PUTTING **REASON** BACK INTO **HANDLERS**

EZ2USE

PUTTING **REASON** BACK INTO **HANDLERS**

EZ2USE

PUTTING **REASON** BACK INTO **HANDLERS**

now with up to 37% shorter proofs*

*Findings based on a survey of 2 proofs conducted today

EZ2USE

90%

PUTTING **REASON** BACK INTO **HANDLERS**

now with up to 37% shorter* proofs

*Findings based on a survey of 2 proofs conducted today every **computation**

either

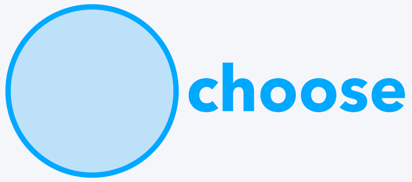
returns a **value**

or

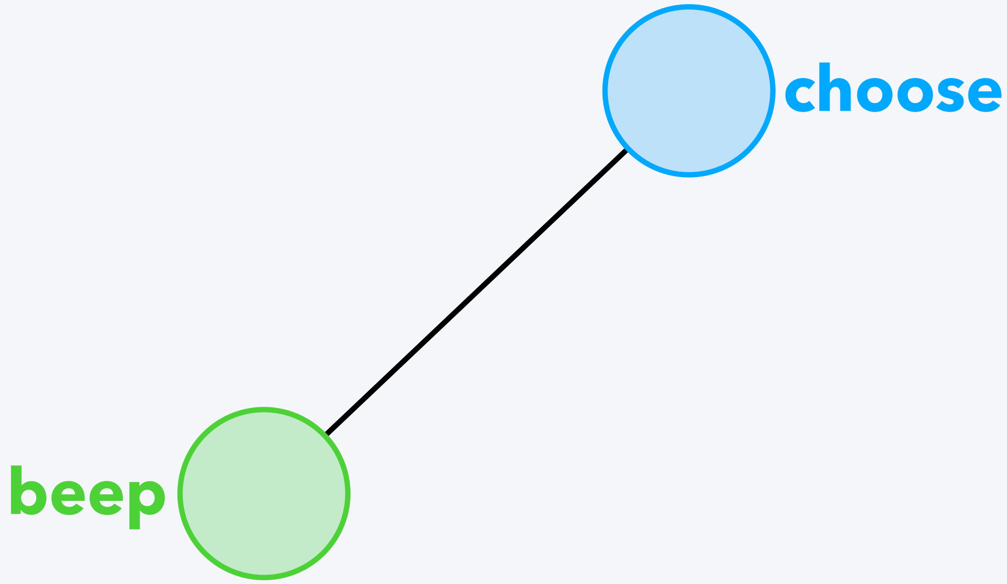
calls an **operation**

```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```

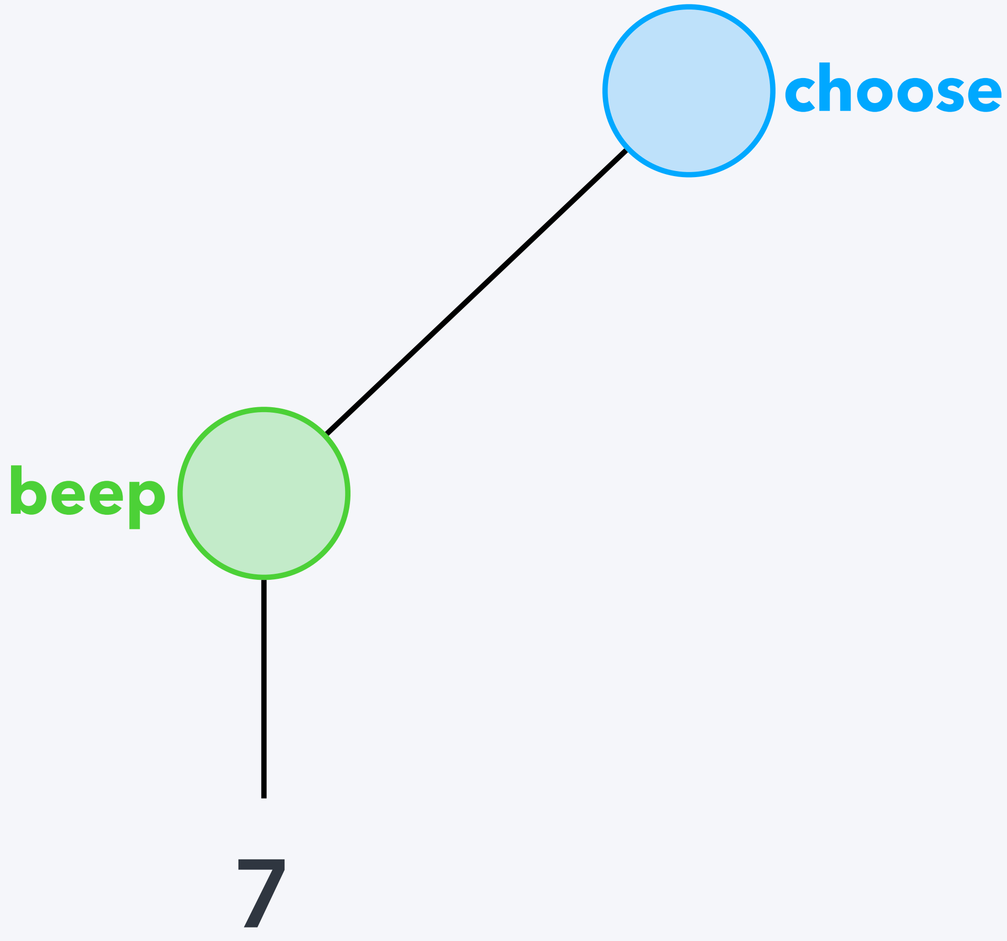
```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



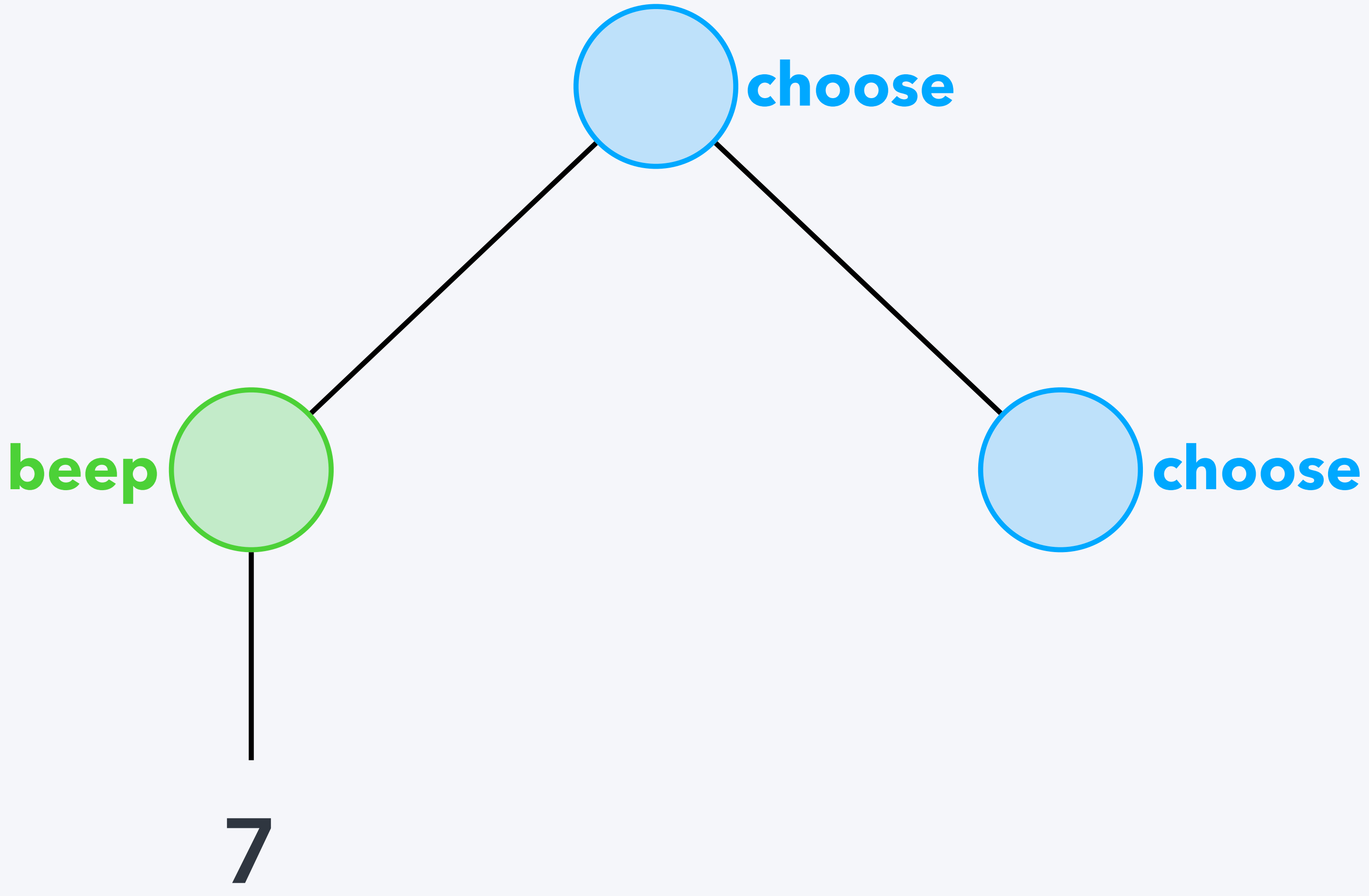
```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



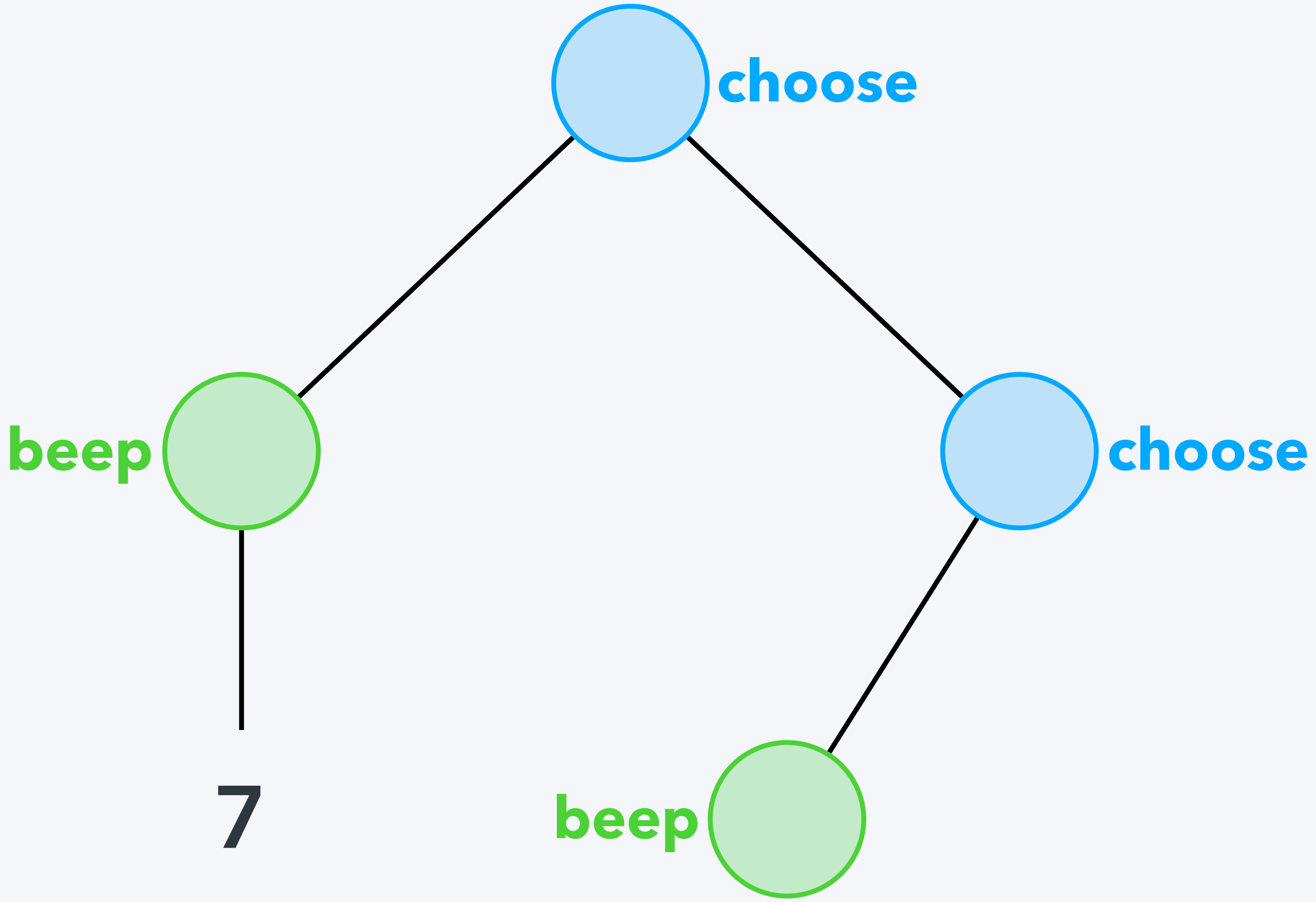
```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



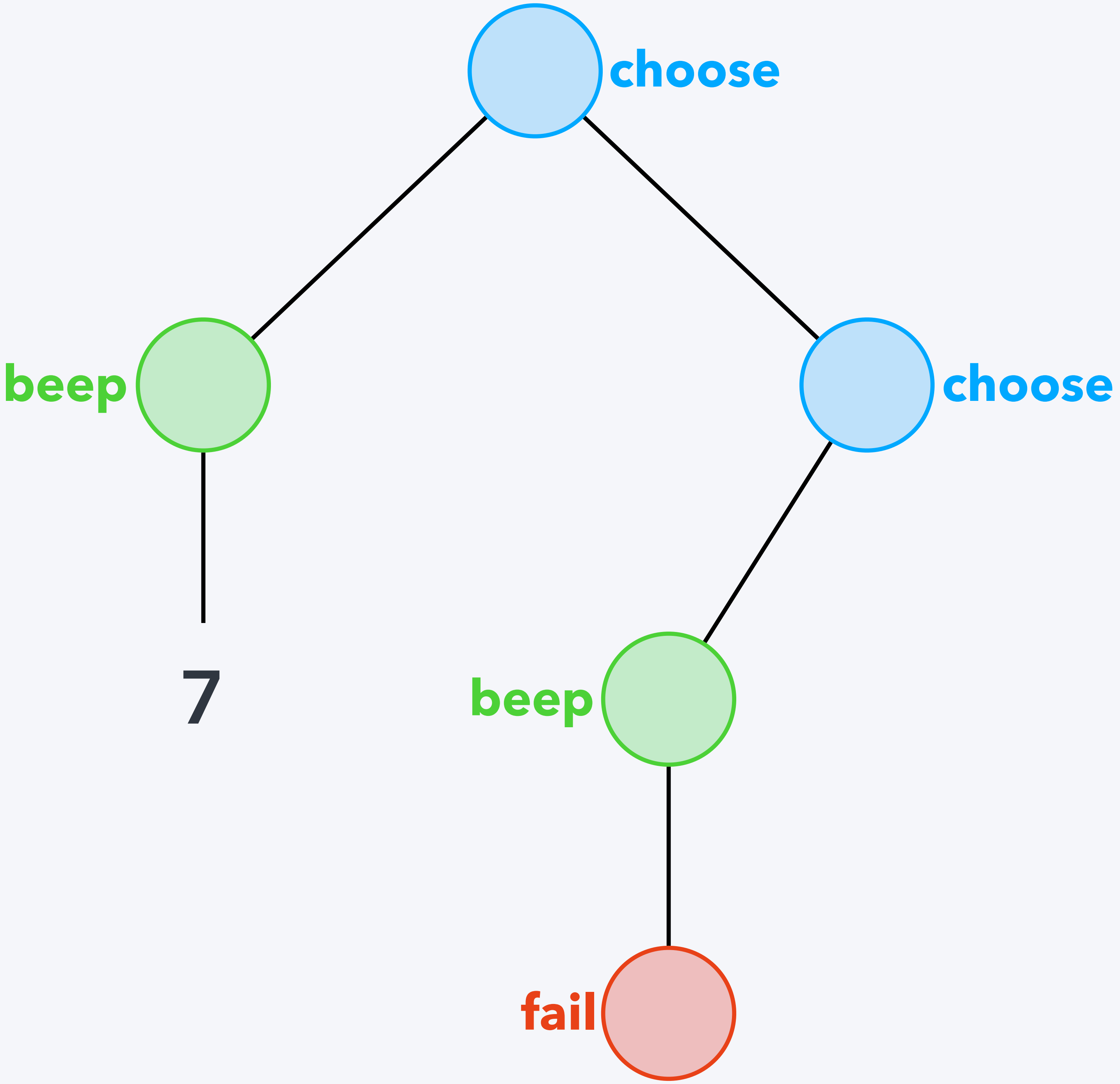
```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



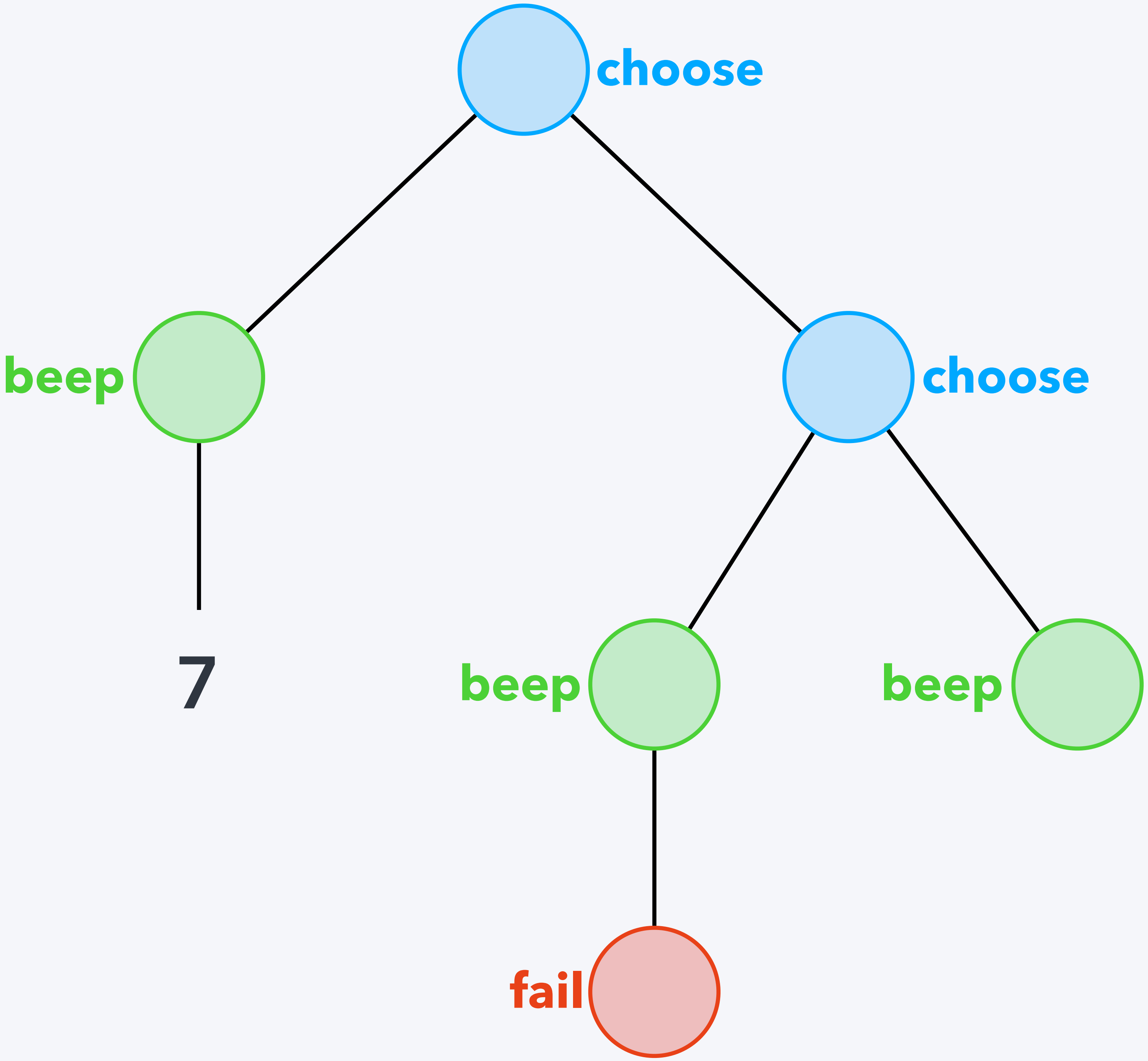
```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



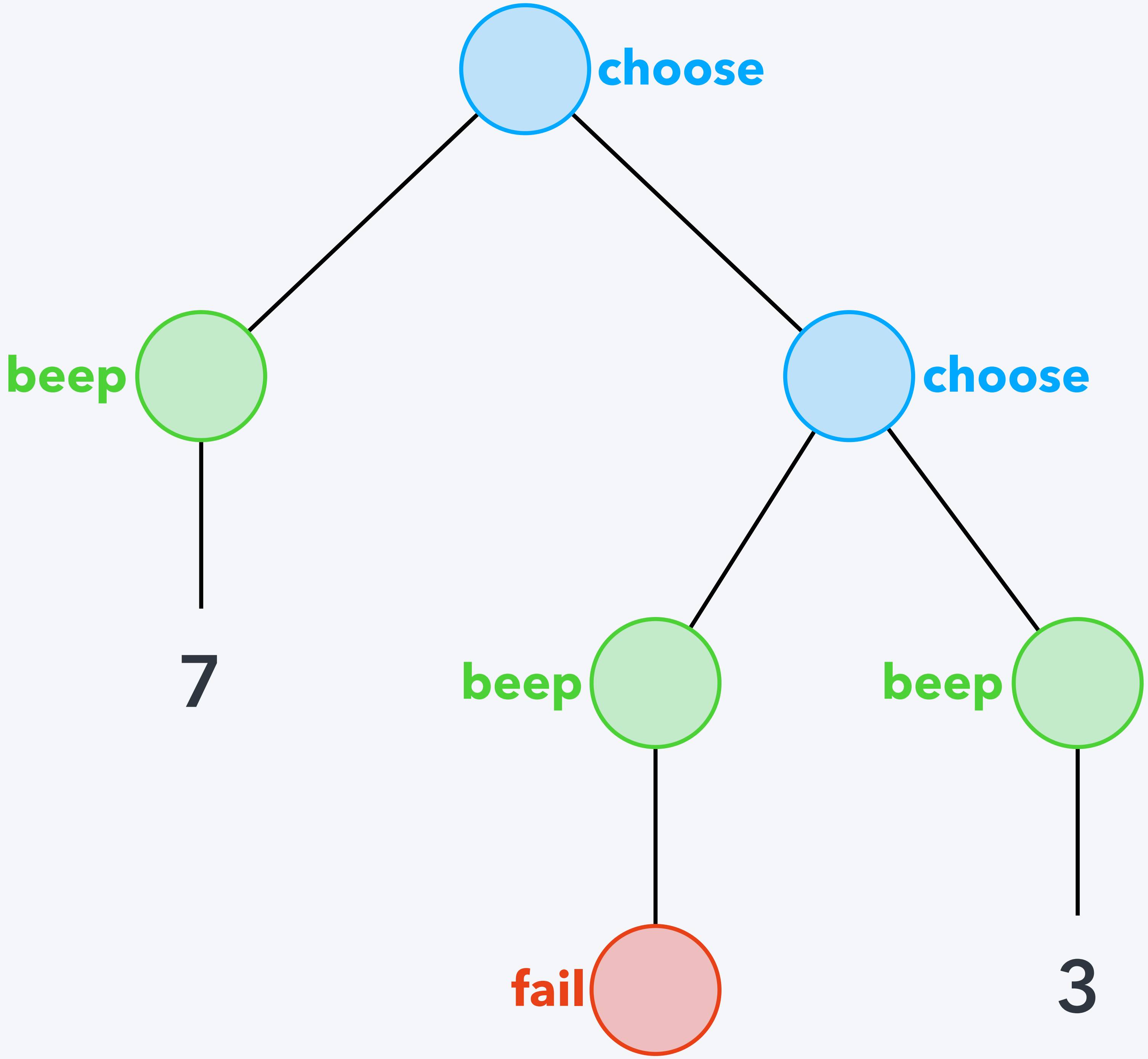
```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



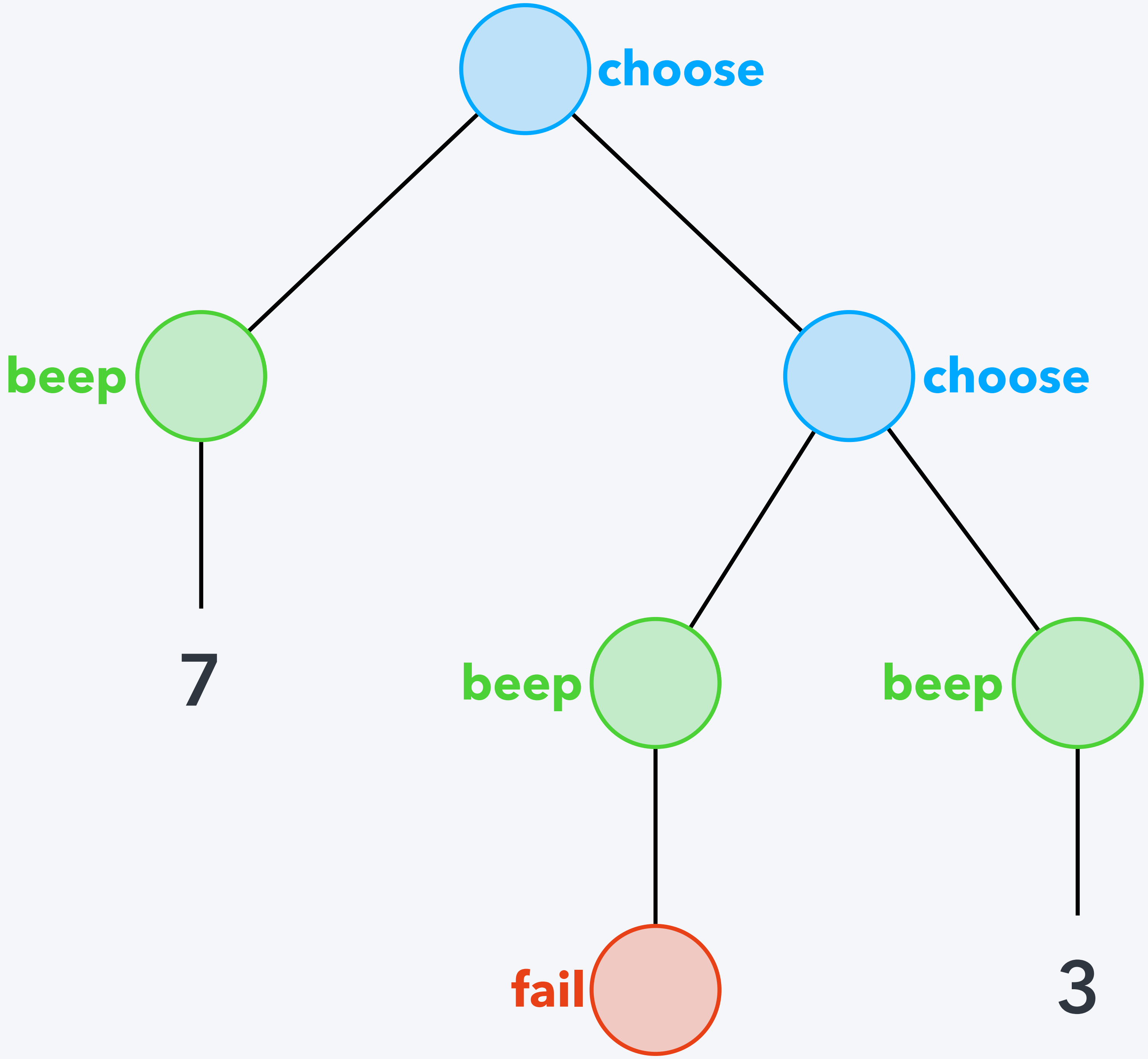
```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



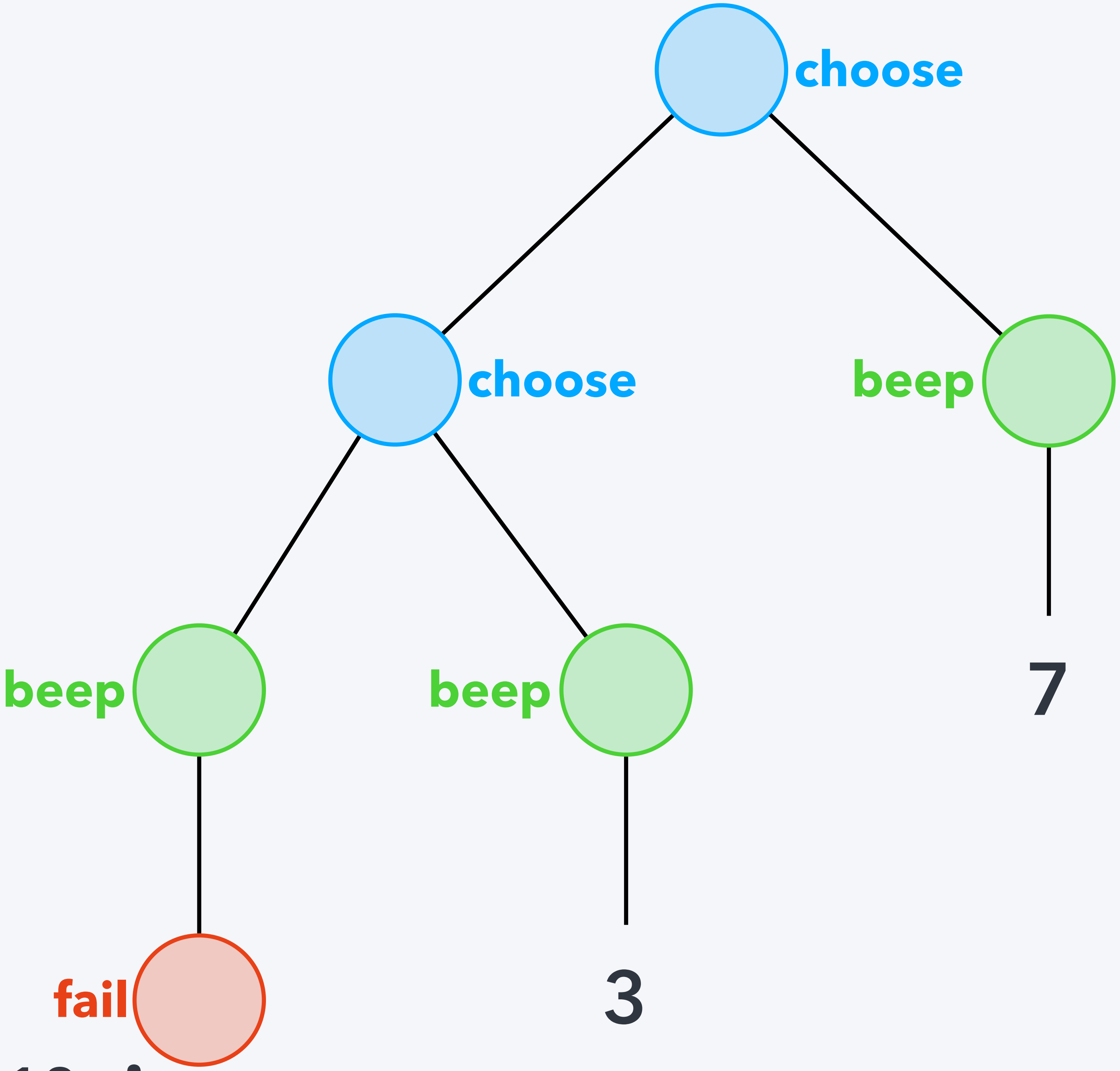
```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```

signature

$$\oplus : 2 \qquad \mathrm{beep} : 1 \qquad \mathrm{fail} : 0$$

$$z \oplus z = z$$

$$z_1 \oplus z_2 = z_2 \oplus z_1$$

$$(z_1 \oplus z_2) \oplus z_3 = z_1 \oplus (z_2 \oplus z_3)$$

$$\mathrm{beep}(z_1) \oplus \mathrm{beep}(z_2) = \mathrm{beep}(z_1 \oplus z_2)$$

$$\mathrm{fail}() \oplus \mathrm{fail}() = \mathrm{fail}()$$

$$\oplus : 2 \qquad \mathrm{beep} : 1 \qquad \mathrm{fail} : 0$$

$$z \oplus z = z$$

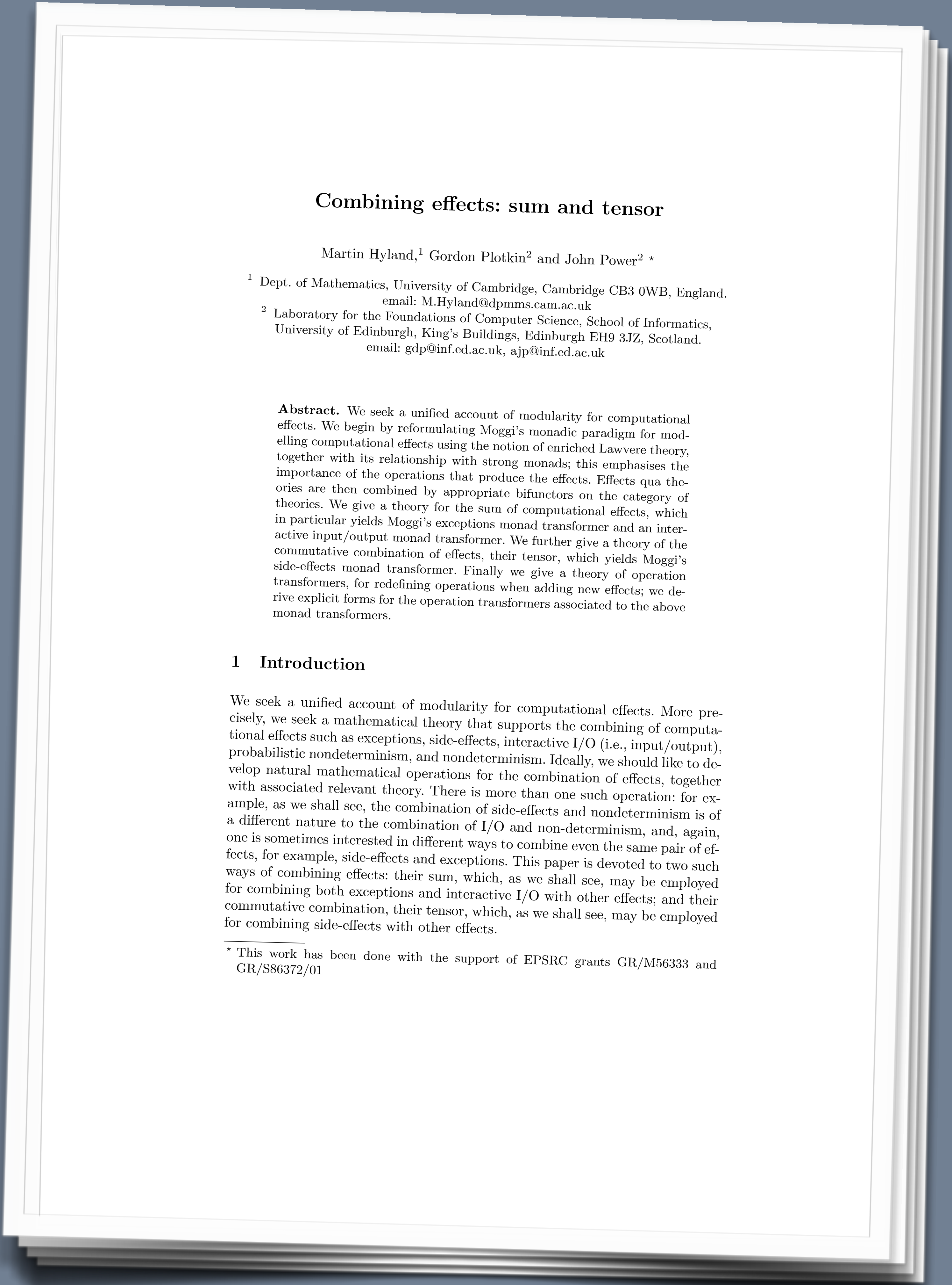
$$z_1 \oplus z_2 = z_2 \oplus z_1$$

$$(z_1 \oplus z_2) \oplus z_3 = z_1 \oplus (z_2 \oplus z_3)$$

$$\mathrm{beep}(z_1) \oplus \mathrm{beep}(z_2) = \mathrm{beep}(z_1 \oplus z_2)$$

$$\mathrm{fail}() \oplus \mathrm{fail}() = \mathrm{fail}()$$

equational theory



Combining effects: sum and tensor

Martin Hyland,[1] Gordon Plotkin[2] and John Power[2] *

[1] Dept. of Mathematics, University of Cambridge, Cambridge CB3 0WB, England.
email: M.Hyland@dpmms.cam.ac.uk

[2] Laboratory for the Foundations of Computer Science, School of Informatics,
University of Edinburgh, King's Buildings, Edinburgh EH9 3JZ, Scotland.
email: gdp@inf.ed.ac.uk, ajp@inf.ed.ac.uk

Abstract. We seek a unified account of modularity for computational effects. We begin by reformulating Moggi's monadic paradigm for modelling computational effects using the notion of enriched Lawvere theory, together with its relationship with strong monads; this emphasises the importance of the operations that produce the effects. Effects qua theories are then combined by appropriate bifunctors on the category of theories. We give a theory for the sum of computational effects, which in particular yields Moggi's exceptions monad transformer and an interactive input/output monad transformer. We further give a theory of the commutative combination of effects, their tensor, which yields Moggi's side-effects monad transformer. Finally we give a theory of operation transformers, for redefining operations when adding new effects; we derive explicit forms for the operation transformers associated to the above monad transformers.

1 Introduction

We seek a unified account of modularity for computational effects. More precisely, we seek a mathematical theory that supports the combining of computational effects such as exceptions, side-effects, interactive I/O (i.e., input/output), probabilistic nondeterminism, and nondeterminism. Ideally, we should like to develop natural mathematical operations for the combination of effects, together with associated relevant theory. There is more than one such operation: for example, as we shall see, the combination of side-effects and nondeterminism is of a different nature to the combination of I/O and non-determinism, and, again, one is sometimes interested in different ways to combine even the same pair of effects, for example, side-effects and exceptions. This paper is devoted to two such ways of combining effects: their sum, which, as we shall see, may be employed for combining both exceptions and interactive I/O with other effects; and their commutative combination, their tensor, which, as we shall see, may be employed for combining side-effects with other effects.

* This work has been done with the support of EPSRC grants GR/M56333 and GR/S86372/01

signature

$$\Sigma ::= \{\mathrm{op}_1 : k_1, \ldots, \mathrm{op}_n : k_n\}$$

$$T ::= z \mid \mathrm{op}(T_1, \ldots, T_n)$$

$$\mathscr{E} ::= \{T_1 = T'_1, \;\ldots\;, \; T_n = T'_n\}$$

equational theory

signature

$$\Sigma ::= \{\mathrm{op}_i : k_i\}_i$$

$$T ::= z \mid \mathrm{op}(T_i)_i$$

$$\mathscr{E} ::= \left\{T_i = T'_i\right\}_i$$

equational theory

values

$$v ::= x \mid () \mid \mathbf{fun}\ x \mapsto c$$

$$c ::= \mathbf{ret}\ v \mid \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 \mid v_1\, v_2 \mid \mathrm{op}(c_i)_i$$

computations

$$A ::= \mathtt{unit} \mid A \to \underline{C}$$

$$\underline{C} ::= A\,!\,\{\mathrm{op}_i\}_i$$

$$v ::= x \mid () \mid \mathbf{fun}\ x \mapsto c$$

$$\begin{aligned} c ::= {} & \mathbf{ret}\ v \mid \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 \mid v_1\, v_2 \mid \\ & \mathrm{op}(c_i)_i \end{aligned}$$

value types

$$A ::= \mathtt{unit} \mid A \to \underline{C}$$

$$\underline{C} ::= A\,!\,\{\mathrm{op}_i\}_i$$

computation types

value typing

$$x_1 : A_1, \ldots, x_n : A_n \vdash v : A$$

$$x_1 : A_1, \ldots, x_n : A_n \vdash c : \underline{C}$$

computation typing

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash () : \texttt{unit}}$$

$$\frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \mathbf{fun}\; x \mapsto c : A \to \underline{C}}$$

$$\frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\, v_2 : \underline{C}}$$

$$\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{ret}\; v : A \,!\, \Delta}$$

$$\frac{\Gamma \vdash c_1 : A \,!\, \Delta \qquad \Gamma, x : A \vdash c_2 : B \,!\, \Delta}{\Gamma \vdash \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 : B \,!\, \Delta}$$

$$\frac{\left[\Gamma \vdash c_i : A \,!\, \Delta\right]_i \qquad \mathrm{op} \in \Delta}{\Gamma \vdash \mathrm{op}(c_i)_i : A \,!\, \Delta}$$

$$\Gamma \vdash v =_A v'$$

$$\Gamma \vdash c =_{\underline{C}} c'$$

standard congruence equations

$$\frac{(T = T') \in \mathscr{E}_{\text{global}} \qquad [c_i : \underline{C}]_i}{T[c_i/z_i]_i =_{\underline{C}} T'[c_i/z_i]_i}$$

$$\begin{aligned} x &=_{\texttt{unit}} () \\ (\mathbf{fun}\, x \mapsto c)\, v &=_{\underline{C}} c[v/x] \\ \mathbf{fun}\, x \mapsto v\, x &=_{A \to \underline{C}} v \end{aligned}$$

$$\mathbf{do}\ x \leftarrow \mathbf{ret}\ v\ \mathbf{in}\ c \ =_{\underline{C}}\ c[v/x]$$

$$\mathbf{do}\ x \leftarrow \mathrm{op}(c_i)_i\ \mathbf{in}\ c$$
$$=_{\underline{C}}$$
$$\mathrm{op}(\mathbf{do}\ x \leftarrow c_i\ \mathbf{in}\ c)_i$$

$$\mathbf{do}\; x \leftarrow c \;\mathbf{in}\; \mathbf{ret}\; x \overset{?}{=} c$$

$$\mathbf{do}\; x_1 \leftarrow c_1 \;\mathbf{in}\; (\mathbf{do}\; x_2 \leftarrow c_2 \;\mathbf{in}\; c)$$

$$\overset{?}{=}$$

$$\mathbf{do}\; x_2 \leftarrow (\mathbf{do}\; x_1 \leftarrow c_1 \;\mathbf{in}\; c_2) \;\mathbf{in}\; c$$

$$\frac{\begin{array}{c}\forall v : \boldsymbol{A}.\ \phi(\mathbf{ret}\ v) \\[1em] \left[\forall c_i : \boldsymbol{A} \,!\, \Delta.\ \bigwedge_i \phi(c_i) \Rightarrow \phi(\mathrm{op}(c_i)_i\right]_{\mathrm{op}\in\Delta}\end{array}}{\forall c : \boldsymbol{A} \,!\, \Delta.\ \phi(c)}$$

base case

$$\forall v : A.\ \phi(\mathbf{ret}\ v)$$

$$\frac{\left[\forall c_i : A \,!\, \Delta.\ \bigwedge_i \phi(c_i) \Rightarrow \phi(\mathrm{op}(c_i)_i)\right]_{\mathrm{op} \in \Delta}}{\forall c : A \,!\, \Delta.\ \phi(c)}$$

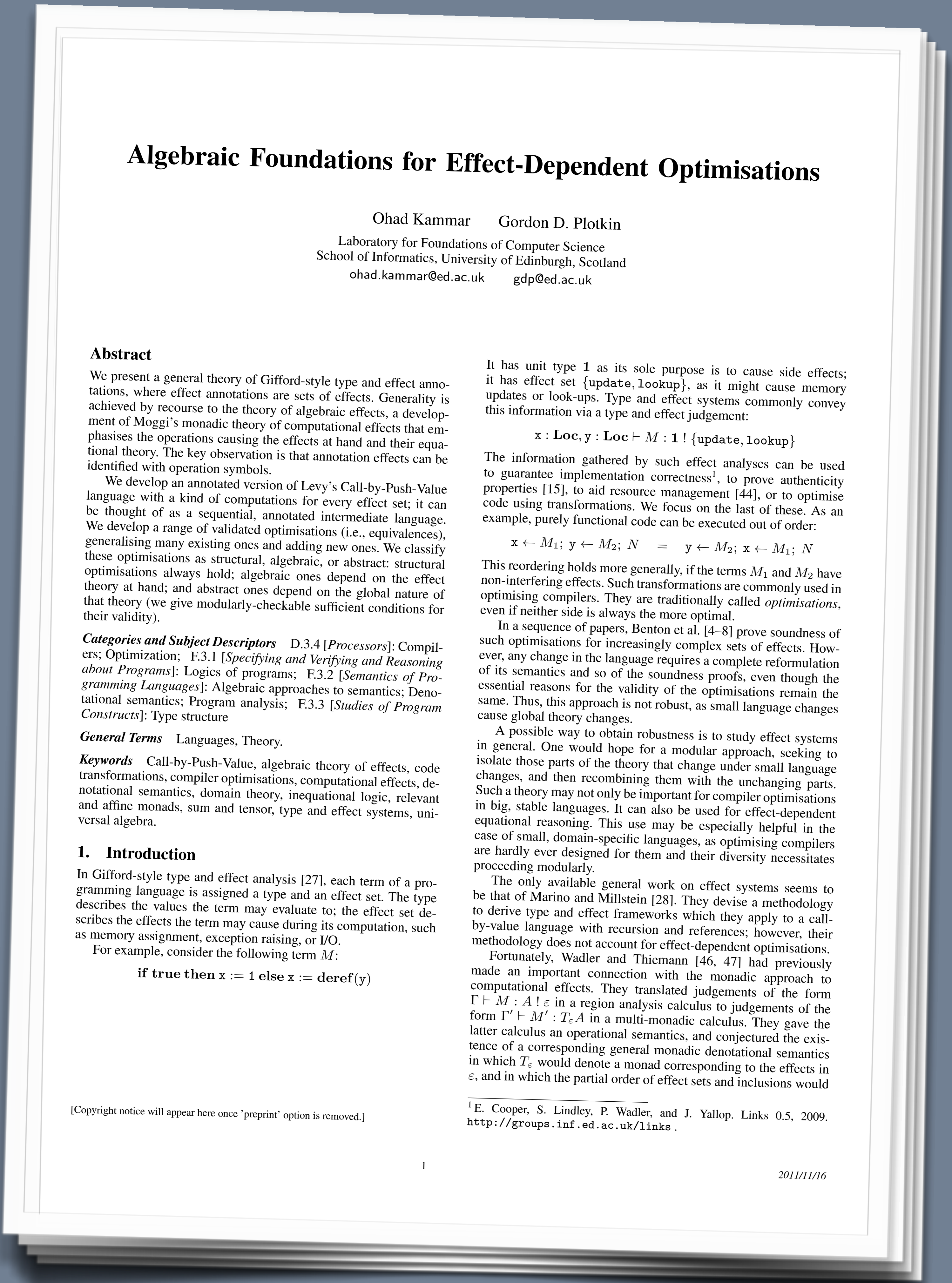
$$
\begin{array}{c}
\forall v : A.\ \phi(\mathbf{ret}\ v) \\
\text{operation cases} \\
\left[\forall c_i : A \,!\, \Delta.\ \bigwedge_i \phi(c_i) \Rightarrow \phi(\mathrm{op}(c_i)_i)\right]_{\mathrm{op} \in \Delta} \\
\hline
\forall c : A \,!\, \Delta.\ \phi(c)
\end{array}
$$

for $\Sigma = \{\oplus : 2, \mathrm{beep} : 1\}$, we have:

$$\begin{gathered}\mathbf{do}\; x \leftarrow c \;\mathbf{in}\; (c_1 \oplus c_2) \\ = \\ (\mathbf{do}\; x \leftarrow c \;\mathbf{in}\; c_1) \oplus (\mathbf{do}\; x \leftarrow c \;\mathbf{in}\; c_2)\end{gathered}$$

$$\begin{gathered}\mathbf{do}\; x \leftarrow c_1 \;\mathbf{in}\; \mathrm{beep}(c_2) \\ = \\ \mathrm{beep}(\mathbf{do}\; x \leftarrow c_1 \;\mathbf{in}\; c_2)\end{gathered}$$

$$\begin{gathered}\mathbf{do}\; x_1 \leftarrow c_1 \;\mathbf{in}\; (\mathbf{do}\; x_2 \leftarrow c_2 \;\mathbf{in}\; c) \\ = \\ \mathbf{do}\; x_2 \leftarrow c_2 \;\mathbf{in}\; (\mathbf{do}\; x_1 \leftarrow c_1 \;\mathbf{in}\; c)\end{gathered}$$



Algebraic Foundations for Effect-Dependent Optimisations

Ohad Kammar Gordon D. Plotkin

Laboratory for Foundations of Computer Science
School of Informatics, University of Edinburgh, Scotland
ohad.kammar@ed.ac.uk gdp@ed.ac.uk

Abstract

We present a general theory of Gifford-style type and effect annotations, where effect annotations are sets of effects. Generality is achieved by recourse to the theory of algebraic effects, a development of Moggi's monadic theory of computational effects that emphasises the operations causing the effects at hand and their equational theory. The key observation is that annotation effects can be identified with operation symbols.

We develop an annotated version of Levy's Call-by-Push-Value language with a kind of computations for every effect set; it can be thought of as a sequential, annotated intermediate language. We develop a range of validated optimisations (i.e., equivalences), generalising many existing ones and adding new ones. We classify these optimisations as structural, algebraic, or abstract: structural optimisations always hold; algebraic ones depend on the effect theory at hand; and abstract ones depend on the global nature of that theory (we give modularly-checkable sufficient conditions for their validity).

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; Optimization; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Logics of programs; F.3.2 [*Semantics of Programming Languages*]: Algebraic approaches to semantics; Denotational semantics; Program analysis; F.3.3 [*Studies of Program Constructs*]: Type structure

General Terms Languages, Theory.

Keywords Call-by-Push-Value, algebraic theory of effects, code transformations, compiler optimisations, computational effects, denotational semantics, domain theory, inequational logic, relevant and affine monads, sum and tensor, type and effect systems, universal algebra.

1. Introduction

In Gifford-style type and effect analysis [27], each term of a programming language is assigned a type and an effect set. The type describes the values the term may evaluate to; the effect set describes the effects the term may cause during its computation, such as memory assignment, exception raising, or I/O.

For example, consider the following term M:

$$\mathbf{if\ true\ then}\ \mathtt{x} := 1\ \mathbf{else}\ \mathtt{x} := \mathbf{deref}(\mathtt{y})$$

It has unit type $\mathbf{1}$ as its sole purpose is to cause side effects; it has effect set $\{\mathtt{update}, \mathtt{lookup}\}$, as it might cause memory updates or look-ups. Type and effect systems commonly convey this information via a type and effect judgement:

$$\mathtt{x} : \mathbf{Loc}, \mathtt{y} : \mathbf{Loc} \vdash M : \mathbf{1}\ !\ \{\mathtt{update}, \mathtt{lookup}\}$$

The information gathered by such effect analyses can be used to guarantee implementation correctness[1], to prove authenticity properties [15], to aid resource management [44], or to optimise code using transformations. We focus on the last of these. As an example, purely functional code can be executed out of order:

$$\mathtt{x} \leftarrow M_1;\ \mathtt{y} \leftarrow M_2;\ N \quad = \quad \mathtt{y} \leftarrow M_2;\ \mathtt{x} \leftarrow M_1;\ N$$

This reordering holds more generally, if the terms M_1 and M_2 have non-interfering effects. Such transformations are commonly used in optimising compilers. They are traditionally called *optimisations*, even if neither side is always the more optimal.

In a sequence of papers, Benton et al. [4–8] prove soundness of such optimisations for increasingly complex sets of effects. However, any change in the language requires a complete reformulation of its semantics and so of the soundness proofs, even though the essential reasons for the validity of the optimisations remain the same. Thus, this approach is not robust, as small language changes cause global theory changes.

A possible way to obtain robustness is to study effect systems in general. One would hope for a modular approach, seeking to isolate those parts of the theory that change under small language changes, and then recombining them with the unchanging parts. Such a theory may not only be important for compiler optimisations in big, stable languages. It can also be used for effect-dependent equational reasoning. This use may be especially helpful in the case of small, domain-specific languages, as optimising compilers are hardly ever designed for them and their diversity necessitates proceeding modularly.

The only available general work on effect systems seems to be that of Marino and Millstein [28]. They devise a methodology to derive type and effect frameworks which they apply to a call-by-value language with recursion and references; however, their methodology does not account for effect-dependent optimisations.

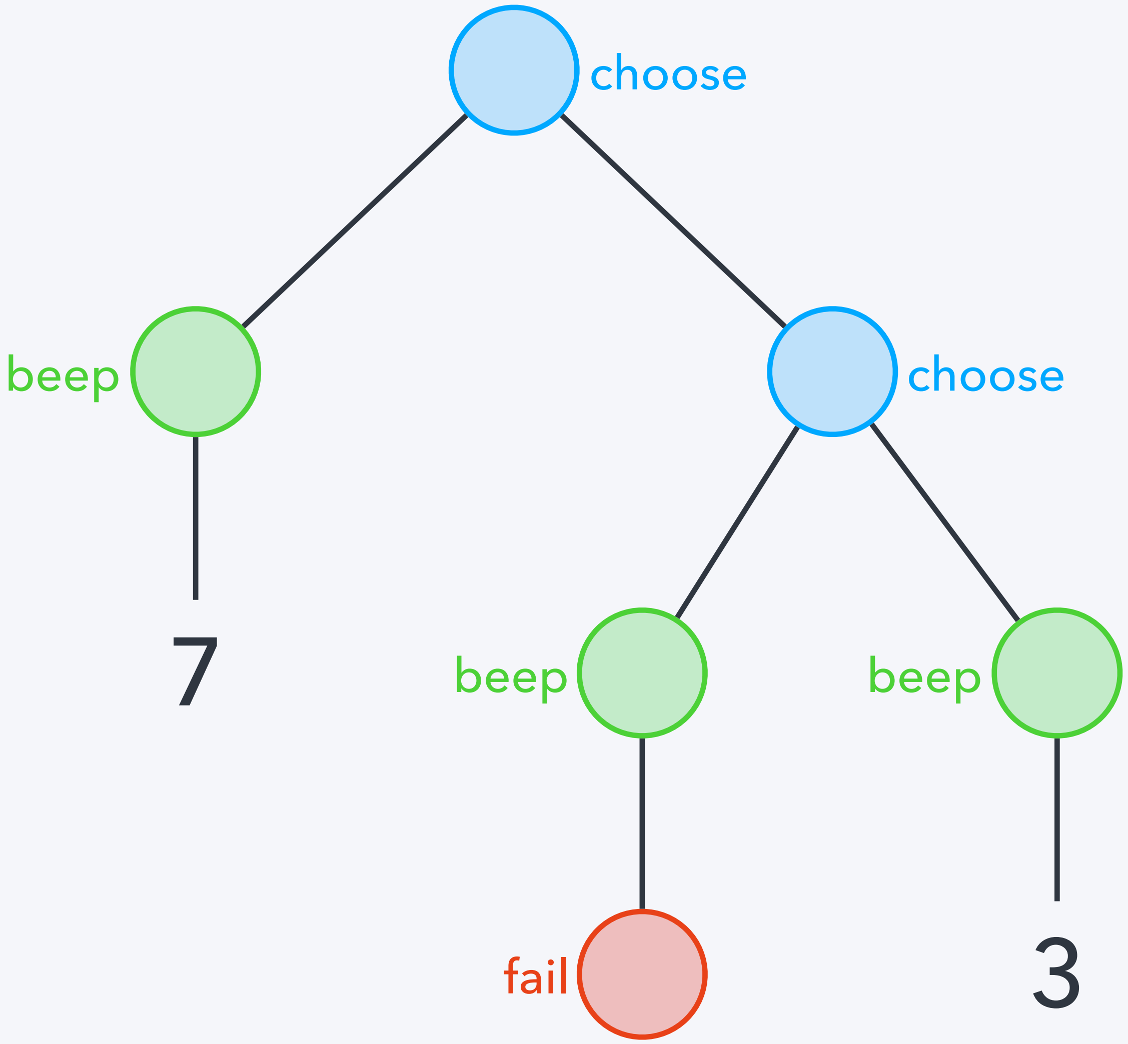
Fortunately, Wadler and Thiemann [46, 47] had previously made an important connection with the monadic approach to computational effects. They translated judgements of the form $\Gamma \vdash M : A\ !\ \varepsilon$ in a region analysis calculus to judgements of the form $\Gamma' \vdash M' : T_\varepsilon A$ in a multi-monadic calculus. They gave the latter calculus an operational semantics, and conjectured the existence of a corresponding general monadic denotational semantics in which T_ε would denote a monad corresponding to the effects in ε, and in which the partial order of effect sets and inclusions would

[1] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links 0.5, 2009. http://groups.inf.ed.ac.uk/links .

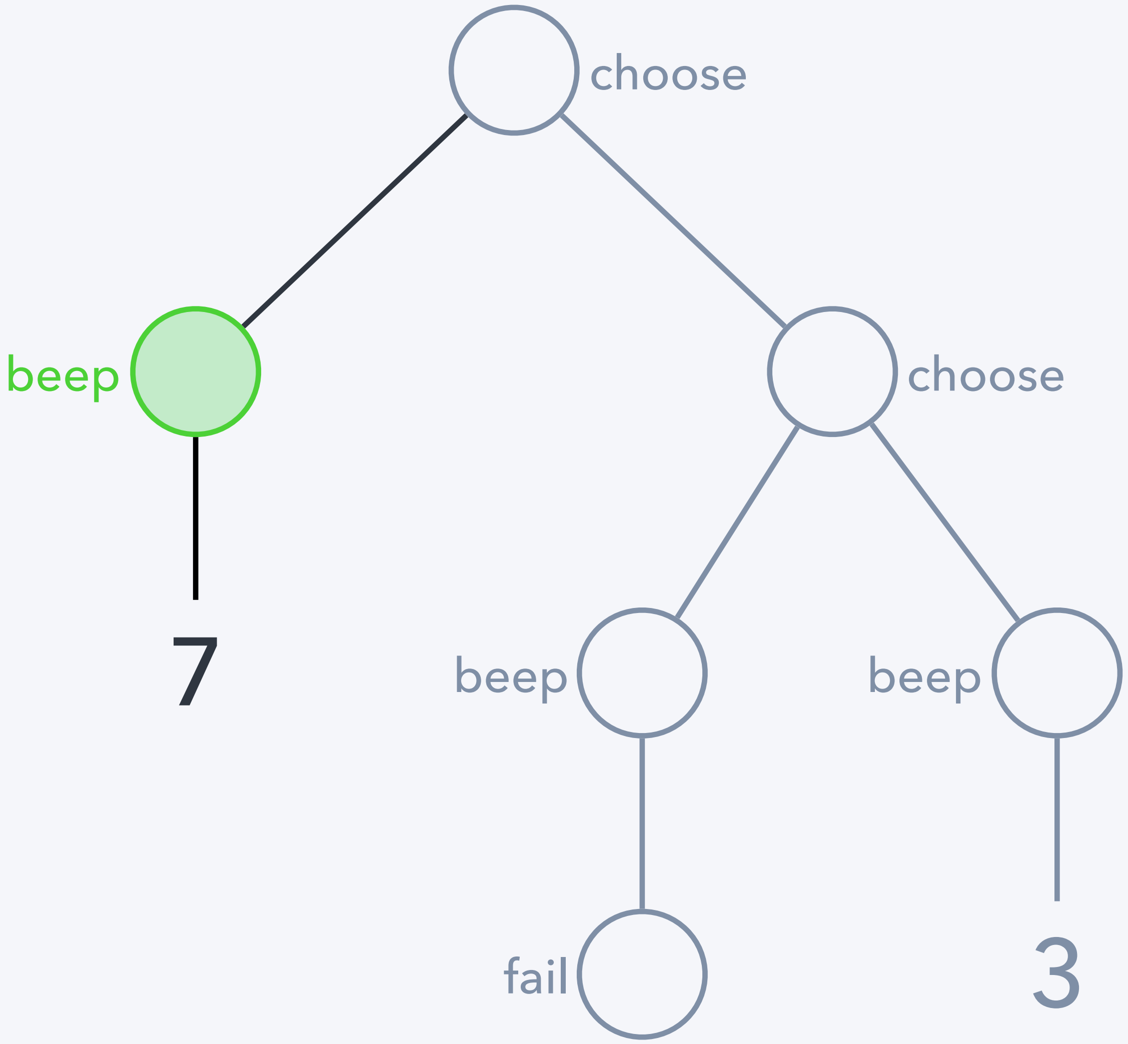
LET ME

HANDLE THIS!

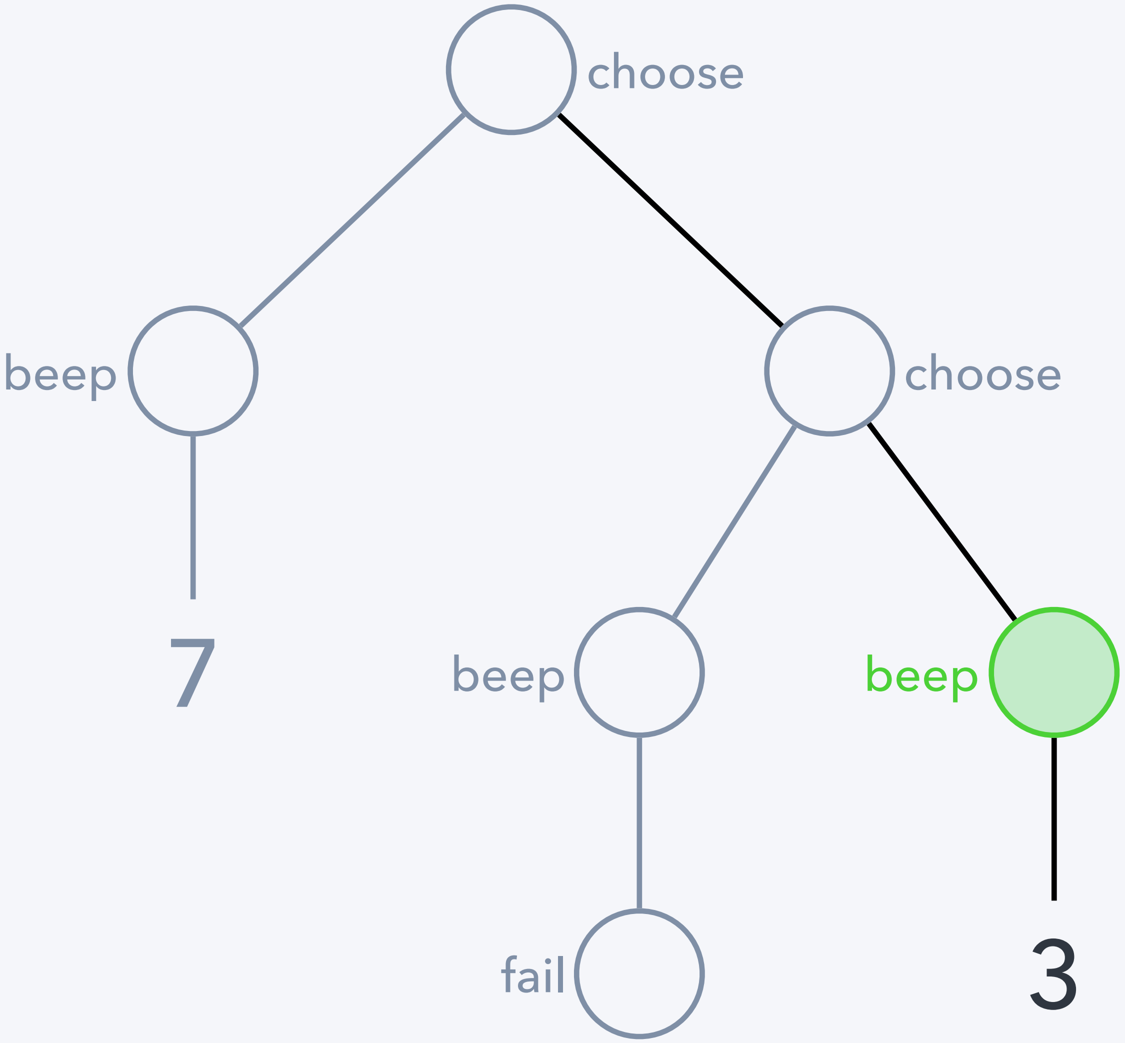
```
let divide m n =
  beep (); m / n
in
let x = choose 42 12 in
if x > 20 then
  divide x 6
else
  divide x (choose 0 4)
```



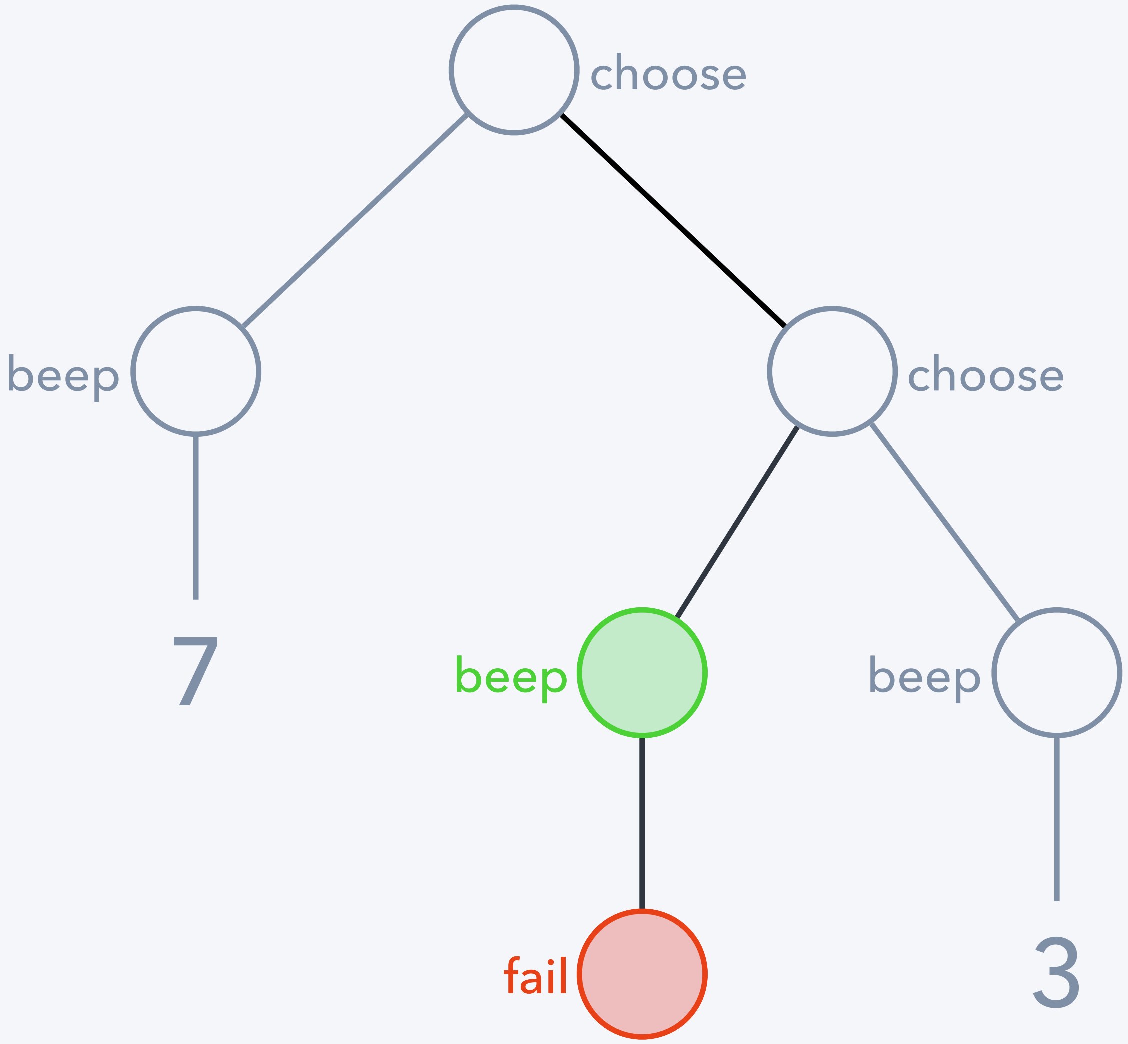
```
let goLeft = handler
  choose k₁ k₂ → k₁ ()
in
with goLeft handle
  let divide m n =
    beep (); m / n
  in
  let x = choose 42 12 in
  if x > 20 then
    divide x 6
  else
    divide x (choose 0 4)
```



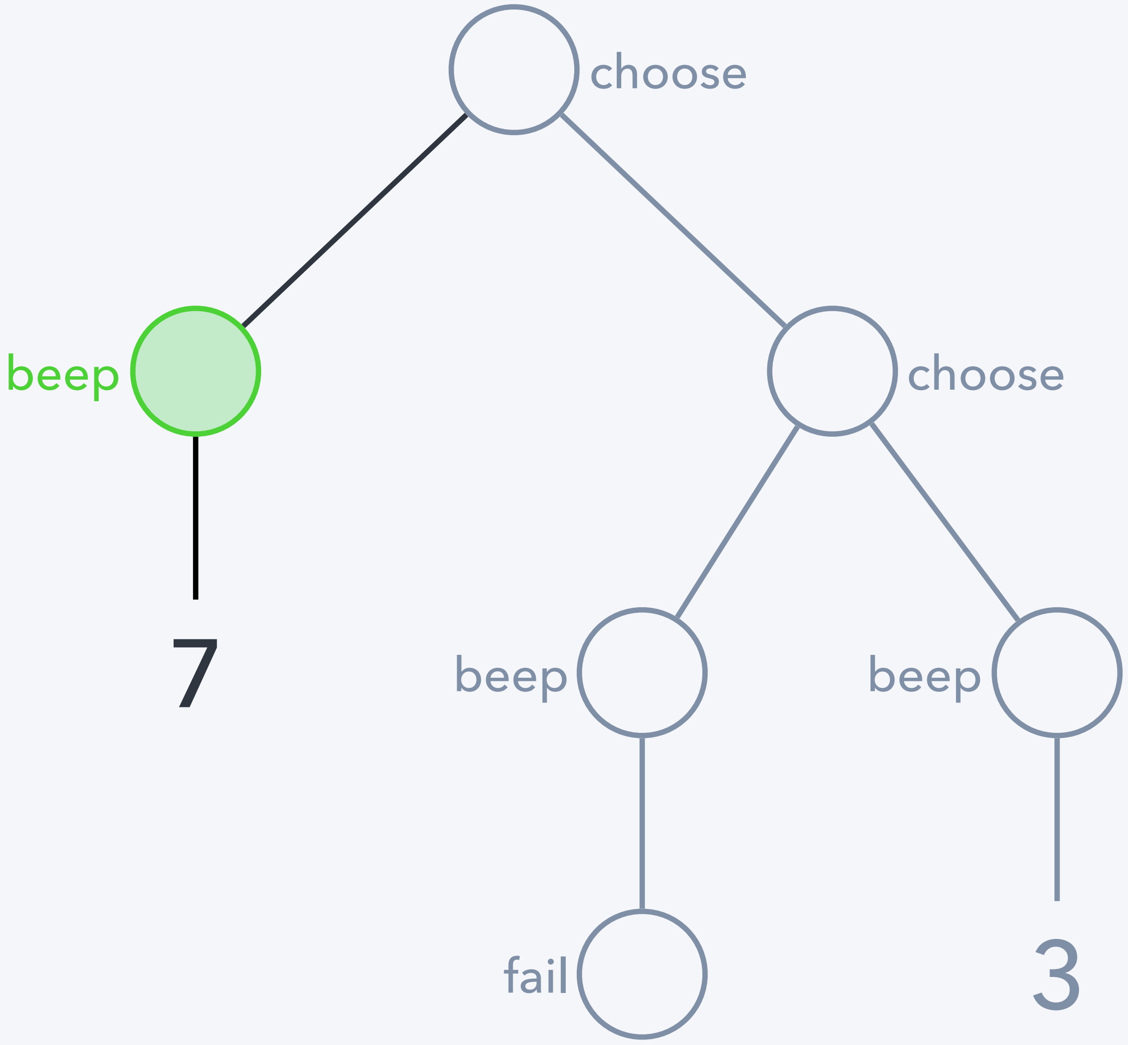
```
let goRight = handler
  choose k₁ k₂ → k₂ ()
in
with goRight handle
  let divide m n =
    beep (); m / n
  in
  let x = choose 42 12 in
  if x > 20 then
    divide x 6
  else
    divide x (choose 0 4)
```



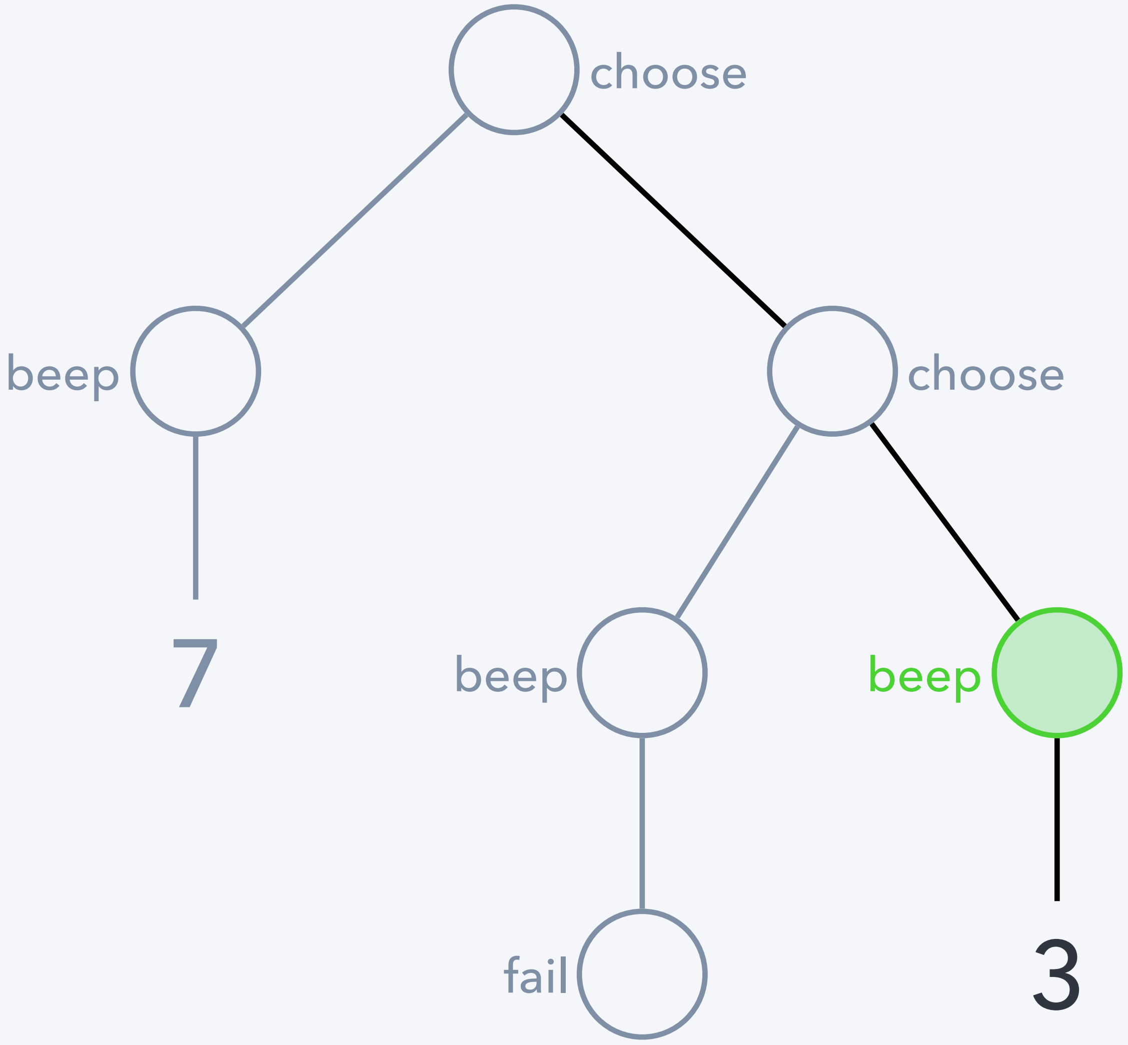
```
let random = handler
  choose k1 k2 →
    if randomFloat () > 0.5
    then k1 ()
    else k2 ()
in
with random handle
  let divide m n =
    beep (); m / n
  in
  let x = choose 42 12 in
  if x > 20 then
    divide x 6
  else
    divide x (choose 0 4)
```



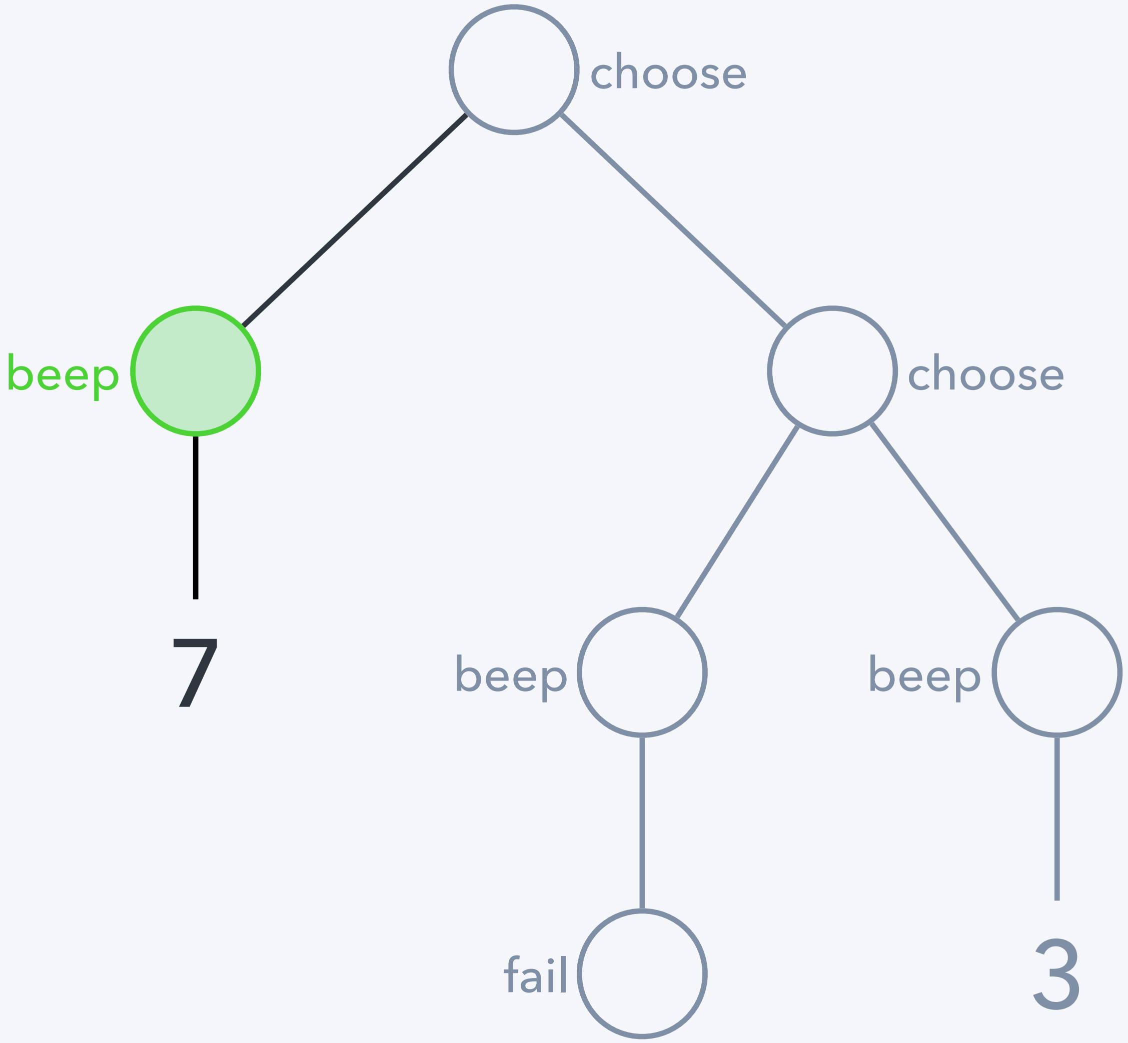
```
let random = handler
  choose k1 k2 →
    if randomFloat () > 0.5
    then k1 ()
    else k2 ()
in
with random handle
  let divide m n =
    beep (); m / n
  in
  let x = choose 42 12 in
  if x > 20 then
    divide x 6
  else
    divide x (choose 0 4)
```



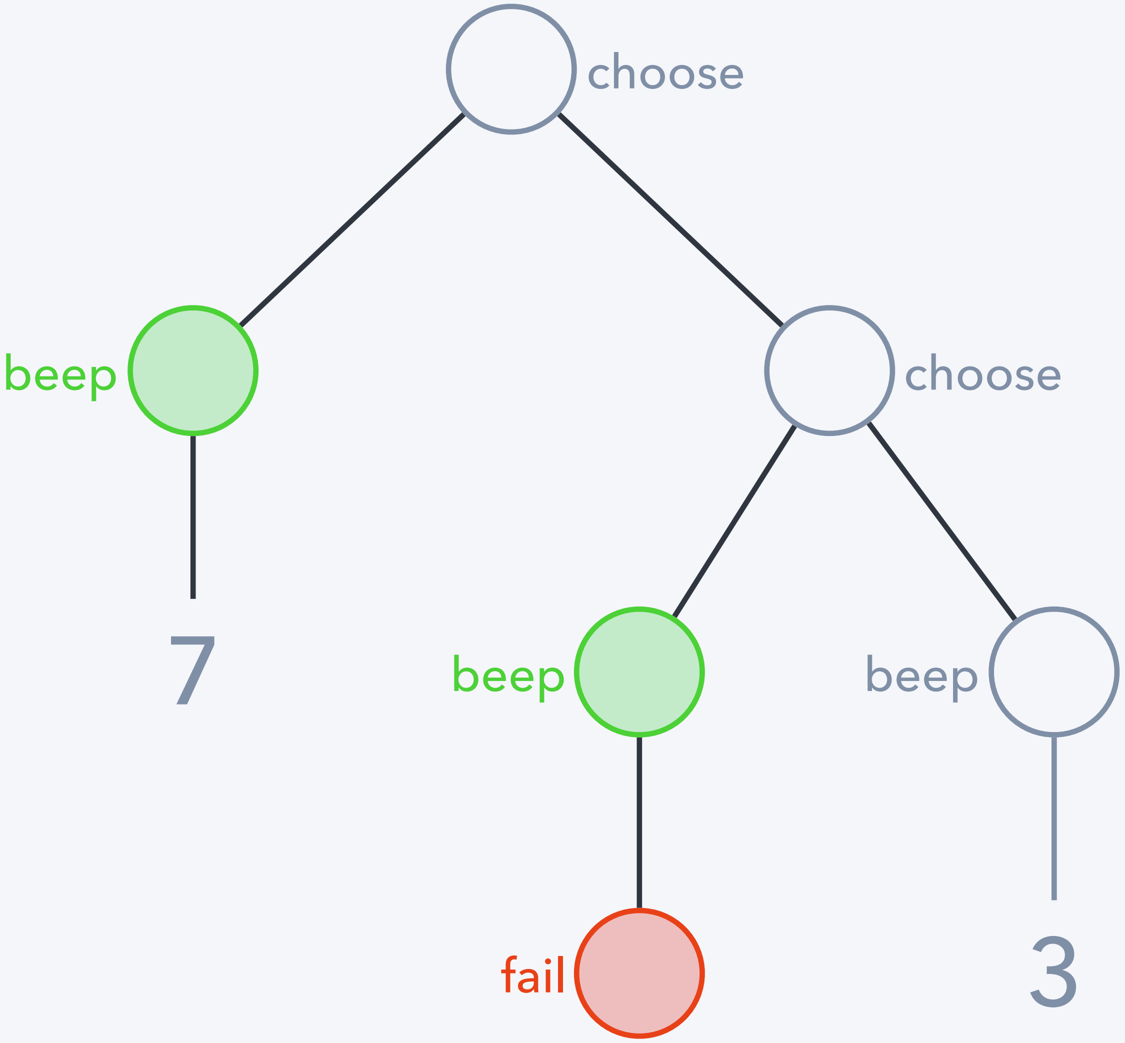
```
let random = handler
  choose k1 k2 →
    if randomFloat () > 0.5
    then k1 ()
    else k2 ()
in
with random handle
  let divide m n =
    beep (); m / n
  in
  let x = choose 42 12 in
  if x > 20 then
    divide x 6
  else
    divide x (choose 0 4)
```



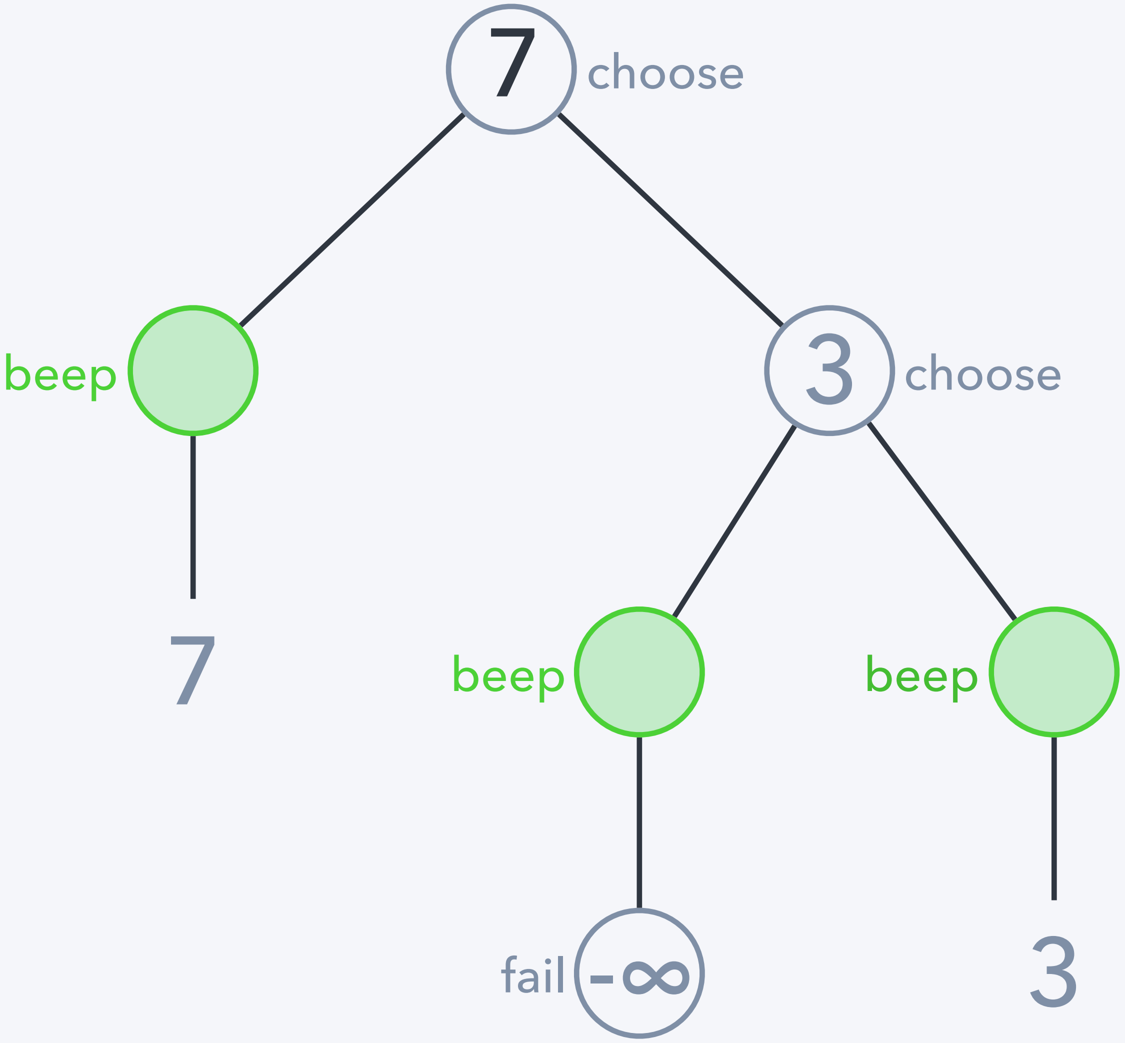
```
let random = handler
  choose k1 k2 →
    if randomFloat () > 0.5
    then k1 ()
    else k2 ()
in
with random handle
  let divide m n =
    beep (); m / n
  in
  let x = choose 42 12 in
  if x > 20 then
    divide x 6
  else
    divide x (choose 0 4)
```



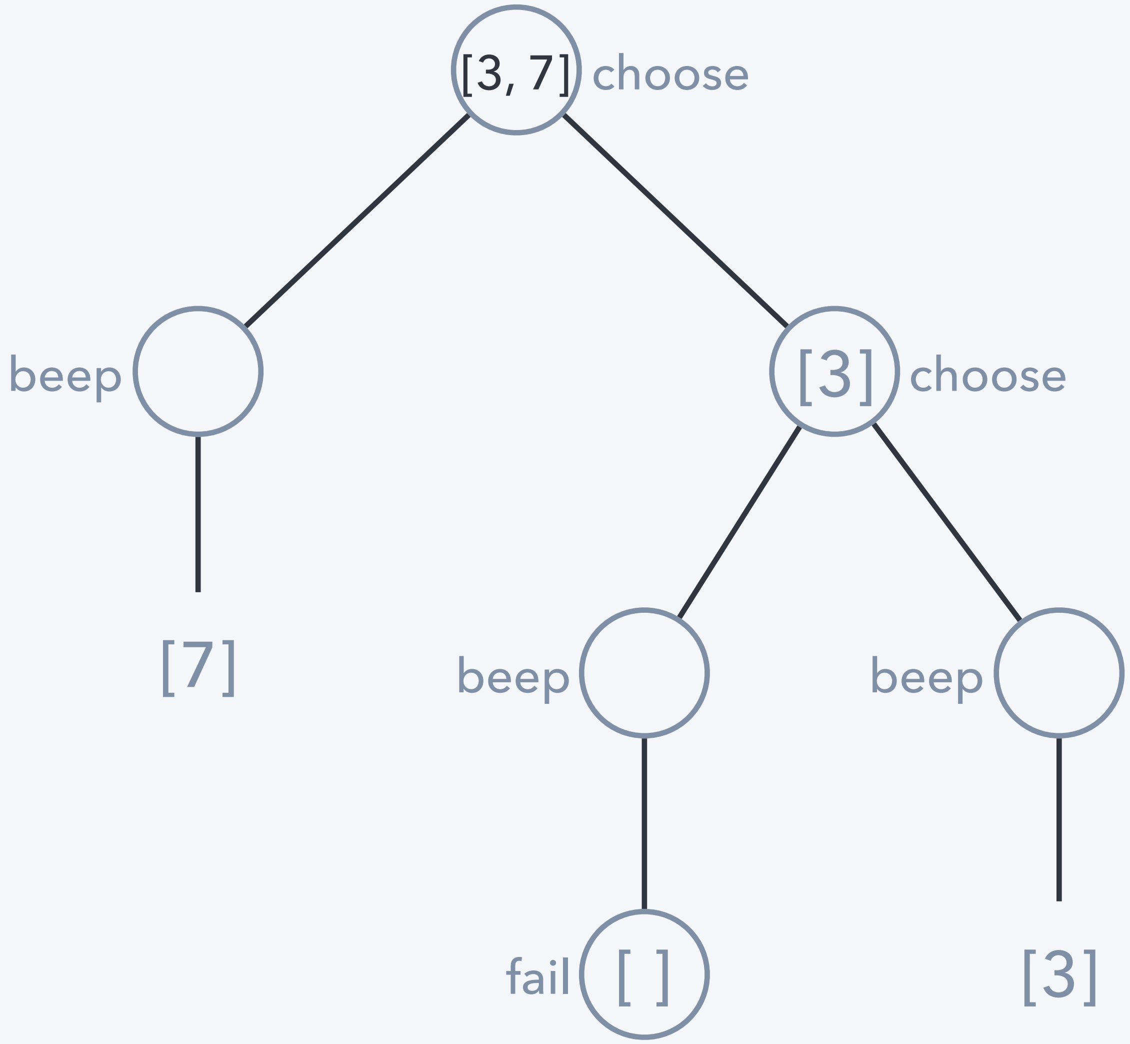
```
let pickMax = handler
  choose k1 k2 →
    max (k1 ()) (k2 ())
in
with pickMax handle
  let divide m n =
    beep (); m / n
  in
  let x = choose 42 12 in
  if x > 20 then
    divide x 6
  else
    divide x (choose 0 4)
```



```
let pickMax = handler
  choose k1 k2 →
    max (k1 ()) (k2 ())
  fail → -inf
in
with pickMax handle
  let divide m n =
    beep (); m / n
  in
  let x = choose 42 12 in
  if x > 20 then
    divide x 6
  else
    divide x (choose 0 4)
```



```
let toList = handler
  choose k1 k2 →
    k1 () ++ k2 ()
  fail → []
  beep k → k ()
  ret x → [x]
in
with toList handle
  let divide m n =
    beep (); m / n
  in
  let x = choose 42 12 in
  if x > 20 then
    divide x 6
  else
    divide x (choose 0 4)
```



$$v ::= \cdots \mid \mathbf{handler}\, h$$

$$c ::= \cdots \mid \mathbf{with}\ v\ \mathbf{handle}\ c$$

$$h ::= \{\mathbf{ret}\, x \mapsto c, [\mathrm{op}_j(k_i)_i \mapsto c_j]_j\}$$

$$\underline{A} ::= \cdots \mid \underline{C}_1 \Rrightarrow \underline{C}_2$$

$$\underline{C} ::= \cdots$$

values & computations

$$v ::= \cdots \mid \mathbf{handler}\, h$$

$$c ::= \cdots \mid \mathbf{with}\ v\ \mathbf{handle}\ c$$

$$h ::= \{\mathbf{ret}\, x \mapsto c, [\mathrm{op}_j(k_i)_i \mapsto c_j]_j\}$$

$$A ::= \cdots \mid \underline{C}_1 \Rightarrow \underline{C}_2$$

$$\underline{C} ::= \cdots$$

$$v ::= \cdots \mid \mathbf{handler}\, h$$

$$c ::= \cdots \mid \mathbf{with}\ v\ \mathbf{handle}\ c$$

$$h ::= \{\mathbf{ret}\, x \mapsto c, [\mathrm{op}_j(k_i)_i \mapsto c_j]_j\}$$

value types

$$A ::= \cdots \mid \underline{C}_1 \Rightarrow \underline{C}_2$$

$$\underline{C} ::= \cdots$$

computation types

$$v ::= \cdots \mid \mathbf{handler}\, h$$

$$c ::= \cdots \mid \mathbf{with}\ v\ \mathbf{handle}\ c$$

$$h ::= \{\mathbf{ret}\, x \mapsto c, [\mathrm{op}_j(k_i)_i \mapsto c_j]_j\}$$

handlers

$$A ::= \cdots \mid \underline{C}_1 \Rightarrow \underline{C}_2$$

$$\underline{C} ::= \cdots$$

value & computation typing

$$\Gamma \vdash v : A$$

$$\Gamma \vdash c : \underline{C}$$

$$\Gamma \vdash h : A \,!\, \Delta \Rightarrow \underline{C}$$

handler typing

$$\frac{\Gamma \vdash h : A \,!\, \Delta \rightrightarrows \underline{C}}{\Gamma \vdash \mathbf{handler}\, h : A \,!\, \Delta \Rightarrow \underline{C}}$$

$$\frac{\Gamma \vdash v : \underline{C}_1 \Rightarrow \underline{C}_2 \qquad \Gamma \vdash c : \underline{C}_1}{\Gamma \vdash \mathbf{with}\ v\ \mathbf{handle}\ c : \underline{C}_2}$$

$$\frac{\begin{array}{c}\Gamma, x : A \vdash c : \underline{C} \\ \left[\Gamma, (k_i : \mathtt{unit} \to \underline{C})_i \vdash c_j : \underline{C}\right]_j\end{array}}{\Gamma \vdash h : A \,!\, \{\mathrm{op}_j\}_j \Rightarrow \underline{C}}$$

$h = \{\mathbf{ret}\; x \mapsto c, [\mathrm{op}_j(k_i)_i \mapsto c_j]_j\}$

$$\textbf{with}\ h\ \textbf{handle}\ (\textbf{ret}\ v)$$

$$=$$

$$c[v/x]$$

$$\textbf{with}\ h\ \textbf{handle}\ (\mathrm{op}_j(c'_i)_i)$$

$$=$$

$$c_j[\textbf{fun}\ ()\mapsto(\textbf{with}\ h\ \textbf{handle}\ c'_i)/k_i]_i$$

$$h = \textbf{handler}\ \{\textbf{ret}\ x \mapsto c, [\mathrm{op}_j(k_i)_i \mapsto c_j]_j\}$$

ret 0

$$
\begin{aligned}
& \textbf{ret}\ 0 \\
= {} & \textbf{with}\ \mathrm{goLeft}\ \textbf{handle}\ (\textbf{ret}\ 0 \oplus \textbf{ret}\ 1)
\end{aligned}
$$

$$
\begin{aligned}
&\ \mathbf{ret}\ 0 \\
&= \mathbf{with}\ \mathrm{goLeft}\ \mathbf{handle}\ (\mathbf{ret}\ 0 \oplus \mathbf{ret}\ 1) \\
&= \mathbf{with}\ \mathrm{goLeft}\ \mathbf{handle}\ (\mathbf{ret}\ 1 \oplus \mathbf{ret}\ 0)
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{ret}\ 0 \\
= {} &\mathbf{with}\ \mathrm{goLeft}\ \mathbf{handle}\ (\mathbf{ret}\ 0 \oplus \mathbf{ret}\ 1) \\
= {} &\mathbf{with}\ \mathrm{goLeft}\ \mathbf{handle}\ (\mathbf{ret}\ 1 \oplus \mathbf{ret}\ 0) \\
= {} &\mathbf{ret}\ 1
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{ret}\ 0\\
=\ &\mathbf{with}\ \mathrm{goLeft}\ \mathbf{handle}\ (\mathbf{ret}\ 0 \oplus \mathbf{ret}\ 1)\\
=\ &\mathbf{with}\ \mathrm{goLeft}\ \mathbf{handle}\ (\mathbf{ret}\ 1 \oplus \mathbf{ret}\ 0)\\
=\ &\mathbf{ret}\ 1
\end{aligned}
$$

$$\frac{\left[T_1^h = T_2^h\right]_{(T_1 = T_2) \in \mathscr{E}_{\text{global}}}}{h \text{ is correct}}$$

$$z_j^h = f_j : \texttt{unit} \to \underline{C}$$

$$\mathrm{op}_j(T_i)_i^h = c_j[(\mathbf{fun}\ () \mapsto T_i^h)/k_i]_i$$

$h = \{\mathbf{ret}\ x \mapsto c, [\mathrm{op}_j(k_i)_i \mapsto c_j]_j\}$

	`goLeft` `goRight`	`random`	`pickMax`	`toList`	`toSet`
$z \oplus z = z$	✓	✓	✓	✗	✓
$z_1 \oplus z_2$ $= z_2 \oplus z_1$	✗	✓	✓	✗	✓
$(z_1 \oplus z_2) \oplus z_3$ $= z_1 \oplus (z_2 \oplus z_3)$	✓	✗	✓	✓	✓
$\mathrm{beep}(z_1) \oplus \mathrm{beep}(z_2)$ $= \mathrm{beep}(z_1 \oplus z_2)$	✓	✓	✗	✗	✗
$\mathrm{fail}() \oplus \mathrm{fail}()$ $= \mathrm{fail}()$	✓	✓	✓	✓	✓

signature

$$\oplus : 2 \qquad \mathrm{beep} : 1 \qquad \mathrm{fail} : 0$$

$$z \oplus z = z$$

$$z_1 \oplus z_2 = z_2 \oplus z_1$$

$$(z_1 \oplus z_2) \oplus z_3 = z_1 \oplus (z_2 \oplus z_3)$$

$$\mathrm{beep}(z_1) \oplus \mathrm{beep}(z_2) = \mathrm{beep}(z_1 \oplus z_2)$$

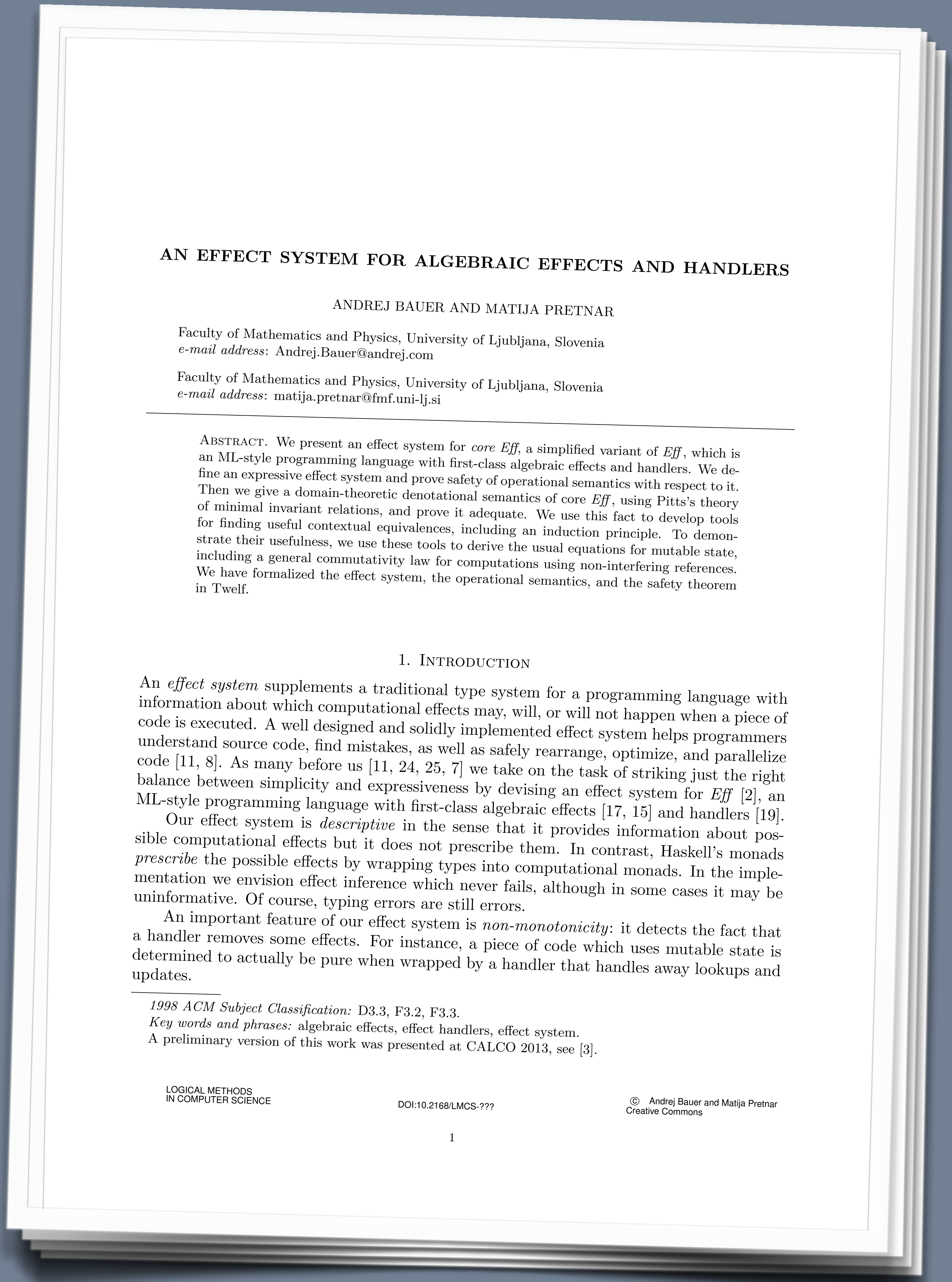
$$\mathrm{fail}() \oplus \mathrm{fail}() = \mathrm{fail}()$$

equational theory

signature

$\oplus : 2 \qquad \mathrm{beep} : 1 \qquad \mathrm{fail} : 0$

equational theory



AN EFFECT SYSTEM FOR ALGEBRAIC EFFECTS AND HANDLERS

ANDREJ BAUER AND MATIJA PRETNAR

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
e-mail address: Andrej.Bauer@andrej.com

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
e-mail address: matija.pretnar@fmf.uni-lj.si

Abstract. We present an effect system for *core Eff*, a simplified variant of *Eff*, which is an ML-style programming language with first-class algebraic effects and handlers. We define an expressive effect system and prove safety of operational semantics with respect to it. Then we give a domain-theoretic denotational semantics of core *Eff*, using Pitts's theory of minimal invariant relations, and prove it adequate. We use this fact to develop tools for finding useful contextual equivalences, including an induction principle. To demonstrate their usefulness, we use these tools to derive the usual equations for mutable state, including a general commutativity law for computations using non-interfering references. We have formalized the effect system, the operational semantics, and the safety theorem in Twelf.

1. Introduction

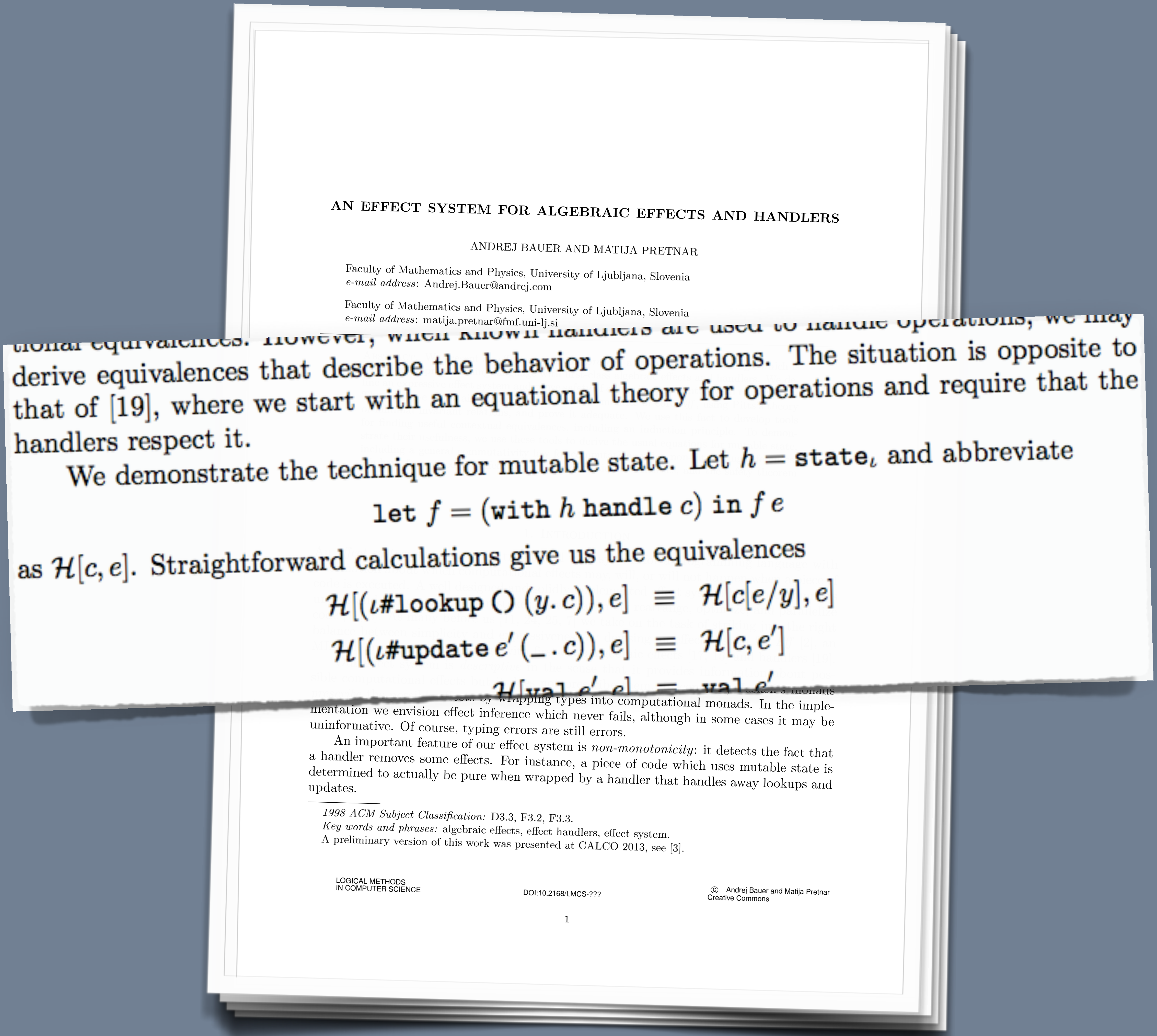
An *effect system* supplements a traditional type system for a programming language with information about which computational effects may, will, or will not happen when a piece of code is executed. A well designed and solidly implemented effect system helps programmers understand source code, find mistakes, as well as safely rearrange, optimize, and parallelize code [11, 8]. As many before us [11, 24, 25, 7] we take on the task of striking just the right balance between simplicity and expressiveness by devising an effect system for *Eff* [2], an ML-style programming language with first-class algebraic effects [17, 15] and handlers [19].

Our effect system is *descriptive* in the sense that it provides information about possible computational effects but it does not prescribe them. In contrast, Haskell's monads *prescribe* the possible effects by wrapping types into computational monads. In the implementation we envision effect inference which never fails, although in some cases it may be uninformative. Of course, typing errors are still errors.

An important feature of our effect system is *non-monotonicity*: it detects the fact that a handler removes some effects. For instance, a piece of code which uses mutable state is determined to actually be pure when wrapped by a handler that handles away lookups and updates.

1998 ACM Subject Classification: D3.3, F3.2, F3.3.
Key words and phrases: algebraic effects, effect handlers, effect system.
A preliminary version of this work was presented at CALCO 2013, see [3].

DOI:10.2168/LMCS-???



AN EFFECT SYSTEM FOR ALGEBRAIC EFFECTS AND HANDLERS

ANDREJ BAUER AND MATIJA PRETNAR

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
e-mail address: Andrej.Bauer@andrej.com

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
e-mail address: matija.pretnar@fmf.uni-lj.si

tional equivalences. However, when known handlers are used to handle operations, we may derive equivalences that describe the behavior of operations. The situation is opposite to that of [19], where we start with an equational theory for operations and require that the handlers respect it.

We demonstrate the technique for mutable state. Let $h = \mathtt{state}_\iota$ and abbreviate

$$\mathtt{let}\ f = (\mathtt{with}\ h\ \mathtt{handle}\ c)\ \mathtt{in}\ f\ e$$

as $\mathcal{H}[c, e]$. Straightforward calculations give us the equivalences

$$\begin{aligned}
\mathcal{H}[(\iota\#\mathtt{lookup}\ ()\ (y.\, c)), e] &\equiv \mathcal{H}[c[e/y], e] \\
\mathcal{H}[(\iota\#\mathtt{update}\ e'\ (\_ .\, c)), e] &\equiv \mathcal{H}[c, e'] \\
\mathcal{H}[\mathtt{val}\ e', e] &\equiv \mathtt{val}\ e'
\end{aligned}$$

Haskell's monads effects by wrapping types into computational monads. In the implementation we envision effect inference which never fails, although in some cases it may be uninformative. Of course, typing errors are still errors.

An important feature of our effect system is *non-monotonicity*: it detects the fact that a handler removes some effects. For instance, a piece of code which uses mutable state is determined to actually be pure when wrapped by a handler that handles away lookups and updates.

1998 ACM Subject Classification: D3.3, F3.2, F3.3.
Key words and phrases: algebraic effects, effect handlers, effect system.
A preliminary version of this work was presented at CALCO 2013, see [3].

DOI:10.2168/LMCS-???

$$\mathscr{H}_{\max}[c] \overset{\text{def}}{=} \textbf{with } \text{pickMax } \textbf{handle } c$$

$$\begin{gathered}\mathscr{H}_{\max}\big[\textbf{do } x \leftarrow c \textbf{ in } (c_1 \oplus c_2)\big] \\ = \\ \mathscr{H}_{\max}\big[(\textbf{do } x \leftarrow c \textbf{ in } c_1) \oplus (\textbf{do } x \leftarrow c \textbf{ in } c_2)\big]\end{gathered}$$

$$\begin{gathered}\mathscr{H}_{\max}\big[\textbf{do } x_1 \leftarrow c_1 \textbf{ in } (\textbf{do } x_2 \leftarrow c_2 \textbf{ in } c)\big] \\ = \\ \mathscr{H}_{\max}\big[\textbf{do } x_2 \leftarrow c_2 \textbf{ in } (\textbf{do } x_1 \leftarrow c_1 \textbf{ in } c)\big]\end{gathered}$$

MAKE **EQUATIONS**
GREAT AGAIN!

#mega

values & computations

$$v ::= \cdots$$

$$c ::= \cdots$$

$$h ::= \cdots$$

handlers

value types

$$A ::= \cdots$$

$$\underline{C} ::= A \,!\, \Delta \,/\, \mathscr{E}$$

computation types

$$\frac{\left[T_1^h = T_2^h\right]_{(T_1 = T_2) \in \mathscr{E}_{\text{global}}}}{h \textbf{ is correct}}$$

$$\frac{\left[T_1^h = T_2^h\right]_{(T_1=T_2)\in\mathscr{E}}}{h \text{ respects } \mathscr{E}}$$

$$\frac{\begin{array}{c}\Gamma \vdash h : A \mathbin{!} \Delta \Rightarrow \underline{C} \\ h \ \textbf{respects} \ \mathscr{E}\end{array}}{\Gamma \vdash \textbf{handler}\, h : A \mathbin{!} \Delta / \mathscr{E} \Rightarrow \underline{C}}$$

$$\frac{\Gamma \vdash v : A}{\Gamma \vdash \mathbf{ret}\, v : A \,!\, \Delta \,/\, \mathcal{E}}$$

$$\frac{\Gamma \vdash c_1 : A \,!\, \Delta \,/\, \mathcal{E} \qquad \Gamma, x : A \vdash c_2 : B \,!\, \Delta \,/\, \mathcal{E}}{\Gamma \vdash \mathbf{do}\, x \leftarrow c_1 \,\mathbf{in}\, c_2 : B \,!\, \Delta \,/\, \mathcal{E}}$$

$$\frac{\left[\Gamma \vdash c_i : A \,!\, \Delta\right]_i \qquad \mathrm{op} \in \Delta}{\Gamma \vdash \mathrm{op}(c_i)_i : A \,!\, \Delta \,/\, \mathcal{E}}$$

$$\frac{(T = T') \in \mathscr{E} \qquad \left[c_i : A \,!\, \Delta \,/\, \mathscr{E}\right]_i}{T[c_i/z_i]_i \;=_{A!\Delta/\mathscr{E}}\; T'[c_i/z_i]_i}$$

$$\mathscr{H}_{\mathrm{rand}}[c] \overset{\mathrm{def}}{=} \mathbf{with}\ \mathrm{random}\ \mathbf{handle}\ c$$

$$\mathscr{H}_{\mathrm{list}}[c] \overset{\mathrm{def}}{=} \mathbf{with}\ \mathrm{toList}\ \mathbf{handle}\ c$$

$$\begin{aligned}&\mathbf{do}\ x \leftarrow \mathscr{H}_{\mathrm{rand}}[c_2 \oplus c_1]\ \mathbf{in}\\ &\mathbf{do}\ xs \leftarrow \mathscr{H}_{\mathrm{list}}[c_1 \oplus (c_2 \oplus c_3)]\ \mathbf{in}\\ &\mathbf{ret}\ (x \in xs)\end{aligned}$$

$$=$$

$$\mathbf{ret}\ \mathbf{true}$$

```
let pickMax = handler
  choose k1 k2 →
    max (k1 ()) (k2 ())
  fail → -inf
```

$$A \,!\, \{\oplus, \mathrm{fail}\} \,/\, \mathcal{E}_{\mathrm{semilattice}} \Rightarrow A \,!\, \emptyset \,/\, \emptyset$$

$$\begin{aligned}&A \,!\, \{\oplus, \mathrm{beep}, \mathrm{fail}\} \,/\, \mathcal{E}_{\mathrm{loudSemilattice}}\\ &\quad\Rightarrow A \,!\, \{\mathrm{beep}\} \,/\, \{\mathrm{beep}(\mathrm{beep}(z)) = \mathrm{beep}(z)\}\end{aligned}$$

$$\mathrm{myQuery} : A \to B \; ! \; \Sigma_{\mathrm{SQL}} \, / \, \emptyset$$

$$\mathrm{queryBraga} : B \; ! \; \Sigma_{\mathrm{SQL}} \, / \, \mathscr{E}_{\mathrm{SQL}} \Rightarrow B \; ! \; \emptyset \, / \, \emptyset$$

$$\begin{aligned}
\mathrm{myQuery} &: A \to B \,!\, \Sigma_{\mathrm{SQL}} \,/\, \emptyset \\
\mathrm{queryBraga} &: B \,!\, \Sigma_{\mathrm{SQL}} \,/\, \mathscr{E}_{\mathrm{SQL}} \Rightarrow B \,!\, \emptyset \,/\, \emptyset \\
\mathrm{optimizeSQL} &: B \,!\, \Sigma_{\mathrm{SQL}} \,/\, \emptyset \Rightarrow B \,!\, \Sigma_{\mathrm{SQL}} \,/\, \mathscr{E}_{\mathrm{SQL}}
\end{aligned}$$

$$\mathrm{swap} \overset{\mathrm{def}}{=} \mathbf{handler}\,\{k_1 \oplus k_2 \mapsto k_2 \oplus k_1\}$$

$$\mathrm{swap} : A \,!\, \Delta \,/\, \mathscr{E} \Rightarrow A \,!\, \Delta \,/\, \mathscr{E}$$

$$\text{for all } \mathscr{E} \subseteq \mathscr{E}_{\mathrm{semilattice}}$$

FUTURE WORK

usual suspects

- semantics
- safety
- soundness

useful implementation

- QuickCheck / QuickSpec
- compiler optimizations

powerful logic

- extended reasoning
- more powerful specification