Implicit Parameters for Logic Programming

Magnus Madsen
Aalborg University
Denmark
magnus@cs.aau.dk

Ondřej Lhoták
University of Waterloo
Canada
olhotak@uwaterloo.ca

ABSTRACT

Implicit parameters allow programmers to omit certain arguments from function calls and have them automatically inferred by the compiler based on their types. At every call site, the compiler determines the values of the implicit parameters based on their declared types and the bindings currently in implicit scope. The programmer controls this mechanism in two ways: by adding bindings to the implicit scope, or by explicitly providing the implicit parameters for the function call.

Implicit parameters are known from functional and object-oriented languages such as Haskell and Scala. In recent years, more languages have added support for implicit parameters, including Agda, Coq, and Idris. Implicit parameters have played an impressive role as the foundation for a broad range of language features such as type classes, capability and effect systems, software transactional memory, macros, and more.

In this paper, we propose a design of implicit parameters for typed Horn clause based logic programming languages, such as Datalog and Prolog. We illustrate the usefulness of implicit parameters and show how they support logic programming in the large. We explore some of the differences that arise between implicit parameters in functional languages and in logic languages.

CCS CONCEPTS

• **Theory of computation** → **Constraint and logic programming**; • **Software and its engineering** → **Constraint and logic languages**;

KEYWORDS

typed logic programming, implicit parameters

ACM Reference Format:
Magnus Madsen and Ondřej Lhoták. 2018. Implicit Parameters for Logic Programming. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3236950.3236953

1 INTRODUCTION

Implicit parameters are a programming language feature that allows certain arguments to be omitted from a function call. The values of the omitted arguments are inferred by the compiler based on their declared types and the bindings in implicit scope. The implicit scope is composed of implicitly passed arguments and bindings explicitly put into the implicit scope. Implicit parameters allow parameterization of functions and methods with extra arguments without requiring those arguments to be explicitly passed through every function call by the programmer.

For example, in the Scala programming language we can declare a method with the signature:

```
def evaluate(q: Query)(implicit c: Conn): Result = ...
```

The method expects two parameters: the explicit parameter q which is the query to run and the implicit parameter c which is the connection to the database. Whenever the programmer wants to evaluate a query, he or she may call the evaluate method omitting the second implicit parameter:

```
val result = evaluate(someQuery)
```

In this case, the compiler automatically infers the implicit parameter c based on its declared type. The question of *how* the value of the implicit parameter is found opens up a large design space with many proposals [8, 11, 12].

Implicit parameters were originally proposed by Lewis et al. [8] as a mechanism to pass an argument to a function deep inside multiple levels of recursion. Since the original work, implicit parameters have played an impressive role as the foundation for a broad range of other language features including type classes [11], capability and effect systems [6, 13], software transactional memory [3], language virtualization [10], and macros [4].

In this paper, we show that implicit parameters are useful for typed logic programming. A logic program, in the context of this paper, is a collection of declarative rules. Each rule is a Horn clause and consists of a head predicate and zero

PPDP '18, September 3–5, 2018, Frankfurt am Main, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6441-6/18/09…$15.00
https://doi.org/10.1145/3236950.3236953

or more body predicates. A predicate consists of a predicate symbol and one or more terms, either variables or constants.

Intuitively, we can think of a predicate like a function call in other languages, but there are differences. In most other programming languages, a variable is *declared* in one place in the program and can then be *used* in multiple other places. There are two parts to the implementation of implicits in these languages: at the declaration site, a variable may optionally be designated as a *candidate* for implicit resolution and assigned a type, and then at an implicit use site, the compiler uses the type of term required by the use to select a candidate variable of compatible type. Logic languages like Datalog lack variable declarations: all occurrences of a variable are uses. Thus, there can be implicit use sites, but there is no obvious place to introduce candidate variables for implicit resolution. In our system, the general solution to this issue is for the compiler to effectively invent new variables, one corresponding to each unique type that occurs at any implicit use site within the Horn clause.

In this work, we are interested in logic programming on a large scale: We do not expect that implicit parameters will add much value to programs that are already small and elegant. Implicit parameters will probably not simplify an implementation of the Eight Queens Problem. Rather, our inspiration is large-scale program analyses that runs to tens of thousands of lines of Datalog code [2, 14].

Implicit parameters have been a powerful vessel as the foundation for other programming language features. We hope that by bringing implicit parameters to logic languages we can help spur similar developments in this space.

The main contributions of this paper are:

- We illustrate how implicit parameters are useful for logic programming.
- We present a design of implicit parameters for typed logic programming languages.
- We present a program transformation that makes implicit parameters explicit.
- We formulate a set of requirements that a design of implicit parameters should satisfy and we prove that our design conforms to these properties, including type-safety of the transformation.

2 MOTIVATION

We motivate the need for implicit parameters through an example: We show parts of a flow– and context-sensitive points-to analysis implemented in Flix. Flix is a functional and logic language inspired by Datalog extended with user-defined lattices and functions. A specific use case of Flix is for the implementation of large-scale program analyses [9]. The use of Flix in this paper is immaterial; the work is equally applicable to any typed logic programming language.

2.1 Example: A Points-To Analysis

Assume we want to implement a *context-sensitive* and *flow-sensitive* subset-based points-to analysis for an object-oriented language like Java. Let us begin with a simplified version of such an analysis. We start by declaring the input relations along with their attributes and their types:

```
rel CFG(s1: Stm, s2: Stm)
rel New(s: Stm, r: Var, o: Obj)
rel Load(s: Stm, r: Var, b: Var, f: Fld)
rel Store(s: Stm, b: Var, f: Fld, v: Var)
```

Each of these relations is populated by a set of facts derived from the program under analysis. For example, a fact $(s_1, s_2) \in$ `CFG` represents that there is a control-flow graph edge from the statement s_1 to the statement s_2. Similarly, a fact $(s, r, o) \in$ `New` represents that the abstract object o is allocated at statement s and assigned to the result variable r.

Next, we declare the variable and heap points-to relations which are the result of the analysis:

```
rel VarPtsToIn(c: Ctx, s: Stm, v: Var, o: Obj)
rel VarPtsToOut(c: Ctx, s: Stm, v: Var, o: Obj)
rel HeapPtsToIn(c: Ctx, s: Stm, b: Obj, f: Fld, t: Obj)
rel HeapPtsToOut(c: Ctx, s: Stm, b: Obj, f: Fld, t: Obj)
```

We declare two copies of each relation; each copy represents the variable and heap points-to relations *immediately before* and *immediately after* some program point identified by a context and a statement. For example, a fact $(c, s, v, o) \in$ `VarPointsToIn` represents that the local variable v may point-to the abstract object o before the statement s in context c. Similarly, a fact $(c, s, b, f, t) \in$ `HeapPointsToOut` represents that the field f of the base object b may point-to the target object t after the statement s in the context c.

We can express the abstract semantics of a field read as:

```
VarPointsToOut(ctx, stm, resultVar, targetObj) :-
  Load(stm, resultVar, baseVar, field),
  VarPointsToIn(ctx, stm, baseVar, baseObj),
  HeapPointsToIn(ctx, stm, baseObj, field, targetObj).
```

Informally, this rule states that if there is some field read `resultVar = baseVar.field` at a statement where the local variable `baseVar` points to some abstract object `baseObj` *before* the same statement in the same context, and the field `field` of the abstract object `baseObj` points to some abstract object `targetObj` again before the same statement in the same context, *then* we infer that the local variable `resultVar` points to the abstract object `targetObj` *after* the statement `stm` in context `ctx`.

We can express the abstract semantics for the rest of the analysis in a similar fashion. For example, we would need to add rules for allocation of objects, for field stores, and for propagating dataflow between control-flow graph edges. A realistic program analysis may require more than a hundred relations and several hundred rules totalling several thousands of lines of code.

If we look closely at the rule above, we see that many parameters are repeated. Specifically, the rule is always relative to the same context `ctx` and statement `stm` and these occur in every predicate. Repetition is a code smell and calls for refactoring. What we would like is to make the context and statement parameters *implicit* and omit them from each predicate in the rule. This has two benefits: it makes the rule more concise and it makes it clear that the semantics is the same regardless of the current context and statement.

For this purpose, we allow the attributes of a predicate to be declared implicit. For example,

```
rel VarPtsToIn(implicit c: Ctx, implicit s: Stm,
               v: Var, o: Obj)
rel VarPtsToOut(implicit c: Ctx, implicit s: Stm,
                v: Var, o: Obj)
rel HeapPtsToIn(implicit c: Ctx, implicit s: Stm,
                b: Obj, f: Fld, t: Obj)
rel HeapPtsToOut(implicit c: Ctx, implicit s: Stm,
                 b: Obj, f: Fld, t: Obj)
```

declares that the attributes `c` and `s` may be treated as implicit in the variable and heap points-to relations. Any attribute may be declared implicit regardless of its position in the relation or of any other implicit attributes. With the context and statement attributes declared as implicit, the previous rule can be simplified to:

```
VarPointsToOut(resultVar, targetObj) :-
  Load(resultVar, baseVar, field),
  VarPointsToIn(baseVar, baseObj),
  HeapPointsToIn(baseObj, field, targetObj).
```

This rule is equivalent to the previous, but omits the parameters for the context and statement. The omitted parameters are automatically inferred by the compiler, and it ensures that the *same* context and statement is used in each predicate.

Implicit attributes eliminate one form of redundancy, but there is a another form of redundancy that we would like to eliminate. Consider the dataflow propagation rule:

```
VarPointsToIn(c, s2, v, o) :-
  CFG(s1, s2),
  VarPointsToOut(c, s1, v, o).
```

This rule states that if there is a points-to fact $(c, s_1, v, o) \in$ `VarPointsToOut` immediately after some statement s_1 in context c and if there is a control-flow graph edge from s_1 to s_2, then the same dataflow fact is available immediately before the statement s_2 in the same context. Locally, in this specific rule, we would like to treat the parameters c, v and o as implicit. However, we *do not* want to globally declare these attributes as implicit since that would affect every rule.

For this purpose, we introduce *implicified* predicates. We mark a predicate, within a rule, as implicified by prefixing it with an at-sign (@). Intuitively, an implicified predicate treats every attribute as *if it* were declared implicit. With implicified predicates, we can simply write the rule as:

```
@VarPointsToIn(s2) :-
  CFG(s1, s2),
  @VarPointsToOut(s1).
```

In this rule, the predicates `VarPointsToIn` and `VarPointsToOut` are implicified whereas `CFG` is not. The `CFG` predicate binds the parameters `s1` and `s2` to values of type `Stm`. The implicified predicate `@VarPointsToIn(s2)` treats every attribute as implicit and requires that the parameter `s2` matches one of these attributes. Intuitively, the rule can be understood as follows: The `CFG` predicate determines the type of the `s1` and `s2` parameters. This type information determines which attribute `s1` corresponds to in the implicified predicate `@VarPointsToIn`. The remaining parameters are then treated as implicit. Similarly for `@VarPointsToOut`.

The difference between predicates and implicified predicates is that in the former we are only permitted to omit the implicit parameters (if any) whereas in the latter we are permitted to omit *any* parameter, as long as the types are sufficient to disambiguate the omitted parameters. In fact, in an implicified predicate it is *only* the types that are used for resolution; the order of parameters does not matter.

3 IMPLEMENTATION

We now present our design of implicit parameters for typed logic programming languages. We begin with a discussion of some requirements and design choices. We then present a minimal logic calculus Δ_{DAT}, in the spirit of Datalog, and show how to extend it with implicit parameters, and ultimately how to replace the implicit parameters with explicit parameters through a translation scheme.

3.1 Requirements

We consider four requirements that any reasonable design of implicit parameters should satisfy:

- **Type Safety.** The design should preserve type safety. Specifically, the declared types of every predicate should agree with the types of the terms everywhere the predicate is used.
- **Consistency.** The design should not change the semantics of any program that has no implicit parameters. Moreover, a program where every implicit parameter is explicitly given should have the same semantics *as if* the program declared no implicit parameters.
- **Determinism.** The use of implicit parameters should be unambiguous and the translation scheme should always produce exactly one program.
- **Predictability.** The meaning of explicit parameters, i.e. parameters written by the programmer, should not be changed. The compiler should only “fill-in” implicit parameters and leave explicit parameters untouched.

3.2 Design Choices

With the above as a guide, we turn to some more practical concerns and design choices.

- We assume a logic programming language with a static type system where every predicate symbol must be declared along with the types of its attributes.
- We assume that a predicate has a *single* parameter list.
- We assume that *any* parameter(s), at any position in the list, may be declared as implicit.
- We consider two notions of implicit parameters: *implicit attributes* and *implicified predicates*.

It is instructive to compare these design choices to those of Scala. Other than our two notions of implicits, the most striking difference is that we restrict ourselves to a single parameter list and that implicit parameters may occur anywhere in the parameter list. In Scala, a method may have multiple parameter lists, to allow currying, and only the last parameter list may be declared implicit.

Other languages that support implicits do so using a combination of two mechanisms. First, at a variable declaration site, a variable may optionally be designated as a candidate for implicit resolution and assigned a type. Second, at a use site that requires an implicit term of some type, the compiler searches the current scope to find a candidate variable of that type. Logic languages like Datalog do not have variable declarations, so all occurrences of a variable are uses. Thus, we require a language design that does not depend on a scope of candidate variables, and can synthesize the required variables based only on the set of implicit uses that occur in each Horn clause.

3.3 Introductory Example

We now present the technical developments necessary for adding implicit parameters to a logic programming language. We begin with an informal description of the key ideas.

At a high level, our goal is to transform a logic program P with some parameters left implicit to another program P' where every parameter is explicit. As discussed earlier, this transformation must preserve type safety along with several other important properties.

Assume we have a program, similar to the one from Section 2, with the declarations:

```
rel VarPtsToIn(implicit c: Ctx, implicit s: Stm,
               v: Var, o: Obj)
rel VarPtsToOut(implicit c: Ctx, implicit s: Stm,
                v: Var, o: Obj)
rel CFG(s1: Stm, s2: Stm)
```

We introduce the notion of *attribute slots* for a rule. An *attribute slot* is a pair of indices i and j where i is the index of an atom in the rule and j is the index of an attribute of the predicate symbol.[1] Notice that j ranges over both explicit and implicit attributes of the predicate. We will write the pair as s^i_j to remind us that this is an attribute slot.

For example, given the rule:

```
VarPointsToIn(c, s2, v, o) :-
  CFG(s1, s2),
  VarPointsToOut(c, s1, v, o).
```

its attribute slots are:

$$s^0_0, s^0_1, s^0_2, s^0_3, \quad s^1_0, s^1_1, \quad \text{and} \quad s^2_0, s^2_1, s^2_2, s^2_3$$

Intuitively, the attribute slot s^0_0 represents the *first attribute of the first atom*, i.e. the attribute `c: Ctx` of the atom A_0 = `VarPointsToIn`(c, s_2, v, o). Similarly, the attribute slot s^1_0 represents the *first attribute of the second atom*, i.e. the attribute `s1: Stm` of the atom A_1 = `CFG`(s_1, s_2).

It is instructive to view the attribute slots at their proper positions in the rule:

```
VarPointsToIn($s^0_0$, $s^0_1$, $s^0_2$, $s^0_3$) :-
  CFG($s^1_0$, $s^1_1$),
  VarPointsToOut($s^2_0$, $s^2_1$, $s^2_2$, $s^2_3$).
```

We remark that the *types* of the attributes of the predicate symbols which occur in this rule are: `Ctx`, `Stm`, `Var`, and `Obj`.

We can now state our desired goal: We want to compute a partitioning of the attribute slots, such that slots that the rule is intended to force to be equal are in the same partition. In the rule above, the programmer has explicitly given a variable for each slot, with the intended meaning that different uses of the same variable should denote the same value. For example, the variable `s2` is used twice, in two atoms, to indicate that the value of the second attribute of the `VarPointsToIn` predicate must be equal to the value of the second attribute of the `CFG` predicate.

When variables explicitly indicate which attribute slots need to be unified to have equal values, the variables themselves can be used to identify the equivalence classes of equated attribute slots. Thus, we can phrase the problem of partitioning the attribute slots as computing a map (a function) from the slots of a rule to its variables. Each variable designates a partition. The rule above denotes the map:

$$\{s^0_0, s^2_0\} \mapsto c, \quad \{s^1_0, s^2_1\} \mapsto s_1, \quad \{s^0_1, s^1_1\} \mapsto s_2,$$
$$\{s^0_2, s^2_2\} \mapsto v, \quad \text{and} \quad \{s^0_3, s^2_3\} \mapsto o$$

Here $\{a, b\} \mapsto x$ means that both a and b map to x. Observe that we can obtain the original rule by applying the map as a substitution to the rule with the attribute slots.

Now, let us consider a rule with implicit parameters:

```
@VarPointsToIn(s2) :-
  CFG(s1, s2),
  @VarPointsToOut(s1).
```

[1] An atom is a predicate symbol followed by a sequence of terms, e.g. `CFG(s1, s2)` is an atom.

Because this rule refers to the same predicates as before, it gives rise to the same attribute slots. The types of those slots are also the same. The task is also the same: to partition the attribute slots so that slots that should require equal values are in the same partition. However, we now have only the two variables s_1 and s_2, because some of the variables have been left implicit. This suggests that we cannot represent the partitioning of the slots as just a mapping to variables.

Instead, we will extend the map so that it maps each slot to either a variable or to a type. Intuitively, if the programmer has not specified any explicit variable for a given slot, we will map the slot to its type, rather than to a variable. As a result, all attribute slots of a given type that have been left implicit by the programmer will be unified in the same partition, identified by the type. Figure 1 shows this process on the rule above. We now explain how this process works. Conceptually we perform a sequence of steps to compute a set of edges in a bipartite graph of attribute slots, variables, and types. Two attribute slots are in the same partition if and only if they are adjacent to some common variable or type in the graph.

Later, we shall see that only two distinct steps are necessary, but for the purpose of exposition we use more steps in this example.

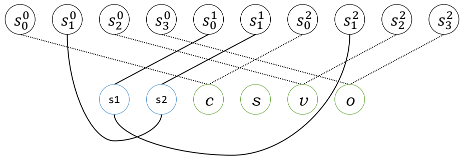

Figure 1: Example of the bipartite graph used to compute the partitioning.

1) We begin with the atom A_1 = CFG(s_1, s_2). We say that this atom is *complete* since the predicate symbol CFG is declared with two attributes and the atom has two arguments, the variables s_1 and s_2. We know that the attribute slots of A_1 are s_0^1 and s_1^1 and that the variables s_1 and s_2 appear at these locations, hence we introduce an edge from s_0^1 to s_1 and an edge from s_1^1 to s_2, as shown in Figure 1.

2) We now consider the atom A_0 = @VarPointsToIn(s_2). We say that this atom is *implicified* since it is prefixed by an at-sign @. We want to determine to which attribute slot we should assign the variable s_2. However, we cannot do so syntactically, because in an implicified atom, not all variables are present, i.e. its arity is less than its declared arity. However, we know that there is an edge from s_1^1 to s_2, which was added in Step (1). Since the type of s_1^1 is Stm, the type of s_2 must also be Stm. The VarPointsToIn predicate declares only one attribute of type Stm, the second attribute. Hence we conclude that we should add an edge from the attribute slot s_1^0, i.e. the attribute of type Stm, to the variable s_2, as shown in Figure 1.

3) We now consider the atom A_2 = @VarPointsToOut(s_1). We use the exact same reasoning as in Step (2). In fact, we could have swapped the order of Step (2) and Step (3), with no difference in result. By the same line of reasoning as in Step (2), we conclude that we should add an edge from the attribute slot s_1^2 to s_1, as shown in Figure 1.

4) At this point, we have processed every atom in the rule. However, we are not yet finished: We have several attribute slots which are not yet connected to any variable or type. We take the remaining unconnected attribute slots and connect them to their type. For example, the attribute slot s_0^0 has the type Ctx, so we connect it to the Ctx type in the graph. We repeat this for every unconnected attribute slot. The end result is shown in Figure 1.

The final graph has the following edges:

$$\{s_0^0, s_0^2\} \mapsto \texttt{Ctx}, \quad \{s_0^1, s_1^2\} \mapsto s_1, \quad \{s_1^0, s_1^1\} \mapsto s_2,$$
$$\{s_2^0, s_2^2\} \mapsto \texttt{Var}, \quad \text{and} \quad \{s_3^0, s_3^2\} \mapsto \texttt{Obj}$$

This graph defines a map from attribute slots to variables and types. If we substitute the types Ctx, Var, and Obj for the variables $x_{\texttt{Ctx}}$, $x_{\texttt{Var}}$, and $x_{\texttt{Obj}}$ and apply the resulting map to the rule with the attribute slots we obtain the rule:

```
VarPointsToIn(x_Ctx, s_2, x_Var, x_Obj) :-
  CFG(s_1, s_2),
  VarPointsToOut(x_Ctx, s_1, x_Var, x_Obj).
```

In this rule, every variable is explicit, and the rule has the same semantics as the original rule.

We can apply this translation scheme to every rule in a logic program P to obtain another logic program P' where every implicit parameter has been made explicit. In the above discussion, we did not cover all the cases of the translation nor did we cover the situations when the map may fail to exist or fail to satisfy some of the requirements or design choices we discussed earlier. With these examples in mind, we now turn to a general and formal treatment of these ideas.

3.4 A Minimal Logic Calculus

We illustrate the translation scheme on a minimal logic programming language similar to Datalog. Figure 2 shows the grammar of this language which we name Δ_{DAT}.

Syntax. A program $\langle D_1, \cdots, D_n, R_1, \cdots, R_n \rangle$ is a pair of *predicate declarations* $D_1, \cdots, D_n$ and *logic rules* $R_1, \cdots, R_n$. A *predicate declaration* $p(a_1, \cdots, a_n)$ associates a predicate symbol p with a sequence of *attributes* $a_1, \cdots, a_n$. An attribute is an identifier *ident* and a type τ possibly marked as implicit with the implicit keyword. If an attribute is not marked implicit, we say that it is explicit. A *rule* $A_0 \Leftarrow A_1, \cdots, A_n$ is a Horn clause where A_0 is the *head atom* and

$$\begin{array}{rcl}
P \in \mathit{Program} & = & \langle D_1, \cdots, D_n,\, R_1, \cdots, R_n \rangle \\
D \in \mathit{Declarations} & = & p(a_1, \cdots, a_n) \\
a \in \mathit{Attributes} & = & \mathit{ident} : \tau \mid \textbf{implicit}\ \ \mathit{ident} : \tau \\
R \in \mathit{Rules} & = & A_0 \Leftarrow A_1, \cdots, A_n \\
A \in \mathit{Atoms} & = & p(t_1, \cdots, t_n) \mid @p(t_1, \cdots, t_n) \\
t \in \mathit{Terms} & = & x \\
 & & \\
\tau \in \mathit{Types} & = & \text{is a set of base types.} \\
\mathit{ident} \in \mathit{Identifiers} & = & \text{is a set of attribute names.} \\
x, y, z \in \mathit{Variables} & = & \text{is a set of variable symbols.} \\
p, q \in \mathit{Predicates} & = & \text{is a set of predicate symbols.} \\
s^i_j \in \mathit{Slots} & = & \text{is a set of attribute slots.}
\end{array}$$

Figure 2: Syntax of Δ_{DAT}

$A_1, \cdots, A_n$ are *body* atoms. An atom $p(t_1, \cdots, t_n)$ is a predicate symbol p followed by a sequence of terms $t_1, \cdots, t_n$. An atom is *implicified* if an at-sign @ is placed in front of it. Finally, a term is simply a variable.

We impose three additional restrictions which are not naturally captured by the grammar:

- Every predicate symbol which appears in a logic rule must be declared.
- No two predicate declarations may share the same predicate symbol.
- The arity of an atom $p(t_1, \cdots, t_k)$ must be less than or equal to the declared arity of p.

With these restrictions in place, we define a few functions:

The decl function returns the declaration $p(a_1, \cdots, a_n)$ of a predicate symbol p:

$$\mathsf{decl} : \mathit{Predicates} \longrightarrow \mathit{Declarations}$$

The functions explicitArity : $\mathit{Predicates} \longrightarrow \mathcal{N}$, implicitArity : $\mathit{Predicates} \longrightarrow \mathcal{N}$, and totalArity : $\mathit{Predicates} \longrightarrow \mathcal{N}$ return respectively the number of explicit, implicit, and total attributes of a predicate symbol.

Atoms. We distinguish three types of atoms:

- An atom $p(t_1, \cdots, t_k)$ is *complete* if its arity k is equal to the arity of its predicate p, i.e. $k = \mathsf{totalArity}(p)$.
- An atom $p(t_1, \cdots, t_k)$ is *partial* if its arity k is equal to the explicit arity of its predicate p, i.e. $k = \mathsf{explicitArity}(p)$.
- An atom $@p(t_1, \cdots, t_k)$ is *implicified* if it is is prefixed by an at-sign @. Its arity must be less than or equal to the total number of attributes of its predicate, i.e. $k \leq \mathsf{totalArity}(p)$.

Every atom, in every rule, must fall into one of the above categories, otherwise the program is illegal and must be rejected by the compiler.

Attribute Slots. Given a rule $R = A_0 \Leftarrow A_1, \cdots, A_n$, we introduce a set of *attribute slots* slots(R). An attribute slot is a pair of a *atom index i* and an *attribute index j*. We write an attribute slot as s^i_j where i is the atom index and j is the attribute index. The attribute slot s^i_j is in slots(R) if and only if the ith atom in the rule has some predicate p and p has at least j attributes.

For example, given a program with the declarations:

$$p_1(a : \tau_1,\ b : \tau_1, \textbf{implicit}\, c : \tau_2),$$
$$p_2(a : \tau_1,\ b : \tau_1, \textbf{implicit}\, c : \tau_2), \quad p_3(c : \tau_2)$$

the rule:

$$p_2(x, z) \Leftarrow p_2(x, y),\ p_1(y, z),\ p_3().$$

gives rise to the attribute slots:

$$s^0_0, s^0_1, s^0_2, \quad s^1_0, s^1_1, s^1_2, \quad s^2_0, s^2_1, s^2_2, \quad \text{and} \quad s^3_0$$

which are easier to understand when placed at their appropriate positions in the rule:

$$p_2(s^0_0, s^0_1, s^0_2) \Leftarrow p_2(s^1_0, s^1_1, s^1_2),\ p_1(s^2_0, s^2_1, s^2_2),\ p_3(s^3_0).$$

The type of each attribute slot is uniquely determined by its position in an atom. For example, the type of s^0_0 is τ_1 and the type of s^3_0 is τ_2. The type utility function returns the type of an attribute slot:

$$\mathsf{type} : \mathit{Slot} \longrightarrow \mathit{Type}$$

We need one other helper function before we can proceed: The offset function takes a predicate symbol and the index of a variable term in a partial atom and returns the index of its attribute.

$$\mathsf{offset} : \mathit{Predicates} \times \mathcal{N} \longrightarrow \mathcal{N}$$

For example, given a program with the declaration:

$$p_1(a : \tau_1, \textbf{implicit}\, b : \tau_1, c : \tau_2)$$

and the atom:

$$p_1(x, z)$$

the offset of x, the variable at index 0, is $\mathsf{offset}(p_1, 0) = 0$ whereas the offset of z, the variable at index 1, is $\mathsf{offset}(p_1, 1) = 2$. Intuitively, x corresponds to the first attribute of p_1, the second attribute of p_1 is left implicit, and z corresponds to the last attribute of p_1.

3.5 Translation Scheme

Given a rule R, we consider three sets:

- The set of variables $\mathcal{V}$ in the rule, i.e. the variables written by the programmer.
- The set of types $\mathcal{T}$ in the rule, i.e. the types of the attributes of the predicates in the rule.
- The set of attribute slots $\mathcal{S}$ of the rule, i.e. the set introduced by slots(R).

We want to compute a total map ζ from attribute slots $\mathcal{S}$ to variables $\mathcal{V}$ and types $\mathcal{T}$:

$$\zeta : \mathcal{S} \rightarrow \mathcal{V} \cup \mathcal{T}$$

The function ζ assigns every attribute slot to a variable or type. Intuitively, two slots mapped to the same variable (or type) become the same parameter in the translated rule.

We construct the ζ map as a set of edges E in a bipartite graph between the attribute slots $\mathcal{S}$ on one side and the variables $\mathcal{V}$ and types $\mathcal{T}$ on the other side. We infer edges in this bipartite graph based on the rules shown in Figure 3 and discussed below:

- [E-Complete] If there is a *complete* atom $A_i = p(\cdots, x_j, \cdots)$, then we introduce an edge from the attribute slot s^i_j to x_j. Intuitively, in a complete atom, every variable corresponds to exactly one attribute slot of the atom.
- [E-Partial] If there is a *partial* atom $A_i = p(\cdots, x_k, \cdots)$, then we introduce an edge from the attribute slot s^i_j to x_k where $j = \text{offset}(p, k)$. Intuitively, in a partial atom, a variable corresponds to an attribute slot with some offset from the position of the variable. Informally, we must "skip" the attribute slots that correspond to implicit attributes which are omitted from a partial atom.
- [E-Implicified] If there is an *implicified* atom $@A_i = p(\cdots, x_k, \cdots)$ with a variable x_k, then we introduce an edge from x_k to the attribute slot s^i_j if there is *some other existing* edge $s^{i'}_{j'}$ to x_k and $s^{i'}_{j'}$ has the same type as s^i_j. Intuitively, the variable x_k appears in some complete or partial atom and some edge is introduced according to either [E-Complete] or [E-Partial]. This gives us the type of x_k, and we use this type to determine which of the attribute slots of the implicified atom A_i the variable x_k should be linked to.
- [E-Implicit] For each attribute slot s^i_j which is not paired to any variable x by the former rules, we introduce an edge from s^i_j to its type τ. Intuitively, this rule links attribute slots to their types if they have not already been linked to a variable by one the previous rules.

$$\frac{\begin{array}{c} R = A_0 \Leftarrow A_1, \cdots, A_n \\ A_i = p(\cdots, x_j, \cdots) \qquad A_i \text{ is complete} \\ s^i_j \in \mathcal{S} \qquad x_j \in \mathcal{V} \end{array}}{(s^i_j, x_j) \in E_v}\ [\text{E-}\textsc{Complete}]$$

$$\frac{\begin{array}{c} R = A_0 \Leftarrow A_1, \cdots, A_n \\ A_i = p(\cdots, x_k, \cdots) \qquad A_i \text{ is partial} \\ s^i_j \in \mathcal{S} \qquad x_k \in \mathcal{V} \qquad j = \text{offset}(p, k) \end{array}}{(s^i_j, x_k) \in E_v}\ [\text{E-}\textsc{Partial}]$$

$$\frac{\begin{array}{c} R = A_0 \Leftarrow A_1, \cdots, A_n \\ A_i = @p(\cdots, x_k, \cdots) \qquad A_i \text{ is implicified} \\ (s^{i'}_{j'}, x_k) \in E_v \qquad \text{type}(s^i_j) = \text{type}(s^{i'}_{j'}) \\ s^i_j \in \mathcal{S} \qquad s^{i'}_{j'} \in \mathcal{S} \qquad x_k \in \mathcal{V} \end{array}}{(s^i_j, x_k) \in E_v}\ [\text{E-}\textsc{Implicified}]$$

$$\frac{(s^i_j, x) \in E_v}{s^i_j \in S_v}\ [\text{E-}\textsc{VarSlots}] \qquad \frac{(s^i_j, x) \in E_v}{(s^i_j, x) \in E}\ [\text{E-}\textsc{Vars}]$$

$$\frac{\begin{array}{c} s^i_j \notin S_v \\ \text{type}(s^i_j) = \tau \qquad s^i_j \in \mathcal{S} \qquad x \in \mathcal{V} \qquad \tau \in \mathcal{T} \end{array}}{(s^i_j, \tau) \in E}\ [\text{E-}\textsc{Implicit}]$$

Figure 3: Inference Rules for Partitioning

The rules are stratified. The rules [E-Complete], [E-Partial], [E-Implicified], and [E-VarSlots] do not contain any occurrences of negation, so they can be fully applied in the first stratum until the graph is saturated. The [E-Implicit] rule needs to be computed in a second stratum because it involves negation of S_v, which is in the goal of [E-VarSlots]. The goal of [E-Implicit] is E, which does not occur in the body of any of the rules in the first stratum, and thus this stratification is valid. It is valid to apply the [E-Vars] rule in either the first or the second stratum because it does not involve negation and E does not occur in its premises (body). We choose to apply it in the second stratum so that both of the rules with goal E, [E-Implicit] and [E-Vars], are in the same stratum. The final set of edges E define the ζ map. We use the ζ map to translate a rule with implicit parameters into one where every parameter has been made explicit. Concretely, we define two functions $\textit{trRule} : \textit{Rule} \times Z \longrightarrow \textit{Rule}$ and $\textit{trAtom} : \textit{Atom} \times Z \longrightarrow \textit{Atom}$:

$$\begin{array}{l} \text{trRule}(A_0 \Leftarrow A_1, \cdots, A_n, \zeta) = \\ \quad \text{trAtom}(A_0, \zeta) \Leftarrow \text{trAtom}(A_1, \zeta), \cdots, \text{trAtom}(A_n, \zeta) \end{array}$$

and

$$\begin{array}{l} \text{trAtom}(A_i(t_1, \cdots t_n), \zeta) = p(\zeta(s^i_0), \cdots, \zeta(s^i_m)) \\ \quad \text{where } m = \text{totalArity}(p) \end{array}$$

The trRule function applies trAtom to every atom in the rule. The trAtom computes the attribute slots of the predicate p with atom index i and then applies the zeta function to each attribute slot. Here ζ yields either a variable or type; if it yields a type we consider that type as a variable name.

Example I. Given a program with the declarations:

$$p_1(a : \tau_1,\ b : \tau_1), \quad p_2(c : \tau_1,\ \textbf{implicit}\ d : \tau_2)$$

then the rule:

$$p_1(x, y) \Leftarrow p_2(x, w),\ p_2(y, w).$$

has three complete atoms $p_1(x, y)$, $p_2(x, w)$, and $p_2(y, w)$. This rule has no implicit parameters since every atom is complete (and hence no parameters are absent). The attribute slots introduced by this rule and placed at their proper locations are:

$$p_1(s^0_0, s^0_1) \Leftarrow p_2(s^1_0, s^1_1),\ p_2(s^2_0, s^2_1).$$

In this rule, the edges between the attribute slots and the variables are determined fully syntactically. By the rule [E-Complete], we infer the edges:

$$\{s^0_0, s^1_0\} \mapsto x, \quad \{s^0_1, s^2_0\} \mapsto y, \quad \text{and} \quad \{s^1_1, s^2_1\} \mapsto w$$

Applying this substitution to the attribute slots yields the original rule, as expected.

Example II. Given a program with the declarations:

$$p_1(\textbf{implicit}\, a : \tau_1,\ b : \tau_2),\ p_2(\textbf{implicit}\, a : \tau_1,\ b : \tau_2),\ p_3(b : \tau_2)$$

then the rule:

$$p_1(x) \Leftarrow p_1(x),\ p_2(x),\ p_3(x).$$

has three partial atoms $p_1(x)$, $p_1(x)$, $p_2(x)$ and one complete atom $p_3(x)$. Intuitively, the attributes a of the predicates p_1 and p_2 are intended to be passed implicitly. The attribute slots of this rule at their proper locations are:

$$p_1(s^0_0, s^0_1) \Leftarrow p_1(s^1_0, s^1_1),\ p_2(s^2_0, s^2_1),\ p_3(s^3_0).$$

By the rules [E-Partial] and [E-Implicit], we infer:

$$\{s^0_1, s^1_1, s^2_1, s^3_0\} \mapsto x \quad \text{and} \quad \{s^0_0, s^1_0, s^2_0\} \mapsto \tau_1$$

If we give the type τ_1 the variable name y_{τ_1}, then the rule is translated to:

$$p_1(y_{\tau_1}, x) \Leftarrow p_1(y_{\tau_1}, x),\ p_2(y_{\tau_1}, x),\ p_3(x).$$

where every parameter is explicit.

Example III. Given a program with the declarations:

$$p_1(a : \tau_1,\ b : \tau_2,\ c : \tau_3, d : \tau_4),$$
$$p_2(a : \tau_1,\ b : \tau_2,\ c : \tau_3, d : \tau_4), \quad p_3(b_1 : \tau_2,\ b_2 : \tau_2)$$

then the rule:

$$@p_1(y) \Leftarrow @p_2(x),\ p_3(x, y).$$

has two implicified atoms $@p_1(y)$ and $@p_2(x)$ and one complete atom $p_3(x, y)$. Intuitively, the implicified atoms are used to implicitly pass the attributes a, c, and d between the predicates p_1 and p_2. The attribute slots introduced by this rule and placed at their proper locations are:

$$@p_1(s^0_0, s^0_1, s^0_2, s^0_3) \Leftarrow @p_2(s^1_0, s^1_1, s^1_2, s^1_3),\ p_3(s^2_0, s^2_1).$$

By the rules [E-Complete] and [E-Implicit] we infer:

$$\{s^1_1, s^2_0\} \mapsto x, \quad \{s^0_1, s^2_1\} \mapsto y, \quad \{s^0_0, s^1_0\} \mapsto \tau_1,$$
$$\{s^0_2, s^1_2\} \mapsto \tau_3, \quad \text{and} \quad \{s^0_3, s^1_3\} \mapsto \tau_4$$

If we give the types τ_1, τ_3, and τ_4 the variable names u_{τ_1}, v_{τ_3}, and w_{τ_4}, then the rule is translated to:

$$p_1(u_{\tau_1}, y, v_{\tau_3}, w_{\tau_4}) \Leftarrow p_2(u_{\tau_1}, x, v_{\tau_3}, w_{\tau_4}),\ p_3(x, y).$$

where every parameter is explicit.

In the next section, we present well-formedness requirements for programs with implicit parameters. But before that, we introduce a function to return the type of a term given its index in a partial or complete atom:

$$\mathsf{termType} : \mathit{Predicate} \times \mathit{Bool} \times \mathit{Int} \longrightarrow \mathit{Type}$$

Intuitively, given a partial or complete atom $p(t_1, \cdots, t_n)$ the termType function returns the type of the term t_i according to the declared attributes of p. If the atom is complete, the type of t_i is simply the type of the declared attribute a_i. If, however, the atom is partial then the type of t_i is a_j where $j = \mathsf{offset}(p, i)$. The boolean argument tells us whether the atom is partial or complete and the integer argument tells us the index of the term in the atom. We will omit the boolean argument whenever it is clear from the context if the atom is partial or complete.

We now discuss the well-formedness requirements for implicit parameters. Our ultimate goal is to prove that the design satisfies the properties stated in Section 3.2.

$$\frac{\begin{array}{c} A_1 = p_1(x_1, \cdots, x_n) \qquad A_2 = p_2(y_1, \cdots, y_m) \\ A_1 \text{ and } A_2 \text{ are complete or partial} \end{array}}{x_i = y_j \Rightarrow \mathsf{termType}(p_1, i) = \mathsf{termType}(p_2, j)}\ [\text{WF-Types-1}]$$

$$\frac{R = A_0 \Leftarrow A_1, \cdots, A_n \qquad A_i = @p_i(\cdots, x, \cdots)}{\begin{array}{c} \exists j.\ A_j = p_j(\cdots, x, \cdots) \\ A_j \text{ is partial or complete} \end{array}}\ [\text{WF-Types-2}]$$

Figure 4: Well-Formedness Requirements: Type Safety

3.6 Well-Formedness: Type Safety

We use the well-formedness requirements, [WF-Types-1] and [WF-Types-2], shown in Figure 4, to ensure that the programs are well-typed. The [WF-Types-1] requirement states that if the same variable appears in two or more atoms, then it must have the same type in each of those atoms. The [WF-Types-2] requirement states that every variable that appears in an implicified atom must also appear in a non-implicified atom.

For example, consider a program with the declarations:

$$p_1(\textbf{implicit}\, a : \tau_1,\ b : \tau_2), \quad p_2(\textbf{implicit}\, a : \tau_1,\ c : \tau_3)$$

with the rule:

$$p_1(x, y) \Leftarrow p_2(x, y).$$

The variable y does not have the same type in p_1 as in p_2. Hence this program is not well-formed. The [WF-Types-1]

$$\frac{\begin{array}{c}@p(\cdots, x, \cdots) \\ \mathsf{decl}(p) = \mathit{ident} : \tau_1, \cdots, \mathit{ident} : \tau_n\end{array}}{\mathsf{varType}(x) \in \{\tau_1, \cdots, \tau_n\}} \;\text{[WF-TYPE-COMPATIBLE]}$$

$$\frac{\begin{array}{c}A = @p(\cdots, x, \cdots, y, \cdots) \\ \mathsf{varType}(x) = \mathsf{varType}(y)\end{array}}{x = y} \;\text{[WF-AMBIGUITY]}$$

Figure 5: Well-Formedness Requirements: Type Compatibilty and Ambiguity

well-formedness requirement ensures that such programs are rejected.

As another example, consider a program with the declarations:

$$p_1(a : \tau_1, b : \tau_2), \quad p_2(a : \tau_1, b : \tau_2), \quad p_3(a : \tau_1, b : \tau_2)$$

and with the rule:

$$@p_1(x) \Leftarrow @p_2(x),\ @p_3(x).$$

Since x does not occur in any complete or partial atom, it is impossible to determine if its type should be τ_1 or τ_2. The [WF-TYPES-2] well-formedness requirement ensures such programs are rejected.

If either of [WF-TYPES-1] or [WF-TYPES-2] is not satisfied, the program is rejected. Otherwise, we can define a function from variables to types:

$$\mathsf{varType} : \mathit{Variable} \longrightarrow \mathit{Type}$$

that uniquely assigns a type to every variable based on its occurence in a complete or partial atom.

3.7 Additional Requirements

We introduce two additional well-formedness requirements to ensure the existence of a translation from a program with implicit parameters to one without. The [WF-TYPE-COMPATIBLE] requirement states that if the variable x is one of the terms of an implicified atom with predicate symbol p, then the type of x must match one of the declared attribute types of p. The [WF-AMBIGUITY] requirement states that if two explicit variables x and y occur in the same implicified atom and they have the same type then x and y must be the same variable.

For example, consider a program with the declarations:

$$p_1(a : \tau_1,\ b : \tau_2), \quad p_2(c : \tau_3,\ d : \tau_4)$$

and with the rule:

$$p_1(x, \_) \Leftarrow @p_2(x).$$

Here, the variable x appears in a complete and in an implicified atom. The complete atom assigns x the type τ_1. However, the attribute types of p_2 are τ_3 and τ_4, so it is impossible for x to have a consistent type in both atoms. The [WF-TYPE-COMPATIBLE] well-formedness requirement ensures that such programs are rejected.

As another example, consider the declarations:

$$p_1(a : \tau, b : \tau) \quad p_2(a : \tau, b : \tau)$$

with the rule:

$$p_1(x, y) \Leftarrow @p_2(x, y).$$

Here, the problem is that x and y both have the type τ, but then it is ambiguous whether x corresponds to the first or second attribute slot in the implicified atom with predicate symbol p_2. The [WF-AMBIGUITY] well-formedness requirement ensures that such programs are rejected.

3.8 Theoretical Properties

We can now present the main theoretical properties of our design. For brevity, we include only the most important lemmas here. All lemmas and their proofs are available in the appendix.

Lemma A.3. If a rule R is well-formed, the edge set E defines a total function ζ.

Lemma A.5. The ζ function is type safe; in other words, if the ζ function maps two slots s_1 and s_2 to the same variable or type, i.e. $\zeta(s_1) = \zeta(s_2)$, then the type of the slots s_1 and s_2 is the same.

Lemma A.6. A well-formed rule R without any implicit parameters is translated to itself.

These results ensure that our design satisfies the requirements stated in Section 3.1, i.e. type-safety, consistency, determinism, and predictability.

4 PRACTICAL APPLICATIONS

In this section, we describe some envisioned practical applications of implicit parameters for logic programs.

4.1 Dataflow and Points-To Analysis

The original motivation for adding implicit parameters to a logic programming language came from our experience with implementing points-to and dataflow analyses in FLIX. In this line of work, which can involve thousands of lines of code, we discovered that many rules require contextual information that rarely changes within the rule. As shown in Section 2, a context- and flow-sensitive analysis tracks information for every program point typically identified by a context and statement. However, most rules operate on the *same* context and statement. Only a few rules actually change the context and statement, e.g. upon method entry or exit, or to propagate dataflow information to the successor nodes in the control-flow graph.

4.2 Global Variables

We can use implicit parameters to thread global variables through a program. The Mercury compiler is an interesting case study. Mercury is a functional and logic programming language [15, 16].

The compiler uses a global object named `Globals` which is manually threaded through many of its rules. For example, here is a code fragment from `mercury_compile_middle_passes.m`:

```
middle_pass(!HLDS, !DumpInfo, !IO) :-
  module_info_get_globals(!.HLDS, Globals),
  globals.lookup_bool_option(Globals, verbose, Verbose),
  globals.lookup_bool_option(Globals, statistics, Stats),
  maybe_read_experimental_complexity_file(!HLDS, !IO),
  tabling(Verbose, Stats, !HLDS, !IO),
  maybe_dump_hlds(!.HLDS, 105, "tabling", !DumpInfo, !IO),
  expand_lambdas(Verbose, Stats, !HLDS, !IO),
  maybe_dump_hlds(!.HLDS, 110, "lambda", !DumpInfo, !IO),
  expand_stm_goals(Verbose, Stats, !HLDS, !IO),
  maybe_dump_hlds(!.HLDS, 113, "stm", !DumpInfo, !IO),
  expand_equiv_types_hlds(Verbose, Stats, !HLDS, !IO),
  maybe_dump_hlds(!.HLDS, 115, "equiv_types", ...),
  // ...
```

In this file, which is `1424` lines of source code, the `Globals` parameter appears a total of `116` times, on average an occurence on every 12th line. The `Verbose` object, which is used to track the level of verbosity, occurs `242` times, an average occurence on every 6th line. With implicit parameters this tedius repetition can be avoided.

As another example, in the `mercury_compiler_main.m` file, which is `1879` lines of code, the `Globals` parameter appears `247` times, on average an occurence on every 7th line.

As a third and final example, the compiler often needs to perform I/O which is handled with access to a special parameter named `!IO`. In the `bytecode.m` file, which is `1028` lines of source code, the `!IO` parameter appears a total of `548` times, on average an occurence on every 2nd line!

As these three examples demonstrate, implicit parameters can significantly help reduce the tediousness and repetition of passing global parameters through a logic program.

4.3 Monotonic Timestamps

The inspiration for implicit parameters partly came from temporal logic programming languages where every predicate is equipped with a timestamp and some systems allow this timestamp to be omitted.

Dedalus is one such system which extends Datalog with timestamps [1]. In Dedalus, every fact is associated with a timestamp $\mathcal{T}$ that represents the time at which the fact is true. Dedalus enables a form of stateful programming where facts can be added or retracted over time. Monotonicity requirements on the timestamps ensure that the semantics of Dedalus are well-defined.

In Dedalus, a fact $p(t_1, \cdots, t_n, \mathcal{T})$ is interpreted as a fact $p(t_1, \cdots, t_n)$ which is true at time $\mathcal{T}$.

We classify the behaviour of Dedalus rules into three types: *deductive*, *inductive*, and *persistent*.

A *deductive* rule, such as

$$p_1(x, z, \mathcal{T}) \Leftarrow p_2(x, y, \mathcal{T}),\ p_3(y, z, \mathcal{T}).$$

is instantaneous in the sense that the head predicate holds in the same time instant $\mathcal{T}$ as the body predicates.

An *inductive* rule, such as

$$p_1(x, z, \mathcal{S}) \Leftarrow p_2(x, y, \mathcal{T}),\ p_3(y, z, \mathcal{T}),\ \text{successor}(\mathcal{T}, \mathcal{S}).$$

is temporal in the sense that the head predicate holds in the next time instant $\mathcal{T}$ of the body predicates.

We can use implicit parameters to significantly simplify a Dedalus program. In a deductive rule, the timestamp does not change and so can be left entirely implicit. This is already supported by our design. In an inductive rule, the timestamp is the same in the body predicates and advances to the next time instant in the head predicate. To support this, the current time should be declared implicit in the successor predicate:

$$\textbf{rel}\ \text{successor}(\textbf{implicit}\ t : \texttt{Time}, s : \texttt{Time})$$

The current time can then be omitted from inductive rules:

$$p_1(x, z, t) \Leftarrow p_2(x, y),\ p_3(y, z), \text{successor}(t).$$

The current time is now implicit, and the future time t is explicit.

Finally, a *persistence* rule, such as

$$p(x, y, z, \mathcal{S}) \Leftarrow p(x, y, z, \mathcal{T}),\ \text{successor}(\mathcal{T}, \mathcal{S}).$$

propagates facts from one time instant $\mathcal{T}$ to the next $\mathcal{S}$. In other words, the rule ensures that a fact continues to exist as the time advances.

A persistence rule is reminiscent of the dataflow propagation rule we showed in Section 2. We can simplify persistence rules with implicified predicates.

$$@p(t) \Leftarrow @p(), \text{successor}(t).$$

This rule copies facts from one time instant to the next. Note that this rule is amenable to refactoring: If an attribute is added to or removed from p, the rule does not have to be changed. In some sense, this rule is the most concise encoding of the property that every fact in p is preserved over time.

4.4 Information Flow

We can use an interesting combination of implicit parameters and the lattice semantics of Flix to automatically track the secrecy of information computed by a logic program reminiscent of policy-agnostic programming [17].

Imagine that we have a logic program with public and secret information about a company. The program has public information such as the names of its employees, the addresses of its buildings and so forth, and secret information such as its payroll. An ordinary logic program would be unrestricted in mixing and matching public and secret information. In particular, we would not know whether a derived fact should be considered public or private information.

We can use implicit parameters to automatically and transparently determine the secrecy of every derived fact in the program. For this purpose, we introduce a lattice `Secrecy` with the two elements `Secret` and `Public` ordered by `Secret` $\sqsubset$ `Public`. The least upper bound and greatest lower bound are defined in the obvious way. We can then define lattices that hold the public and secret information of the company:

```
lat Employee(name: Str, bldg: Str, implicit s: Secrecy)
lat Building(bldg: Str, addr: Str, implicit s: Secrecy)
lat Salary(name: Str, amount: Int, implicit s: Secrecy)
```

Notice that every extensional fact is associated with an element of the secrecy lattice. Let us assume we have some employees and buildings:

```
Employee("Peter␣Gibbons", "Initech", Public).
Employee("Stephen␣Root", "Initech", Public).
Building("Initech", "4120␣Freidrich␣Lane", Public).
Salary("Peter␣Gibbons", 42, Secret).
Salary("Stephen␣Root", 43, Secret).
```

We can compute the address of an employee, leaving the `Secrecy` parameter implicit, with the rule:

```
AddressOf(name, address) :-
  Employee(name, building),
  Building(building, address).
```

This rule is equivalent to:

```
AddressOf(name, address, s) :-
  Employee(name, building, s),
  Building(building, address, s).
```

The implicit parameter s occurs in the lattice position of the predicates. In Flix semantics, this means that the value of s in the head predicate `AddressOf` is the *greatest lower bound* of the value(s) of s in the predicates `Employee` and `Building`. In other words, we derive the facts:

```
AddressOf("Peter␣Gibbons", "4120␣Freidrich␣Lane", Public).
AddressOf("Stephen␣Root", "4120␣Freidrich␣Lane", Public).
```

since the secrecy of the employee fact(s) and the building fact(s) are both `Public` and the greatest lower bound of these is `Public`. On the other hand, if we compute the salaries of employees in a specific building,

```
SalaryOf(name, amount) :-
  Employee(name, "4120␣Freidrich␣Lane"),
  Salary(name, amount).
```

we get the result:

```
SalaryOf("Peter␣Gibbons", 42, Secret).
SalaryOf("Stephen␣Root", 43, Secret).
```

since the `Employee` information is `Public`, but the `Salary` information is `Secret`, and the result is the greatest lower bound of the two. Note that the computation is *per employee*. Hence, if the salary of `Stephen` was public then the derived salary would be public for `Stephen`, but not for `Peter`.

If two rules derive the same fact, but with different levels of secrecy, then the Flix semantics computes the *least upper bound* of the two. In other words, if a fact can be inferred from public information exclusively or with some secret information, then the fact is still considered public. This makes intuitive sense, since if there is a way to derive a fact from public information there is no reason it should be classified as secret.

This scheme is not limited to public and secret information. We can use any arbitrary privilege lattice. Similarly, we can use the same scheme to compute other meta data about derived facts. For example, we could express a notion of “confidence” or “correctness” as a lattice and automatically compute it for any derived facts.

4.5 Discussion

We have presented several examples of how implicit parameters are useful for logic programming. The extension of object-oriented and functional programming languages with implicit parameters have borne rich fruit, and we hope to spur similar developments for logic programming languages.

Implicit parameters are undoubtedly a powerful feature that should be used with care. A common complaint from Scala programmers is that implicit parameters can lead to code that is fragile and hard to understand. We are sympathetic to these arguments, but we believe that appropriate use can significantly simplify code and reduce boiler-plate. Ultimately, striking the right balance is the responsibility of the programmer.

5 RELATED WORK

Implicit Parameters. Implicit parameters were originally proposed by Lewis et al. [8] as an extension of the Hugs Haskell interpreter. In their work, the authors studied the problem of how to elegantly pass a parameter through several levels of recursive calls. For example, a command line argument or an environment variable, which are both accessible from main, must be passed to somewhere deep down in the call stack. The authors considered two existing solutions, neither satisfactory: (a) modifying every function to carry the extra parameter, or (b) storing the value in a global variable. As a better alternative, the authors proposed implicit parameters: a dynamically scoped value that the compiler automatically threads through the program.

After the work of Lewis et al., but before its publication, Jones [7] discusses the relationship between implicit parameters and type classes in a technical report.

Devriese and Piessens [5] present a design of *instance arguments* for the Agda programming language. Instance arguments are intended to support the implementation of type classes in Agda. Like in Scala, but unlike in Haskell, the idea is that instance arguments provide a useful mechanism to support overloadable type classes, without requiring the language itself to be extended with type classes. Where Scala relies on traits and objects to model type classes, Agda uses its own dependently-typed records.

Oliveira et al. [12] present the *implicit calculus* $\lambda_{\Rightarrow}$ for generic programming with implicit parameters. In the view of the authors, generic programming techniques typically consist of two high-level components: (a) a special type of interface and (b) a mechanism to implicitly instantiate that interface. For example, the Haskell type class `Ord` is an interface and the mechanism to instantiate the interface is the use of a qualified type like `Ord ⇒ a ...` which instructs the compiler to provide an instance of `Ord` to the generic function. An important feature of the implicit calculus is to get rid of the first component, i.e. the need for a special type of interface. Instead, the implicit instantiation is generalized to work for any type. In other words, an implicit parameter can be of *any* type, not just those types that belong to some type class, and the compiler will use a resolution mechanism to find the appropriate instance.

Practical Applications. Implicit parameters have played a major role as a means to support or implement many other programming language features, including type classes [11], capability and effect systems [6, 13], software transactional memory [3], language virtualization [10], and macros [4].

Oliveira et al. [11] propose implicit parameters as a mechanism to implement type classes in Scala. In their work, a type class is expressed as a trait, with a type parameter, which declares the operations supported by the type class. A type class instance is an object that implements the trait. A polymorphic function uses a type class by declaring an implicit parameter which is the type class name parameterized by the type variable. To call the function, the type class is either explicitly passed by the programmer or resolved through the implicit scope. For example, to sort the elements of a polymorphic list with elements of type `A`, a `sort` function would require that its elements are members of the `Ord[A]` type class if the elements are of type `A`.

Osvald et al. [13] use implicit parameters as a mechanism to propagate *capabilities*. A capability is a value that represents a permission to perform some action. For example, reading a file may require a *capability token* of type `CanRead`, writing a file may require another token of type `CanWrite`, and throwing an exception may require yet another token of type `CanThrow`. Unless a token is passed to a function, we can rest assured that it cannot perform the associated action. Implicit parameters are a useful mechanism to propagate capabilities since they automatically allow the permissions of a caller to propagate to all of its callees.

Haller and Loiko [6] use implicit parameters to control access to protected objects in an actor-based system.

Logic languages occasionally deal with resources such as files and streams. A capability system with implicit parameters could be useful to control and propagate access to these resources. Moreover, as discussed in Section 4, implicit parameters can help control access to sensitive information.

Burmako [4] proposes a macro system for Scala. An interesting feature of his system is its ability to automatically generate type class instances based on macro invocations. For example, if a method requires a specific type class instance, specified as an implicit parameter, the implicit resolution procedure finds an implicit macro of the appropriate type and invokes it to automatically generate the required instance.

Implicit parameters were originally introduced in Haskell to solve a specific problem, but turned out to be useful with many diverse applications. After adding implicit parameters to Datalog, it will be interesting to see what applications carry over and whether any interesting new features arise.

6 CONCLUSION

We have proposed implicit parameters for logic programming languages. Implicit parameters allow the programmer to omit certain arguments from predicates in logic rules and have them automatically inferred by the compiler based on their types. We have presented several practical applications and shown how implicit parameters simplify logic programming. Our proposed design introduces two notions of implicit parameters: *implicit attributes* and *implicified predicates* to support different use cases that arise in logic programming. We have presented a translation scheme that given a logic program with implicit parameters, computes another program where every parameter is explicit. Finally, we have proven that our design satisfies a range of desirable properties, including type-safety, consistency, and determinism.

ACKNOWLEDGMENT

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in Time and Space. In *Datalog Reloaded*. https://doi.org/10.1007/978-3-642-24206-9_16

[2] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-To Analyses. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/1640089.1640108

[3] Nathan G Bronson, Hassan Chafi, and Kunle Olukotun. 2010. CCSTM: A Library-based STM for Scala. In *Proc. Workshop on Scala (SCALA)*.

[4] Eugene Burmako. 2013. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types work with Metaprogramming. In *Proc. Workshop on Scala (SCALA)*. https://doi.org/10.1145/2489837.2489840

[5] Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. In *Proc. International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2034574.2034796

[6] Philipp Haller and Alex Loiko. 2016. LaCasa: Lightweight Affinity and Object Capabilities in Scala. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/3022671.2984042

[7] Mark Jones. 1999. *Exploring the Design Space for Typebased Implicit Parameterization*. Technical Report. Oregon Graduate Institute.

[8] Jeffrey R Lewis, John Launchbury, Erik Meijer, and Mark B Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *Proc. Principles of Programming Languages (POPL)*. https://doi.org/10.1145/325694.325708

[9] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proc. Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2980983.2908096

[10] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized. In *Partial Evaluation and Program Manipulation (PEPM)*. https://doi.org/10.1145/2103746.2103769

[11] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes as Objects and Implicits. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/1932682.1869489

[12] Bruno CdS Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. 2012. The Implicit Calculus: A New Foundation for Generic Programming. In *Proc. Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2345156.2254070

[13] Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I González Alayón, and Tiark Rompf. 2016. Gentrification Gone Too Far? Affordable 2nd-class Values for Fun and (co-) Effect. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/3022671.2984009

[14] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proc. Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1925844.1926390

[15] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *The Journal of Logic Programming* (1996).

[16] Zoltan Somogyi, Fergus J Henderson, and Thomas Charles Conway. 1995. Mercury, an Efficient Purely Declarative Logic Programming Language. (1995).

[17] Jean Yang et al. 2015. *Preventing Information Leaks with Policy-Agnostic Programming*. Ph.D. Dissertation. Massachusetts Institute of Technology.

A APPENDIX

A.1 Proofs

Lemma A.1. *The rules shown in Figure 3 are a stratified Datalog program.*

Proof. Each rule is a Horn clause. Each premise, in every rule, is a finite relation (for any specific input program). The offset function is also a finite relation. The [E-Implicit] rule uses negation, but the program can be stratified as follows: E_v and S_v are computed first using rules [E-Complete], [E-Partial], [E-Implicified], and [E-VarSlots], and then E is computed using [E-Vars] and [E-Implicit]. □

Lemma A.2. *The minimal edge set E satisfying the rules shown in Figure 3 exists and is unique.*

Proof. By Lemma 1, the rules are a stratified Datalog program, and any stratified Datalog program has a minimal model. □

Lemma A.3. *If a rule R is well-formed, the edge set E defines a total function ζ.*

Proof. We must show that every attribute slot of the rule R is associated with exactly one variable or type according to the edge set E. We split the proof into two parts:

Part I. : E maps every attribute slot s to *at least* one variable or type. Proof: If E_v maps the attribute slot s to variable x, then so does E by [E-Vars]. If E_v does not map the attribute slot s to any variable, then $s \notin S_v$, so by rule [E-Implicit], E maps s to type(s).

Part II:. Every attribute slot s is mapped to *at most* one variable or type. Proof: We consider three cases: could a slot be mapped to (i) more than one type?, (ii) both a variable and a type?, and (iii) more than one variable? For (1), [E-Implicit] maps a slot to its own type, and every slot has exactly one type. Hence a slot cannot be mapped to two types. For (2), [E-Implicit] will map a slot to a type, but only if the slot is not already mapped to a variable. Hence a slot cannot be mapped to both a type and a variable. For (3), we observe that an attribute slot belongs to an atom which is either complete, partial, or implicified. Hence we only need to ensure that each of [E-Complete], [E-Partial], and [E-Implicified] do not map the same attribute slot to multiple variables. For [E-Complete] and [E-Partial], this follows immediately from the fact that there is exactly one variable at every position in the atom. For [E-Implicified], the well-formedness requirement [WF-Ambiguity] ensures that if the slot s has type τ, then there is at most one explicit variable x of type τ in the implicified atom. Hence [E-Implicified] can only map an attribute slot to at most one variable. □

Lemma A.4. *For every variable x in a rule R, there exists an attribute slot s such that $\zeta(s) = x$.*

Proof. If the variable x occurs in a complete or partial atom, this follows directly from [E-Complete] and [E-Partial]. If, on the other hand, the variable x occurs in an implicified atom, then by [WF-Types-2] it also occurs in a partial or complete atom, and hence the former argument applies. □

Lemma A.5. *The ζ function is type safe; in other words, if the ζ function maps two slots s_1 and s_2 to the same variable or type, i.e. $\zeta(s_1) = \zeta(s_2)$, then the type of the slots s_1 and s_2 is the same.*

Proof. We consider two cases: (i) if ζ maps the two slots to the same type, and (ii) if ζ maps the two slots to the same variable. In the former case, by the rule [E-Implicit], the two attribute slots must have the same type. In the latter case, the well-formedness requirement [WF-Types-1] ensures that variables have consistent types in complete and partial atoms, hence [E-Complete] and [E-Partial] preserve type safety. Finally, the [E-Implicified] rule preserves type safety through an explicit type check. □

Lemma A.6. *A well-formed rule R without any implicit parameters is translated to itself.*

Proof. If a rule has no implicit parameters, then it has no partial or implicified atoms. Hence the rules [E-Partial] and [E-Implicified] are not applicable. Instead, the [E-Complete] rule is applicable for every atom. Consequently, every slot of every atom is mapped to its explicit variable. Thus the translation substitutes every attribute slot for its explicit variable, reproducing the original rule. □