculus and can be adapted to handle more practical programming languages. Those extensions seem to reflect a more general pattern in denotational semantics. It also enriches the landscape of partial evaluation, by introducing a type-directed approach that is fairly different from the traditional, syntax-based methods.

5.1 Limitations and Future Work

The primary limitation of our normalizer is, of course, that it only handles a small fragment of Erlang; in particular, no concurrency (via `!` and `receive`) is supported — the reason being that, unlike remote functions, there is no easy way to assign type information to these primitives. Enabling this would likely require some sort of type annotation at every use site in order to reify and reflect the values, respectively, which would be rather unidiomatic and inconvenient, and would require significantly more invasive changes to the code to be normalized than just adding top-level type specifications. Thus supporting these concurrency primitives would require either sophisticated type inference — which is far beyond the scope of this project, but prior works exist in this field — or alleviating the need for types altogether, which would significantly alter the basic nature of the project.

Another significant limitation is in pattern matching. As discussed in section 3.3, supporting more sophisticated patterns would require changing reflection to return a pattern, rather than receiving a variable name. We have attempted this several times, unsuccessfully: implementing this correctly seems to require complex use of continuations, but we have not been able to deduce the correct strategy.

Our support for recursion is very rudimentary: we can recurse over purely static terms, but recursing over a dynamic term will cause the evaluator to diverge. Enabling this would likely require keeping track of the call stack in some way, and changing strategies if encountering a call where the static parts are identical to something earlier.

Finally, the strategy used to collect type signatures is very rudimentary. Supporting `-type` aliases would go a long way toward improving ergonomics, but it is fairly complicated, especially with generic types. As this is orthogonal to the main part of this project, and would only improve ergonomics rather than functionality, we did not think this was a very pressing concern.

The major work left is to enrich Erlang semantics that encompasses more concurrent operations of Erlang. We also want our work to benefit from (but not dependent on) type analyzers. The work also includes a further step towards type-directed partial evaluation that involves binding-time annotations. Recursion over dynamic terms would also be a significant benefit.

5.2 Reflections

Many of the largest difficulties in this project lie in Erlang's lack of static types. The first is that this makes it difficult to formalize the exact semantics of Erlang, and due to being *type-driven* NbE, types are rather important — hence the difficulties with the aforementioned concurrency primitives.

The second problem is that type specifications in Erlang are only advisory, which means that our normalized functions are only necessarily equivalent to the original for correctly typed inputs. Passing an incorrect input to a function is of course an error, but in certain cases, clients may depend on the exact behavior of such faulty inputs.

In addition, the lack of static types made implementing the program, especially the continuation-heavy parts, tedious and error-prone. In several cases we resorted to implementing parts of the required functionality in a statically typed language such as Haskell or OCaml before translating it to Erlang.

This means that type-driven NbE might not be the best choice for a dynamically typed language such as Erlang. It might be interesting to try to build a NbE system which makes use of how values are *used*, rather than declared. This would need a significantly more complicated implementation of `case` expressions and pattern matching, but would possibly not require a reflection step at all, since there is not much the reflection can do without a type specification. It is however difficult to foresee what other issues such a normalizer would run into.

Despite this, we believe that we have shown that NbE, although developed for call-by-name lambda calculus, is a viable choice for call-by-value languages and works reasonably well with very modest type specifications only on top-level functions.

Bibliography

[1] Joe Armstrong. "A history of Erlang". In: *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*. Ed. by Barbara G. Ryder and Brent Hailpern. ACM, 2007, pp. 1–26. DOI: 10.1145/1238844.1238850. URL: https://doi.org/10.1145/1238844.1238850.

[2] Kenichi Asai and Oleg Kiselyov. *Introduction to Programming with Shift and Reset*. 2011. URL: http://pllab.is.ocha.ac.jp/~asai/cw2011tutorial/main-e.pdf (visited on 06/10/2023).

[3] Ulrich Berger and Helmut Schwichtenberg. "An Inverse of the Evaluation Functional for Typed lambda-calculus". In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 1991, pp. 203–211. DOI: 10.1109/LICS.1991.151645. URL: https://doi.org/10.1109/LICS.1991.151645.

[4] Thierry Coquand and Peter Dybjer. "Intuitionistic Model Constructions and Normalization Proofs". In: *Math. Struct. Comput. Sci.* 7.1 (1997), pp. 75–94. DOI: 10.1017/S0960129596002150. URL: https://doi.org/10.1017/S0960129596002150.

[5] Olivier Danvy. "Type-Directed Partial Evaluation". In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. Ed. by Hans-Juergen Boehm and Guy L. Steele Jr. ACM Press, 1996, pp. 242–257. DOI: 10.1145/237721.237784. URL: https://doi.org/10.1145/237721.237784.

[6] Olivier Danvy and Andrzej Filinski. "Abstracting Control". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. Ed. by Gilles Kahn. ACM, 1990, pp. 151–160. DOI: 10.1145/91556.91622. URL: https://doi.org/10.1145/91556.91622.

[7] Olivier Danvy, Chantal Keller, and Matthias Puech. "Typeful Normalization by Evaluation". In: *20th International Conference on Types for Proofs and Programs, TYPES 2014, May 12-15, 2014, Paris, France*. Ed. by Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau. Vol. 39. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014, pp. 72–88. DOI: 10.4230/LIPIcs.TYPES.2014.72. URL: https://doi.org/10.4230/LIPIcs.TYPES.2014.72.

[8] Peter Dybjer and Andrzej Filinski. "Normalization and Partial Evaluation". In: *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*. Ed. by Gilles Barthe et al.

Vol. 2395. Lecture Notes in Computer Science. Springer, 2000, pp. 137–192. DOI: 10.1007/3-540-45699-6_4. URL: https://doi.org/10.1007/3-540-45699-6_4.

[9] Andrzej Filinski. "A Semantic Account of Type-Directed Partial Evaluation". In: *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*. Ed. by Gopalan Nadathur. Vol. 1702. Lecture Notes in Computer Science. Springer, 1999, pp. 378–395. DOI: 10.1007/10704567_23. URL: https://doi.org/10.1007/10704567_23.

[10] Andrzej Filinski. "From Normalization-by-Evaluation to Type-Directed Partial Evaluation". In: *1998 APPSEM Workshop on Normalization by Evaluation*. Ed. by Olivier Danvy and Peter Dybjer. Basic Research in Computer Science, 1999.

[11] Andrzej Filinski. "Normalization by Evaluation for the Computational Lambda-Calculus". In: *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*. Ed. by Samson Abramsky. Vol. 2044. Lecture Notes in Computer Science. Springer, 2001, pp. 151–165. DOI: 10.1007/3-540-45413-6_15. URL: https://doi.org/10.1007/3-540-45413-6_15.

[12] Andrzej Filinski. "Representing Monads". In: *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*. Ed. by Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin. ACM Press, 1994, pp. 446–457. DOI: 10.1145/174675.178047. URL: https://doi.org/10.1145/174675.178047.

[13] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993. ISBN: 978-0-13-020249-9.

[14] Per Martin-Löf. "About Models for Intuitionistic Type Theories and the Notion of Definitional Equality". In: *Studies in logic and the foundations of mathematics* 82 (1975), pp. 81–109.

[15] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Studies in logic and the foundations of mathematics* 80 (1975), pp. 73–118.

[16] Eugenio Moggi. "Computational Lambda-Calculus and Monads". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 1989, pp. 14–23. DOI: 10.1109/LICS.1989.39155. URL: https://doi.org/10.1109/LICS.1989.39155.