Thesis report

Albert ten Napel

1 Simply typed lambda calculus

As the core of the calculi that follow we have chosen the fine-grain call-by-value[1] variant of the simply typed lambda calculus (STLC-fg). Terms are divided in values and computations, which allows the system to be extended to have effects more easily, since values never have effects but computations do.

$\tau ::=$		(value types)
	$()$	(unit type)
	$\tau \to \tau$	(type of functions)
$\nu ::=$		(values)
	x, y, z, k	(variables)
	$()$	(unit value)
	$\lambda x.c$	(abstraction)
$c ::=$		(computations)
	$\mathit{return}\ \nu$	(return value as computation)
	$\nu\ \nu$	(application)
	$x \leftarrow c; c$	(sequencing)

For the typing rules there are two judgements, $\Gamma \vdash \nu : \tau$ for assigning types to values and $\Gamma \vdash c : \tau$ for assigning types to computations.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash () : ()} \qquad \frac{\Gamma, x : \tau_1 \vdash c : \tau_2}{\Gamma \vdash \lambda x.c : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash \nu : \tau}{\Gamma \vdash \mathit{return}\ \nu : \tau} \qquad \frac{\Gamma \vdash \nu_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash \nu_2 : \tau_1}{\Gamma \vdash \nu_1\ \nu_2 : \tau_2} \qquad \frac{\Gamma \vdash c_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash c_2 : \tau_2}{\Gamma \vdash (x \leftarrow c_1; c_2) : \tau_2}$$

We define the relation $\rightsquigarrow$ for the small-step operational semantics.

$$\frac{}{(\lambda x.c)\ \nu \rightsquigarrow c[\nu/x]} \qquad \frac{c_1 \rightsquigarrow c_1'}{(x \leftarrow c_1; c_2) \rightsquigarrow (x \leftarrow c_1'; c_2)}$$

$$\frac{}{(x \leftarrow \mathit{return}\ \nu; c) \rightsquigarrow c[\nu/x]}$$

```
-- As an example we define a forking combinator.

-- In the usual formulation of
-- the simply typed lambda calculus:
\f g h x . f (g x) (h x)

-- In the fine-grain call-by-value
-- simply typed lambda calculus:
\f g h x . (y <- g x; z <- h x; f y z)
-- In this system we have to be explicit
-- about the order of evaluation.
```

2 Algebraic effects

We extend the calculus with basic algebraic effects. We assume there is a predefined set of effects $E := \{\varepsilon_1, ..., \varepsilon_n\}$, where ε is a single effect name and ε^* is some subset of E. In a real programming language these effects would include IO, non-determinism, concurrency, mutable state and so on. For every effect ε we have a set of operations O^ε, where op is a single operation and op^* is some set of operations (possible from different effects). Each operation op has a parameter type τ^0_{op} and a return type τ^1_{op}. Rules are taken from [2]. We extend and update the syntax of the previous calculus as follows:

$$
\begin{array}{lr}
\tau ::= ... & \text{(value types)} \\
\quad \tau \to \underline{\tau} & \text{(type of functions)} \\
\quad \underline{\tau} \Rightarrow \underline{\tau} & \text{(type of handlers)} \\
\underline{\tau} ::= \tau ! R & \text{(computation types)} \\
R ::= op^* & \text{(effect annotations)} \\
\nu ::= ... & \text{(values)} \\
\quad handler\,\{return\ x \to c, op_1(x; k) \to c, ..., op_n(x; k) \to c\} & \text{(handler)} \\
c ::= ... & \text{(computations)} \\
\quad op(\nu; \lambda x.c) & \text{(operation call)} \\
\quad with\ \nu\ handle\ c & \text{(handle computation)}
\end{array}
$$

We add computation types, which are value types together with a set of operations that are performed in a computation. Function types now have a computation type as return type.
Note that the operation call also packages a value and a continuation inside of it, having the continuation makes the semantics easier. We can get back the simpler operation calls such as seen in Eff and Koka by defining $op := \lambda x.op(x; \lambda y.return\ y)$ (Pretnar calls these Generic Effects in [2]).

2.1 Typing rules

In the typing judgment of computations we assign a computation type to a computation: $\Gamma \vdash c : \underline{\tau}$.

The typing rules for the values of the previous calculus stay the same, though we have to update the rule for abstractions to account for computation types.

$$\frac{\Gamma, x : \tau_1 \vdash c : \underline{\tau}_2}{\Gamma \vdash \lambda x.c : \tau_1 \to \underline{\tau}_2}$$

For return we allow the operation set of the computation to be polymorphic, this simplifies type checking. For the sequencing the operation sets have to match.

$$\frac{\Gamma \vdash \nu : \tau}{\Gamma \vdash \mathit{return}\ \nu : \tau ! R} \qquad \frac{\Gamma \vdash \nu_1 : \tau_1 \to \underline{\tau}_2 \quad \Gamma \vdash \nu_2 : \tau_1}{\Gamma \vdash \nu_1\ \nu_2 : \underline{\tau}_2} \qquad \frac{\Gamma \vdash c_1 : \underline{\tau}_1 \quad \Gamma, x : \tau_1 \vdash c_2 : \underline{\tau}_2}{\Gamma \vdash (x \leftarrow c_1; c_2) : \underline{\tau}_2}$$

For the handler we check that both the return case and the operation cases agree on the effects. A handler is allowed to have more effects in its type then it handles, these effects will simply remain unhandled and they will appear in both the input and output effect sets. The input effect set must atleast contain all effects that the operations belong to and the output effect set must agree on the unhandled effects of the input set.

$$\frac{\begin{array}{l}\Gamma, x_r : \tau_1 \vdash c_r : \tau_2 ! R_2 \\ \Gamma, x_i : \tau^0_{op_i}, k_i : \tau^1_{op_i} \to \tau_2 ! R_2 \vdash c_i : \tau_2 ! R_2 \\ R_1 \setminus op_i^* \subseteq R_2\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{handler}\,\{\mathit{return}\ x_r \to c_r, op_1(x_1; k_1) \to c_1, ..., op_n(x_n; k_n) \to c_n\} \\ \quad : \tau_1 ! R_1 \Rightarrow \tau_2 ! R_2\end{array}}$$

The typing rules of the computations of the previous calculus stay the same. For operation calls we have to check that the effect that belongs to the operation is contained in the resulting effect set.

$$\frac{\begin{array}{l}\Gamma \vdash \nu : \tau^0_{op} \\ \Gamma, x : \tau^1_{op} \vdash c : \tau ! R \\ op \in R\end{array}}{\Gamma \vdash op(\nu; \lambda x.c) : \tau ! R} \qquad \frac{\begin{array}{l}\Gamma \vdash \nu : \underline{\tau}_1 \Rightarrow \underline{\tau}_2 \\ \Gamma \vdash c : \underline{\tau}_1\end{array}}{\Gamma \vdash \mathit{with}\ \nu\ \mathit{handle}\ c : \underline{\tau}_2}$$

2.2 Semantics

Following are the small-step operational semantics of the calculus taken from [2]. The rule for abstractions stays the same. With the computations we can always either get to *return* ν or $op(\nu; \lambda x.c)$ by floating out the operation call.

This makes the semantics of the handle computation easier since we only have to consider the cases of return and operation calls.

$$\frac{c_1 \rightsquigarrow c_1'}{(x \leftarrow c_1; c_2) \rightsquigarrow (x \leftarrow c_1'; c_2)} \qquad \frac{}{(x \leftarrow return\ \nu; c) \rightsquigarrow c[\nu/x]}$$

$$\frac{}{(x \leftarrow op(\nu; \lambda y.c_1); c_2) \rightsquigarrow op(\nu; \lambda y.(x \leftarrow c_1; c_2))}$$

$h := handler\ \{return\ x_r \rightarrow c_r, op_1(x_1; k_1) \rightarrow c_1, ..., op_n(x_n; k_n) \rightarrow c_n\}$, in the following rules:

$$\frac{c \rightsquigarrow c'}{with\ h\ handle\ c \rightsquigarrow with\ h\ handle\ c'} \qquad \frac{}{with\ h\ handle\ (return\ \nu) \rightsquigarrow c_r[\nu/x_r]}$$

$$\frac{op_i \in \{op_1, ..., op_n\}}{with\ h\ handle\ op_i(\nu; \lambda x.c) \rightsquigarrow c_i[\nu/x_i, (\lambda x.with\ h\ handle\ c)/k_i]}$$

$$\frac{op \notin \{op_1, ..., op_n\}}{with\ h\ handle\ op(\nu; \lambda x.c) \rightsquigarrow op(\nu; \lambda x.with\ h\ handle\ c)}$$

2.3 Examples

```
-- Assume the following effects, operations and operation signatures.
effect Flip {
  flip : () -> Bool
}

-- Defining generic effects for easier use
flip : () -> {flip} Bool
flip = \u. flip(); \x.return x)

-- Defining a handler for Flip
flipTrue : {flip} Bool => Bool
flipTrue = handler {
  return x -> return x
  flip(x; k) -> k True
}
```

```
-- A program that uses Flip
flipProgram : Bool -> {flip} Bool
flipProgram = \b.
  x <- flip ();
  b && x

-- Handling the program
flipProgramResult : () -> Bool
flipProgramResult =
  \u. with flipTrue handle (flipProgram False)
```

3 Algebraic effects with static instances

Static instances allow multiple "versions" of some effect to be used. This allows you to handle the same effect in multiple ways in the same program. We assume there is some statically known set of instances I, for each effect ε there is a set of instances $I^{\varepsilon} \subseteq I$. A single instance is denoted as ι and a set of instances as ι^*. We denote the pair of an instance and an operation as $\iota\#op$ and the set of pairs $\iota\#op^*$. We call operations on instances, using $\iota\#op$. The rules are taking from [3]. We add instances to the values and instance types to the value types. Instance types are an effect name together with a set of instances. We update the syntax of the previous calculus as follows:

$$
\begin{array}{lll}
\tau ::= \ldots & & \text{(value types)} \\
\quad \tau \to \underline{\tau} & & \text{(type of functions)} \\
\quad \underline{\tau} \Rightarrow \underline{\tau} & & \text{(type of handlers)} \\
\quad \varepsilon^{\iota^*} & & \text{(type of instances)} \\
\underline{\tau} ::= \tau ! R & & \text{(computation types)} \\
R ::= \iota\#op^* & & \text{(effect annotations)} \\
\nu ::= \ldots & & \text{(values)} \\
\quad \iota & & \text{(instances)} \\
\quad \mathit{handler}\,\{\mathit{return}\,x \to c, \nu_1\#op_1(x;k) \to c, \ldots, \nu_n\#op_n(x;k) \to c\} & & \text{(handler)} \\
c ::= \ldots & & \text{(computations)} \\
\quad \nu\#op(\nu; \lambda x.c) & & \text{(operation call)}
\end{array}
$$

3.1 Subtyping

The rules are fairly straightforward, for the subtyping of computation types we check that the left operations is a subset of the right operations.

$$\frac{}{() <: ()} \qquad \frac{\tau_3 <: \tau_1 \quad \underline{\tau}_2 <: \underline{\tau}_4}{\tau_1 \to \underline{\tau}_2 <: \tau_3 \to \underline{\tau}_4}$$

$$\frac{\underline{\tau}_3 <: \underline{\tau}_1 \quad \underline{\tau}_2 <: \underline{\tau}_4}{\underline{\tau}_1 \Rightarrow \underline{\tau}_2 <: \underline{\tau}_3 \Rightarrow \underline{\tau}_4} \qquad \frac{\tau_1 <: \tau_2 \quad R_1 \subseteq R_2}{\tau_1!R_1 <: \tau_2!R_2}$$

3.2 Typing rules

We had weakening rules to both value and computations.

$$\frac{\Gamma \vdash \nu : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash \nu : \tau_2} \qquad \frac{\Gamma \vdash c : \underline{\tau}_1 \quad \underline{\tau}_1 <: \underline{\tau}_2}{\Gamma \vdash c : \underline{\tau}_2}$$

To simplify the typing rules we allow instances to be polymorphic in the instance set.

$$\frac{\iota \in I^{\varepsilon} \quad \iota \in \iota^*}{\Gamma \vdash \iota : \varepsilon^{\iota^*}}$$

$$\frac{\begin{array}{l}\Gamma, x_r : \tau_1 \vdash c_r : \tau_2 \,;\iota\#op_2^* \\ \Gamma, x_i : \tau_{op_i}^0, k_i : \tau_{op_i}^1 \to \iota\#op_1^*\ \tau_2 \vdash c_i : \tau_2 \,;\iota\#op_2^* \\ \iota\#op_1^* \setminus \iota\#op_i^* \subseteq \iota\#op_2^*\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{handler}\,\{\mathit{return}\, x_r \to c_r, \iota_1\#op_1(x_1; k_1) \to c_1, ..., \iota_n\#op_n(x_n; k_n) \to c_n\} \\ \quad : \iota\#op_1^* \Rightarrow \tau_1 \iota\#op_2^* \tau_2\end{array}}$$

$$\frac{\begin{array}{l}\Gamma \vdash \nu_1 : \varepsilon^{\iota^*} \\ \Gamma \vdash \nu_2 : \tau_{op}^0 \\ \Gamma, x : \tau_{op}^1 \vdash c : \tau!R \\ \forall \iota \in \iota^*.\iota\#op \in R\end{array}}{\Gamma \vdash \nu_1\#op(\nu_2; \lambda x.c) : \tau!R}$$

3.3 Semantics

The small-step semantics stay very similar to the semantics of the basic algebraic effects, we just have to add the instances to the operation calls.

$$\frac{}{(x \leftarrow \iota\#op(\nu; \lambda y.c_1); c_2) \rightsquigarrow \iota\#op(\nu; \lambda y.(x \leftarrow c_1; c_2))}$$

$h := handler\ \{return\ x_r \to c_r, \iota_1\#op_1(x_1; k_1) \to c_1, ..., \iota_n\#op_n(x_n; k_n) \to c_n\}$, in the following rules:

$$\frac{\iota\#op_i \in \{\iota_1\#op_1, ..., \iota_n\#op_n\}}{with\ h\ handle\ \iota\#op_i(\nu; \lambda x.c) \rightsquigarrow c_i[\nu/x_i, (\lambda x.with\ h\ handle\ c)/k_i]}$$

$$\frac{\iota\#op \notin \{\iota_1\#op_1, ..., \iota_n\#op_n\}}{with\ h\ handle\ \iota\#op(\nu; \lambda x.c) \rightsquigarrow \iota\#op(\nu; \lambda x.with\ h\ handle\ c)}$$

3.4 Examples

```
-- Assume the following effects, operations and operation signatures.
effect Flip {
  flip : () -> Bool
}

-- Creating instances
instance Flip flip1
instance Flip flip2

-- Defining generic effects for easier use
flip1g : () -> {flip1#flip} Bool
flip1g = \u. flip1#flip(); \x.return x)

flip2g : () -> {flip2#flip} Bool
flip2g = \u. flip2#flip(); \x.return x)

-- Defining a handler for flip1 and flip2
flip1True : {flip1#flip} Bool => Bool
flip1True = handler {
  return x -> return x
  flip1#flip(x; k) -> k True
}

flip2True : {flip2#flip} Bool => Bool
flip2True = handler {
  return x -> return x
  flip2#flip(x; k) -> k True
```

```
}

-- A program that uses flip1 and flip2
flipProgram : () -> {flip1#flip, flip2#flip} Bool
flipProgram = \u.
  x <- flip1g ();
  y <- flip2g ();
  x && y

-- Handling the program
flipProgramResult : () -> Bool
flipProgramResult =
  \u. with flip2True handle
      with flip1True handle (flipProgram ())
```

4 Algebraic effects with first-class static instances

4.1 Typing rules

4.2 Semantics

4.3 Examples

```
-- Assume the following effects, operations and operation signatures.
effect Flip {
  flip : () -> Bool
}

-- Creating instances
instance Flip flip1
instance Flip flip2

-- Defining generic effects for easier use
flip1g : () -> {flip1#flip} Bool
flip1g = \u. flip1#flip(); \x.return x)

flip2g : () -> {flip2#flip} Bool
flip2g = \u. flip2#flip(); \x.return x)

-- Defining a handler for all flip instances
flipTrue : Inst ? -> ({f#flip} Bool => Bool)
flipTrue i = handler {
  return x -> return x
  i#flip(x; k) -> k True
}

-- A program that uses flip1 and flip2
flipProgram : () -> {flip1#flip, flip2#flip} Bool
flipProgram = \u.
  x <- flip1g ();
  y <- flip2g ();
  x && y
```

```
-- Handling the program
flipProgramResult : () -> Bool
flipProgramResult =
  \u. with (flipTrue flip1) handle
      with (flipTrue flip2) handle (flipProgram ())
```

5 Algebraic effects with dynamic instances

5.1 Typing rules

5.2 Semantics

5.3 Examples

References

[1] Levy, PaulBlain, John Power, and Hayo Thielecke. "Modelling environments in call-by-value programming languages." Information and computation 185.2 (2003): 182-210.

[2] Pretnar, Matija. "An introduction to algebraic effects and handlers. invited tutorial paper." Electronic Notes in Theoretical Computer Science 319 (2015): 19-35.

[3] Bauer, Andrej, and Matija Pretnar. "An effect system for algebraic effects and handlers." International Conference on Algebra and Coalgebra in Computer Science. Springer, Berlin, Heidelberg, 2013.