Monadification of Attribute Grammars

Dawn Michaelson
micha576@umn.edu
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, MN, USA

Eric Van Wyk
evw@umn.edu
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, MN, USA

Abstract

We describe a monadification process for attribute grammars for more concisely written attribute equations, closer to the style of inference rules used in traditional typing and evaluation specifications. Inference rules specifying, for example, a typing relation typically consider only typable expressions, whereas well-defined attribute grammars explicitly determine attribute values for any term, including untypable ones. The monadification approach lets one represent, for example, types as monadic optional/maybe values, but write non-monadic equations over the value inside the monad that only specify the rules for a correct typing, leading to more concise specifications. The missing failure cases are handled by a rewriting that inserts monadic return, bind, and failure operations to produce a well-defined attribute grammar that handles untypable trees. Thus, one can think in terms of a type T and not the actual monadic type $M(T)$. To formalize this notion, typing and evaluation relations are given for the original and rewritten equations. The rewriting is total, preserves types, and a correctness property relating values of original and rewritten equations is given. A prototype implementation illustrates the benefits with examples such as typing of the simply-typed lambda calculus with Booleans, evaluation of the same, and type inference in Caml Light.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**.

Keywords: attribute grammars, monads, monadification

ACM Reference Format:
Dawn Michaelson and Eric Van Wyk. 2020. Monadification of Attribute Grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3426425.3426941

```
Expr: t ::=  x          (variable)
             \x:T. t    (abstraction)
             t t        (application)
Type: T ::=  T → T      (arrow type)
```

$$\frac{\Gamma; x : T_1 \vdash body : T_2}{\Gamma \vdash \lambda x : T_1 .\, body : T_1 \rightarrow T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash f : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash a : T_{11}}{\Gamma \vdash f\ a : T_{12}} \quad \text{(T-APP)}$$

Figure 1. Syntax of the simply-typed lambda calculus and inference rules for typing abstractions and applications.

1 Introduction and Motivation

Inference rules used in specifying type systems [20] and in structural operational semantics (SOS) [21] provide a clear and concise method for describing analyses over programming languages in part because only the cases where the analysis is successful are explicitly considered. For example, in typing a program, only rules for identifying well-typed programs are considered; the drawback to this approach is that this does not give any feedback when the program is ill-typed and thus there exists no typing derivation for the program. Contrast this with attribute grammars (AGs), which evaluate attributes for all syntax trees, even for ill-formed trees such as when there is a type error. This is necessary for providing feedback on erroneous programs, but handling error cases explicitly clutters the specification, obscuring the cases where the analysis succeeds and making reasoning about the specification more difficult. Monads in functional languages provide a mechanism for handling error cases implicitly, but are still more verbose and less straightforward than inference rules. The work presented here allows for the writing of specifications with the concision and clarity of inference rules; these are rewritten to insert monadic operations to handle errors (monadification). This provides specifications that enjoy the benefits of both approaches.

Motivating Example. In Figure 1, we consider typing and evaluation in the simply-typed lambda calculus with Boolean operations. In this figure are two inference rules for typing abstractions (λ-expressions) and function applications. The T-ABS rule cleanly and concisely describes the

```
 1  arrow: ty:Type ::= in:Type out:Type { }
 2  abs: t:Expr ::= x:String Ty:Type body:Expr
 3  { body.env = (x,Ty) :: t.env;
 4    -- using just and nothing directly
 5    t.type = match body.type with
 6               | just(T2) -> just(arrow(Ty, T2))
 7               | nothing() -> nothing()
 8    -- using explicit monads
 9    t.type = body.type >>=
10               (\T2:Type. return(arrow(Ty, T2)))
11    -- using do-notation
12    t.type = do { T2 <- body.type;
13                  return (arrow(Ty, T2)) }
14    -- using implicit monads
15    t.type = arrow(Ty, body.type) }
```

Figure 2. Attribute grammar productions for types and various equations for typing abstractions in the λ-calculus. Here the attribute `type` has type `Maybe(Type)`.

case when an abstraction is typable, making it easy to read and reason about. It states that if the typing environment Γ extended with the binding giving x the type T_1 can be used to show that the body of the λ-abstraction has type T_2, then the abstraction will have type $T_1 \to T_2$. Unfortunately, if the abstraction is not typable, this rule gives us no information.

In Figure 2 we show four attribute grammar equations for defining the type of an abstraction to illustrate the monadification process proposed in this paper. On line 1 is an (abstract) production for representing function types, corresponding to that in Figure 1. Next is the production for abstractions. Here production names precede the production signature, where nonterminal symbols are labeled for accessing attributes on the corresponding tree nodes in the production's equations. The first equation, for `body.env` on line 3, specifies the typing environment, represented as a list of pairs of names and types, for the body of the abstraction; it has the abstraction's bound name (`x`) and the type of this name (`Ty`) added to the environment given to the abstraction itself (`t.env`). Because AGs must handle trees that may not be typable, the type of the `type` attribute on expressions is `Maybe(Type)` which encodes success as a `just(_)` value and failure as `nothing()`. The equation on lines 5–7 defines the `type` attribute for abstractions by pattern matching on `body.type`. If the body was successfully typed, this value is `just(T2)` for some type `T2`, and we can successfully type the full abstraction as the function type `arrow(Ty, T2)`, wrapped in `just()`. If the body was not typed, `body.type` is `nothing()`, then the abstraction cannot be typed, so the result is `nothing()`. While this is complete, the failure case and the necessity of considering whether something was a `just()` or a `nothing()` interferes with understanding the success case and makes any reasoning about it more difficult.

The equation on lines 9–10 makes explicit use of `Maybe()` as a monad. A monad datatype `M(T)` includes operations $\gg=$ `: M(T)` $\to$ `(T` $\to$ `M(S))` $\to$ `M(S)`, called bind, which unwraps a value and provides it to a function or propagates failure, and `return : T` $\to$ `M(T)`. For the `Maybe()` monad, if the monadic value to bind is `just(`v`)` it applies the function argument to v, otherwise it returns `nothing()`. This matches the behavior in the previous case. While it hides the failure case, the bind and function are a bit clumsy and obscure the intention slightly.

Lines 12–13 are based on Haskell's do-notation for easier specification of monadic computations. This is better than the explicit case above, but we need to give a name to `body.type` instead of just accessing the type directly.

The final equation on line 15 uses monads implicitly. The monadification process described in this paper will rewrite this `t.type` equation to something similar to what is on lines 9–10, providing both success and failure cases, but without the need to write anything but the success case, as in traditional inference rules.

Example: Typing and Error Reporting. In attribute grammars, equations for reporting type errors often end up duplicating the structure of equations for typing, since both are looking to identify situations where the production is typable and where it is not. We can reduce the redundancy by using an `Either` type. Similar to the `Maybe` type, a value of type `Either(S, T)` is often thought of as having a result of type `T` in `right(_)` or an error message of type `S` in `left(_)`. This allows us to use the typing equation to generate a type when it is typable, or an error message when it is not.

Since `Either` can also be formulated as a monad, our rewriting allows us to use values of an `Either` type implicitly while still generating error messages. This is shown in Figure 3. The `type` attribute has type `Either(String, Type)`, but we are able to use it as if it had type `Type`. We write a `match` expression which gives a type when it is typable but also generates error messages for the cases where both `t1` and `t2` are typable but the overall application is not. The `errs` attribute can then identify a new error by checking whether the application has a type and whether its children have types. These two equations give us good error messages without duplicating the typing structure or requiring us to consider error cases from further down in the tree.

Attribute Monadification. Our scheme is built on having different modes of attributes, where modes dictate whether one is allowed to use monads implicitly or explicitly. Two of these modes can be seen in Figure 3. The `type` attribute is in the implicit mode because it treats the `Either` monad implicitly. The `errs` attribute treats the `Either` monad explicitly by matching on `left` and `right`. This is the unrestricted mode. The third mode, restricted mode, also treats monads

```
app: t:Expr ::= f:Expr a:Expr
{ t.type = match f.type with
    | arrow(T1, T2) when T1 == a.type -> T2
    | arrow(_, _) -> left("App type mismatch")
    | _ -> left("Non-function applied")
  t.errs = match t.typ, f.typ, a.typ of
    | left(s), right(_), right(_) ->
        [s] ++ f.errs ++ a.errs
    | _, _, _ -> f.errs ++ a.errs }
```

Figure 3. Typing and error-reporting equations for application where the type attribute has type `Either(String, T)` so that typing failures can include an error message.

explicitly, but may not access implicit attributes. The three modes are necessary for us to be able to relate the properties of the attribute grammar before and after the rewriting. The relationship between the two is important for ensuring our rewriting does not change the intended semantics.

Our goal with this rewriting is to allow a programmer to think about expressions and values of a monadic type as if they were not monadic, ignoring, to a large degree, the presence of the monad. In Figure 2, the equation on line 15 uses an attribute of type `Maybe(Type)` as if it had type `Type`. The resulting equation is similar to the inference rule T-Abs in Figure 1. Similarly, in Figure 3, we use the `type` attribute as if it simply had type `Type`. We do not completely ignore the presence of the monad here, since we use it to output error messages, but we are able to ignore it in determining typability and in giving the resulting type.

To be able to handle error cases and other effects in evaluating equations in the attribute grammar, we rewrite equations for implicit attributes, translating them into the base attribute grammar language that the other modes use. This rewriting completes the clauses of `match` expressions when they do not match all possible values. It also inserts monadic operations to create an expression typable in the base language. Our rewriting process translates each equation for an implicit attribute while leaving equations for restricted and unrestricted attributes as they are, since they are already valid in the base language.

We want to ensure that the semantics written while ignoring monads in the implicit mode are valid after the rewriting, and that the rewriting process itself is not problematic. To be able to formalize these properties, we define typing and evaluation rules for the extended language including the three modes of attributes, which we relate to the same for the basic language after the rewriting.

The properties state that rewriting preserves the types of the expressions and equations it processes, rewriting is total on equations that are typable in the extended language, and rewriting preserves evaluation results, so an equation which evaluates to a value will have its rewritten version evaluate to a related value, and vice versa, *e.g.* relating v with `just(`v`)`. Because of these properties, it is valid to think about the semantics of a language defined by an attribute grammar by ignoring the presence of a monad. This allows us to prove properties of language semantics using the simple equations before rewriting, and have them apply to the semantics of the equations the rewriting gives us.

Contributions. We present, in Section 3 a method for writing attribute equations that allows one to largely ignore the presence of a monad, thinking in terms of the type it contains. This includes a notion of attribute modes and rules for interpreting expressions using monads in an implicit fashion. In Section 4 we present a rewriting process to monadify such expressions and equations. In Section 5 we present properties to show our rewriting is correct with respect to the semantics of the extended language. Section 6 describes our prototype implementation in Silver, of which an artifact is available, and discusses the effectiveness of our scheme in practice. We describe related work in Section 7 and close in Section 8, discussing points of our work and possible future extensions.

2 Background

We first provide some background on attribute grammars, the typing and evaluation rules for them, and monads.

2.1 Attribute Grammar Paradigm

Attribute grammars [19] have a long history, beginning with Knuth [14]. An attribute grammar associates attributes with nodes in an abstract syntax tree to compute analyses on the program it represents. An attribute grammar $AG = \langle G, A, @, \Gamma, EQ \rangle$ consists of several parts. The grammar G contains a set of nonterminal types (NT), primitive types (such as integers, and strings), and productions. A production has the form $pn : x_0 : T_0 ::= \overline{x_i : T_i}$ where pn is the production name, x_0 is the label on the node (of nonterminal type T_0) the production builds, and each x_i refers to a child argument to the production of type T_i (either a nonterminal type or a primitive type).

The set A contains all attributes, each of which has a declared type. An attribute is either inherited or synthesized; inherited attributes are given to a node from above to provide information about the node's context, such as a typing context, while synthesized attributes are defined by the node itself to pass information up, such as the type of the node. The @ occurs-on relation ($@ \subseteq A \times NT$) indicates which attributes decorate which nonterminals. The typing environment Γ maps production names and attributes to their types; it is used and extended in the typing rules below. Note that in the sample AG specifications in the paper we leave out nonterminal and occurrence declarations since they can be inferred. Finally, EQ is the set of equations of the form

$x_j.a = e$, where each equation belongs to a specific production. Equations in a production may define synthesized attributes on the current node (x_0) or inherited attributes on children ($x_i, i > 0$). They may also define local attributes, values local to the current production used to help define other attributes, or inherited attributes on these.

Since Knuth, many extensions [2, 7] to AGs have been made. These include higher-order attributes (such as `type` of nonterminal type `Type`) which allow (undecorated) syntax trees to be passed around as attribute values and then decorated and used [25].

2.2 Attribute Grammar Typing and Evaluation

Our expression language for writing equations can be viewed as a functional programming language, with the addition of the construct for accessing attributes. Due to this, most constructs and their rules for typing and evaluation are familiar.

Typing. Our expression typing relation, written $\Gamma \vdash e : \tau$, indicates that, under typing environment Γ, expression e has type τ. In typing equations and expressions, Γ is as above but extended to include node labels and their types based on the production signature of the equation. The typing rules are as expected for common constructs such as conjunction and pattern matching, but a collection is shown in Appendix A.1. The main rule of interest, as the one not found in normal functional languages, is T-AttrAccess, found in Figure 4. This requires the expression on which the attribute is accessed be typable, the attribute have a declared type, and the attribute be declared to occur on the type of the tree. The access then has the type of the attribute.

We also have a typing relation over equations with one rule, T-Eq, also found in Figure 4. This rule checks that the tree is known, the attribute has a type, the attribute occurs on trees of the tree type, and the expression defining the attribute has the attribute's type.

Evaluation. Evaluation of expressions in attribute equations is specified in the style of big-step semantics, with a judgment of the form $\gamma \vdash e \Downarrow v$ indicating that, under the value store γ, expression e evaluates to value v. The store γ is extended or changed, as expected, when evaluating let-expression bodies or closures to map the introduced names to values. As above, for most constructs these are exactly as expected, but a few are shown in Appendix A.2. For attribute grammar-specific concerns, the store also maps the labels of nonterminals in a production to tree nodes so attribute values can be accessed and is initiated as such when the equation is evaluated. A tree node, tn, is simply another store mapping attribute names to values. The only interesting evaluation rules, from an attribute grammar perspective are E-AttrAccess, E-Eq, and E-Eq-Store, all in Figure 4. In E-AttrAccess, when e evaluates to a tree node tn, the attribute value can be looked up in tn. Since we later discuss properties of the rewriting process in terms of equations, the

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash a : \tau_2 \qquad \Gamma \vdash a@\tau_1}{\Gamma \vdash e.a : \tau_2} \quad (\text{T-AttrAccess})$$

$$\frac{t : NT \in \Gamma \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash a@NT \qquad \Gamma \vdash e : \tau}{\Gamma \vdash t.a = e\ OK} \quad (\text{T-Eq})$$

$$\frac{\gamma \vdash e \Downarrow tn \qquad (a, v) \in tn}{\gamma \vdash e.a \Downarrow v} \quad (\text{E-AttrAccess})$$

$$\frac{\gamma \vdash e.a \Downarrow v}{\gamma \vdash^{EQ} t.a = e; \Downarrow v} \quad (\text{E-Eq})$$

$$\frac{\gamma \vdash^{EQ} t.a = e; \Downarrow v \qquad \gamma \vdash t \Downarrow tn}{tn \leftarrow (a, v)} \quad (\text{E-Eq-Store})$$

Figure 4. Basic typing and evaluation rules for attribute accesses and equations. Other rules in the expression relations are as expected in any functional language.

rule E-Eq evaluates an equation to the value of its expression. The process of determining an order for evaluating equations and storing attributes in tree nodes is orthogonal to the work in this paper. It could be driven by a schedule as used in ordered attribute grammars [13] or the demand-driven approach [9] used in systems such as JastAdd, Kiama, and Silver. Nevertheless, the rule E-Eq-Store can be used to update a tree node tn so the computed attribute value can be looked up in an evaluation of another equation.

We will abuse notation slightly and write $\gamma \vdash e \Downarrow \bot$ when there is no evaluation derivation for e.

2.3 Monads

A monad [16] is a type constructor with two special operations, bind (written infix as $\gg=$) and `return`. Monads are useful for handling effects in functional languages, and can be generally thought of as the result of a computation [26] or as a box holding zero or more values. Throughout this work, we will write a generic monad over the type `T` as `M(T)`. An example of a monad is `Maybe`. The type `Maybe(T)` has two constructors, `just` of type `T` $\rightarrow$ `Maybe(T)` and `nothing` of type `Maybe(T)`. `Maybe` can be used to represent a computation which may not succeed, such as assigning a type to an expression. Another example is `Either(S, T)` for a particular `S`. The `Either(S, T)` type has constructors `left` of type `S` $\rightarrow$ `Either(S, T)` and `right` of type `T` $\rightarrow$ `Either(S, T)`, often viewed as representing a failure with some kind of error message of type `S` or a result of type `T`.

Bind has type `M(T`$_1$`)` $\rightarrow$ `(T`$_1$ $\rightarrow$ `M(T`$_2$`))` $\rightarrow$ `M(T`$_2$`)`. Binding a value into a function can be viewed as taking the result of a computation and using it in another computation, the given function, or as taking a value out of a box and applying the function to it. The second operation, `return`, has type `T` $\rightarrow$ `M(T)`. This wraps a value to create an instance of the monad. For `Maybe`, binding a `just` takes the value out of the `just` and applies the function to it (`just`(x) $\gg=$ $f = f\ x$). Binding a `nothing` results in a `nothing`. The `return` operation wraps its argument in `just`. The operations for `Either` are similar to those for `Maybe`, with bind either unwrapping and applying the function for `right` or giving the same `left` value back, and `return` wrapping its argument in `right`.

Another monad operation is `fail`, representing a failing computation or an empty box. For some monads, like `Maybe`, it can be viewed as a single value (`nothing`). For others, it is a function which takes an argument, as with `Either`'s `left`. We will write `fail` as a value throughout, with the understanding that it may sometimes require an argument.

For a monad to be valid, there are some laws which it must follow. The left identity law requires that `(return a` $\gg=$ `f) = f a`, and right identity requires `(m` $\gg=$ `return) = m`. Bind must also be associative. The `fail` operation must be a zero of bind, meaning `fail` $\gg=$ `f = fail`.

Monadic Containment. Our rewriting is concerned with monads which fit the view of a box containing at most a single value. We define a relation $v \in_M v'$ where v' is a monadic value and v is a value in the box v'. We use this definition to define how to think about evaluation using implicit monads prior to rewriting. For the monads we have discussed, we have $v \in_M$ `just`(v) and $v \in_M$ `right`(v). Any monadic value which does not contain a value is a monadic failure, possibly distinct from `fail`, since it is a value of type `M(T)` but has no values of type `T` associated with it.

Our final property, named evaluation correctness, will require this monadic containment relation to satisfy some properties in relation to the monadic operations because we will use monadic containment to define evaluation before the rewriting. Any monad for which a definition can be given satisfying these requirements will also satisfy our properties. The first is that `fail` must not contain any values ($\neg\exists v.\ v \in_M$ `fail`). Secondly, if there is a value contained in the result of a bind ($v \in_M (m \gg= f)$), there must be a value contained in the original monadic value ($\exists v.\ v \in_M m$). Next, if a value is contained in a monadic value ($v \in_M m$), the result of a bind is the function applied to this value ($m \gg= f = f\ v$). This requirement implies that there is at most one value contained in a monadic value. Finally, a value must be contained in the `return` of it ($v \in_M$ `return` v). These requirements fit the view of a monad as a box, with bind operating on the value inside the box and `fail` as an empty box. An intuitive definition of containment should satisfy these requirements, as long as it only defines a single value as being contained.

Lists are also monads, but their normal implementation would not satisfy the requirements above, due to the third requirement. We further discuss lists, their uses, and their challenges in Section 8.

```
restricted inh attr env : [ (String, Type)];
implicit syn attr type : Maybe(Type);
unrestricted syn attr errs : [String];
app : t:Expr ::= f:Expr a:Expr
{ f.env = t.env; a.env = t.env;
  t.type = match f.type with
    | arrow(T11, T12)
        when T11 == a.type -> T12;
  t.errs = f.errs ++ a.errs ++
     match f.type, a.type with
     | just(arrow(T11, T12)), just(T2) ->
        if T11 == T2 then []
        else ["Type mismatch"]
     | just(_), _ -> ["Non function applied"]
     | _, _ -> []; }
var : t:Expr ::= x:String
{ t.type = lookup (t.env, x);
t.errs = ... ; }
```

Figure 5. Using 3 attribute modes for typing and error reporting of function application and variable reference.

3 Implicit Monads in Attribute Grammars

Here we discuss the three different attribute modes in our scheme to develop an intuition for how they are related. We also introduce formal evaluation and typing rules for the three modes of attributes and describe how they relate to the intuition for their use.

3.1 Attribute and Equation Modes

Our scheme includes three different modes of attributes: *implicit*, *restricted*, and *unrestricted* attributes, each with an associated equation mode. Having different modes allows us to treat monad-typed values implicitly in some equations and explicitly in others. Figure 5 shows all three attribute modes in another version of typing an application and name reference: a restricted attribute `env` for the typing environment; an implicit attribute `type` as before; and an unrestricted attribute `errs` for error messages.

Both restricted and unrestricted attribute equations treat monadic values explicitly and are written in the basic language. To satisfy the correctness properties of the monadification (see Section 5), we restrict how information flows between attributes of different modes. One aspect of this is that a `fail` in an implicit attribute should not be explicitly examined and converted into a non-failure value as this breaks the abstraction of implicit monads. The mode of an equation determines which attributes it can access, as specified below:

eqn. mode	can access attributes of mode:
restricted	restricted
implicit	restricted, implicit with same monadic type
unrestricted	restricted, implicit, unrestricted

Implicit attributes can only access implicit attributes of the same monadic type since effects or failures in the accessed attribute must flow through. This hierarchy of information flow ensures that nothing in an implicit attribute can affect the evaluation of a restricted attribute, and nothing in an unrestricted attribute can affect the evaluation of an implicit attribute or a restricted attribute. Our type system, discussed below, ensures this, even with closures in our language.

Implicit Attributes. Implicit attributes are the only mode where monads may be used implicitly. This is done in the `type` equation for `app` on line 6 in Figure 5; it matches directly on `f.type` for an `arrow` type, leaving the other cases unstated. In the equation for abstraction on line 15 in Figure 2, a type term is constructed directly. In fact, equations for implicit attributes, called *implicit equations*, may not use monads explicitly, meaning they cannot use operations expecting a monad such as the monadic bind construct or matching against monadic patterns. Implicit failure cases will be filled in by the rewriting with monadic `fails`.

Implicit attributes are required to have a monadic type. They are intended to represent semantic attributes of the language with more information than just a simple value, such as potential failure with the `Maybe` monad. Implicit attributes of type `M(T)` should generally be thought of as having type `T`, since the monad is used to represent failure or an effect on a computation of type `T`. In Figure 5, we see the `type` attribute, of type `Maybe(Type)`, being treated as having type `Type`.

Restricted Attributes. Restricted attributes may have any type and manipulate monadic values directly. If a restricted equation includes expressions of a monadic type, it treats these explicitly. Because of this, restricted equations may only access other restricted attributes; this restriction is the source of the name "restricted". The `env` attribute is restricted and is a list. This maps names to their types and is accessed, via the `lookup` function, in the equation for `type` for variable references, line 17 in Figure 5.

Unrestricted Attributes. Unrestricted attributes are not intended for standard semantic attributes of the language, such as typing, but for the extras above the language semantics, such as error messages. On lines 10–15 the equation for `errs`, the unrestricted attribute, matches on both `f.type` and `a.type`, checking not only whether the types correspond when `f.type` is an arrow type, but also checking whether `f.type` is an arrow type and whether they are typable at all. Unrestricted attribute equations may access any attribute, regardless of mode, and treat monads explicitly. Indeed, they must be able to do so, if they are to be able to generate error messages based on an implicit attribute which has a monadic type possibly representing failure. Because these have no restrictions on mode access and their evaluation is unaffected by monadification, they are written in the basic language.

3.2 Evaluation and Typing Relations

We specify evaluation and typing rules for our language with attribute modes to formalize what it means to think in terms of type `T` instead of `M(T)` in implicit equations. The rules for equations in the restricted and unrestricted modes are very similar to the basic rules, while the rules for implicit equations are more complex. In what follows we will reference monadic types and monadic values. In rules for typing and evaluating implicit equations and expressions, these will always refer to the monad in the type of the attribute being defined. For example, if an attribute has type `Maybe(Bool)`, the monadic type constructor M in the typing rules always refers to `Maybe`, and monadic containment $v \in_M m$ will mean m being `just(`v`)`. The full set of all versions of evaluation and typing rules is given in Appendix A.

3.2.1 Evaluation. Before we look at evaluation rules, we note that we do not require a particular evaluation strategy for attributes. Techniques such as ordered [13] or demand-driven [9] attributes grammars both suffice and are orthogonal to the concerns here. Also note that these evaluation rules are for thinking about implicit evaluation, and are not used for evaluation. The rewriting will provide a more complete version of evaluation for implicit equations, maintaining the monadic values and effects associated with them, which our rules here intentionally drop.

The relation for evaluating restricted attributes is the same as the basic relation with the exception of evaluating abstractions, a difference which is necessary due to interactions with implicit attributes; we discuss this in Appendix A.5. Because we think of implicit attributes as having non-monadic types, they will evaluate to non-monadic values. Because of this, we cannot define evaluation for unrestricted attributes, which expect implicit attributes to be monadic.

The evaluation rules for implicit attributes mirror those for the base language with some exceptions. We do not include rules for bind or `return` for the equation's monad, as these constructs are not allowed. Thinking about expressions of type `M(T)` as having type `T`, we will want to be able to evaluate constructs with a monadic type where a non-monadic type is expected. This requires that we have a way to get the value from a monad. We might think of doing this with the following rule, allowing us to extract the value at any point:

$$\frac{\gamma \vdash_I e \Downarrow v \qquad v' \in_M v}{\gamma \vdash_I e \Downarrow v'}\ (\textsc{IE-DropMonad})$$

While sufficient, a more precise formulation is possible. The only constructs which may introduce monadic values in implicit expressions are restricted attribute accesses and function applications. We extract the value where it arises by adding the monadic unwrapping into the attribute access rule. This gives us two separate rules. The first is

$$\frac{\gamma \vdash_I e \Downarrow tn \qquad (a, v') \in tn \qquad v \in_M v' \qquad v \text{ is not monadic in } M}{\gamma \vdash_I e.a \Downarrow v} \quad \text{(IE-AttrAccess\_E)}$$

This evaluates as in the E-AttrAccess rule in Figure 4, but then extracts the non-monadic value v from the attribute's original value v'. The second rule directly uses the attribute's value if it is not monadic in the current monad:

$$\frac{\gamma \vdash_I e \Downarrow tn \qquad (a, v) \in tn \qquad v \text{ is not monadic in } M}{\gamma \vdash_I e.a \Downarrow v} \quad \text{(IE-AttrAccess)}$$

We have a similar duplication of rules for application, discussed in Appendix A.5.

We also exclude evaluation rules for `fail` constructs, such as `left` on lines 4–5 in Figure 3, in the implicit evaluation. A monadic failure value cannot be used in the implicit evaluation, so being unable to evaluate a monadic `fail` is essentially the same as evaluating it to a monadic failure value.

Using the attribute-access and application rules for monadic values instead of IE-DropMonad institutes a boundary that monadic values may not cross. Because of these rules and the lack of a rule for evaluating monadic failures, equations for attributes of type `M(T)` always evaluate to values of type `T`. This means one may think in terms of the underlying type, ignoring the effects held by the monad. The full set of implicit evaluation rules are in Appendix A.4.

3.2.2 Typing. The extended typing system is intended to check that the attribute mode hierarchy is obeyed, that type-driven rewriting can proceed, and that the rewritten equations respect implicit uses of monads in evaluation. Unlike the evaluation rules, which are only for reasoning about the implicit semantics, these rules are used to check the validity of equations and expressions before rewriting occurs.

Since unrestricted equations have no additional restrictions, they use the basic typing relation. Restricted attributes also use the basic typing relation, but with a different attribute access rule that checks the attribute mode hierarchy requirements. This rule adds a mode restriction:

$$\frac{\Gamma \vdash_R e : NT \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash a@NT \qquad a \text{ restricted}}{\Gamma \vdash_R e.a : \tau} \quad \text{(RT-AttrAccess)}$$

Otherwise the typing rules for restricted equations are the same as for the basic language.

Our implicit typing relation, written $\Gamma \vdash_I e : \tau$, types an expression e to type τ. As in evaluation, there are no typing rules for bind and `return` for the current equation's monad. We have restrictions on attribute accesses, as in RT-AttrAccess, ensuring that only restricted attributes and implicit attributes with types built by the same monad are accessed. This avoids explicit uses of an implicit monadic value as discussed below, and because all implicit uses in the same equation must represent the same effect.

To ensure implicit monad uses are respected in evaluation after rewriting, we must ensure no implicit monad is ever matched in a `match` with explicit monadic patterns:

$$\frac{\Gamma \vdash_I e : \tau_1 \qquad \Gamma \Vdash_I cs : \tau_1 : \tau_2 \qquad \tau_1 \neq M(\tau_1') \qquad \Gamma \vdash cs \textit{ complete}}{\Gamma \vdash_I \texttt{match}\ e\ \texttt{with}\ |\ cs : \tau_2} \quad \text{(IT-Match)}$$

The requirement that τ_1 (the type of the patterns in clauses cs) is not a monadic type ensures that we are not matching against the monad. To see why we must do this, consider the following expression with monadic patterns:

```
  match t.a with | just(x) -> x | nothing() -> 5
```

If `t.a` evaluates to `nothing()` after rewriting, as our properties will show, it did not evaluate before rewriting. The `match` above turns that failure into a non-failure value which has no relation to the original evaluation. This would invalidate thinking in terms of the original equation, because we could get unrelated answers after rewriting.

We also need to ensure implicit monads do not get passed as arguments to functions which would explicitly match on them, since the function may not be type checked with these implicit typing rules. While the above restriction prevents such computations from being built in an implicit equation, they can be built in restricted attributes and passed into implicit attributes. To prevent this, we require the argument in an application not be a monad and not ultimately result in a monad type, encoded in the *non-monadic-result* predicate:

$$\frac{\Gamma \vdash_I e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash_I e_2 : \tau_1 \qquad \textit{non-monadic-result}(\tau_1)}{\Gamma \vdash_I e_1\ e_2 : \tau_2} \quad \text{(IT-App)}$$

This bans the application of a function expecting any function which may ultimately produce a monadic value, no matter how many arguments it takes. To see why this is necessary, consider the following function: `λx:Bool. t.a` If `a` is an implicit attribute, this function allows us to pass an implicit monadic value to another function, even if indirectly, which might match on it, breaking our evaluation properties. Banning direct matching on monadic types and passing monadic arguments ensures implicit monads are respected.

For thinking of expressions of type `M(T)` as type `T` to be valid, we must be able to use a monadic type where the type inside was expected, such as `M(Bool)` where `Bool` was expected. One approach is to use a rule to drop the monad from the type, so an expression of type `M(T)` can also have type `T`. If higher-order functions are not used, this rule is sufficient for reasoning about implicit expressions and ensuring rewriting a typable expression will produce a well-typed expression in the basic language, where the monads in types will propagate upwards to maintain their effects.

Our actual typing rules for implicit equations thus propagate monadic types. They also ban matching against monadic patterns and taking monadic types or functions with monadic results as function arguments, as well as the ban on monadic bind and `return` while adding rules to propagate monadic types from subexpressions. To see how this propagation occurs, consider two of the four rules for typing conjunctions:

$$\frac{\Gamma \vdash_I e_1 : \texttt{Bool} \qquad \Gamma \vdash_I e_2 : \texttt{Bool}}{\Gamma \vdash_I e_1 \&\& e_2 : \texttt{Bool}} \quad (\text{IT-AND})$$

$$\frac{\Gamma \vdash_I e_1 : M(\texttt{Bool}) \qquad \Gamma \vdash_I e_2 : \texttt{Bool}}{\Gamma \vdash_I e_1 \&\& e_2 : M(\texttt{Bool})} \quad (\text{IT-AND\_M1})$$

The first rule corresponds directly to the basic typing rule for conjunctions, since both subexpressions have type `Bool` and the conclusion has type `Bool`. The second rule has one subexpression of type `M(Bool)` and one of type `Bool`, with the conclusion having type `M(Bool)`. This demonstrates what we mean by propagating the monadic type, adding the monadic constructor to the expected conclusion type from the basic typing rule the implicit rule is based on, whenever a monadic type occurs on a subexpression.

We have implicit typing rules corresponding to each basic typing rule other than the explicit monadic constructs, with each basic rule expanding into a set of implicit rules for each possible combination of implicit monad uses. The rewriting rules are based on the implicit typing rules, and seeing the typing rules in conjunction with how an expression is rewritten is more informative than seeing the typing alone, so we shall combine our discussion of the finer points of propagating monads in typing with our discussion of rewriting. Our discussion of rewriting is picked up in the next section. The full set of implicit typing rules can be found in Appendix A.3.

4 Rewriting

While the implicit monad equations are a convenient way to specify and reason about some attribute computations, we ultimately want a complete computation, with failures and other effects included. Instead of evaluating implicit equations directly, we rewrite them into the same basic language as the unmodified restricted and unrestricted equations.

This rewriting has two purposes: to complete incomplete `match` expressions by adding default monadic failure cases and to insert monadic bind and `return` operations as necessary to make a basic-typable expression. The equation rewriting uses the expression rewriting to rewrite the full equation into a version which is acceptable under the basic typing rules. The monadification rewriting is only carried out if all equations in the attribute grammar are well-typed, which includes hierarchy mode checking.

The expression rewriting relation, written $\Gamma \vdash e \rightsquigarrow e' : \tau$, rewrites e into e' under the typing context Γ in which τ is the type of e under the implicit typing rules, and also the type of e' under the basic typing rules. We have a rewriting rule corresponding to each implicit typing rule. Because we assume the equation was well-typed before rewriting, we can leave out some requirements that the typing rules include.

On many forms of expressions, such as conjunction, a common pattern in monadification plays out. If the types involved are not monadic in the type of the implicit attribute, the rewriting reconstructs the expression with the rewritten components. This can be seen in RW-AND in Figure 6. If a subexpression has a monadic type, the rewritten version of that child is bound into an abstraction with the original operation as its body. If the return type of this expression is not a monad (as is the case for conjunction with a return type of `Bool`), the original operation is wrapped in `return`. This pattern can be seen in Figure 6 in the rule RW-AND_M1. Because we are using a bind, our return type must be the same monad, so we need a `return`. In addition to having the correct type for the rewritten expression, the evaluation order from the original term must be preserved. In RW-AND_M2, our rewriting turns conjunctions into `if-then-else`s to evaluate e_1' before e_2'. Because conjunction is short-circuiting, this gives them related termination behavior. If e_2' were bound into a function containing e_1', e_2' would be evaluated even if e_1' were false, changing the behavior of the evaluation.

Our rules for applications, such as RW-APP_MARG1 in Figure 6, do not include the requirement of *not-monadic-result*, as that was checked during typing. In RW-APP_MARG1 maintaining the evaluation order, as in RW-AND_M2, requires using a `let` to evaluate the function first, followed by the argument as we bind it in. To propagate the monad, we wrap the result in `return`. If the result type of the function were monadic, say `M(T)`, we would not need to insert this `return` to get the type `M(T)`. We also do this in the typing rules, not wrapping a type in the same monadic type constructor twice. Our typing and rewriting rules will only unwrap one instance of a monadic type constructor from a type when used implicitly, so a doubly-wrapped type is not useful here. If `Maybe(T)` represents possible failure, `Maybe(Maybe(T))` represents a possible failure of a possible failure, which is likely better represented by combining the failures.

We can see the use of RW-APP_MARG1 in rewriting the implicit equation for `type` in Figure 2, which read:

```
    t.type = arrow(Ty, body.type);
```

For brevity, we consider the `arrow` constructor to be curried

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \mathit{Bool} \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \mathit{Bool}}{\Gamma \vdash e_1 \mathbin{\&\&} e_2 \rightsquigarrow e_1' \mathbin{\&\&} e_2' : \mathit{Bool}}\ (\text{RW-}\textsc{And})$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : M(\mathit{Bool}) \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \mathit{Bool} \qquad x \text{ fresh in } e_2'}{\Gamma \vdash e_1 \mathbin{\&\&} e_2 \rightsquigarrow e_1' \gg\!= \lambda x : \mathit{Bool}.\ \texttt{return}\ (x \mathbin{\&\&} e_2') : M(\mathit{Bool})}\ (\text{RW-}\textsc{And\_M1})$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \mathit{Bool} \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : M(\mathit{Bool})}{\Gamma \vdash e_1 \mathbin{\&\&} e_2 \rightsquigarrow \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else return false} : M(\mathit{Bool})}\ (\text{RW-}\textsc{And\_M2})$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : M(\mathit{Bool}) \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : M(\mathit{Bool}) \qquad x \text{ fresh in } e_2'}{\Gamma \vdash e_1 \mathbin{\&\&} e_2 \rightsquigarrow e_1' \gg\!= \lambda x : \mathit{Bool}.\ \texttt{if}\ x\ \texttt{then}\ e_2'\ \texttt{else return false} : M(\mathit{Bool})}\ (\text{RW-}\textsc{And\_MBoth})$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : M(\tau_1) \qquad \tau_2 \neq M(\tau_2') \qquad f \text{ fresh in } e_2' \qquad x \neq f}{\Gamma \vdash e_1\ e_2 \rightsquigarrow \texttt{let}\ f = e_1'\ \texttt{in}\ (e_2' \gg\!= \lambda x : \tau_1.\ \texttt{return}\ (f\ x)) : M(\tau_2)}\ (\text{RW-}\textsc{App\_MArg1})$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : M(\tau_1) \qquad \neg(\Gamma \vdash cs\ \mathit{complete}) \qquad \Gamma \Vdash cs \mid \_ \Rightarrow \texttt{fail} \rightsquigarrow cs' : \tau_1 : M(\tau_2) \qquad x \text{ fresh in } cs'}{\Gamma \vdash \texttt{match}\ e\ \texttt{with} \mid cs \rightsquigarrow e' \gg\!= \lambda x : \tau_1.\ \texttt{match}\ x\ \texttt{with} \mid cs' : M(\tau_1)}\ (\text{RW-}\textsc{Match\_MD\_Incomplete})$$

$$\frac{\Gamma \vdash p\ \mathit{match}\ \tau_1; \Gamma' \qquad \Gamma' \vdash e \rightsquigarrow e' : M(\tau_2) \qquad \Gamma \Vdash cs \rightsquigarrow cs' : \tau_1 : \tau_2 \qquad \mathit{returnify}\ cs'\ cs''}{\Gamma \Vdash p \Rightarrow e \mid cs \rightsquigarrow p \Rightarrow e' \mid cs'' : \tau_1 : M(\tau_2)}\ (\text{RW-CS-}\textsc{Add\_MHere})$$

$$\frac{\Gamma \vdash p\ \mathit{match}\ \tau_1; \Gamma' \qquad \Gamma' \vdash e \rightsquigarrow e' : \tau_2 \qquad \Gamma \Vdash cs \rightsquigarrow cs' : \tau_1 : M(\tau_2)}{\Gamma \Vdash p \Rightarrow e \mid cs \rightsquigarrow p \Rightarrow \texttt{return}\ e' \mid cs' : \tau_1 : M(\tau_2)}\ (\text{RW-CS-}\textsc{Add\_MLater})$$

Figure 6. Select rules for rewriting expressions.

here to treat this as two applications of single-argument functions. The first applies `arrow` to the non-monadic value `Ty`, yielding a function of type `Type`⟶`Type` and is unchanged by the rewriting. The second application matches rule RW-App_MArg1 since the argument `body.type` is monadic with type `Maybe(Type)`. Using this rule, we rewrite into

```
    let f = λb:Type. arrow(Ty, b) in
        body.type ≫= λty:Type. return (f ty)
```

We can see that this is semantically equivalent to the explicit monad version on lines 9 and 10 in Figure 2.

In our rewriting rules for `match`, we keep the restriction from the typing rules that the pattern type cannot be monadic. Unlike argument types in applications, we cannot drop this requirement, since the pattern type is not necessarily unique due to variable patterns, and we ban any type of matching on monadic values.

In Figure 6, we show RW-Match_MD_Incomplete for `match`. The expression being matched has a monadic type, and, because the patterns are not allowed to have a monadic type, we bind this into a new function containing a new `match`. This rule, and others like it, are predicated on whether the clauses *cs* are complete, meaning they match all possible patterns. In RW-Match_MD_Incomplete, there are some values not matched by the existing clauses. These are completed with a wildcard pattern yielding a monadic failure result before rewriting them. This ensures the clauses result in a monadic type. Because we add a monadic failure clause to any `match` with incomplete clauses, the typing rules require such a `match` have a monadic type, and thus the typing rules discriminate between complete and incomplete clauses in `match` expressions the same as the rewriting rules.

In typing and rewriting clauses, we must consider that some clauses in a set may result in type `T` and some in type `M(T)`, as seen in lines 3–5 in Figure 3, since we allow implicit use of monads and may think of an expression of type `M(T)` as if it had type `T`. The type of such sets of clauses will be `M(T)`, with the rules always choosing the monadic type when both the monadic and non-monadic types are present. In RW-CS-Add_MHere we see the use of *returnify* to wrap `return`s around clause expressions as needed.

We use the expression rewriting described above to rewrite implicit equations. As above, the equation rewriting rules correspond to the implicit typing rules for equations. The rules for this rewriting are found in Figure 7. The first rule, RW-EQ_Basic, requires the expression in the equation to have the same type as the implicit attribute being defined. This corresponds exactly to the basic rule for typing equations. The second rule, RW-EQ_M, allows the expression in the equation to have type `T` when the attribute has type `M(T)`. This fits with thinking of the attribute's type `M(T)` as if it were type `T`. We might take advantage of this rule by writing the equation `t.type = bool();` for typing a production for the constant `true`. To make the rewritten equation typable, this rule wraps the rewritten expression in `return`. Finally, we have a rule for typing an implicit equation with no expression, `t.a = ;`, which is rewritten as an equation with `fail` as its expression. This allows us to write an equation acknowledging there is no value of type `T` for the semantic attribute, while getting a monadic failure when actually running the code. An empty equation might be used to define an attribute for an evaluation step in small-step evaluation for a production representing a value, as discussed in Section 6.

$$\frac{\Gamma \vdash e \rightsquigarrow e' : M(\tau) \qquad \Gamma \vdash a : M(\tau) \qquad t : NT \in \Gamma \qquad \Gamma \vdash a@NT \qquad a \text{ implicit}}{\Gamma \vdash t.a = e;\ \rightsquigarrow t.a = e';}\ (\text{RW-EQ\_Basic})$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : \tau \qquad \Gamma \vdash a : M(\tau) \qquad t : NT \in \Gamma \qquad \Gamma \vdash a@NT \qquad a \text{ implicit}}{\Gamma \vdash t.a = e;\ \rightsquigarrow t.a = \texttt{return}\ e';}\ (\text{RW-EQ\_M})$$

$$\frac{\Gamma \vdash a : M(\tau) \qquad t : NT \in \Gamma \qquad \Gamma \vdash a@NT \qquad a \text{ implicit}}{\Gamma \vdash t.a = ;\ \rightsquigarrow t.a = \texttt{fail};}\ (\text{RW-EQ\_Empty})$$

Figure 7. Rewriting rules for equations and the definition of the full expression-rewriting relation used in rewriting equations.

The full attribute grammar is rewritten by rewriting every implicit equation in it, leaving restricted and unrestricted equations as they are. Above we have discussed the more interesting and illustrative rewriting and typing rules. The complete set of rewriting rules can be found in Appendix B.

5 Properties

To show that it is correct to reason about implicit equations and expressions using their typing and evaluations rules shown above, we have three properties about the monadification rewriting: types are preserved from the original to the rewritten form, it is total when the original was typable in the extended language, and its results are correct with respect to the extended language's semantics. We walk through each property and sketch its proof in turn.

5.1 Types Preserved

Any attribute grammar which was typable using the extended language before the rewriting is typable after the rewriting using the basic language. It is clear that restricted and unrestricted equations are typable after the rewriting, since these only add to the basic requirements.

For implicit equations, we prove that any equation which was typable under the implicit equation typing relation before the rewriting is typable under the basic equation typing relation after the rewriting:

$$\Gamma \vdash_I eq\ OK \Rightarrow \Gamma \vdash eq \rightsquigarrow eq' \Rightarrow \Gamma \vdash eq'\ OK$$

This is, in turn, supported by a lemma for typing being preserved by rewriting for expressions:

$$\Gamma \vdash_I e : \tau \Rightarrow \Gamma \vdash e \rightsquigarrow e' : \tau \Rightarrow \Gamma \vdash e' : \tau$$

It is clear, from the definition of the rewriting rules, that the rewritten expressions will have the type in the rewriting relation under the basic rules. This property is the direct result of that.

5.2 Totality of Rewriting

For any attribute grammar typable using the extended language, there is an attribute grammar to which it rewrites. Because restricted and unrestricted equations are not changed in rewriting, they do not affect the totality of rewriting.

Rewriting is also total for implicit equations if they are typable under the implicit equation typing relation:

$$\Gamma \vdash_I eq\ OK \Rightarrow \exists eq'.\ \Gamma \vdash eq \rightsquigarrow eq'$$

This holds based on a similar lemma for expressions:

$$\Gamma \vdash_I e : \tau \Rightarrow \exists e'.\ \Gamma \vdash e \rightsquigarrow e' : \tau$$

The proof of this relies on the expression rewriting rules being based on the implicit typing rules. Since there is a rewriting rule for each typing rule, we have a rule to fit every case in an inductive proof on the typing derivation.

5.3 Evaluation Correctness

More important than maintaining typing and rewriting totality is that the evaluation results, meaning the values of attributes, before and after the rewriting are related to each other. For restricted attributes, the results are exactly the same before and after the grammar is rewritten. This is because the explicit evaluation relation used in the extended language for restricted attributes is essentially the same as the basic evaluation relation.

The relationship for implicit equations is more complex, as we need to relate non-monadic values in the implicit evaluation with (values wrapped up in) monadic ones in the rewritten base evaluation. This is due to our thinking of attributes of a monadic type `M(T)` as having type `T` in the implicit language. Below, $v \sim_v v'$ is equality for non-closure values, but compares closures slightly differently. The full details of this relation are in Appendix C.

We have two related properties for implicit equations:

- If $\Gamma \vdash eq \rightsquigarrow eq'$ and $\gamma \vdash_I^{EQ} eq \Downarrow v$, then $\exists v'\ v_0'.\ \gamma \vdash^{EQ} eq' \Downarrow v' \ \wedge\ v_0' \in_M v' \ \wedge\ v \sim_v v_0'$
- If $\Gamma \vdash eq \rightsquigarrow eq'$ and $\gamma \vdash^{EQ} eq' \Downarrow v'$, then
  - $\exists v_0.\ v_0 \in_M v' \ \wedge\ \exists v.\ \gamma \vdash_I^{EQ} eq \Downarrow v \ \wedge\ v \sim_v v_0$ or
  - $(\neg \exists v_0.\ v_0 \in_M v') \ \wedge\ \gamma \vdash_I^{EQ} eq \Downarrow \bot$

In text:

- If the original equation evaluates to a value v, the rewritten equation evaluates to a monadic value containing a value related to v.
- If the rewritten equation evaluates, it either contains a value v_0 and the original equation can evaluate to a value related to v_0 or it is a monadic failure and the original equation cannot evaluate.

Both of these rely on related properties about expression evaluation. The proofs of the properties proceed by induction on the derivation given, either the original, implicit derivation or the basic derivation from after rewriting. These proofs rely on the monad laws as well as the requirements for monadic containment in Section 2.3. The monad laws aren't enough to prove these because they don't allow us to relate the results of bind with our monadic containment back to containment in the original monadic value in the general case, and thus relate the original results and rewritten results. A more detailed proof sketch can be found in Appendix D.

As a corollary to these properties, we can also state that the rewritten equation being unevaluable implies the original equation is unevaluable, because having a derivation of evaluation for the original equation implies a derivation for the rewritten equation.

6 Prototype Implementation and Its Use

We have a prototype implementation of implicit monads as an extension to the Silver[1] [24] Attribute Grammar system. We used implicit monads in specifying typing and small-step evaluation for the simply-typed lambda calculus (STLC) with Boolean operations[2] and for type inference for Caml Light[3] [15]. A software artifact including the prototype in Silver and these examples can be found in the Silver archive.[1] In this, we found it quite convenient to write equations for typing without needing to pattern match on the monadic value.

Using an attribute of type `Either(String, Type)` for typing in STLC, we were able to create a list of error messages for typing without duplicating the type checking logic in equations for types and error messages, as seen in comparing Figure 3 to Figure 5. Using an implicit `Either` to create the error message in the type attribute avoids this duplication; the error (an unrestricted attribute) is then extracted from the type (an implicit one) and propagated up the tree. The use of `Either` as an implicit monad allows us to only write the error messages for the error cases where the subexpressions have types, letting the rewriting fill in the cases where the subexpressions are not typable.

We used the `Maybe` monad for small-step evaluation in STLC. Because typing and evaluation do not interact, we are able to use different monads for the two attributes. We used empty equations, *e.g.* `e.next = ;`, to define the evaluation step attribute for productions representing values, such as abstractions, because they cannot step.

In using implicit monads, the attributes which must be implicit, and therefore must have a monadic type, tend to spread, as happens when using monads in Haskell. Because our Caml Light type attribute is an implicit `Maybe` and the type substitution and typing context attributes both use the type attribute and are used in the type attribute equations, both of these had to have implicit `Maybe` types as well. As can be seen in Figure 8, which shows the production for

```
production ifthenelse
top::Expr ::= c::Expr th::Expr el::Expr
{
  implicit c.gamma = top.gamma;
  implicit th.gamma = top.gamma;
  implicit el.gamma = top.gamma;

  implicit c.subst = top.subst;
  implicit th.subst = c.subst_out;
  implicit el.subst= th.subst_out;
  implicit top.subst_out =
        typeUnify(th.type, el.type,
          typeUnify(c.type, boolType(),
            el.subst_out));

  implicit top.type =
     case typeSubst(c.type, top.subst_out) of
     | boolType() when
       typeEqual(th.type, el.type,
                  top.subst_out) -> th.type
     end;
}
```

Figure 8. The production for `if-then-else` in our Caml Light implementation using implicit monads using the syntax of the Silver prototype. The `gamma`, `subst`, `subst_out`, and `type` attributes all have `Maybe` types.

`if-then-else` in our Caml Light implementation, this actually made the equations for both of these easier to write, since we avoided matching on the type attribute to determine if a type existed before trying to unify it with the expected type. In typing an `if-then-else`, we simply write to unify the types (using a function `typeUnify`) of the two branches and the condition's type with the Boolean type to get a new type substitution in the equation for `subst_out`, then check in the `type` equation that the unification was successful by checking that the types are as expected under the substitution. All the unwrapping is taken care of automatically.

The prototype is built as an extension to the core language of Silver. It type checks restricted equations for mode hierarchy access errors and type checks implicit equations before translating them down to the core language, which is then compiled normally. As an extension, the core Silver language is not changed and the extension satisfies certain composabililty criteria [10, 22]. Because this implementation is an extension, it should work well with existing Silver specifications that do not use it. Therefore only implicit and restricted attributes need to be marked with their intended mode. Because unrestricted attributes only use the basic typing rules, not requiring marking them is safe.

[1]Available at http://melt.cs.umn.edu/silver and https://github.com/melt-umn/silver, archived, with artifact, at https://doi.org/10.13020/D6QX07.
[2]Found in the tutorials that come with Silver.
[3]Our implementation is available at https://github.com/melt-umn/caml-light, archived at https://doi.org/10.13020/866e-0d92.

While Silver is designed to be extended, it has some shortcomings that affect the usability of implicit monads. Since equation syntax is not overloadable, new syntax must distinguish implicit and restricted equations from existing (unrestricted) ones to trigger their type checking and translation. This is done by marking them with their mode, much like the attribute declarations in Figure 3. Silver also does not support extension-introduced error-checking analyses on existing language constructs, and thus we cannot check these mode markers on equations are present, as without them they are treated as unrestricted attributes. At this time, Silver does not include type classes, so there are no general monad operations to insert. Instead, we use only the monads built-in to Silver, with monad operations implemented as functions. These include `Maybe`, `Either`, and `List`. For these reasons the examples in the paper use an abbreviated form of the actual Silver extension syntax. Only Figure 8 uses the current syntax with marked equations. Both of these features are present in other extensible languages [11] implemented in Silver and, since Silver is written in itself, we are currently adding these capabilities to Silver. Once we no longer need to mark equations with their modes, we intend to make this extension part of the standard Silver distribution.

7 Related Work

Prior work has examined monadifying functional languages. Erwig and Ren [5] monadified functional programs by taking functions returning type `T` and rewriting them to return `M(T)`. The rest of the program was then changed to work with the monadic type being returned. Our work is more general in that we monadify general code, not just that associated with function calls. Our monadification can also be viewed as fulfilling the types written, rather than changing them. This permits us to have unrestricted attributes act on the monadic types. Other work [23] monadified ML programs to allow a normal let-expression to be used to bind a value of a monadic type `M(T)` which was then used as if it had type `T` in the body of the `let`. The `let` would then be turned into a monadic bind operation in rewriting. Different monads are allowed in the same expression, with the result being a combined monad of all those used and morphisms placed to convert everything to the final monadic type. This monadification is at once simpler and more complex than ours. It is simpler because all monadic expressions are bound by a `let`, which may be turned into a monadic bind, where we have attribute accesses which may introduce monads without a `let`. It is more complex because of the mixing of different monads, requiring morphisms between monads. Neither of these works have the same difficulties with nondeterminism that we discuss in Section 8 because they are working in functional languages. No external names can occur in their expressions not originating in their expressions which require them to find a consistent location to insert a bind.

There are also efforts to improve the readability of language specifications. The work of Mosses [17] is most related to our work. It also examined the idea of making language specifications easier to write by hiding details, but in SOS, by introducing Modular SOS (MSOS). This bundles parts of relations for defining language semantics in SOS rules, allowing the rules to be reused when more features are added to the language. It also allows for errors to be propagated back through rules automatically. This is similar to how our implicit attributes allow failures to automatically propagate through the tree. More work [18] introduced Implicit MSOS (I-MSOS), formalizing an implicit threading-through of parts of semantic relations, further simplifying SOS rules. This work, allowing the automatic propagation of *non-failure* values, is similar to previous work on modularity in AGs [12] that alleviates so-called copy-rules for copying information down the tree, equations that collect and combine information flowing up the tree, like lists of error messages, and *chaining-rules* that automatically propagate information in a left-to-right pattern in the tree. Eli [6] was one of the first AG systems to incorporate these conveniences, but they are relatively common in AG systems now. Pretty big-step semantics [4] reduces duplication of premises in big-step rules by breaking them down into smaller rules, each of which does part of the work done in the original big-step rules. This approach also simplifies handling errors, since exceptions in the language for which the rules are being written and errors of an unevaluable term can be propagated by the same rules, rather than needing special cases to be written for each one.

Many systems have offered solutions for handling potential failures without much added effort, as our monadification allows us to do. The Haskell language, among others, provides `do`-notation for working with monads, similar to what is seen on lines 12–13 in Figure 2. This is easier to use than the monad operations. Some work has used `do`-notation outside of Haskell, where the authors had to implement it themselves, because `do`-notation is much easier to write than the basic monad operations [1]. As noted above, however, it still requires overhead over our monadification. The Alloy language for models takes a different approach to potential failures. Scalar values are represented as sets, with failures being represented by empty sets [8]. Since failures are valid values of the set type, they can be easily handled. The Object Constraint Language (OCL), part of UML, has a similar approach. OCL has a `null` value for failures which is a member of any type due to its subtyping system [3]. This value passes through operations in most cases, percolating the failure up through evaluation. In both Alloy and OCL, because their failure values are normal members of the types in which they represent failure, some operations may not result in failures when they have failures as operands, unlike our monadification, which always passes monadic failures through. Because these languages are concerned with properties of systems holding, not with propagating the nonexistence of values, this approach suffices. In defining language semantics, we generally want to know if a failure occurred, making monadification a better choice.

8 Discussion and Future Work

Challenges in Modeling Nondeterminism. The `List` monad may be used to model nondeterminism, with each element as a possible result. Its bind operation maps the function over the list and appends the results. We have a `match_any` construct to use lists implicitly, defined to evaluate to the result of any matching clause. This translates to an expression appending the results of all the matching clauses.

We can't always think of `List(T)` as `T`, however. If we think of `t.a` as having type `Int`, we expect the result of `t.a + t.a` to be even. If, however, it has type `List(Int)`, this rewrites to

`t.a ≫= λx:Int. t.a ≫= λy:Int. return (x + y)`

If `t.a` has the value `[3, 4]`, we would like the result to be `[6, 8]`, but it actually adds all combinations of values, yielding `[6, 7, 7, 8]`. This means evaluation correctness does not hold in this case, and thus not for implicit lists in general.If an attribute access only occurs once in an expression, this issue does not arise. The requirement of a single occurrence also means it cannot occur by an alias, meaning `t.b` is assigned the value of `t.a` and they are used together. Without aliasing, this problem can be solved by using a `let` to rename an attribute, which will turn into a bind in the rewriting. For the addition above, we could instead write `let x = t.a in x + x`. This changes the `let` into a bind, and both addends will have the same value.

Our implementation permits implicit uses of the `List` monad, and it includes the `match_any` construct. We used this to implement small-step parallel evaluation of conjunction and disjunction in the simply-typed lambda calculus with Booleans. Implementing this with implicit monads was much simpler than writing the parallelism by hand. This closely corresponds to the nondeterministic inference rules for evaluation.

Implicit Monads and Do-Notation. A natural comparison is our implicit monads and do-notation as found in Haskell, which lets users sequence monadic operations without writing the monadic binds. One benefit to using implicit monads is that it allows us to leave off the `return` operation for the final result. This will be inserted in the rewriting. Another benefit is that do-notation requires the results of monadic computations to be bound to a variable before use, where we allow using them directly. For example, in Figure 2, do-notation requires us to bind `body.type` to a variable to use it. With implicit monads, we simply write the arrow type.

Sometimes this binding is helpful, however. As discussed above, our rewriting will bind an attribute access wherever it occurs, which may lead to unexpected results with some monads. The use of do-notation avoids this specifically because it requires binding to a variable.

Implicit Equations, Bind, and Return. In implicit equations, we have banned the use of monadic bind and return operations. These operations are safe to use, with regard to our properties, as long as we appropriately augment the typing, evaluation, and rewriting relations. We ban them because they are unnecessary and do not follow the philosophy of implicit use. They are unnecessary because they will be inserted as needed. Then we can just write the expression or function application we want to get the result of `return` or bind. Using such operations explicitly means values are being thought of as type `M(T)` rather than as type `T`, as they ought to be thought of in implicit equations

Explicitly using the `fail` operation of a monad also breaks this view, and `fail`s can be achieved by using an empty equation or an incomplete `match`. However, we sometimes want particular uses of `fail`. For example, in Figure 3, we use the `type` attribute to also generate error messages. Since there is only one use of bind or `return` in any given situation, but there are possibly different uses of `fail`, we disallow the former and allow the latter.

Future Work. One area of future work is finding a solution to allow monads with multiple values in them, such as `List`, while maintaining the ability to view monadic expressions of type `M(T)` as if they had type `T`.

A second area of future work is examining whether it is possible to create a finer-grained set of modes. If we did not require equations to have a mode, but instead inferred which expressions were implicitly monadic and which were not, we would be able to mix implicit and explicit uses in the same expression, all while ensuring implicit monads were not used explicitly. This would maintain our properties, but could make it easier to write equations.

Finally, it should be possible to leave both restricted and unrestricted attributes unmarked. The restricted attributes could be inferred by using flow information to check whether they were accessed in implicit equations, and then to check if they accessed unrestricted attributes. Only the implicit attributes would need to be declared with a mode. As noted, it is possible to modify Silver so that equations can be overloaded to check that the mode restrictions are obeyed without requiring the mode marker on equations.

Acknowledgments

We thank members of the IFIP Working Group 2.11 on Program Generation, especially Jacques Carette, Chung-chieh Shan, and Jeremy Gibbons, for comments on this work when presented at the 2019 meeting. We also thank the anonymous reviewers of this paper for many helpful comments and suggestions.

A Elaboration of Language Specifications

In this appendix we provide a more detailed listing of the evaluation and typing relations of the equation and expression languages used in the paper.

A.1 Basic Typing Rules

The typing rules for expressions in the basic language are as expected for a functional language, other than the T-AttrAccess rule.

$$\frac{\Gamma \vdash e : NT \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash a@NT}{\Gamma \vdash e.a : \tau}\ \text{(T-AttrAccess)}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}}\ \text{(T-True)}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}\ \text{(T-False)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \texttt{Bool}}{\Gamma \vdash e_1 \mathbin{\&\&} e_2 : \texttt{Bool}}\ \text{(T-And)}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{(T-Var)}$$

$$\frac{\Gamma; x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 .\ e : \tau_1 \longrightarrow \tau_2}\ \text{(T-Lam)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \longrightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{(T-App)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e : M(\tau)}\ \text{(T-Return)}$$

$$\frac{}{\Gamma \vdash \texttt{fail} : M(\tau)}\ \text{(T-Fail)}$$

$$\frac{\Gamma \vdash e_1 : M(\tau_1) \qquad \Gamma \vdash e_2 : \tau_1 \longrightarrow M(\tau_2)}{\Gamma \vdash e_1 \gg= e_2 : M(\tau_2)}\ \text{(T-Bind)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma; x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{(T-Let)}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \Vdash cs : \tau_1 : \tau_2}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ |\ cs : \tau_2}\ \text{(T-Match)}$$

A set of clauses must all match the same type, and all the clauses must have the same result type:

$$\frac{\Gamma \vdash p\ match\ \tau_1; \Gamma' \qquad \Gamma' \vdash e : \tau_2}{\Gamma \Vdash p \Rightarrow e : \tau_1 : \tau_2}\ \text{(T-CS-Single)}$$

$$\frac{\Gamma \vdash p\ match\ \tau_1; \Gamma' \qquad \Gamma' \vdash e : \tau_2 \qquad \Gamma \Vdash cs : \tau_1 : \tau_2}{\Gamma \Vdash p \Rightarrow e\ |\ cs : \tau_1 : \tau_2}\ \text{(T-CS-Add)}$$

A.2 Basic Evaluation Rules

The evaluation rules for expressions are as expected in a functional language, other than E-AttrAccess. We do not include rules here for the monadic constructs bind, `return`, and `fail`, since their results are dependent on the monad being used.

$$\frac{\gamma \vdash e \Downarrow tn \qquad (a, v) \in tn}{\gamma \vdash e.a \Downarrow v}\ \text{(E-AttrAccess)}$$

$$\frac{}{\gamma \vdash \texttt{true} \Downarrow \texttt{true}}\ \text{(E-True)}$$

$$\frac{}{\gamma \vdash \texttt{false} \Downarrow \texttt{false}}\ \text{(E-False)}$$

$$\frac{\gamma \vdash e_1 \Downarrow \texttt{true} \qquad \gamma \vdash e_2 \Downarrow \texttt{true}}{\gamma \vdash e_1 \mathbin{\&\&} e_2 \Downarrow \texttt{true}}\ \text{(E-And\_True)}$$

$$\frac{\gamma \vdash e_1 \Downarrow \texttt{false}}{\gamma \vdash e_1 \mathbin{\&\&} e_2 \Downarrow \texttt{false}}\ \text{(E-And\_False1)}$$

$$\frac{\gamma \vdash e_1 \Downarrow \texttt{true} \qquad \gamma \vdash e_2 \Downarrow \texttt{false}}{\gamma \vdash e_1 \mathbin{\&\&} e_2 \Downarrow \texttt{false}}\ \text{(E-And\_False2)}$$

$$\frac{x : v \in \gamma}{\gamma \vdash x \Downarrow v}\ \text{(E-Var)}$$

$$\frac{}{\gamma \vdash \lambda x : ty.\ e \Downarrow \texttt{closure}\ \gamma\ x\ e}\ \text{(E-Lam)}$$

$$\frac{\gamma \vdash e_1 \Downarrow \texttt{closure}\ \gamma'\ x\ e' \qquad \gamma \vdash e_2 \Downarrow v_2 \qquad \gamma'; x : v_2 \vdash e' \Downarrow v}{\gamma \vdash e_1\ e_2 \Downarrow v}\ \text{(E-App)}$$

$$\frac{\gamma \vdash e_1 \Downarrow v_1 \qquad \gamma; x : v_1 \vdash e_2 \Downarrow v_2}{\gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2}\ \text{(E-Let)}$$

$$\frac{\gamma \vdash e \Downarrow v_1 \qquad \gamma \Vdash cs, v_1 \Downarrow v}{\gamma \vdash \texttt{match}\ e\ \texttt{with}\ |\ cs \Downarrow v}\ \text{(E-Match)}$$

As expected, evaluation of clauses walks through a set of clauses, evaluating the expression associated with the first clause that matches:

$$\frac{\gamma \vdash^P p\ matches\ vm \Downarrow \gamma' \qquad \gamma' \vdash e \Downarrow v}{\gamma \Vdash p \Rightarrow e, vm \Downarrow v}\ \text{(E-CS-Single)}$$

$$\frac{\gamma \vdash^P p\ matches\ vm \Downarrow \gamma' \qquad \gamma' \vdash e \Downarrow v}{\gamma \Vdash p \Rightarrow e\ |\ cs, vm \Downarrow v}\ \text{(E-CS-Add\_Match)}$$

$$\frac{\neg(\gamma \vdash^P p\ matches\ vm \Downarrow \gamma') \qquad \gamma \Vdash cs, vm \Downarrow v}{\gamma \Vdash p \Rightarrow e \mid cs, vm \Downarrow v}$$
(E-CS-Add_NoMatch)

A.3 Extended Typing Rules

Our extended language includes attribute equations written in all three modes. While unrestricted equations use the basic typing relation, both restricted equations and implicit equations have their own extended typing relations.

Restricted Typing Relation. The typing relation for restricted attributes is essentially the same as the basic typing relation. The only change is to the attribute rule, which adds a restriction that the attribute must be a restricted attribute:

$$\frac{\Gamma \vdash_R e : NT \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash a@NT \qquad a \text{ restricted}}{\Gamma \vdash_R e.a : \tau}$$
(RT-AttrAccess)

This extra restriction enforces the attribute hierarchy, discussed in Section 3.1.

Implicit Typing Relation. The implicit typing relation contains a set of rules corresponding to the basic typing rules, with a rule for each combination of the monad from the equation being used implicitly in subexpressions, while avoiding nesting the same monad inside itself. For example, we have four rules for typing e_1 *&&* e_2: one for when both e_1 and e_2 have type `Bool`, one for when e_1 has type `M(Bool)`, one for when e_2 has type `M(Bool)`, and one for when both have type `M(Bool)`.

We have no rules for typing the explicit monadic constructs bind and `return`. We do, however, retain a typing rule for `fail`. In the same vein of banning explicit monad uses, we ban matching on the equation's monadic type and applying functions expecting arguments of this monadic type. We also ban applying functions whose arguments are functions ultimately resulting in this monadic type.

We require that attribute accesses obey the mode restrictions. This requires that all attributes be either restricted or implicit attributes. It also requires that implicit attributes being accessed must have the same monad as the equation currently being typed.

Finally, while the basic rules require nothing about completeness of pattern matching, if a set of clauses in a `match` expression is incomplete, the implicit typing relation requires the `match` to have a monadic type.

In these rules, M is the monadic type constructor for the type of the implicit attribute under consideration. Also, we use $\tau \neq M(\_)$ to mean τ is not a type built with the monadic type constructor M. Our full set of implicit typing rules for expressions:

$$\frac{\Gamma \vdash_I e : NT \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash a@NT \qquad a \text{ restricted}}{\Gamma \vdash_I e.a : \tau}$$
(IT-AttrAccess_R)

$$\frac{\Gamma \vdash_I e : NT \qquad \Gamma \vdash a : M(\tau) \qquad \Gamma \vdash a@NT \qquad a \text{ implicit}}{\Gamma \vdash_I e.a : M(\tau)}$$
(IT-AttrAccess_I)

$$\frac{\Gamma \vdash_I e : M(NT) \qquad \Gamma \vdash a : \tau \qquad \tau \neq M(\_) \qquad \Gamma \vdash a@NT \qquad a \text{ restricted}}{\Gamma \vdash_I e.a : M(\tau)}$$
(IT-AttrAccess_R_M)

$$\frac{\Gamma \vdash_I e : M(NT) \qquad \Gamma \vdash a : M(\tau) \qquad \Gamma \vdash a@NT \qquad a \text{ restricted}}{\Gamma \vdash_I e.a : M(\tau)}$$
(IT-AttrAccess_R_MAttr)

$$\frac{\Gamma \vdash_I e : M(NT) \qquad \Gamma \vdash a : M(\tau) \qquad \Gamma \vdash a@NT \qquad a \text{ implicit}}{\Gamma \vdash_I e.a : M(\tau)}$$
(IT-AttrAccess_I_MAttr)

$$\frac{}{\Gamma \vdash_I \texttt{true} : \texttt{Bool}}$$ (IT-True)

$$\frac{}{\Gamma \vdash_I \texttt{false} : \texttt{Bool}}$$ (IT-False)

$$\frac{\Gamma \vdash_I e_1 : \texttt{Bool} \qquad \Gamma \vdash_I e_2 : \texttt{Bool}}{\Gamma \vdash_I e_1 \mathbin{\&\&} e_2 : \texttt{Bool}}$$ (IT-And)

$$\frac{\Gamma \vdash_I e_1 : M(\texttt{Bool}) \qquad \Gamma \vdash_I e_2 : \texttt{Bool}}{\Gamma \vdash_I e_1 \mathbin{\&\&} e_2 : M(\texttt{Bool})}$$ (IT-And_M1)

$$\frac{\Gamma \vdash_I e_1 : \texttt{Bool} \qquad \Gamma \vdash_I e_2 : M(\texttt{Bool})}{\Gamma \vdash_I e_1 \mathbin{\&\&} e_2 : M(\texttt{Bool})}$$ (IT-And_M2)

$$\frac{\Gamma \vdash_I e_1 : M(\texttt{Bool}) \qquad \Gamma \vdash_I e_2 : M(\texttt{Bool})}{\Gamma \vdash_I e_1 \mathbin{\&\&} e_2 : M(\texttt{Bool})}$$ (IT-And_MBoth)

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash_I x : \tau}$$ (IT-Var)

$$\frac{\Gamma; x{:}\tau_1 \vdash_I e : \tau_2}{\Gamma \vdash_I \lambda x{:}\tau_1.\ e : \tau_1 \rightarrow \tau_2}$$ (IT-Lam)

$$\frac{\Gamma \vdash_I e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash_I e_2 : \tau_1 \qquad \textit{non-monadic-result}(\tau_1)}{\Gamma \vdash_I e_1\ e_2 : \tau_2}$$ (IT-App)

$$\frac{\Gamma \vdash_I e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash_I e_2 : M(\tau_1) \qquad \textit{non-monadic-result}(\tau_1) \qquad \tau_2 \neq M(\_)}{\Gamma \vdash_I e_1\ e_2 : M(\tau_2)}$$ (IT-App_MArg1)

$$\frac{\Gamma \vdash_I e_1 : \tau_1 \to M(\tau_2) \qquad \Gamma \vdash_I e_2 : M(\tau_1) \qquad \textit{non-monadic-result}(\tau_1)}{\Gamma \vdash_I e_1\ e_2 : M(\tau_2)}\ \text{(IT-App\_MArg2)}$$

$$\frac{\Gamma \vdash_I e_1 : M(\tau_1 \to \tau_2) \qquad \Gamma \vdash_I e_2 : \tau_1 \qquad \textit{non-monadic-result}(\tau_1) \qquad \tau_2 \neq M(\_)}{\Gamma \vdash_I e_1\ e_2 : M(\tau_2)}\ \text{(IT-App\_MFun1)}$$

$$\frac{\Gamma \vdash_I e_1 : M(\tau_1 \to M(\tau_2)) \qquad \Gamma \vdash_I e_2 : \tau_1 \qquad \textit{non-monadic-result}(\tau_1)}{\Gamma \vdash_I e_1\ e_2 : M(\tau_2)}\ \text{(IT-App\_MFun2)}$$

$$\frac{\Gamma \vdash_I e_1 : M(\tau_1 \to \tau_2) \qquad \Gamma \vdash_I e_2 : M(\tau_1) \qquad \textit{non-monadic-result}(\tau_1) \qquad \tau_2 \neq M(\_)}{\Gamma \vdash_I e_1\ e_2 : M(\tau_2)}\ \text{(IT-App\_MFunArg1)}$$

$$\frac{\Gamma \vdash_I e_1 : M(\tau_1 \to M(\tau_2)) \qquad \Gamma \vdash_I e_2 : M(\tau_1) \qquad \textit{non-monadic-result}(\tau_1)}{\Gamma \vdash_I e_1\ e_2 : M(\tau_2)}\ \text{(IT-App\_MFunArg2)}$$

$$\frac{}{\Gamma \vdash_I \texttt{fail} : M(\tau)}\ \text{(IT-Fail)}$$

$$\frac{\Gamma \vdash_I e_1 : \tau_1 \qquad \Gamma; x\!:\!\tau_1 \vdash_I e_2 : \tau_2 \qquad \tau_1 \neq M(\_)}{\Gamma \vdash_I \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{(IT-Let)}$$

$$\frac{\Gamma \vdash_I e_1 : M(\tau_1) \qquad \Gamma; x\!:\!\tau_1 \vdash_I e_2 : \tau_2 \qquad \tau_1 \neq M(\_) \qquad \tau_2 \neq M(\_)}{\Gamma \vdash_I \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : M(\tau_2)}\ \text{(IT-Let\_M)}$$

$$\frac{\Gamma \vdash_I e_1 : M(\tau_1) \qquad \Gamma; x\!:\!\tau_1 \vdash_I e_2 : M(\tau_2) \qquad \tau_1 \neq M(\_)}{\Gamma \vdash_I \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : M(\tau_2)}\ \text{(IT-Let\_MBoth)}$$

$$\frac{\Gamma \vdash_I e : \tau_1 \qquad \Gamma \Vdash_I cs : \tau_1 : \tau_2 \qquad \tau_1 \neq M(\_) \qquad \Gamma \vdash cs\ \textit{complete}}{\Gamma \vdash_I \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ cs : \tau_2}\ \text{(IT-Match)}$$

$$\frac{\Gamma \vdash_I e : M(\tau_1) \qquad \Gamma \Vdash_I cs : \tau_1 : \tau_2 \qquad \tau_1 \neq M(\_) \qquad \tau_2 \neq M(\_)}{\Gamma \vdash_I \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ cs : M(\tau_2)}\ \text{(IT-Match\_MD)}$$

$$\frac{\Gamma \vdash_I e : M(\tau_1) \qquad \tau_1 \neq M(\_) \qquad \Gamma \Vdash_I cs : \tau_1 : M(\tau_2)}{\Gamma \vdash_I \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ cs : M(\tau_2)}\ \text{(IT-Match\_MBoth)}$$

$$\frac{\Gamma \vdash_I e : \tau_1 \qquad \Gamma \Vdash_I cs : \tau_1 : \tau_2 \qquad \neg\Gamma \vdash cs\ \textit{complete} \qquad \tau_1 \neq M(\_) \qquad \tau_2 \neq M(\_)}{\Gamma \vdash_I \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ cs : M(\tau_2)}\ \text{(IT-Match\_Incomplete)}$$

$$\frac{\Gamma \vdash_I e : \tau_1 \qquad \Gamma \Vdash_I cs : \tau_1 : M(\tau_2) \qquad \neg\Gamma \vdash cs\ \textit{complete} \qquad \tau_1 \neq M(\_)}{\Gamma \vdash_I \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ cs : M(\tau_2)}\ \text{(IT-Match\_MIncomplete)}$$

We require clauses to all have the same pattern type, but we allow some clauses to have a monadic result type and some to have a non-monadic result type. If we encounter a mix, we choose the monadic result type:

$$\frac{\Gamma \vdash p\ \textit{match}\ \tau_1; \Gamma' \qquad \Gamma' \vdash_I e : \tau_2}{\Gamma \Vdash_I p \Rightarrow e : \tau_1 : \tau_2}\ \text{(IT-CS-Single)}$$

$$\frac{\Gamma \vdash p\ \textit{match}\ \tau_1; \Gamma' \qquad \Gamma' \vdash_I e : \tau_2 \qquad \Gamma \Vdash_I cs : \tau_1 : \tau_2}{\Gamma \Vdash_I p \Rightarrow e\ \texttt{|}\ cs : \tau_1 : \tau_2}\ \text{(IT-CS-Add)}$$

$$\frac{\Gamma \vdash p\ \textit{match}\ \tau_1; \Gamma' \qquad \Gamma' \vdash_I e : M(\tau_2) \qquad \Gamma \Vdash_I cs : \tau_1 : \tau_2}{\Gamma \Vdash_I p \Rightarrow e\ \texttt{|}\ cs : \tau_1 : M(\tau_2)}\ \text{(IT-CS-Add\_MHere)}$$

$$\frac{\Gamma \vdash p\ \textit{match}\ \tau_1; \Gamma' \qquad \Gamma' \vdash_I e : \tau_2 \qquad \Gamma \Vdash_I cs : \tau_1 : M(\tau_2)}{\Gamma \Vdash_I p \Rightarrow e\ \texttt{|}\ cs : \tau_1 : M(\tau_2)}\ \text{(IT-CS-Add\_MLater)}$$

We allow equations to have expressions with the same type as the implicit attribute being defined or the type inside the monadic type constructor in the implicit attribute. This allows assigning expressions of type `T` to an attribute of type `M(T)`, fitting with our thinking of the type without the monad. We also allow equations with no expression, which represent an attribute having no value of type `T` as a meaningful value. These might be used for defining the next step in small-step evaluation on productions representing values.

$$\frac{\Gamma \vdash_I e : M(\tau) \qquad \Gamma \vdash a : M(\tau) \qquad t\!:\!NT \in \Gamma \qquad \Gamma \vdash a@NT \qquad a\ \text{implicit}}{\Gamma \vdash_I t.a\texttt{=}e;\ OK}\ \text{(IT-EQ\_Basic)}$$

$$\frac{\Gamma \vdash_I e : \tau \qquad \Gamma \vdash a : M(\tau) \qquad t\!:\!NT \in \Gamma \qquad \Gamma \vdash a@NT \qquad a\ \text{implicit}}{\Gamma \vdash_I t.a\texttt{=}e;\ OK}\ \text{(IT-EQ\_M)}$$

$$\frac{\Gamma \vdash a : M(\tau_2) \qquad t\!:\!NT \in \Gamma \qquad \Gamma \vdash a@NT \qquad a\ \text{implicit}}{\Gamma \vdash_I t.a\texttt{=};\ OK}\ \text{(IT-EQ\_Empty)}$$

A.4 Implicit Evaluation

The rules for evaluating implicit expressions are mostly the same as the basic rules. We have two rules for attribute accesses to only evaluate them to non-monadic values, as discussed in Section 3.2. We have three rules for evaluating closures, with the evaluation depending on the closure's tag and ensuring we do not evaluate to a monadic value.

$$\frac{\gamma \vdash_I e \Downarrow tn \qquad (a, v') \in tn \qquad v \in_M v' \qquad v \text{ is not monadic in } M}{\gamma \vdash_I e.a \Downarrow v} \ \text{(IE-AttrAccess\_E)}$$

$$\frac{\gamma \vdash_I e \Downarrow tn \qquad (a, v) \in tn \qquad v \text{ is not monadic in } M}{\gamma \vdash_I e.a \Downarrow v} \ \text{(IE-AttrAccess)}$$

$$\frac{}{\gamma \vdash_I \mathtt{true} \Downarrow \mathtt{true}} \ \text{(IE-True)}$$

$$\frac{}{\gamma \vdash_I \mathtt{false} \Downarrow \mathtt{false}} \ \text{(IE-False)}$$

$$\frac{\gamma \vdash_I e_1 \Downarrow \mathtt{true} \qquad \gamma \vdash_I e_2 \Downarrow \mathtt{true}}{\gamma \vdash_I e_1 \mathbin{\&\&} e_2 \Downarrow \mathtt{true}} \ \text{(IE-And\_True)}$$

$$\frac{\gamma \vdash_I e_1 \Downarrow \mathtt{false}}{\gamma \vdash_I e_1 \mathbin{\&\&} e_2 \Downarrow \mathtt{false}} \ \text{(IE-And\_False1)}$$

$$\frac{\gamma \vdash_I e_1 \Downarrow \mathtt{true} \qquad \gamma \vdash_I e_2 \Downarrow \mathtt{false}}{\gamma \vdash_I e_1 \mathbin{\&\&} e_2 \Downarrow \mathtt{false}} \ \text{(IE-And\_False2)}$$

$$\frac{x : v \in \gamma}{\gamma \vdash_I x \Downarrow v} \ \text{(IE-Var)}$$

$$\frac{}{\gamma \vdash_I \lambda x : \tau .\ e \Downarrow \mathtt{closure}\ \gamma\ x\ e\ \mathtt{implicit}} \ \text{(IE-Lam)}$$

$$\frac{\gamma \vdash_I e_1 \Downarrow \mathtt{closure}\ \gamma'\ x\ e'\ \mathtt{implicit} \qquad \gamma \vdash_I e_2 \Downarrow v_2 \qquad \gamma'; x : v_2 \vdash_I e' \Downarrow v}{\gamma \vdash_I e_1\ e_2 \Downarrow v} \ \text{(IE-App\_Im)}$$

$$\frac{\gamma \vdash_I e_1 \Downarrow \mathtt{closure}\ \gamma'\ x\ e'\ \mathtt{explicit} \qquad \gamma \vdash_I e_2 \Downarrow v_2 \qquad \gamma'; x : v_2 \vdash_E e' \Downarrow v' \qquad v \in_M v' \qquad v \text{ is not monadic in } M}{\gamma \vdash_I e_1\ e_2 \Downarrow v} \ \text{(IE-App\_Ex\_E)}$$

$$\frac{\gamma \vdash_I e_1 \Downarrow \mathtt{closure}\ \gamma'\ x\ e'\ \mathtt{explicit} \qquad \gamma \vdash_I e_2 \Downarrow v_2 \qquad \gamma'; x : v_2 \vdash_E e' \Downarrow v \qquad v \text{ is not monadic in } M}{\gamma \vdash_I e_1\ e_2 \Downarrow v} \ \text{(IE-App\_Ex)}$$

$$\frac{\gamma \vdash_I e_1 \Downarrow v_1 \qquad \gamma; x : v_1 \vdash_I e_2 \Downarrow v_2}{\gamma \vdash_I \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow v_2} \ \text{(IE-Let)}$$

$$\frac{\gamma \vdash_I e \Downarrow v_1 \qquad \gamma \Vdash_I cs, v_1 \Downarrow v}{\gamma \vdash_I \mathtt{match}\ e\ \mathtt{with}\ \mathtt{|}\ cs \Downarrow v} \ \text{(IE-Match)}$$

As expected, evaluation of clauses walks through a set of clauses, evaluating the expression associated with the first clause that matches:

$$\frac{\gamma \vdash^P p\ matches\ vm \Downarrow \gamma' \qquad \gamma' \vdash_I e \Downarrow v}{\gamma \Vdash_I p \Rightarrow e, v \Downarrow} \ \text{(IE-CS-Single)}$$

$$\frac{\gamma \vdash^P p\ matches\ vm \Downarrow \gamma' \qquad \gamma' \vdash_I e \Downarrow v}{\gamma \Vdash_I p \Rightarrow e \mid cs, vm \Downarrow v} \ \text{(IE-CS-Add\_Match)}$$

$$\frac{\neg(\gamma \vdash^P p\ matches\ vm \Downarrow \gamma') \qquad \gamma \Vdash_I cs, vm \Downarrow v}{\gamma \Vdash_I p \Rightarrow e \mid cs, vm \Downarrow v} \ \text{(IE-CS-Add\_NoMatch)}$$

A.5 Closure Evaluation in the Extended Language

The values in the extended language are the same as the values in the basic language other than closures. In the basic language, we have closures (`closure` γ `x e`), where γ is an evaluation context, `x` is the variable bound by the closure, and `e` is the body of the closure. In the extended language, we have tagged closures (`closure` γ `x e tag`), where `tag` is either `implicit` or `explicit`, referring to whether monads which appear in it are treated implicitly or explicitly.

An abstraction in the implicit evaluation evaluates to a closure tagged with `implicit`:

$$\frac{}{\gamma \vdash_I \lambda x : \tau .\ e \Downarrow \mathtt{closure}\ \gamma\ x\ e\ \mathtt{implicit}} \ \text{(IE-Lam)}$$

Similarly, the explicit evaluation relation evaluates to a closure tagged with `explicit`:

$$\frac{}{\gamma \vdash_E \lambda x : ty .\ e \Downarrow \mathtt{closure}\ \gamma\ x\ e\ \mathtt{explicit}} \ \text{(EE-Lam)}$$

In the implicit evaluation relation, an application evaluates the body of the closure using the appropriate relation for the tag of the closure. If it is tagged as implicit, we use the implicit evaluation relation to evaluate the body of the closure:

$$\frac{\gamma \vdash_I e_1 \Downarrow \mathtt{closure}\ \gamma'\ x\ e'\ \mathtt{implicit} \qquad \gamma \vdash_I e_2 \Downarrow v_2 \qquad \gamma'; x : v_2 \vdash_I e' \Downarrow v}{\gamma \vdash_I e_1\ e_2 \Downarrow v} \ \text{(IE-App\_Im)}$$

If it is tagged as explicit, we evaluate the body of the closure using the explicit evaluation relation. However, we need two rules, just as we explained we need two rules for attribute accesses in Section 3.2 due to the possibility of monadic values as results. If the result is a monadic value, the application evaluates to the value inside it:

$$\frac{\gamma \vdash_I e_1 \Downarrow \texttt{closure}\ \gamma'\ x\ e'\ \texttt{explicit} \qquad \gamma \vdash_I e_2 \Downarrow v_2 \qquad \gamma';x\!:\!v_2 \vdash_E e' \Downarrow v' \qquad v \in_M v' \qquad v \text{ is not monadic in } M}{\gamma \vdash_I e_1\ e_2 \Downarrow v} \quad \text{(IE-App\_Ex\_E)}$$

If the result is not a monadic value, the application evaluates to the value to which the closure's body evaluated:

$$\frac{\gamma \vdash_I e_1 \Downarrow \texttt{closure}\ \gamma'\ x\ e'\ \texttt{explicit} \qquad \gamma \vdash_I e_2 \Downarrow v_2 \qquad \gamma';x\!:\!v_2 \vdash_E e' \Downarrow v \qquad v \text{ is not monadic in } M}{\gamma \vdash_I e_1\ e_2 \Downarrow v} \quad \text{(IE-App\_Ex)}$$

We do not need two separate rules for evaluating implicit attributes because the implicit evaluation relation, used for evaluating the closure body, cannot result in a monadic value.

We have a similar split in the explicit evaluation relation. We have two rules, one for explicit closures and one for implicit closures:

$$\frac{\gamma \vdash_E e_1 \Downarrow \texttt{closure}\ \gamma'\ x\ e'\ \texttt{implicit} \qquad \gamma \vdash_E e_2 \Downarrow v_2 \qquad \gamma';x\!:\!v_2 \vdash_I e' \Downarrow v'}{\gamma \vdash_I e_1\ e_2 \Downarrow v} \quad \text{(IE-App\_Im)}$$

$$\frac{\gamma \vdash_E e_1 \Downarrow \texttt{closure}\ \gamma'\ x\ e'\ \texttt{explicit} \qquad \gamma \vdash_E e_2 \Downarrow v_2 \qquad \gamma';x\!:\!v_2 \vdash_E e' \Downarrow v}{\gamma \vdash_E e_1\ e_2 \Downarrow v} \quad \text{(IE-App\_Ex)}$$

As in implicit evaluation, this ensures the body of the closure is evaluated with the relation which can handle it.

B Rewriting Rules

Our rewriting rules correspond very closely with the implicit typing rules, since we are rewriting an implicitly-typable expression into a basic-typable expression. We rewrite to a particular type, which is both the type of the original expression under the implicit typing rules and the type of the rewritten expression under the basic typing rules.

There is one rewriting rule for each typing rule other than for `match` expressions. Because our rewriting completes incomplete sets of clauses, we need to duplicate some typing rules to get our set of rewriting rules.

As in the typing rules, the monadic type constructor M is the monadic type constructor for the type of the equation of which the expression is part. Also as in the typing rules, we use $\tau \neq M(\_)$ to mean that τ is not built with the monadic type constructor M. Our full set of rewriting rules for expressions:

$$\frac{\Gamma \vdash e \rightsquigarrow e' : NT \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash a@NT}{\Gamma \vdash e.a \rightsquigarrow e'.a : \tau} \quad \text{(RW-AttrAccess)}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : M(NT) \qquad \Gamma \vdash a : \tau \qquad \tau \neq M(\_) \qquad \Gamma \vdash a@NT}{\Gamma \vdash e.a \rightsquigarrow e' \gg= \lambda x : NT.\ \texttt{return}\ (x.a) : M(\tau)} \quad \text{(RW-AttrAccess\_M)}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e' : M(NT) \qquad \Gamma \vdash a : M(\tau) \qquad \Gamma \vdash a@NT}{\Gamma \vdash e.a \rightsquigarrow e' \gg= \lambda x : NT.\ x.a : M(\tau)} \quad \text{(RW-AttrAccess\_MBoth)}$$

$$\frac{}{\Gamma \vdash \texttt{true} \rightsquigarrow \texttt{true} : Bool} \quad \text{(RW-True)}$$

$$\frac{}{\Gamma \vdash \texttt{false} \rightsquigarrow \texttt{false} : Bool} \quad \text{(RW-False)}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : Bool \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : Bool}{\Gamma \vdash e_1\ \&\&\ e_2 \rightsquigarrow e_1'\ \&\&\ e_2' : Bool} \quad \text{(RW-And)}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : M(Bool) \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : Bool \qquad x \text{ fresh in } e_2'}{\Gamma \vdash e_1\ \&\&\ e_2 \rightsquigarrow e_1' \gg= \lambda x : Bool.\ \texttt{return}\ (x\ \&\&\ e_2') : M(Bool)} \quad \text{(RW-And\_M1)}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : Bool \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : M(Bool)}{\Gamma \vdash e_1\ \&\&\ e_2 \rightsquigarrow \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else return false} : M(Bool)} \quad \text{(RW-And\_M2)}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : M(Bool) \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : M(Bool) \qquad x \text{ fresh in } e_2'}{\Gamma \vdash e_1\ \&\&\ e_2 \rightsquigarrow e_1' \gg= \lambda x : Bool.\ \texttt{if}\ x\ \texttt{then}\ e_2'\ \texttt{else return false} : M(Bool)} \quad \text{(RW-And\_MBoth)}$$

$$\frac{x\!:\!\tau \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \tau} \quad \text{(RW-Var)}$$

$$\frac{\Gamma;x\!:\!\tau_1 \vdash e \rightsquigarrow e' : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e \rightsquigarrow \lambda x : \tau_1.\ e' : \tau_1 \rightarrow \tau_2} \quad \text{(RW-Lam)}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_1}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau_2} \quad \text{(RW-App)}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : M(\tau_1) \qquad \tau_2 \neq M(\_) \qquad f \text{ fresh in } e_2' \qquad x \neq f}{\Gamma \vdash e_1\ e_2 \rightsquigarrow \texttt{let}\ f = e_1'\ \texttt{in}\ (e_2' \gg= \lambda x : \tau_1.\ \texttt{return}\ (f\ x)) : M(\tau_2)} \quad \text{(RW-App\_MArg1)}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \rightarrow M(\tau_2) \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : M(\tau_1) \qquad f \text{ fresh in } e_2' \qquad x \neq f}{\Gamma \vdash e_1\ e_2 \rightsquigarrow \texttt{let}\ f = e_1'\ \texttt{in}\ (e_2' \gg= \lambda x : \tau_1.\ f\ x) : M(\tau_2)} \quad \text{(RW-App\_MArg2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow e_1' : M(\tau_1 \to \tau_2) \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_1 \\ \tau_2 \neq M(\_) \qquad f \text{ fresh in } e_2'\end{array}}{\begin{array}{c}\Gamma \vdash e_1\ e_2 \rightsquigarrow \\ e_1' \gg= \lambda f : \tau_1 \to \tau_2.\ \texttt{return}\ (f\ e_2') : M(\tau_2)\end{array}}$$

(RW-App_MFun1)

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow e_1' : M(\tau_1 \to M(\tau_2)) \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_1 \\ f \text{ fresh in } e_2'\end{array}}{\begin{array}{c}\Gamma \vdash e_1\ e_2 \rightsquigarrow \\ e_1' \gg= \lambda f : \tau_1 \to M(\tau_2).\ f\ e_2' : M(\tau_2)\end{array}}$$

(RW-App_MFun2)

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow e_1' : M(\tau_1 \to \tau_2) \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : M(\tau_1) \\ \tau_2 \neq M(\_) \qquad f \text{ fresh in } e_2' \qquad f \neq x\end{array}}{\begin{array}{c}\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1' \gg= \\ \lambda f : \tau_1 \to \tau_2.\ e_2' \gg= \lambda x : \tau_1.\ \texttt{return}\ (f\ x) : M(\tau_2)\end{array}}$$

(RW-App_MFunArg1)

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow e_1' : M(\tau_1 \to M(\tau_2)) \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : M(\tau_1) \\ f \text{ fresh in } e_2' \qquad f \neq x\end{array}}{\begin{array}{c}\Gamma \vdash e_1\ e_2 \rightsquigarrow \\ e_1' \gg= \lambda f : \tau_1 \to M(\tau_2).\ e_2' \gg= \lambda x : \tau_1.\ f\ x : M(\tau_2)\end{array}}$$

(RW-App_MFunArg2)

$$\frac{}{\Gamma \vdash \texttt{fail} \rightsquigarrow \texttt{fail} : M(\tau)} \quad \text{(RW-Fail)}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma; x{:}\tau_1 \vdash e_2 \rightsquigarrow e_2' : \tau_2 \qquad \tau_1 \neq M(\_)}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' : \tau_2}$$

(RW-Let)

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow e_1' : M(\tau_1) \qquad \Gamma; x{:}\tau_1 \vdash e_2 \rightsquigarrow e_2' : \tau_2 \\ \tau_2 \neq M(\_)\end{array}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow e_1' \gg= \lambda x : \tau_1.\ \texttt{return}\ e_2' : M(\tau_2)}$$

(RW-Let_M)

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : M(\tau_1) \qquad \Gamma; x{:}\tau_1 \vdash e_2 \rightsquigarrow e_2' : M(\tau_2)}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow e_1' \gg= \lambda x : \tau_1.\ e_2' : M(\tau_2)}$$

(RW-Let_MBoth)

$$\frac{\begin{array}{c}\Gamma \vdash e \rightsquigarrow e' : \tau_1 \qquad \Gamma \Vdash cs \rightsquigarrow cs' : \tau_1 : \tau_2 \\ \Gamma \vdash cs\ complete \qquad \tau_1 \neq M(\_)\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ cs \rightsquigarrow \texttt{match}\ e'\ \texttt{with}\ \texttt{|}\ cs' : \tau_2}$$

(RW-Match)

$$\frac{\begin{array}{c}\Gamma \vdash e \rightsquigarrow e' : \tau_1 \qquad \Gamma \Vdash cs\ \texttt{|}\ \_ \Rightarrow \texttt{fail} \rightsquigarrow cs' : \tau_1 : M(\tau_2) \\ \neg\Gamma \vdash cs\ complete \qquad \tau_1 \neq M(\_)\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ cs \rightsquigarrow \texttt{match}\ e'\ \texttt{with}\ \texttt{|}\ cs' : M(\tau_2)}$$

(RW-Match_Incomplete)

$$\frac{\begin{array}{c}\Gamma \vdash e \rightsquigarrow e' : M(\tau_1) \qquad \Gamma \Vdash cs \rightsquigarrow cs' : \tau_1 : \tau_2 \\ \tau_2 \neq M(\_) \qquad \Gamma \vdash cs\ complete \qquad x \text{ fresh in } cs'\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ cs \rightsquigarrow \\ e' \gg= \lambda x : \tau_1.\ \texttt{return}\ (\texttt{match}\ x\ \texttt{with}\ \texttt{|}\ cs'\ ) : M(\tau_1)\end{array}}$$

(RW-Match_MD)

$$\frac{\begin{array}{c}\Gamma \vdash e \rightsquigarrow e' : M(\tau_1) \qquad \neg(\Gamma \vdash cs\ complete) \\ \Gamma \Vdash cs\ \texttt{|}\ \_ \Rightarrow \texttt{fail} \rightsquigarrow cs' : \tau_1 : M(\tau_2) \qquad x \text{ fresh in } cs'\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ cs \rightsquigarrow \\ e' \gg= \lambda x : \tau_1.\ \texttt{match}\ x\ \texttt{with}\ \texttt{|}\ cs' : M(\tau_1)\end{array}}$$

(RW-Match_MD_Incomplete)

$$\frac{\begin{array}{c}\Gamma \vdash e \rightsquigarrow e' : M(\tau_1) \qquad \Gamma \Vdash cs \rightsquigarrow cs' : \tau_1 : M(\tau_2) \\ x \text{ fresh in } cs' \qquad \Gamma \vdash cs\ complete\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \texttt{|}\ cs \rightsquigarrow \\ e' \gg= \lambda x : \tau_1.\ \texttt{match}\ x\ \texttt{with}\ \texttt{|}\ cs : M(\tau_2)\end{array}}$$

(RW-Match_MBoth)

Some clauses in a set may initially have different result types, some resulting in type `T` and some resulting in type `M(T)`. When we encounter such mismatched types, we add monadic `return` operations, either directly or with *returnify*, to make all clauses result in the same type:

$$\frac{\Gamma \vdash p\ match\ \tau_1; \Gamma' \qquad \Gamma' \vdash e \rightsquigarrow e' : \tau_2}{\Gamma \Vdash p \Rightarrow e \rightsquigarrow p \Rightarrow e' : \tau_1 : \tau_2} \quad \text{(RW-CS-Single)}$$

$$\frac{\begin{array}{c}\Gamma \vdash p\ match\ \tau_1; \Gamma' \qquad \Gamma' \vdash e \rightsquigarrow e' : \tau_2 \\ \Gamma \Vdash cs \rightsquigarrow cs' : \tau_1 : \tau_2\end{array}}{\Gamma \Vdash p \Rightarrow e\ \texttt{|}\ cs \rightsquigarrow p \Rightarrow e'\ \texttt{|}\ cs' : \tau_1 : \tau_2} \quad \text{(RW-CS-Add)}$$

$$\frac{\begin{array}{c}\Gamma \vdash p\ match\ \tau_1; \Gamma' \qquad \Gamma' \vdash e \rightsquigarrow e' : \tau_2 \\ \Gamma \Vdash cs \rightsquigarrow cs' : \tau_1 : M(\tau_2)\end{array}}{\Gamma \Vdash p \Rightarrow e\ \texttt{|}\ cs \rightsquigarrow p \Rightarrow \texttt{return}\ e'\ \texttt{|}\ cs' : \tau_1 : M(\tau_2)}$$

(RW-CS-Add_MLater)

$$\frac{\begin{array}{c}\Gamma \vdash p\ match\ \tau_1; \Gamma' \qquad \Gamma' \vdash e \rightsquigarrow e' : M(\tau_2) \\ \Gamma \Vdash cs \rightsquigarrow cs' : \tau_1 : \tau_2 \qquad returnify\ cs'\ cs''\end{array}}{\Gamma \Vdash p \Rightarrow e\ \texttt{|}\ cs \rightsquigarrow p \Rightarrow e'\ \texttt{|}\ cs'' : M(\tau_2) :}$$

(RW-CS-Add_MHere)

C Value Relations

The relation describing when values are appropriately related as evaluation results before and after rewriting can be thought of as a translation relation for values. As expected, constant values, such as `true` and `false` are related to themselves, but more complex values require a more careful treatment.

Formally, $v \sim_v v'$ holds if and only if one of the following rules holds:

- Anything other than a closure or a tree node requires both values equal:
  - $\neg(\exists \gamma\ x\ e\ t.\ v = closure\ \gamma\ x\ e\ t)$
  - $\neg(\exists tn.\ v = tn)$
  - $v = v'$
- A tree node is related to another tree node if they are references to the same node in the tree.

- An original explicit closure becomes a closure with the same body but a different, related context:
  - $v = closure\ \gamma\ x\ e$ `explicit`
  - $v' = closure\ \gamma'\ x\ e$
  - $\gamma\ \sim_c \langle x, e\rangle\ \gamma'$
- An original implicit closure becomes a closure with a rewritten body and related context:
  - $v = closure\ \gamma\ x\ e$ `implicit`
  - $v' = closure\ \gamma'\ x\ e'$
  - $\exists \Gamma\ \tau.\ \Gamma \vdash e \rightsquigarrow e' : \tau$ and for each name in both Γ and γ', the value in γ' has the type found in Γ
  - $\gamma\ \sim_c \langle x, e\rangle\ \gamma'$

The relation $\gamma\ \sim_c \langle x, e\rangle\ \gamma'$ holds if, for all the free variables in e other than x, if γ' has a value for the free variable, then γ has a related value and vice versa:

- $\forall y\ v'.\ freevar\ y\ e \implies y \neq x \implies x : v' \in \gamma' \implies \exists v.\ y : v \in \gamma\ \wedge\ v \sim_v v'$
- $\forall y\ v.\ freevar\ y\ e \implies y \neq x \implies x : v \in \gamma \implies \exists v'.\ y : v' \in \gamma'\ \wedge\ v \sim_v v'$

The way to think about this is that we want the two contexts to have related values for all the variables which might be referenced in evaluating the closures they are part of, which is the free variables minus the variable for the closures.

D Proof Sketch for Evaluation Correctness

Our evaluation correctness property for rewriting is composed of two properties:

- If the original equation evaluated to a value v, the rewritten equation evaluates to a monadic value containing a value related to v.
- If the rewritten equation evaluates, either there is a value v' in the result and the original equation evaluated to a value related to v' or the result is a monadic failure and the original equation could not evaluate.

Each property follows directly from a related, but more complex, property for expressions. The added complexity comes from the context possibly containing related values rather than exactly the same values. They also become more complex in the expression version because the rewritten expression may not have a monadic type, so we need to consider cases based on the type of the expression.

Because our properties are only concerned with containment and non-containment of values, it doesn't matter how many empty (failure) or non-empty (contain a value) constructors the monad being used has. We can prove the properties as if the monad has one of each, which means we can prove the properties for `Maybe` and have the proofs apply to any other monad with an acceptable definition of monadic containment.

We will look at a sketch of the proof of each property.

Original Evaluation Implies Rewritten Evaluation. We proceed by induction on the derivation of evaluation for the original expression, also considering cases on rewriting.

If no bind is inserted, the values of the subderivations are essentially equal between the original and the rewritten version because there are no monads in the types, and the basic rule corresponding to the original implicit rule can be used for the evaluation.

If the original evaluation used a rule which removed a value v from a monadic value, such as IT-AttrAccess_E, the subderivation in the basic language will result in a monadic value containing a value v' such that we have $v \sim_v v'$. Then the monadic value after the rewriting contains a value related to the value before the rewriting.

If a bind was inserted, the expression being bound in originally evaluated to a value v, and its rewritten version evaluates to a monadic value containing a value v' such that we have $v \sim_v v'$. By our monadic containment requirement that $m \gg= f$ be equal to $f\ v_0$ when $v_0 \in_M m$, the result of the inserted bind is the result of doing the computation originally done with v with v' instead, a value which is essentially equal. If we inserted a `return`, the result is a monadic value containing a value related to the original value, since we require that monadic containment be defined so $v \in_M$ `return` v.

Rewritten Evaluation Implies Original Evaluation. As with the other direction, we proceed by induction on the derivation of evaluation, this time for the rewritten expression, also considering cases on rewriting the expression.

If it evaluates to a non-monadic value, we can use the related implicit evaluation rule to evaluate the original expression, since the values involved are essentially equal.

If it evaluates to a non-failure monadic value and a bind was inserted, our monadic containment requirements tell us there was a value v' in the monadic value bound in, and thus that the result of the bind was the function applied to this value. The inductive hypothesis tells us that the expression bound in originally evaluated to a value v such that $v \sim_v v'$. This value v was used in the computation represented by the function for the bind, and, because v and v' are essentially equal, the result of this computation must be related as well.

Our proof also relies on our ban on using monadic values explicitly. If, for example, we allowed matching on monadic values, a `match` in the rewritten expression could generate spurious values. Consider the following `match` on an attribute of type `Maybe(Int)`:

```
 match t.a with | just(x) ⇒ x | nothing() ⇒ 5
```

If `t.a` were originally unevaluable and is now `nothing()`, we would have produced a non-failure value which had no counterpart in the original evaluation rules. Because the above `match` could not be written in the original typing rules, we avoid this problem.

A monadic failure result in the rewritten evaluation must ultimately come from either a `fail` construct in the expression, an attribute access, or a function evaluation. The implicit rules do not evaluate `fail`, and attribute accesses and function evaluations only allow non-monadic results into the implicit evaluation. Furthermore, only corresponding subexpressions are evaluated in the rewritten and original expressions, so it cannot be that the monadic failure came from a subexpression which may be "skipped" in the original evaluation to produce a value. Therefore a monadic failure result in a rewritten expression implies the original expression cannot evaluate.

References

[1] Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 666âĂŞ679, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3009837.3009866.

[2] John Tang Boyland. Remote attribute grammars. *Journal of the ACM*, 52(4):627–687, 2005. doi: 10.1145/1082036.1082042.

[3] Jordi Cabot and Martin Gogolla. *Object Constraint Language (OCL): A Definitive Guide*, pages 58–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi: 10.1007/978-3-642-30982-3_3.

[4] Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 41–60, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-37036-6_3.

[5] Martin Erwig and Deling Ren. Monadification of functional programs. *Science of Computer Programming*, 52(1):101–129, 2004. doi: 10.1016/j.scico.2004.03.004.

[6] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: a complete, flexible compiler construction system. *CACM*, 35(2):121–130, 1992. doi: 10.1145/129630.129637.

[7] Görel Hedin. Reference attribute grammars. *Informatica*, 24(3):301–317, 2000.

[8] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256âĂŞ290, April 2002. doi: 10.1145/505145.505149.

[9] M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In *Proceedings of 6h International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 1984. doi: 10.1007/3-540-12925-1_37.

[10] Ted Kaminski and Eric Van Wyk. Modular well-definedness analysis for attribute grammars. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE)*, volume 7745 of *Lecture Notes in Computer Science*, pages 352–371. Springer, September 2012. doi: 10.1007/978-3-642-36089-3_20.

[11] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to C: The ableC extensible language framework. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):98:1–98:29, October 2017. doi: 10.1145/3138224.

[12] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994. doi: 10.1007/BF01177548.

[13] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13: 229–256, 1980. doi: 10.1007/BF00288644.

[14] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. doi: 10.1007/BF01692511. Corrections in **5**(1971) pp. 95–96.

[15] Xavier Leroy. *The Caml Light system, documentation and user's guide*, December 1997. URL http://caml.inria.fr/pub/docs/manual-caml-light/. Release 0.74, see also https://ocaml.org/caml-light/.

[16] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. doi: 10.1016/0890-5401(91)90052-4. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[17] Peter D Mosses. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:195 – 228, 2004. doi: 10.1016/j.jlap.2004.03.008. Structural Operational Semantics.

[18] Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009. doi: 10.1016/j.entcs.2009.07.073. Proceedings of the Fifth Workshop on Structural Operational Semantics (SOS 2008).

[19] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2): 196–255, June 1995. doi: 10.1145/210376.197409.

[20] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[21] Gordon D Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:3 – 15, 2004. doi: 10.1016/j.jlap.2004.03.009.

[22] August Schwerdfeger and Eric Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 199–210, New York, NY, USA, June 2009. ACM. doi: 10.1145/1542476.1542499.

[23] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ML. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 15–27. ACM, 2011. doi: 10.1145/2034773.2034778.

[24] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010. doi: 10.1016/j.scico.2009.07.004.

[25] Harold H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 131–145. ACM, 1989. doi: 10.1145/73141.74830.

[26] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995. doi: 10.1007/978-3-662-02880-3_8.