Implementation of static and dynamic semantics for a calculus with algebraic effects and handlers using PLT Redex

(Implementacja statycznej i dynamicznej semantyki rachunku z efektami algebraicznymi i ich obsługą z pomocą biblioteki PLT Redex)

Maciej Buszka

Praca inżynierska

Promotor: dr hab. Dariusz Biernacki

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

5 lutego 2019

Abstract

English abstract

Abstrakt w języku polskim

Contents

Chapter 1

Introduction

Algebraic effects [11] are an increasingly popular technique of structuring computational effects. They allow for seamless composition of multiple effects, while retaining (unlike monads) applicative style of programs. Coupled with handlers [12] which give programmers ability to interpret effects, they provide a disciplined and flexible tool for abstracting over a set of operations which a program may perform and for separating this interface from the semantics of those operations, defined as effect handlers.

As composability and separation of concerns are often sought after, many calculi and languages have been developed in order to get algebraic effects just right; most notable of them being: *Koka* [8] (featuring type inference, effect polymorphism with row-types and *JavaScript*-like syntax), *Links* [5] (featuring *ML*-like syntax, row-typed effect polymorphism and ad-hoc effects), *Helium* [2] (with abstract and local effects, *ML*-style module system and principled approach to effect polymorphism), *Eff* [1] (with implicit effect checking and recent work on direct compilation to *OCaml* [6]) and *Frank* [9] (with bidirectional type and effect system requiring minimal amount of effect variables, and shallow effect handlers).

On a more theoretical side, various approaches to semantics of algebraic effects can be spotted in the literature, both with respect to type systems and run-time semantics. Although most calculi use some form of row-types (with notable exception of *Frank* [9]) to track effects, there are differences in permitted shapes (at most one effect [5] of given type or many effects [3, 8], whether effects must be defined before use [3, 9, 8, 1] or not [5]) and how effects interact with polymorphism and abstraction. At run-time handlers can wrap the captured continuation (giving so-called deep handlers [3, 5, 8, 1]) or not (shallow handlers [9]) and the very act of finding the right handler can be implemented in various ways, mainly depending on some constructs which skip handlers [3].

All this variety naturally invites us to experiment with different features and components of a calculus. An executable model would be excellent for everyone interested in understanding inner workings of algebraic effects. In particular, the ability to perform and visualize reductions would be very helpful for understanding of the dynamic semantics of operations and handlers. Such model should also contain a type system complementing the dynamic semantics and guiding the programmer with usage of effects. In this thesis I will build such a calculus, describing my choices and discussing the trade-offs I faced.

The first goal of this thesis is the design of the calculus with effects and handlers, and its dynamic semantics. Its implementation should be executable and allow for step-by-step reduction of calculus terms by a computer program.

The second goal is the design and implementation of a sound type system for the calculus, preferably with type inference. The implementation should be able to type-check examples with minimal additional context and boilerplate.

The third goal is the design and implementation of abstract machine, which preserves the dynamic semantics of the calculus, yet is easier to translate into low-level virtual machine. It could also be used as a basis for compilation to native code.

In order to rapidly iterate on the design and test the calculus, I decided to use the `PLT Redex` library which allows for building language model with executable type system judgments and reduction relation. It also provides facilities for visualizing rewriting of terms and typesetting all components of the development.

The calculus is designed to be small enough to be easily understood, yet have general language features to allow experimentation with reasonably complex programs. The programmer can use λ-abstractions and recursive functions, numbers with addition, subtraction, multiplication and comparisons, booleans with conditional expressions and lists. The algebraic effects are implemented with ad-hoc operations which take one parameter and return a value, handlers which can handle multiple (different) operations at once and lifts which allow operations to skip handlers. The handler wraps captured continuation, giving the deep handling semantics. The type system for calculus is presented in Curry style, with the implementation inferring simple types for unannotated terms using unification algorithm. Although there is no way to create and bind polymorphic values, the system infers most general (simple) type for an expression which may contain unsolved variables. The abstract machine is implemented with explicit stack of continuations and value environment. It is given deterministic transition system using `PLT Redex`'s reduction relation and meta-function transforming a calculus expression into initial configuration. Additionally to ease experimentation I implemented a language front-end which translates human-friendly programs to calculus terms, integrated with `Racket` environment.

In brief summary, the development consists of:

- Executable reduction semantics most similar to the system of [3].
- Curry style type system with ad-hoc effects in the style of *Links*, effect rows based on *Koka* and *lift* construct of [3], implemented as a unification-based type inference algorithm.
- CEK style abstract machine with stack and *meta*-stack of handlers, based on [5]

The rest of this thesis is structured as follows: in the remainder of this chapter I introduce the main topics of this thesis, in chapter 2 I describe the calculus in greater detail, in chapter 3 I discuss technicalities of implementation and integration with the `Racket` environment and chapter 4 is user's manual.

1.1 Algebraic effects and handlers

Algebraic effects and handlers are a language level framework which allow for coherent presentation, abstraction, composition and reasoning about computational effects. The key idea is to separate invocation of an effectful operation in an expression from the meaning of such an operation. When one invokes an operation, current continuation (up to the nearest handler) is captured and passed along with the operation's argument to the nearest handler. The handler in turn may execute arbitrary expression, using the continuation once, twice, returning a function which calls the continuation or simply ignoring it. This way many control structures can be modeled and generalized by algebraic effects and appropriate handlers. For example, in the following Listing 1, function `exists` returns `true` when the list contains an element that satisfies predicate `p`.

```
let exists = λ p λ xs
  let f = λ x if p x then Break true else x end in
  handle map f xs with
  | Break x r -> true
  | return x  -> false
  end in
exists (λ x ==(x, 3)) (fromTo 0 10)
```

Listing 1: Exception-like usage of algebraic effects

It is implemented in terms of map and a helper function `f` which `Break`s normal control flow when the predicate returns true. This map is then invoked inside the handler that returns `true` on `Break` and `false` otherwise. This usage of operation and its handler is similar to exceptions, as the resumption is discarded. Another example, with a handler for the state-like operations is presented in Listing 2 in the next section.

From the language design standpoint algebraic effects provide single implementation of various phenomena which may happen during execution of a program, for example mutable state, I/O, environment lookup, exceptions, etc., in a sense that every effect is treated the same, the typing rules are defined for invocation of any operation, and handling of any operation. Similarly the operational semantics is also quite simple and succinct thanks to uniform treatment of various effects. This framework is also extendable. With small extension it can handle built-in effects in addition to user-defined ones.

From the language user perspective algebraic effects provide means of abstraction over effects used in a program. Thanks to easy creation of new effects, one can define special purpose operations and their handlers to better represent domain specific problems while simultaneously using well known effects, defined in the standard library. With effects being tracked by the type system, programmers can enforce purity or specific set of used effects at compile-time, or using effect polymorphism they can write reusable functions that abstract over effects which may happen. The separation of definition and implementation of effects allows for various interpretations of operations, for example simulating a database connection or file-system during testing.

1.2 Types and type inference

The most common approach to giving an effectful computation a type uses a type-level data-structure known as a row. Initially developed in order to structurally type records, rows come in two flavors: Remy-style[13] where they are treated as (finite) sets of label-type pairs, and Leijen-style[7] where rows are treated as (finite) lists of label-type pairs that are equivalent up to permutation of different labels. When polymorphism is present a row may have concrete prefix (possibly empty) and polymorphic tail denoted by a type variable. In effectful setting, the type system usually keeps a row of effects which an expression may perform, and suspended computations e.g. functions must have types decorated with a row of operations that may be invoked when their body is evaluated. In the following listing, the function `add` loads a number, sums it with the argument `x` and sets this sum, returning unit value.

```
let add = λ x Set +(Get (), x) in
let comp =
  handle add 5 with
  | Set x r -> λ _ r () x
  | Get _ r -> λ s r s s
  | return _ -> λ s s
  end in
comp 37
```

Listing 2: Stateful computation

It contains two effectful sub-expressions: $Get : Unit \Rightarrow Num$ and $Set\, Num \Rightarrow Unit$. As this function uses both operations, their effects must be combined, giving add following type: $Num \rightarrow (Get : Unit \Rightarrow Num, Set : Num \Rightarrow Unit)\, Unit$ where Num is the type of input, $Unit$ is the type of output and $(Get \ldots)$ is the effect row. The handler which interprets the operations is implemented to return a state transforming function. When the `Set` operation is invoked, the handler returns a function which will ignore its argument and first resume the computation with unit value, that will return a state transformation and then pass it the new state `x`. The `Get` operation handler returns a function which awaits for a state s with which it resumes the continuation and then applies returned function to s. The return clause returns identity state transformation. The type system requires all handler bodies to be of the same type: $Num \rightarrow ()\, Num$ which guides implementation and ensures that operations do not escape the handler.

Type inference is a technique of algorithmic reconstruction of types for various constructions used in a language. It allows programmers to write programs with no type annotations, that often feel redundant and obfuscate the meaning of a program. The most well known type system with inference is a system for *ML* family of languages[10] – *Haskell*, *OCaml*, *SML* which infers the types with no annotations whatsoever. A formal type system defines grammar of types consisting of base types (`int`, `bool` etc.), type constructors (arrows, algebraic data types) and type variables. The typing rules require types which should be compatible (e.g. formal parameter and argument types) to unify. The key feature of this system is the so-called let-polymorphism – generalization of types of let-bound variables. This way code reuse can be accomplished without complicating the type system and compromising type safety. The basis of implementation of this system is first order unification algorithm[10], which syntactically decomposes types and builds a substitution from type variables to types.

x ::= *variable-not-otherwise-mentioned* n ::= *number* v ::= $(\lambda\ x\ e)$ | n e ::= v | $(e\ e)$ | x K ::= [] | $(K\ e)$ | $(v\ K)$

Figure 1.1: λ-calculus abstract syntax

$K[((\lambda\ x\ e)\ v)] \longrightarrow K[\text{substitute}[\![e, x, v]\!]]$ [β]

Figure 1.2: λ-calculus reduction relation

1.3 Reduction semantics and abstract machines

Reduction semantics[4] is a format for specifying dynamic semantics of a calculus in an operational style. The basic idea is to first define redexes – expressions which can be reduced, and contexts in which the reduction can happen. Taking λ-calculus extended with numbers (Figure 1.1) and with call-by-value reduction order as an example, the only redex is application of a function to value $(\lambda x.e)v$ as shown in Figure 1.2. The possible contexts are: empty context $\square$ or evaluation of operator part of application Ke or evaluation of operand vK when left part has already been evaluated to a value. With these possibilities in mind, we will define binary relation $\longrightarrow$ which describes single step of reduction. Such relation can be thought of as a transition system, rewriting terms into 'simpler' ones step by step. There usually are two approaches to definition of such relation:

- Definition of primitive reduction $(\lambda x.e)v \longrightarrow_p e\{v/x\}$ which operates only on redexes and giving it a closure with the following inference rule:
  $$\frac{e \longrightarrow_p e'}{K[e] \longrightarrow K[e']}$$
  which says that if we can primitively reduce some expression, then we can do it in any context.
- Or definition of $\longrightarrow$ directly, with decomposition of terms on both sides: $K[(\lambda x.e)v] \longrightarrow K[e\{v/x\}]$

where the syntax $e\{v/x\}$ means term e with value v substituted for variable x, and $K[e]$ means some context K with expression e inserted into the hole. For both approaches it is important, that any term can be uniquely decomposed into redex and context, because when it is the case, then the relation is deterministic and gives good basis for formulation of abstract machines, interpreters or transformations to some other intermediate representations.

Abstract machine is a mathematical construction, usually defined as a set of

λ-calculus reduction example

Figure 1.3: λ-calculus example reduction sequence

CEK-machine for λ-calculus

Figure 1.4: λ-calculus abstract machine

configurations with deterministic transformations, which are computationally simple. The goal for formulation of an abstract machine is to mechanize evaluation of terms while retaining semantics given in a more abstract format, e.g. reduction semantics, with the correspondence being provable[4]. As an example, Figure 1.4 shows a *CEK*-machine for the λ-calculus defined earlier. The name *CEK* comes from *C*ommand, *E*nvironment and *K*ontinuation. The machine configuration is a triple (e, ρ, κ) where e is an expression which is decomposed or reduced, ρ is an environment mapping variables to values, and the last component κ is a continuation stack, which determines what will happen with value, to which first component eventually reduces. Thanks to the environment we no longer have to explicitly perform substitution, leading to more machine friendly and efficient implementation. Given an initial state, the machine can then repeatedly apply transformation relation, either looping, arriving at a final value, or getting stuck.

1.4 PLT Redex

The `PLT Redex`[4] library provides a comprehensive set of tools for the development of various calculi and language-like artifacts. The work begins with the definition of a language using a familiar BNF-like syntax. The library provides many options for defining patterns which describe the abstract syntax of the language, among them: meta-variables, symbols – playing the role of markers, numbers, object language variables, repetitions of patterns using ellipsis and nonlinear patterns which can for example enforce that all variables in a binding are different. Besides the syntax, the language definition allows for specifying the variable binding structure of the object language, which will be used by built-in meta-functions for substitution. The following Listing 3 shows code extending λ-calculus defined earlier in Figure 1.1 with typed terms E, along with types t and typing contexts Γ.

```
(define-extended-language LC-typed LC
  (E ::= (λ [x t] E) n (E E) x)
  (t ::= num (t → t))
  (Γ ::= · (x t Γ))

  #:binding-forms
  (λ [x t] E #:refers-to x))
```

Listing 3: Typed λ-calculus with numbers in `PLT Redex`

Second feature of the `PLT Redex` library are meta-functions which can pattern match on terms and return other terms. They may use full power of complex and non-linear patterns which the library exposes, additional side conditions and even escape to `Racket` – the host language in which the `PLT Redex` is defined.

```
(define-judgment-form LC-typed
  #:mode (types I I O)

  [(in x Γ t)
   -------------
   (types Γ x t)]

  [---------------
   (types Γ n num)]

  [(types (x t_1 Γ) e t_2)
   -----------------------------------
   (types Γ (λ [x t_1] e) (t_1 → t_2))]

  [(types Γ e_1 (t_1 → t_2))
   (types Γ e_2 t_1)
   -----------------------
   (types Γ (e_1 e_2) t_2)])
```

Listing 4: Type system for λ-calculus in `PLT Redex`

The third aspect of this library are judgment forms which encode inductively defined judgments (e.g. type systems) with a syntax similar to pen-and-paper rules. Listing 4 shows example code, defining a simple type system for annotated λ-calculus with numbers. These rules can use patterns, meta-functions, other judgments and also escape to `Racket`. When defining a judgment, the programmer must specify which parameters are to be treated as inputs and which as outputs using `#:mode` keyword. The library enforces an invariant that inputs `I` of a judgment form must be concrete terms and outputs `O` may be variables. Under the hood the `PLT Redex` library will resolve the rules in depth-first order backtracking on failures and multiple pattern matches.

The last component of language modeled using `PLT Redex` are reduction relations, usually used for specifying semantics. They are defined as a set of clauses, which should rewrite an input term into other term of same syntactic category. In order to find redex, the programmer can define evaluation contexts, and then the library will decompose the terms using `(in-hole K e)` pattern, as in Listing 5.

```
(reduction-relation
 LC
 (--> (in-hole K ((λ x e) v))
      (in-hole K (substitute e x v))
      β))
```

Listing 5: Reduction relation for λ-calculus in `PLT Redex`

Finally `PLT Redex` provides features for automatic testing via term generation facilities and has ability to typeset every component of calculus development. Every figure in this thesis, which shows language grammar, reduction relation, metafunction or judgment has been generated using `PLT Redex`.

Chapter 2

The calculus

The calculus implemented in this thesis is based on λ-calculus with call-by-value semantics. Its abstract syntax is presented in Figure 2.1. Meta-variable x ranges over variables used in value binders and their references, while *op* ranges over operation names, which are distinct from normal variables. Meta-variable v ranges over values, which are one of: boolean b, number m, λ-abstraction $(\lambda\, x\, e)$, recursive function $(rec\, x_f\, x_v\, e)$ or a list of values $(v \ldots)$. Meta-variable e ranges over expressions, which include values v, forms standard to λ-calculus – variables x, function applications $(e\, e)$, conditionals $(if\, e\, e\, e)$ and primitive operations $(prim\, e \ldots)$; they also include three constructs specific to effects – operation invocations $(op\, e)$, lifts $(lift\, op\, e)$ and handlers $(handle\, e\, hs\, ret)$ where ret is return expression $(return\, x\, e)$ and hs is a list of handler clauses.

To achieve call-by-value, left-to-right reduction order I use evaluation contexts E; this choice follows other calculi which allow for computational effects[3, 8, 5]. One interesting aspect of these contexts is notion of *free*ness [3], defined in Figure 2.2. The judgment $free[\,op,\, E,\, n]$ asserts that operation *op* is n-free in evaluation context E, meaning that it will be handled by $(n+1)$st handler for *op outside* the context E.

The syntax of types, ranged over by meta-variable t, comprises base types (Int, $Bool$), lists $List\, t$, arrow types $(t \to row\, t)$, operation types $(t \Rightarrow t)$, row types row and type variables a. Rows are defined inductively as either an empty row $\cdot$, a variable a or an extension $(op\ t\ row)$ of a row row with a type t assigned to an operation label *op*, and are ranged over by meta-variable row. Finally, meta-variable Γ ranges over typing contexts, S over type substitutions and SN denotes a pair of substitution and name supply N – a natural number used to generate fresh type variables.

```
        b ::= true | false
        m ::= number
        v ::= m | b | (λ x e) | (rec x x e) | (v ...)
     prim ::= + | - | * | == | <= | >= | cons | nil | cons? | nil? | hd | tl
        e ::= v | x | (app e e) | (if e e e) | (prim e ...)
            | (op e) | (handle e hs ret) | (lift op e)
       hs ::= ((op!_1 hexpr) ...)
    hexpr ::= (x!_1 x!_1 e)
      ret ::= (return x e)
        h ::= (op hexpr)
        t ::= Int | Bool | (t -> row t) | (List t) | (t => t) | row | a
      row ::= (op t row) | a | ·
        x ::= (variable-prefix v:)
        a ::= (variable-prefix t:)
       op ::= (variable-prefix op:)
        Γ ::= (x t Γ) | ·
        E ::= [] | (app E e) | (app v E) | (prim v ... E e ...) | (if E e e)
            | (op E) | (handle E hs ret) | (lift op E)
        S ::= (a t S) | ·
     N, n ::= natural
       SN ::= (S N)
```

Figure 2.1: Abstract syntax

2.1 Dynamic semantics

The dynamic semantics for a calculus with algebraic effects defines, besides the standard reductions known from λ-calculus, the control structure of operations and handlers. Intuitively, when an operation *op* is invoked, it will be handled by dynamically closest handler, with a caveat that for each lift passed in search of handler, it must skip one handler. Formally, when an operation *op* is invoked, it will be handled by lexically enclosing handler $(handle\, E[op\, e]\, hs\, ret)$ if and only if the intermediate context E is 0-free [3].

The dynamic semantics is defined in the format of contextual reduction semantics in Figure 2.3. All rules perform reduction in a context E leaving it unchanged. The first rule β-λ describes standard β-reduction via substitution of argument value for variable in function body, while the second β-*rec* is the β-reduction of recursive function, where first we substitute the function for function variable and then substitute the argument.

The rule *prim-op* deals with built-in primitive operations using helper metafunction *prim-apply* which pattern matches on *prim* and performs the appropriate operation. Next two rules *if-true* and *if-false* perform a choice of the correct branch in a conditional expression depending on the value of the condition. The rule *lift-compat* returns a value from a lift expression, leaving it unchanged.

The rule *handle-return* handles the case when the inner expression of a handle expression evaluates to a value, which means we have to evaluate the return clause by substituting the result value for x, and plugging this expression into the evaluation

$$\frac{}{\text{free}[\![op,\ [],\ 0]\!]}$$

$$\frac{\text{free}[\![op,\ E,\ n]\!]}{\text{free}[\![op,\ (\text{lift}\ op\ E),\ (+\ n\ \ 1)]\!]}$$

$$\frac{\text{free}[\![op,\ E,\ n]\!] \qquad op = op_{!\_1}}{\text{free}[\![op,\ (\text{lift}\ op_{!\_1}\ E),\ n]\!]}$$

$$\frac{\text{in}[\![op,\ \text{ops}[\![hs]\!]]\!] \qquad \text{free}[\![op,\ E,\ n]\!]}{\text{free}[\![op,\ (\text{handle}\ E\ hs\ ret),\ (-\ n\ \ 1)]\!]}$$

$$\frac{\text{not-in}[\![op,\ \text{ops}[\![hs]\!]]\!] \qquad \text{free}[\![op,\ E,\ n]\!]}{\text{free}[\![op,\ (\text{handle}\ E\ hs\ ret),\ n]\!]}$$

$$\frac{\text{free}[\![op,\ E,\ n]\!]}{\text{free}[\![op,\ (\text{app}\ E\ e),\ n]\!]}$$

$$\frac{\text{free}[\![op,\ E,\ n]\!]}{\text{free}[\![op,\ (\text{app}\ v\ E),\ n]\!]}$$

$$\frac{\text{free}[\![op,\ E,\ n]\!]}{\text{free}[\![op,\ (prim\ v\ \ldots\ E\ e\ \ldots),\ n]\!]}$$

$$\frac{\text{free}[\![op_1,\ E,\ n]\!]}{\text{free}[\![op_1,\ (op_2\ E),\ n]\!]}$$

$$\frac{\text{free}[\![op,\ E,\ n]\!]}{\text{free}[\![op,\ (\text{if}\ E\ e_1\ e_2),\ n]\!]}$$

Figure 2.2: Context *free*ness

$$
\begin{array}{rcll}
E[(\text{app } (\lambda\ x\ e)\ v)] & \longrightarrow & E[\text{substitute}\llbracket e, x, v\rrbracket] & [\beta\text{-}\lambda] \\
E[(\text{app } (\text{rec } x_f\ x_a\ e)\ v)] & \longrightarrow & E[\text{substitute}\llbracket \text{substitute}\llbracket e, x_f, (\text{rec } x_f\ x_a\ e)\rrbracket, x_a, v\rrbracket] & [\beta\text{-rec}] \\
E[(prim\ v\ \ldots)] & \longrightarrow & E[\text{prim-apply}\llbracket prim, v, \ldots\rrbracket] & [\text{prim-op}] \\
E[(\text{if true } e_1\ e_2)] & \longrightarrow & E[e_1] & [\text{if-true}] \\
E[(\text{if false } e_1\ e_2)] & \longrightarrow & E[e_2] & [\text{if-false}] \\
E[(\text{lift } op\ v)] & \longrightarrow & E[v] & [\text{lift-compat}] \\
E[(\text{handle } v\ hs\ (\text{return } x\ e))] & \longrightarrow & E[\text{substitute}\llbracket e, x, v\rrbracket] & [\text{handle-return}] \\
E_1[(\text{handle } E_2[(op\ v)]\ hs\ ret)] & \longrightarrow & E_1[\text{substitute}\llbracket \text{substitute}\llbracket e, x_1, v\rrbracket, & [\text{handle-op}] \\
 & & \quad x_2, (\lambda\ \text{v:z } (\text{handle } E_2[\text{v:z}]\ hs\ ret))\rrbracket] & \\
 & & \multicolumn{2}{l}{\text{where free}\llbracket op, E_2, 0\rrbracket, \text{get-handler}\llbracket op, hs, (x_1\ x_2\ e)\rrbracket, \text{v:z fresh}}
\end{array}
$$

Figure 2.3: Reduction relation

context E.

The last rule *handle-op* describes the behavior when an expression calls some operation. To handle an operation we must find a 0-free inner context E_2 which is directly surrounded by a handle expression which has a case for *op*. Then we substitute the value v for the first variable x_1 of the operation handler and the inner context E_2 surrounded with the very same handler (the continuation delimited by the handler) closed in a lambda for the second argument x_2. This wrapping gives deep handling semantics, as an operation call cannot escape from a handler in the resumption. The operation handler can resume the evaluation of the expression which invoked the handled operation using the function bound by x_2.

2.2 Static semantics

The type system is based on *Koka* [8] (Leijen's style of row types [7]), *Links* [5] (ad-hoc operations) and Biernacki et al's. [3] (lift construct) systems. Initially I implemented a variant of System F extended with row-types but it proved to be a bit of a mouthful to write in it even the simplest programs. Additionally the `PLT Redex`'s facilities for generation of terms were not able to generate sufficiently many well typed ones. As I intended on using the automated counterexample search to test assertions about the calculus, my goal was to minimize the amount of programs rejected by the type system. In order to accomplish this goal, type inference proved to be very helpful as it increased the amount of well typed terms tenfold.

To limit the amount of work and keep the calculus reasonably simple, I decided to present the calculus in the Curry style, with the typing relation inferring the type for unannotated terms, instead of implementing a separate type system and an inference algorithm.

Building on well known foundations[10], types are inferred via first-order unification. While the actual algorithm is presented in section 3.1, the notion of unification is used extensively in the remainder of this chapter and as such I will present an intuitive definition here. Two types t_1 and t_2 *unify* (written $t_1 \sim t_2$) if they are structurally the same, where variables can be substituted with any type. Two rows unify, if they are the same list of operation-type pairs, up to permutation of different operations.

The system does not feature polymorphism in a first-class fashion, as no polymorphic functions can be bound, mainly because there is no rule where types are generalized, but I believe it to be a straightforward addition, following the *Koka* [8] calculus. Still, after inferring the type of an expression, we can see which unification variables are left abstract and could be generalized. There are two main features differentiating this system from *Koka*'s; firstly effects need not be defined before use, their signature is inferred the same way as any other construction; secondly the system is algorithmic, with rules explicitly encoding a recursive function which can infer the type of an expression.

2.2.1 Type inference

The judgment $\Gamma \mid [S_1\, N_1] \vdash e \,:\, t\,!\,row \mid [S_2\, N_2]$ asserts that in a typing context Γ under a type substitution S_1, with name supply state N_1, expression e has type t with effects row under a type substitution S_2 and with a name supply state N_2. Algorithmically this judgment infers a type and an effect row, and calculates new substitution, given typing environment, current substitution and an expression. As in *ML* languages only simple types can be inferred, along with effect rows. The judgment rules are presented in Figure 2.4.

Base rules for constants (*Bool* and *Num*) check whether the value is of the appropriate category and the rule for variables *var* looks the type up in the environment Γ, each introducing fresh effect row variable. To check λ expression with rule λ, we first introduce fresh type variable, and then check the body in the extended environment. The arrow gets annotated with effects which may occur during evaluation of the body and the λ abstraction itself is returned with fresh effect row.

The recursive functions are checked using rule *rec* in a fashion similar to normal functions. First variable x_f denotes function itself, while second x_a its argument. Accordingly, the environment Γ gets extended with functional type $t_1 \to row_1 t_2$ for x_f and argument type t_1 for x_a, to check the body of the function. Afterwards the result type of body t gets unified with the result type of function t_2, same with effect row. The whole function, as it is a value, is returned with a fresh effect row.

The rule *app* for application requires the expression e_1 at function position to

$$\frac{\text{fresh-row}[\![N_1,\ row,\ N_2]\!]}{\Gamma \mid [S\ N_1] \vdash b : \text{Bool} \ !\ row \mid [S\ N_2]}\ [\text{Bool}]$$

$$\frac{\text{fresh-row}[\![N_1,\ row,\ N_2]\!]}{\Gamma \mid [S\ N_1] \vdash m : \text{Int} \ !\ row \mid [S\ N_2]}\ [\text{Num}]$$

$$\frac{\text{lookup}[\![\Gamma,\ x,\ t]\!] \qquad \text{fresh-row}[\![N_1,\ row,\ N_2]\!]}{\Gamma \mid [S\ N_1] \vdash x : t\ !\ row \mid [S\ N_2]}\ [\text{var}]$$

$$\frac{\begin{array}{c}\text{fresh-var}[\![N_1,\ t_1,\ N_2]\!] \qquad \text{fresh-row}[\![N_2,\ row_2,\ N_3]\!] \\ (x\ t_1\ \Gamma) \mid [S_1\ N_3] \vdash e : t_2\ !\ row_1 \mid SN\end{array}}{\Gamma \mid [S_1\ N_1] \vdash (\lambda\ x\ e) : (t_1 \text{ -> } row_1\ t_2)\ !\ row_2 \mid SN}\ [\lambda]$$

$$\frac{\begin{array}{c}\text{fresh-arr}[\![N_1,\ t_1,\ \text{->},\ row_1,\ t_2,\ N_2]\!] \qquad \text{fresh-row}[\![N_2,\ row_2,\ N_3]\!] \\ (x_f\ (t_1 \text{ -> } row_1\ t_2)\ (x_a\ t_1\ \Gamma)) \mid [S_1\ N_3] \vdash e : t\ !\ row \mid SN_1 \\ SN_1\ row_1 \sim row\ SN_2 \qquad SN_2\ t_2 \sim t\ SN_3\end{array}}{\Gamma \mid [S_1\ N_1] \vdash (\text{rec}\ x_f\ x_a\ e) : (t_1 \text{ -> } row_1\ t_2)\ !\ row_2 \mid SN_3}\ [\text{rec}]$$

$$\frac{\begin{array}{c}\Gamma \mid SN_1 \vdash e_1 : t_a\ !\ row_a \mid SN_2 \qquad \text{unify-arr}[\![SN_2,\ t_a,\ t_1,\ \text{->},\ row_1,\ t_2,\ SN_3]\!] \\ \Gamma \mid SN_3 \vdash e_2 : t_3\ !\ row_2 \mid SN_4 \\ SN_4\ t_1 \sim t_3\ SN_5 \qquad SN_5\ row_1 \sim row_2\ SN_6 \qquad SN_6\ row_1 \sim row_a\ SN_7\end{array}}{\Gamma \mid SN_1 \vdash (\text{app}\ e_1\ e_2) : t_2\ !\ row_2 \mid SN_7}\ [\text{app}]$$

$$\frac{\text{check-prim}[\![\Gamma,\ SN_1,\ prim,\ (e\ ...),\ t,\ row,\ SN_2]\!]}{\Gamma \mid SN_1 \vdash (prim\ e\ ...) : t\ !\ row \mid SN_2}\ [\text{prim}]$$

$$\frac{\begin{array}{c}\Gamma \mid SN_1 \vdash e_c : t_c\ !\ row_c \mid SN_2 \qquad SN_2\ t_c \sim \text{Bool}\ SN_3 \\ \Gamma \mid SN_3 \vdash e_t : t_t\ !\ row_t \mid SN_4 \qquad \Gamma \mid SN_4 \vdash e_e : t_e\ !\ row_e \mid SN_5 \\ SN_5\ t_t \sim t_e\ SN_6 \qquad SN_6\ row_c \sim row_t\ SN_7 \qquad SN_7\ row_t \sim row_e\ SN_8\end{array}}{\Gamma \mid SN_1 \vdash (\text{if}\ e_c\ e_t\ e_e) : t_t\ !\ row_t \mid SN_8}\ [\text{if}]$$

$$\frac{\begin{array}{c}\Gamma \mid SN_1 \vdash e : t_1\ !\ row_1 \mid [S_1\ N_1] \qquad \text{fresh-row}[\![N_1,\ row_2,\ N_2]\!] \qquad \text{fresh-var}[\![N_2,\ t_2,\ N_3]\!] \\ [S_1\ N_3]\ (op\ (t_1 \Rightarrow t_2)\ row_2) \sim row_1\ SN_2\end{array}}{\Gamma \mid SN_1 \vdash (op\ e) : t_2\ !\ row_1 \mid SN_2}\ [\text{op}]$$

$$\frac{\Gamma \mid SN_1 \vdash e : t\ !\ row \mid [S_1\ N_1] \qquad \text{fresh-var}[\![N_1,\ a,\ N_2]\!]}{\Gamma \mid SN_1 \vdash (\text{lift}\ op\ e) : t\ !\ (op\ a\ row) \mid [S_1\ N_2]}\ [\text{lift}]$$

$$\frac{\begin{array}{c}\Gamma \mid SN_1 \vdash e : t_1\ !\ row_1 \mid SN_2 \qquad (x\ t_1\ \Gamma) \mid SN_2 \vdash e_{ret} : t_{ret}\ !\ row_{ret} \mid SN_3 \\ \text{infer-handlers}[\![\Gamma,\ SN_3,\ t_{ret},\ hs,\ row_{out},\ row_{handled},\ SN_4]\!] \\ SN_4\ row_{out} \sim row_{ret}\ SN_5 \qquad SN_5\ row_1 \sim row_{handled}\ SN_6\end{array}}{\Gamma \mid SN_1 \vdash (\text{handle}\ e\ hs\ (\text{return}\ x\ e_{ret})) : t_{ret}\ !\ row_{out} \mid SN_6}\ [\text{handle}]$$

Figure 2.4: Type system

$$\frac{\text{fresh-row}[\![N_1,\ a_{handled},\ N_3]\!]}{\text{infer-handlers}[\![\Gamma,\ [S\ N_1],\ t,\ (),\ a_{handled},\ a_{handled},\ [S\ N_3]]\!]}\ \text{[tail]}$$

$$\frac{\begin{array}{c}\text{infer-handlers}[\![\Gamma,\ SN_1,\ t_{ret},\ (h\ ...),\ row_{out},\ row_{handled},\ [S\ N_1]]\!] \\ \text{fresh-var}[\![N_1,\ t_v,\ N_2]\!] \qquad \text{fresh-var}[\![N_2,\ t_r,\ N_3]\!] \\ (x_v\ t_v\ (x_r\ (t_r \text{ -> } row_{out}\ t_{ret})\ \Gamma))\ |\ [S\ N_3] \vdash e : t_h\ !\ row_h\ |\ SN_2 \\ SN_2\ row_{out} \sim row_h\ SN_3 \qquad SN_3\ t_{ret} \sim t_h\ SN_4\end{array}}{\text{infer-handlers}[\![\Gamma,\ SN_1,\ t_{ret},\ ((op\ (x_v\ x_r\ e))\ h\ ...),\ row_{out},\ (op\ (t_v \Rightarrow t_r)\ row_{handled}),\ SN_4]\!]}\ \text{[head]}$$

Figure 2.5: Handlers type inference

be of functional type and the expression e_2 at argument position to have a type that unifies with the parameter type of the function. All effect rows (from evaluation of e_1, e_2 and the body of the λ) must unify as well. Inference for primitive operation call in rule *prim* is deferred to auxiliary judgment *check-prim*, which checks arity and argument types, returning result type and usually fresh effect row. The rule *if* for conditional expression requires the condition to be of type *Bool* and types of two branches to unify. As usual all effect rows must also unify.

Operation invocation, checked by the rule *op*, requires the effect row to contain operation *op* with type $(t_1 \Rightarrow t_2)$ where input type t_1 is the inferred type for e and output type t_2 is fresh. The rule *lift*, checking operation lifting, prepends fresh *op* to the effect row of subexpression e.

Finally, to check handle expression with the rule *handle*, we infer the type t_1 of the enclosed expression e. Then in an environment extended with the type t_1 we infer the type t_{ret} of the return expression. Helper judgment *infer-handlers* returns the result effect row of handlers row_{out} and row marking handled effects $row_{handled}$ with the same tail as row_{out}. By unifying result row with return row and handled row with row_1 we ensure that effects which may occur during handling of operations, evaluation of return clause and leftovers from the inner expression are all accounted for.

2.2.2 Inference for effect handlers

List of effect handlers *hs* is processed right-to-left by the judgment

$$infer\text{-}handlers[\Gamma,\ SN_{in},\ t_{ret},\ hs,\ row_{out},\ row_{handled},\ SN_{out}]$$

presented in Figure 2.5. The t_{ret} is the type of the return clause, row_{out} is the combined row of effects which may occur in any handler and $row_{handled}$ is the row of handled operations, with appropriate types. The base case of empty list initializes both rows with the same type variable. This way $row_{handled}$ returned by the $infer\text{-}handlers$ judgment will consist of all handled operations and its tail will be row_{out}. The inductive case first calculates row_{out} and $row_{handled}$ for the tail of the

$$
\begin{aligned}
C &::= (e\ \rho\ \Sigma\ \phi\ K) \mid (\mathsf{val}\ V\ \rho\ \Sigma\ \phi\ K) \mid (op\ V\ n\ K\ K) \mid V \\
V &::= (\lambda\ \rho\ x\ e) \mid (\mathsf{rec}\ \rho\ x\ x\ e) \mid m \mid b \mid (V\ ...) \mid K \\
\rho &::= ([x\ V]\ ...) \\
\sigma &::= (\mathsf{arg}\ e\ \rho) \mid (\mathsf{app}\ V) \mid (\mathsf{do}\ op) \mid (prim\ \rho\ V\ ...\ /\ e\ ...) \mid (\mathsf{if}\ e\ e\ \rho) \\
\Sigma &::= (\sigma\ ...) \\
\phi &::= (\mathsf{handle}\ hs\ ret\ \rho) \mid (\mathsf{lift}\ op) \mid \mathsf{done} \\
\kappa &::= (\Sigma\ \phi) \\
K &::= (\kappa\ ...)
\end{aligned}
$$

Figure 2.6: Abstract machine configurations

$$\text{initial-conf}[\![e]\!] \ = \ (e\ ()\ ()\ \mathsf{done}\ ())$$

Figure 2.7: Abstract machine – initial configuration

list. Then, two new fresh variables are created: t_v for the type of value passed to handler and t_r for the type of resumption parameter. Resumption's result type is the result type of return clause as it should eventually evaluate to a value, which will be transformed by that clause. Its effect row is the same as the result row of whole handle expression, which means that any subsequent uses of operations will be handled by this handler. With the environment extended with t_v for the first parameter – an argument to the handler expression and $t_r \rightarrow row_{out}\, t_{ret}$ for second parameter – the resumption, we check the handler expression e. We then unify the effects of evaluating e with all effects of handlers, and result type t_h with the return type t_{ret}. Finally we extend the handled row with the current operation $(op(t_v \Rightarrow t_r))$.

2.3 Abstract machine

The abstract machine is based on the *CEK*-machine of Hillerström and Lindley[5], with the difference that during the search for operation handler, the machine must count handlers and lifts it passes. It is similar to the abstract machine presented in extended version of Biernacki et al's [3] as both calculi have the *lift* construct.

Possible configurations are given in Figure 2.6. Meta-variable C ranges over shapes of machine configurations, meta-variable V ranges over machine values, which are distinct from the calculus values, e.g., function values must now keep their environment ρ which maps variables to machine values. Additionally one of the possible values is a *meta-stack* which one may think of as a continuation captured during operation invocation. Meta-variable σ defines normal (or pure) continuation frames and Σ denotes pure continuation *stack*, while ϕ ranges over effect frames – `handle`, `lift` and `done` token which marks final continuation. Meta-variable κ denotes meta-frame which consists of a pure stack and one effect frame. Finally K ranges over stacks of meta-frames, which I will call *meta-stack*s. Meta-function *initial-conf* in Figure 2.7 transforms an expression into initial configuration – initializing machine with empty stack, `done` effect frame and empty meta-stack.

The first group of transitions depicted in Figure 2.8 performs mostly adminis-

$$
\begin{aligned}
((\lambda\ x\ e)\ \rho\ \Sigma\ \phi\ K) &\longrightarrow (\mathsf{val}\ (\lambda\ \rho\ x\ e)\ \rho\ \Sigma\ \phi\ K)\\
((\mathsf{rec}\ x_f\ x_a\ e)\ \rho\ \Sigma\ \phi\ K) &\longrightarrow (\mathsf{val}\ (\mathsf{rec}\ \rho\ x_f\ x_a\ e)\ \rho\ \Sigma\ \phi\ K)\\
(m\ \rho\ \Sigma\ \phi\ K) &\longrightarrow (\mathsf{val}\ m\ \rho\ \Sigma\ \phi\ K)\\
(b\ \rho\ \Sigma\ \phi\ K) &\longrightarrow (\mathsf{val}\ b\ \rho\ \Sigma\ \phi\ K)\\
((prim)\ \rho\ \Sigma\ \phi\ K) &\longrightarrow (\mathsf{val}\ \mathsf{prim\text{-}apply}[\![prim]\!]\ \rho\ \Sigma\ \phi\ K)\\
(\mathsf{val}\ V\ \rho_1\ ([\mathsf{arg}\ e\ \rho_2]\ \sigma\ \ldots)\ \phi\ K) &\longrightarrow (e\ \rho_2\ ([\mathsf{app}\ V]\ \sigma\ \ldots)\ \phi\ K)\\
(\mathsf{val}\ V\ \rho_1\ ([prim\ \rho_2\ V_1\ \ldots\ /\ e\ e_1\ \ldots]\ \sigma\ \ldots)\ \phi\ K) &\longrightarrow (e\ \rho_2\ ([prim\ \rho_2\ V_1\ \ldots\ V\ /\ e_1\ \ldots]\ \sigma\ \ldots)\ \phi\ K)\\
(\mathsf{val}\ V\ \rho\ ([\mathsf{do}\ op]\ \sigma\ \ldots)\ \phi\ (\kappa\ \ldots)) &\longrightarrow (op\ V\ 0\ ([(\sigma\ \ldots)\ \phi]\ \kappa\ \ldots)\ ())\\
(\mathsf{val}\ V\ \rho\ ()\ \mathsf{done}\ ()) &\longrightarrow V
\end{aligned}
$$

Figure 2.8: Abstract machine – administrative transitions

trative functions – capturing environment for functional values, transforming from value terms to machine values, sequencing binary operations, switching to special configuration for search of handler and a transition for returning final value.

$$
\begin{aligned}
((\mathsf{app}\ e_1\ e_2)\ \rho\ (\sigma\ \ldots)\ \phi\ K) &\longrightarrow (e_1\ \rho\ ([\mathsf{arg}\ e_2\ \rho]\ \sigma\ \ldots)\ \phi\ K)\\
((op\ e)\ \rho\ (\sigma\ \ldots)\ \phi\ K) &\longrightarrow (e\ \rho\ ([\mathsf{do}\ op]\ \sigma\ \ldots)\ \phi\ K)\\
((prim\ e\ e_1\ \ldots)\ \rho\ (\sigma\ \ldots)\ \phi\ K) &\longrightarrow (e\ \rho\ ([prim\ \rho\ /\ e_1\ \ldots]\ \sigma\ \ldots)\ \phi\ K)\\
((\mathsf{if}\ e_{cond}\ e_{then}\ e_{else})\ \rho\ (\sigma\ \ldots)\ \phi\ K) &\longrightarrow (e_{cond}\ \rho\ ([\mathsf{if}\ e_{then}\ e_{else}\ \rho]\ \sigma\ \ldots)\ \phi\ K)\\
((\mathsf{handle}\ e\ hs\ ret)\ \rho\ \Sigma\ \phi\ (\kappa\ \ldots)) &\longrightarrow (e\ \rho\ ()\ (\mathsf{handle}\ hs\ ret\ \rho)\ ([\Sigma\ \phi]\ \kappa\ \ldots))\\
((\mathsf{lift}\ op\ e)\ \rho\ \ \Sigma\ \phi\ (\kappa\ \ldots)) &\longrightarrow (e\ \rho\ ()\ (\mathsf{lift}\ op)\ ([\Sigma\ \phi]\ \kappa\ \ldots))
\end{aligned}
$$

Figure 2.9: Abstract machine – continuation building

The second group of transitions (Figure 2.9) decomposes current expression and builds continuation by growing either stack or meta-stack. First four rules – for function application, operation invocation, primitive operation call and conditional expression – all create a new pure frame and push it onto stack. Last two transitions deal with effectful operations – handle and lift; they build meta-stack by bundling previous effect frame with current stack into meta-frame, pushing it onto meta-stack and then installing a fresh stack and a new effect frame into the machine configuration.

The third group of transitions (Figure 2.10) perform various reductions. The first rule looks up value of a variable in current environment, the second rule performs contraction of a normal function, by extending the environment with mapping from the function's formal parameter x to the calculated value V. The third rule reduces recursive function, extending the environment with the argument value V and the function value, allowing for recursive calls. The last rule handling application deals

$(x\ \rho\ \Sigma\ \phi\ K) \longrightarrow$
(val lookup-ρ$[\![\rho, x]\!]\ \rho\ \Sigma\ \phi\ K$)

(val $V\ \rho_2$ ([app (λ $\rho_1\ x\ e$)] σ ...) $\phi\ K$) $\longrightarrow$
(e extend$[\![\rho_1, x, V]\!]$ (σ ...) $\phi\ K$)

(val $V\ \rho_2$ ([app (rec $\rho_1\ x_f\ x_a\ e$)] σ ...) $\phi\ K$) $\longrightarrow$
(e extend$[\![$extend$[\![\rho_1, x_f,$ (rec $\rho_1\ x_f\ x_a\ e$)$]\!], x_a, V]\!]$ (σ ...) $\phi\ K$)

(val $V\ \rho$ ([app ([$\Sigma\ \phi_1$] κ_1 ...)] σ ...) ϕ_2 (κ_2 ...)) $\longrightarrow$
(val $V\ \rho\ \Sigma\ \phi_1$ (κ_1 ... [(σ ...) ϕ_2] κ_2 ...))

(val $V\ \rho_1$ ([$prim\ \rho_2\ V_1$ ... /] σ ...) $\phi\ K$) $\longrightarrow$
(val prim-apply$[\![prim, V_1, ..., V]\!]\ \rho_2$ (σ ...) $\phi\ K$)

(val true ρ_1 ([if e any ρ] σ ...) $\phi\ K$) $\longrightarrow$
($e\ \rho$ (σ ...) $\phi\ K$)

(val false ρ_1 ([if any e ρ] σ ...) $\phi\ K$) $\longrightarrow$
($e\ \rho$ (σ ...) $\phi\ K$)

Figure 2.10: Abstract machine – contractions

with continuation resumption, by pushing the current stack and the effect frame ($[(\sigma \ldots)\ \phi_1]$) onto the meta-stack, installing the top ($[\Sigma\ \phi_2]$) of captured meta-stack and prepending its tail ($(\kappa_1 \ldots)$) to the meta-stack.

($op\ V\ n$ ([Σ (lift op)] κ_1 ...) (κ_2 ...)) $\longrightarrow$
($op\ V$ `(+ n 1)` (κ_1 ...) (κ_2 ... [Σ (lift op)]))

($op_1\ V\ n$ ([Σ (lift op_2)] κ_1 ...) (κ_2 ...)) $\longrightarrow$
($op\ V$ `(+ n 1)` (κ_1 ...) (κ_2 ... [Σ (lift op)]))
where $op_1 = op_{!\_1}$, $op_2 = op_{!\_1}$

($op\ V\ n$ ([Σ (handle hs ret ρ)] κ_1 ...) (κ_2 ...)) $\longrightarrow$
($op\ V$ `(- n 1)` (κ_1 ...) (κ_2 ... [Σ (handle hs ret ρ)]))
where in$[\![op,$ ops$[\![hs]\!]]\!]$, `(> n 0)`

($op\ V\ n$ ([Σ (handle hs ret ρ)] κ_1 ...) (κ_2 ...)) $\longrightarrow$
($op\ V\ n$ (κ_1 ...) (κ_2 ... [Σ (handle hs ret ρ)]))
where not-in$[\![op,$ ops$[\![hs]\!]]\!]$

($op\ V$ 0 ([Σ (handle hs ret ρ)] [$\Sigma_2\ \phi$] κ_1 ...) (κ_2 ...)) $\longrightarrow$
(e extend$[\![$extend$[\![\rho, x_1, V]\!], x_2,$ (κ_2 ... [Σ (handle hs ret ρ)])$]\!]\ \Sigma_2\ \phi$ (κ_1 ...))
where get-handler$[\![op, hs, (x_1\ x_2\ e)]\!]$

(val $V\ \rho_1$ () (handle hs (return $x\ e$) ρ) ([$\Sigma\ \phi$] κ ...)) $\longrightarrow$
(e extend$[\![\rho, x, V]\!]\ \Sigma\ \phi$ (κ ...))

(val $V\ \rho$ () (lift op) ([$\Sigma\ \phi$] κ ...)) $\longrightarrow$
(val $V\ \rho\ \Sigma\ \phi$ (κ ...))

Figure 2.11: Abstract machine – effect handling

Transitions in the last group (Figure 2.11) form the essence of this machine, performing all tasks related to effect handling. First four rules search for the appropriate handler for *op* by maintaining a counter n which is incremented by every lift for *op* and decremented by every handler for *op*. The fifth rule matches when the counter is 0 and the handler has a case for the operation which was invoked. In this situation, the machine must begin evaluation of the handling expression in the environment extended both with the passed argument, and the captured continuation with current meta-frame appended. Two last rules deal with returning values – in case of a handler the return clause is installed, and in case of a lift its frame is simply discarded.

Chapter 3

Implementation

3.1 Unification

The unification algorithm is loosely based on [10], but instead of keeping a set of constraints to be solved, it solves sub-problems recursively. It is implemented as a `PLT Redex` judgment, which takes two types, the initial substitution and returns the substitution extended with their unifier. This judgment must also pass around the name supply token – a natural number which is incremented every time a new type variable is created. Variables in types are substituted lazily – whenever algorithm encounters variable which is in domain of the substitution, it looks it up and continues unification. The Figure 3.1 shows the unification algorithm.

Terminal rules *refl-var* and *refl-const* handle unification of equal variables and constants respectively. In a special case *refl-var-lookup*, when both types are variables, the left variable a_1 is not in the domain of substitution and the right variable a_2 is mapped to a_3 which is equal to a_1, then they are in fact the same variable and should unify. This corner-case arises due to lazy application of substitution to types.

The rule *ext* extends substitution, when the left-hand-side type is a variable which is not in the domain of substitution. It is important to note, that the type which is inserted into substitution is fully substituted. This way we can maintain an invariant, that every type in the substitution has only type variables which are not in the domain of the substitution. The last side condition, that a is not a free type variable in t is the occurs-check, which ensures that algorithm doesn't try to unify cyclic types, which would loop.

When the left type is a variable, in the domain of substitution, rule *lookup* looks it up, and continues unification. When the left type is not a variable, and right type is, rule *flip* flips them around and continues unification. This rule is needed because the judgment performs extension and lookup only on left variable. The next three rules ->, => and *List* decompose the type constructor and unify all respective sub-types.

The last rule *row* unifies two rows. It requires the left row to be non empty, and the right row not to be a variable (to ensure determinism with respect to the rule *flip*). Then it uses helper judgment *unify-row* which rewrites row_2 such that its head is *op* t_2 and rest is bound row_r. The side condition, requiring the variable at the end of (substituted) row_1 not to be in domain of substitution which was build by the rewriting ensures that algorithm does not try to unify rows with different labels, but same type variable in the tail (similar to occurs-check in rule *ext*).

The helper judgment *unify-row*, shown in Figure 3.2 rewrites the row such that it begins with desired label *op*. It is based on similar judgment in [7]. There are two base cases: either the row already begins with *op* (rule *row-head*) and is decomposed and returned or the row is already a variable which is not bound by substitution (rule *row-var*). In this case the substitution is extended with a row ($op\, t\, row$) where both t and row are fresh variables.

The third rule *row-lookup* handles variables which are bound by the substitution, looking up the appropriate row and continuing row rewriting. The last rule *row-swap* matches rows that have a label $o1$ different to desired label o. In this case, the rest of the row must rewritten recursively yielding type t_2 for o and new tail. The type is returned and the tail is extended with original head.

3.2 Automatic testing

3.3 The `Racket` environment

$$\frac{}{SN\ a \sim a\ SN}\ [\text{refl-var}]$$

$$\frac{\text{in}\llbracket t,\ (\text{Int Bool}\ \cdot)\rrbracket}{SN\ t \sim t\ SN}\ [\text{refl-const}]$$

$$\frac{\text{not-in}\llbracket a_1,\ \text{dom-S}\llbracket S\rrbracket\rrbracket \qquad a_3 = \text{apply}\llbracket S,\ a_2\rrbracket \qquad \text{eq}\llbracket a_1,\ a_3\rrbracket}{[S\ N]\ a_1 \sim a_2\ [S\ N]}\ [\text{refl-var-lookup}]$$

$$\frac{\text{not-in}\llbracket a,\ \text{dom-S}\llbracket S\rrbracket\rrbracket \qquad t = \text{apply}\llbracket S,\ t_1\rrbracket \qquad \text{not-in}\llbracket a,\ \text{ftv}\llbracket t\rrbracket\rrbracket}{[S\ N]\ a \sim t_1\ [\text{ext}\llbracket a,\ t,\ S\rrbracket\ N]}\ [\text{ext}]$$

$$\frac{\text{in}\llbracket a,\ \text{dom-S}\llbracket S\rrbracket\rrbracket \qquad [S\ N]\ \text{apply}\llbracket S,\ a\rrbracket \sim t\ SN}{[S\ N]\ a \sim t\ SN}\ [\text{lookup}]$$

$$\frac{\text{not-var}\llbracket t\rrbracket \qquad SN_1\ a \sim t\ SN_2}{SN_1\ t \sim a\ SN_2}\ [\text{flip}]$$

$$\frac{SN_1\ t_{1\text{-}l} \sim t_{1\text{-}r}\ SN_2 \qquad SN_2\ row_l \sim row_r\ SN_3 \qquad SN_3\ t_{2\text{-}l} \sim t_{2\text{-}r}\ SN_4}{SN_1\ (t_{1\text{-}l}\ \text{->}\ row_l\ t_{2\text{-}l}) \sim (t_{1\text{-}r}\ \text{->}\ row_r\ t_{2\text{-}r})\ SN_4}\ [\text{->}]$$

$$\frac{SN_1\ t_{1\text{-}l} \sim t_{1\text{-}r}\ SN_2 \qquad SN_2\ t_{2\text{-}l} \sim t_{2\text{-}r}\ SN_3}{SN_1\ (t_{1\text{-}l}\ \text{=>}\ t_{2\text{-}l}) \sim (t_{1\text{-}r}\ \text{=>}\ t_{2\text{-}r})\ SN_3}\ [\text{=>}]$$

$$\frac{SN_1\ t_1 \sim t_2\ SN_2}{SN_1\ (\text{List}\ t_1) \sim (\text{List}\ t_2)\ SN_2}\ [\text{List}]$$

$$\frac{\begin{array}{c}\text{not-var}\llbracket row_2\rrbracket \qquad \text{unify-row}\llbracket [S_1\ N_1],\ row_2,\ op,\ t_2,\ row_r,\ [S_2\ N_2]\rrbracket \\ a = \text{last}\llbracket \text{apply}\llbracket S_1,\ row_1\rrbracket\rrbracket \qquad \text{not-in}\llbracket a,\ \text{dom-S}\llbracket S_2\rrbracket\rrbracket \\ [S_2\ N_2]\ t_1 \sim t_2\ SN_3 \qquad SN_3\ row_1 \sim row_r\ SN_4\end{array}}{[S_1\ N_1]\ (op\ t_1\ row_1) \sim row_2\ SN_4}\ [\text{row}]$$

Figure 3.1: Unification algorithm

$$\frac{}{\text{unify-row}[\![SN, (op\ t\ row), op, t, row, SN]\!]}\ \text{[row-head]}$$

$$\frac{\text{not-in}[\![a, \text{dom-S}[\![S]\!]]\!] \qquad \text{fresh-row}[\![N_1, row, N_2]\!] \qquad \text{fresh-var}[\![N_2, t, N_3]\!]}{\text{unify-row}[\![[S\ N_1], a, op, t, row, [\text{ext}[\![a, (op\ t\ row), S]\!]\ N_3]]\!]}\ \text{[row-var]}$$

$$\frac{\text{in}[\![a, \text{dom-S}[\![S]\!]]\!] \qquad \text{unify-row}[\![[S\ N], \text{apply}[\![S, a]\!], op, t, row, SN]\!]}{\text{unify-row}[\![[S\ N], a, op, t, row, SN]\!]}\ \text{[row-lookup]}$$

$$\frac{\begin{array}{c}\text{unify-row}[\![SN_1, row_1, \text{o}, t_2, row_2, SN_2]\!] \\ \text{o1} = op_{!\_1} \qquad \text{o} = op_{!\_1}\end{array}}{\text{unify-row}[\![SN_1, (\text{o1}\ t_1\ row_1), \text{o}, t_2, (\text{o1}\ t_1\ row_2), SN_2]\!]}\ \text{[row-swap]}$$

Figure 3.2: Row unification algorithm

Chapter 4

User's manual

Bibliography

[1] Andrej Bauer and Matija Pretnar. ?Programming with Algebraic Effects and Handlers? In: *CoRR* abs/1203.1539 (2012). arXiv: `1203.1539`. URL: `http://arxiv.org/abs/1203.1539`.

[2] Dariusz Biernacki et al. ?Abstracting Algebraic Effects? In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 6:1–6:28. ISSN: 2475-1421. DOI: `10.1145/3290319`. URL: `http://doi.acm.org/10.1145/3290319`.

[3] Dariusz Biernacki et al. ?Handle with Care: Relational Interpretation of Algebraic Effects and Handlers? In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 8:1–8:30. ISSN: 2475-1421. DOI: `10.1145/3158096`. URL: `http://doi.acm.org/10.1145/3158096`.

[4] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. 1st. The MIT Press, 2009. ISBN: 0262062755, 9780262062756.

[5] Daniel Hillerström and Sam Lindley. ?Liberating Effects with Rows and Handlers? In: *Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. Nara, Japan: ACM, 2016, pp. 15–27. ISBN: 978-1-4503-4435-7. DOI: `10.1145/2976022.2976033`. URL: `http://doi.acm.org/10.1145/2976022.2976033`.

[6] Oleg Kiselyov and KC Sivaramakrishnan. ?Eff Directly in OCaml? In: *arXiv e-prints*, arXiv:1812.11664 (Dec. 2018), arXiv:1812.11664. arXiv: `1812.11664 [cs.PL]`.

[7] Daan Leijen. ?Extensible records with scoped labels? In: *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05), Tallin, Estonia*. Sept. 2005. URL: `https://www.microsoft.com/en-us/research/publication/extensible-records-with-scoped-labels/`.

[8] Daan Leijen. ?Koka: Programming with Row Polymorphic Effect Types? In: *Mathematically Structured Functional Programming 2014*. EPTCS, Mar. 2014. URL: `https://www.microsoft.com/en-us/research/publication/koka-programming-with-row-polymorphic-effect-types-2/`.

[9] Sam Lindley, Conor McBride, and Craig McLaughlin. ?Do Be Do Be Do? In: *SIGPLAN Not.* 52.1 (Jan. 2017), pp. 500–514. ISSN: 0362-1340. DOI: `10.1145/3093333.3009897`. URL: `http://doi.acm.org/10.1145/3093333.3009897`.

[10] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091, 9780262162098.

[11] Gordon Plotkin and John Power. ?Algebraic Operations and Generic Effects? In: *Applied Categorical Structures* 11.1 (Feb. 2003), pp. 69–94. ISSN: 1572-9095. DOI: `10.1023/A:1023064908962`. URL: `https://doi.org/10.1023/A:1023064908962`.

[12] Gordon Plotkin and Matija Pretnar. ?Handling Algebraic Effects? In: *Logical Methods in Computer Science* 9.4 (Dec. 2013). Ed. by Andrzej Tarlecki. ISSN: 1860-5974. DOI: `10.2168/lmcs-9(4:23)2013`. URL: `http://dx.doi.org/10.2168/LMCS-9(4:23)2013`.

[13] Didier Rémy. ?Theoretical Aspects of Object-oriented Programming? In: ed. by Carl A. Gunter and John C. Mitchell. Cambridge, MA, USA: MIT Press, 1994. Chap. Type Inference for Records in Natural Extension of ML, pp. 67–95. ISBN: 0-262-07155-X. URL: `http://dl.acm.org/citation.cfm?id=186677.186689`.