INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

Types in Logic Programming

An overview of type systems proposed for logic programming languages

Wolfgang Witt

Masterarbeit

Betreuer Prof. Dr. François Bry

Abgabe am 14. Dezember 2021

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 14. Dezember 2021

Wolfgang Witt
Matrikelnummer: 11860082

Abstract

Types play an important role in modern software development. They help to identify many possible errors at compile-time, before a program is run. Additionally, types can serve as an added documentation for the intended use of a procedure. Further benefits are the possibilities for compiler optimization, e.g. through exhaustiveness checks, and further program analysis, e.g. program verification.
Type systems have to strike a fine balance to achieve this goal. If they are too simplistic, they forbid many programming techniques common to untyped languages of the same programming paradigm. In logic programming, meta-programming has for example proven to be difficult to include into a type system, resulting in different suggested type systems allowing higher-order programming. On the other hand, type systems can become too complex. Type analysis (mostly type inference) might become undecidable, rendering the type system basically worthless. It might also lead to confused developers baffled by error messages or even accepted programs, leaving them with a feeling of an arbitrary type checker.
Logic Programming allows a *declarative* view of programs, providing a very elegant and simple way to reason about a program. A type system would help by providing an additional tool to talk about the program in an abstract way. However, due to its roots in untyped first-order predicate logic, logic programming languages have traditionally been untyped.
This work provides an overview over type systems proposed for logic programming. Their defining features are briefly described, mentioning some of the challenges arising from introducing them in the context of logic programming, as well as suggested solutions to these problems.

Zusammenfassung

Typen spielen eine wichtige Rolle in moderner Software-Entwicklung. Sie helfen dabei, viele potenzielle Fehler zur Kompilierzeit zu finden, bevor ein Programm überhaupt ausgeführt wird. Zusätzlich können Typen als zusätzliche Dokumentation für die angedachte Verwendung einer Prozedur dienen. Weitere Vorteile sind die Möglichkeiten zu Compiler-Optimierungen, z.B. durch Vollständigkeitsanalysen, und zu weiterer Programmanalyse, z.B. Software Verifikation.
Typsysteme müssen dabei gegensätzliche Ziele berücksichtigen. Sind sie zu simpel, verbieten sie viele Programmiertechniken, die in untypisierten Sprachen desselben Programmierparadigma üblich sind. In der Logik-Programmierung war z.B. Meta-Programmierung nur sehr schwierig in Typsysteme zu integrieren. Als Resultat wurden verschiedene Typsysteme vorgeschlagen, die Programmieren mit Prädikaten höherer Ordnung erlauben. Andererseits kann ein Typsystem zu komplex werden. Typanalyse (vor allem Typ-Inferenz) kann unentscheidbar werden, wodurch das Typsystem praktisch nutzlos wird. Es kann auch Entwickler verwirren, die von Fehlermeldungen oder unerwartet akzeptieren Programmen vor ein Rätsel gestellt werden. Dies führt zu dem Gefühl eines willkürlichen Typ-Checkers.
Logikprogrammierung erlaubt eine *deklarative* Sichtweise auf das Programm, mit der es auf elegante und einfache Art möglich ist, Schlussfolgerungen über das Programm aufzustellen. Ein Typsystem würde dabei helfen, indem es ein weiteres Werkzeug bereitstellt, über das Programm in abstrakter Art und Weise zu sprechen. Allerdings sind Sprachen der Logikprogrammierung aufgrund seiner Wurzeln in untypisierter Prädikatenlogik erster Stufe ebenfalls untypisiert.
Diese Arbeit gibt einen Überblick über in der Literatur vorgeschlagene Typsysteme für die Logikprogrammierung. Ihre besonderen Eigenschaften werden kurz beschrieben. Zusätzlich werden Herausforderungen, die durch eine Einführung im Kontext der Logikprogrammierung entstehen, sowie dazu vorgeschlagene Lösungen genannt.

Acknowledgments

I thank Prof. Dr. François Bry for many helpful discussions, as well as always lending an ear both concerning this work and other topics related to studying. Without him, this work probably wouldn't have been possible.
I also thank the librarians at the "Universitätsbibliothek der LMU" who helped me to find some of the more difficult to obtain articles.
A special thanks goes to the developers of *Mercury*. Their language helped me to stir my focus in the right direction, preventing me from getting lost in literature.

Contents

CHAPTER 1

Introduction

In programming, types are a way to formalize given meaning to raw data, e.g. bits and bytes. In literature, especially in articles heavily influenced by mathematics, types are sometimes called *sorts*. The exact definition of a type may vary between type systems, but in most cases the intuition of types as sets of values with some shared properties and operations is reasonably close, although Cardelli and Wegner [CW85] point out other interpretations exist as well. A type system is a collection of types. It is monomorphic if every value has only one type and polymorphic if a value can have multiple types.

Empirical evidence shows a good type system helps finding many common programming mistakes, e.g. switching the arguments of a function, even before a program is run. Therefore, a common opinion states programs above a certain size, e.g. once multiple developers are involved, should be created with a typed language.

On the other hand, type systems might feel restrictive or confusing, leaving the programmer with the feeling of fighting the type checker, instead of it being another pair of helpful eyes. Together with the added verbosity of type annotations, this might contribute to explain why Python and JavaScript, both untyped, are currently two of the most-used languages (e.g. see [Zap21]).

Similarly, Prolog as the main logic programming language is untyped as well. This can be traced back to first-order logic serving as the basis for the logic programming paradigm. This work provides an overview of type systems proposed for logic programming by various authors.

Chapter 2 provides a brief description of logic programming, introducing commonly used terminology and providing references for further reading. It doesn't provide detailed insight, just enough to ensure a common terminology. The interested reader is referred to other literature. [Llo87; BG94]

Chapter 3 focuses on descriptive type systems. They take an untyped program, describe its clauses with types, in this context subsets of the Herbrand Universe as an over-approximation of potentially successful values, and use them to find errors, allow for compiler optimizations, or help with further analysis of the program.

Chapter 4 looks at prescriptive type systems. Here, types are part of and essential to the meaning of programs. Usually, type declarations have to be provided by the programmer at strategic locations, e.g. for all predicates, allowing the type of other expressions to be

inferred. Special attention is brought to the type system of Mercury[1], since it has a ready-to-use compiler, allowing to try it hands-on.
Chapter 5 looks into type system features which, to my knowledge, haven't been explored in the context of logic programming.
Chapter 6 provides a short summary and some concluding thoughts.

[1]Mercury is available at `https://mercurylang.org/`

CHAPTER 2

Logic Programming

Logic Programming stems from the idea of assigning a *procedural interpretation* to first order logic. Its main building block are clauses, also called rules, taking the following form:

$$A_0 \leftarrow A_1, \dots, A_m, \text{not} A_{m+1}, \dots, \text{not} A_n$$

A_is are atoms and *not* a logical connective called *negation as failure*. A_0 is called the *head* (or conclusion) of the clause and everything to the right side of the arrow its body (or premise). A clause without a head is called a *goal*, one with an empty body a *fact*. [Llo87; BG94] Negation as failure gives rise to non-monotonic logic (e.g. Chitta Baral and Michael Gelfond use stable model semantics [BG94]), but is not addressed further since it has no (significant) impact on possible type systems. For easier writing, in logic programming languages the arrow is often replaced by the ASCII sequence `:-` and each clause ends with a dot "`.`".
Atoms, also known as atomic formulas, have the form $p(t_1, \dots, t_n)$, where t_is are terms and p is a n-ary predicate symbol. For 0-ary predicates the parentheses are usually dropped. When talking about predicates, it is helpful to include the arity in the name to avoid confusion. The predicate `p/n` specifies a predicate with name p and arity n, and is in fact different to the predicate `p/m` with $m \neq n$.
Following [Llo87] terms are defined inductively:

- A variable is a term.
- A constant is a term.
- If f is a n-ary function symbol and $t_1, \dots, t_n$ are terms, then $f(t_1, \dots, t_n)$ is a term.

Variables are usually written with a name starting with a capital letter, constants with a lower case letter. Functions are in general not evaluated, instead a term consisting of the function symbols and arguments is build. A variant called *functional logic programming* exists, where evaluated functions exist in addition to predicates. Since it is possible to encode a function as a predicate (and vice-versa if tuples exist), both from a semantics and typing perspective, evaluated functions won't be discussed in detail. In fact, *Mercury* internally encodes every function of arity n as a predicate of arity $n + 1$, automatically replacing function calls with predicate calls and unification of the extra result argument. [Jef02]
Terms, and by extension atoms and formulas, are called *ground* if they do not contain any variable. The set of all ground terms is called the Herbrand universe, the set of all ground atoms the Herbrand base. Both are defined in terms of some program.

A program is run by giving an initial goal, e.g. following C convention "`:- main.`". Let $\leftarrow C_1, \dots, C_k$ be the current goal, then a computation step tries to unify some C_i with the head A of a program clause $A \leftarrow B_1, \dots, B_n$. If unification succeeds with unifying substitution ϑ, the next current goal is $\leftarrow (C_1, \dots, C_{i-1}, B_1, \dots, B_n, C_{i+1}, \dots, C_k)\vartheta$. Once the empty clause is produced (or no unification is possible) terminate the calculation. [Llo87]
A substitution is a syntactical operation on terms and, by extension, atoms and formulas which replaces variables with terms. A substitution ϑ is unifying for two terms t_1, t_2 if both terms are equal after the substitution $t_1\vartheta = t_2\vartheta$. In literature, multiple different notations are used for substitutions. In this work, choosing one notation is avoided when possible, describing them only informally.
Example 1:
Let `X` be a variable, `a` an atom and `:- p(X).` the current goal. Further, let `p(Y) :- u(Y,a).` and `u(Z, Z).` be clauses in the program. A valid computation step unifies `p(X)` with `p(Y)`, where the unifying substitution ϑ_p replaces the variable `X` with the variable `Y`. The new current goal is "`:- u(Y, 1).`". A possible next (and final) computation step unifies `u(Y, 1)` with `u(Z, Z)`. The corresponding substitution ϑ_u replaces the variable `Y` with the variable `Z` and the variable `Z` with the atom `a`. Since the relevant clause for `u/2` has no body, the empty goal is produced resulting in the successful termination of the calculation.
In general, multiple unifications are possible for the current goal, e.g. in example 1, the program might contain multiple clauses for `p/1`. Due to the deterministic nature of current computer systems, this is usually implemented via backtracking. The selection order (which goal atom is tried to unify with what clause head first) is dependent on the language and sometimes even its implementation.

CHAPTER 3

Descriptive Type Systems

Descriptive type systems assign types to the terms of a previously untyped language, e.g. *Prolog*. These types help to statically reason about the code, e.g. for compiler optimizations, or identifying unsatisfiable clauses. Since the source code doesn't contain any type information, the type of every term has to be calculated. This process is called *type inference* in contrast to *type checking* and *type reconstruction* used for prescriptive type systems. The main advantage is the ability to continue using the same language, e.g. to continue using well-tested libraries. In logic programming, descriptive types are usually interpreted as a subset of the Herbrand universe, approximating the meaning of a predicate as a superset of potentially successful program derivations.

3.1 Regular Trees as Types

One of the first type system for logic programming was proposed by Prateek Mishra in [Mis84]. It is purely descriptive, so the user doesn't have to provide any type information since every type is inferred. Types are interpreted as subsets of the Herbrand universe and described using regular trees. The type for a term `t` is written as `T(t)`.
Ground regular trees (GRT) are defined in [Mis84, p.292] by the following rules:

* The empty tree Φ is a GRT.
* Any constant symbol a,b,c.. is a GRT.
* Given n GRTs, $r_1, \ldots, r_n$ the tuple $(r_1, \ldots, r_n)$ is a GRT.
* Given function symbol f and GRT r, an application fr is a GRT.
* If t_1 and t_2 are GRTs, then $t_1 + t_2$ is a GRT.
* If $g(z)$ is a GRT involving variable z then $z!g(z)$ is a GRT.

Their interpretation is specified by an interpretation function **I** as expected for the first four cases. The type $t_1 + t_2$ is interpreted as the (set) union of t_1 and t_2. The interpretation for $z!g(z)$ is the smallest set of terms X satisfying $X = \{g(x)|x \in X\}$. The author adds the minimal solution always exists and is given by $I : z!g(z) = \cup_{i=0}^{\text{inf}}\{g^i(\Phi)\}$.

The author also defines parameterized regular trees (PRT), denoting sets of GRTs. However, due to issues with type inference, partially similar to those discussed in section 4.4, they are not discussed beyond addressing problems with the presented inference algorithm.
Polymorphism in the type system is achieved through subtypes, where one type may be fully contained in another type. Its presence is obvious from the type $t_1 + t_2$, but other types may be supertypes as well. Since many types are infinite, solving subset relations is not a realistic option. Instead, Mishra provides an algorithm using the GRT representation of the involved types.
Types of predicates are over-approximations of their success sets, and are written as a tuple of the types of its arguments. E.g. examples 3.1.1 and 3.2.1[1] from [Mis84], using # as an infix tuple operator.

```
sub1(succ(zero), zero).
sub1(succ(succ(X)), succ(Y)) :- sub1(succ(X), Y).

T(sub1) = z ! succ(zero) + succ(z)
        # z ! zero + succ(z)
```

A full type inference algorithm is provided. One of its noteworthy properties is it only produces tuple distributive types. Therefore, the described type system only includes tuple distributive types as well.
Example 2:

```
p(a, c).
p(b, d).

T(p) = (a+b, c+d)
```

Although the predicate `p` will only succeed with the argument-combinations `(a,c)` and `(b,d)`, the inferred type includes `(b,c)` and `(b,d)` as well. While some precision is lost, the type is clearly correct since all success cases are included.
One of the main results is the ability to identify clauses with an empty success set, i.e. clauses that will never succeed regardless of their arguments. This led the author to suggest replacing the famous claim "well-typed programs do not wrong" used in prescriptive type systems (e.g. [MO84]) with "Ill-types programs cannot succeed" as their correctness coin phrase.
The work was one of the first on types in logic programming, thus it is mostly of theoretical nature. Additionally, some parts of logic programming weren't discussed.
First, negation wasn't considered and might prove difficult to integrate. Using the naive set difference with the Herbrand universe won't work with supersets of the success set (everything else will fail), so a second type interpreted as subset of the success set (guaranteed to succeed) would be required. Interaction with negation-as-failure is unclear, so it must also be investigated. However, there might be a more elegant way, so further research is required to provide an answer.
Second, higher-order programming wasn't covered. There is no obvious reason for it not working in the presented type system, but introducing it to prescriptive type systems (see section 4.5) has proven to be difficult.
Third, only types representable by GRTs are included. The author notes "[s]ome simple polymorphic predicates (e.g. **append** and **reverse**) can be analyzed by an ad-hoc expansion of the standard algorithms". [Mis84, p.297] However, a dedicated algorithm for dealing with PRTs is required for most polymorphic predicates.

[1]A typo in the type has been corrected.

3.2 Operator Fixpoints

Nevin Heintze and Joxan Jaffar in [HJ92] explore descriptive types, they call them semantic types, defined and calculated with different approaches. The first approach uses a collection of formulas describing relationships between sets of atoms, e.g. used by Mishra in [Mis84]. A type is defined as a specific model, e.g. the least model, of the collection of formulas. The second approach is based on an adaption, usually introducing an approximation, of the immediate consequence operator T_P associated with the program P. A type is defined as the fixpoint of the adapted operator. They also mention a third approach based on flow analysis, but don't include it in their analysis.
Heintze and Jaffar show the types defined using set formulas and those defined with fixpoints of approximated immediate consequence operators are equivalent, hence establishing an equivalence between both formalism. They also present three specific type variants of varying accuracy, only the least accurate one using tuple distributive types. The two more accurate types are recursive sets and can be represented by a regular tree grammar. Whether this is possible for the third type is left as an open question, but representation with a regular tree grammar is conjectured to be possible.

3.3 Regular Types

Philip W. Dart and Justin Zobel in [DZ92] investigate *regular types* represented by a regular term grammar (not to be confused with a regular grammar). These types are a generalization of those used by Mishra in [Mis84].
With the goal of identifying whether a given type is a regular type they start by providing a pumping lemma which is a necessary, but not sufficient condition for regular types. Improving the lemma to a more specific, at best necessary condition is left as an open problem.
Type inference and analysis based on types require reasoning about types. The authors provide algorithms to decide whether a *regular type* is empty, a subset of another *regular type* and to calculate the intersection between two *regular types*. They show the intersection of two *regular types* is a *regular type* as well. All three algorithms are exact and don't rely on approximations, e.g. in contrast to Mishra types don't need to adhere to tuple distributivity. For type inference, Dart and Jaffar provide an algorithm *unify* for type unification, generating a *type unifier*. Due to problems comparable to tuple-distributivity, not all types have a most general type unifier. In consequence, *unify* returns a weak type unifier as an approximation. The authors note that for all their algorithms using *unify* a weak type unifier is sufficient. Furthermore, they show for types defined by deterministic type rules the result is always exact. However, the question whether *unify* returns a minimal weak type unifier in general is unknown and left as an open question.
The work can be viewed as a framework for creating a type system based on regular types. It doesn't provide any specific type system, but provides algorithms for and identifies some properties of type systems built upon regular types. As a caveat, parametric types are only allowed as a notational convenience. According to the authors, the results can "easily be modified to use instantiated parametric type[s]" [DZ92, p.184], but won't work if type variables are involved.

3.4 Discussion

Descriptive type systems are defined without introducing new syntax or special constructs into the language. Therefore, they allow reasoning about code based on types even for existing code without any additional developer effort. Program verification of an untyped

language for example usually starts by inferring types based on a descriptive type system. In logic programming, they allow to identify unsatisfiable clauses usually caused by a programming error. Additionally, the inferred type might provide insight into the operational semantics of a predicate.
Nonetheless, the inferred type doesn't always match the type intended by the developer as shown by Lee Naish in [Nai92]. Consider for example the `append` predicate with its usual definition. Then the following query is successful, even though two of the arguments are not lists.

```
?- append([], a, a).
true.
```

This manifests in the inferred type for `append`, described as `[list(any), any, any]` in [WK20]. In a descriptive system, outside of the developer carefully checking every inferred type, there is no way to find this mismatch since the desired type is unknown.
As a way to bridge the gap to prescriptive type systems, Tom Schrijvers, Vitor Santos Costa, Jan Wielemaker and Bart Demoen propose an optional prescriptive type system based on the one introduced by Mycroft and O'Keefe discussed in section 4.1. A type-safe interface with untyped code is achieved through introduction of runtime type checking. According to [WK20] the project appears to be abandoned, however the prolog package *type_check* is available as an artifact[2].
Isabel Wingen and Philipp Körner use a similar approach in [WK20] with the tool *plspec*. Instead of runtime type checks, their type system includes an *any* type, including *no type information available* as part of their type system. Type information is inferred from builtin predicates with manually provided type information. Their type system is based on a set of builtin types, some of them polymorphic like list(α), with subtype relations between them. They might be combined with a union or intersection operation, resulting in types looking very similar (not identical at least through inclusion of polymorphic types) to *regular types* described in section 3.3. Their type system distinguishes between variables and ground terms, allowing predicates to have pre- and postconditions on their arguments, thus including mode information discussed in section 4.2. They notice problems with library code since they are pre-compiled by modern Prolog systems. Their clauses can't be accessed or are wrappers of a C library. Without annotations to these libraries, the amount of inferred types remains relatively low resulting in many variables with inferred type *any*. Nonetheless, the authors report successful identification of type errors in several Prolog repositories. While a full implementation exists, the work appears to be at an early stage resulting in sometimes drastic performance issues. Some ideas for improving both performance and handling of libraries are presented. Time will tell, whether they are successful resulting in a useful tool based on static type analysis.

[2] *type_check* is available at `https://www.swi-prolog.org/pack/file_details/type_check/prolog/type_check.pl?show=src`

CHAPTER 4

Prescriptive Type Systems

In prescriptive type systems, types are a part of the programming language. The developer can provide a type signature, signalling his intent in a machine-verifiable way avoiding the problems described in section 3.4. Here, the goal `append([], 3, 3)` would be ill-typed and therefore without meaning. According to Frank Pfenning [Pfe92], this notion of types goes back to Church [Chu40].
Since providing a type declarations for every value is quite tedious it is often possible to omit them for many values. The process of filling in missing type information is called *type reconstruction* and is a generalization of *type checking*.
Only well-typed clauses and goals have a meaning. Therefore, the well-typing has to be ensured before the execution of a program and goal. Consequently, both type checking and type reconstruction have to be decidable. Otherwise, a program might never be able to run, indefinitely waiting for the type checker to report a result. In addition, type system might have a runtime cost, e.g. by requiring type information and type checks at runtime to ensure an execution step can only produce well-typed goals. Ideally, this is not necessary resulting in an identical, or even improved runtime behaviour when compared to a semantically identical code in an untyped language.

4.1 Parametric Polymorphism

The basis for basically all prescriptive type systems is the one introduced by Alan Mycroft and Richard A. O'Keefe in [MO84], from now on abbreviated as MO type system. It is a translation of Milner's type system for ML [Mil78] featuring parametric polymorphism to logic programming. Types may be defined in term of type arguments, usually called parameters to avoid confusion with value level arguments. The first name of a type is called a *type constructor* and the number of parameters specifies its arity, e.g. considering the type `list(`α`)` its type constructor is `list` of arity `1`, in a logic programming context often written as `list/1`. The full power of parametric polymorphism is achieved through type variables. They can be substituted with any type to form an instance of the original type, with a pure renaming of type variables forming a type variant. For example, let α and β be different type variables. Then `list(`α`)` and `list(`β`)` are type variants (and instances) of each other, while `list(int)` is an instance (but not a variant) of both.

More formally, let $\texttt{TCons}_n$ be the set of all type constructors of arity `n` and `TVar` the set of all type variables. Types are defined by the following Grammar:

$$\texttt{Type::= TVar | TCons}_n\texttt{(Type}^n\texttt{)}$$

A type without any type variables, e.g. `bool` or `list(int)`, is called a monotype. Otherwise, e.g. `list(`α`)` or `pair(int, `β`)`, it is a polytype. Type instances (and variants) are usually described through type substitutions which are a mapping of type variables to types. The syntax for definition and application of type substitutions differs between authors, thus requiring careful reading.
In difference to [Mil78] the restriction where "mutually recursive definitions can only be used nongenerically within their bodies" [MO84, p.297] is lifted. For example, the following ML program presented in [MO84, p.297] is ill-typed, but its Prolog equivalent is allowed in the described type system:
Example 3:

```
let rec I x = x
    and f x = I(x + 1)
    and g x = if I(x) then 1 else 2
```

Since mutual recursion is prevalent in Prolog clauses, the authors note keeping this restriction would make polymorphism practically useless.
Although no typed execution semantics has been given, the authors show that using SLD-resolution on the corresponding untyped Prolog program, a well-typed program and resolvent will never produce an ill-typed resolvent, i.e. won't "go wrong". Additionally, a typed semantics together with a type reconstruction algorithm was provided in [LR91] by Lakshman and Reddy, addressing some of the initial concerns about the type system.
Mycroft and O'Keefe propose developers supply type declarations for predicates and functors, automatically inferring the types of variables. In addition to documentation by the written form of the types, this also reduces the risk of unintended positives like `append([],3,3)`. Lakshman and Reddy's algorithm allows to even omit type declarations for predicates, providing them only for functors (e.g. as a meta-predicate `:- type bool=false,true.`). However, they note type reconstruction for recursive polymorphic predicates is undecidable, citing an earlier work showing an equivalence to semi-unification [LR91]. As a solution, the algorithm assumes the type of a body occurrence in recursive definitions is *identical* to (not just an instance of) its type signature. This would make example 3 ill-typed as well. The authors believe this is not a serious problem in practice, referring to some functional programming languages using the same type reconstruction algorithm.
Execution Semantics of well-typed programs in the described type system don't depend on the type of variables and values. Thus, it is not necessary for type information to be present at runtime, reducing the required memory and resulting in faster execution of the program.
The MO type system doesn't allow higher-order programming, but the authors propose ways how it might be extended. It is interesting to note, there is no typing rule for negation present. Adding one would pose no problem from a typing perspective (see [LR91, p.14]), but might become redundant with the introduction of higher-order (predicate) types.

4.1.1 Head Condition

One unexpected property of the MO type system is the so-called *head condition*, not explicitly stated in [MO84] and named "Definitional Genericity" in [LR91]. It restricts the use of any arbitrary type instance of a predicate in the head of a clause, instead only type variants are allowed. For example, consider this extended definition presented in [Han89a] of the `append` predicate violating the head condition (both type and term variables are written with an uppercase letter):

Example 4:

```
:- pred append(list(A), list(A), list(A)).
append([1,2], [3,4], [1,2,3,4]).
append([], T, T).
append([X | L], Y, [X | T]) :- append(L, Y, T).
```

In the first clause, all arguments have the type `list(int)` instead of the declared `list(A)`. Thus, the head of the clause is only an instance `(list(int), list(int), list(int))` instead of (a variant of) the declared type `(list(A), list(A), list(A))`. This is obviously a very constructed example; it was chosen, since it showcases problems with rejecting the head condition very well. A more realistic example violating the head condition is a generic `print` predicate.
Assuming the head condition wasn't enforced, it would be mandatory for type information to be present at runtime. Otherwise, there would be no way to tell which clause to try first in the recursive call of the third clause.
Example 5:
Compare the following two queries, both initially unifying with the third clause.

```
?- append([5,1,2], [3,4], [5,1,2,3,4]).
?- append(["hello"], ["world"], ["hello", "world"]).
```

The integer-query will short-cut evaluation with the specialized clause. Conversely, the string-query will directly continue to the base case of an empty first list without considering the specialized clause.
Among other things, Pierre Deransart and Jan–Georg Smaus discuss the head condition in more detail in [DS01]. According to them, it can be viewed as an essential characteristic of *generic* polymorphism, opposed to *ad-hoc* polymorphism, i.e. different behaviour based on the type of the arguments.
Michael Hanus proposes a type system without this restriction [Han89a]. Here, normal unification has to be replaced by a typed unification, but many cases allow for optimizations where the type can be ignored in the unification procedure. One of the main benefits they gain by lifting the head condition is a way of higher-order programming, using constants to represent predicates and functions described in more detail in section 4.5.
Pascale Louvet and Olivier Ridoux take a different approach in [LR96]. They notice typing problems with useful program identities/transformations if the head condition isn't enforced. As an answer, they define an extension to λProlog called λ_2Prolog. They introduce a new type $\Pi\sigma.\sigma'$, which can be interpreted as a function `Type` $\rightarrow$ `Type`. In this system, the extended *append* predicate would have the type $\Pi\alpha$`.(list(`α`), list(`α`), list(`α`))`.
Additionally, terms allow for *type guards* providing the ability for different runtime behaviour depending on the type, without sacrificing the head condition in the process.They note these type guards act like additional constraints on unification, therefore implementation can be similar to unification in a *constraint logic programming* language. For type reconstruction the authors provide an decidable algorithm based on anti-unification instead of the usual unification-based approach. The cost for this expressiveness is a complicated type system which might deter developers new to statically typed languages.
Another way to achieve different runtime behaviour depending on the type are type classes described in section 4.6.

4.1.2 Interaction with Theorems for Free

Theorems for free is a concept in functional programming introduced by Philip Wadler. It states "[e]very function of the same type satisfies the same theorem". [Wad89, p.1]
Example 6:

Let A, B be types, $a : A \to B$ a function from A to B and $I : \forall X.X \to X$ a polymorphic function from any type to itself. Then it doesn't matter in which order a and I are applied $a \circ I_A = I_{A'} \circ a$. Since I has to be defined independent of the type, it has to behave identical for every type. This example was originally stated in [Wad89, Figure 1].
Functions can be encoded with predicates, so a translation of the concept to logic programming should be possible. Due to the difference of functional and logic programming, a mode and determinism system (see sections 4.2 and 4.3) might be necessary to do so. Lee Naish provides a similar example in [Nai96a, p.6]: A predicate with type $\forall T.(T, T)$ and mode (in, out) (he doesn't consider determinism, leaving it arbitrary) must be a subset of the identity relation.
However, if the *head condition* is not enforced the theorems no longer hold.
Example 7:
Let `A=int, B=float` and `a: pred(int::in, float::out) is det` be the usual cast from `int` to `float` values, e.g. `a(1, 1.0)` is a solution. Further, consider the following polymorphic predicate `p: ∀ X. pred(X::in, X::out) is multi`, where the first clause is only defined for the type `int`. It is only possible to define this way if the *head condition* is not enforced.

```
:- pred p(A::in, A::out) is multi.
p(X, Y) :- succ(X, Y).
p(X, X).

:- pred succ(int::in, int::out) is det.
succ(X, Y) :- Y is X + 1.
```

Clearly, the following two goals produce different results even though they only differ in application order of the predicates (contradicting the translated theorem from example 6).

```
?- p(4, Y), a(Y, Z).
Y = 5, Z = 5.0.
Y = 4, Z = 4.0.
?- a(4.0, Y), p(Y, Z).
Y = 4.0, Z = 4.0.
```

It is unclear how strong a translation of theorems for free would be for logic programming, if it is possible at all. Nonetheless, example 7 clearly demonstrated enforcing the *head condition* is a requirement for it to be possible. Further research is to required to decide whether a type system enforcing the head condition produces useful free theorems.

4.2 Modes

Modes describe the instantiation of arguments prior to and after successful evaluation of a predicate. If an argument is a variable it is `free`, if it is fully evaluated it is `ground`. Depending on the instantiation of its arguments a predicate might have drastically different runtime behaviour, demonstrated in example 8 originally given in [ZY92].
Example 8:

```
reverse1([X | L1], L2) :- reverse1(L1, L3), append(L3, [X], L2).
reverse1([], []).

reverse2(L1, [X | L2]) :- reverse2(L3, L2), append(L3, [X], L1).
reverse2([], []).

append([], L, L).
append([X | L1], L2, [X | L3]) :- append(L1, L2, L3).
```

The declarative semantics for both `reverse1` and `reverse2` are identical, both reversing a list. However, using usual Prolog execution semantics, if the second argument is `free` prior to evaluation of the predicate, evaluation of `reverse1` produces the expected result and `reverse2` diverges. Both queries have been tested with SWI-Prolog 8.2.2.

```
?- reverse1([a,b], L).
L = [b, a]
:- reverse2([a,b], L).
% infinite loop
```

In [Jac92], Dean Jacobs only looks at instantiation prior to predicate evaluation. He proposes using a set of integers to mark ground arguments of a predicate. Later chapters already suggest linking modes to types modeled after [Mis84] using regular trees, but only considers ground arguments.
Joseph L. Zachary and Katherine A. Yelick explore the effect of different evaluation semantics. By imposing a youngest-literal-first restriction on computation rules, they identify a class of interpreters which diverge on exactly the same inputs for predicates (and programs by extension) with equivalent declarative semantics and mode specification. They also introduce a way to define modes for types (even though they call them sorts): For every type `T` the mode $\texttt{any}_{\texttt{T}}$ exists and contains all terms (including variables). In addition, the user can define their own modes, based on functors of the type together with the modes of their arguments. This allows to properly account for partially instantiated variables, e.g. a list where only its length, but none of its arguments are known, which are a feature exclusive to logic programming.
[DH88] also considers instantiation of arguments after a successful evaluation of predicate. Its three modes are called *i*, *o* and *io*, linking them to data flow in the predicate.
Mercury allows mode specification by giving pre- and post-conditions about instantiation of an argument [Jef02]. Both are automatically verified by the compiler and an important aspect of its semantics.
Mode analysis can be done independently of a type system, thus might be regarded orthogonal to it. Nonetheless, since a mode violation feels similar to a type error, e.g. a program gets rejected by the compiler, it should be treated as an extension to type checking. Note that one predicate may have multiple valid mode declarations.
Example 9:

```
:- pred length(list(A), uint).
:- mode length(in(list_skel(I)), out).
:- mode length(out(list_skel(free)), in).
length([], 0u).
length([_|T], X) :- X = 1u + Y, length(T, Y).
```

The mode `in(I')` is an abbreviation for an unchanging instantiation `I'`, `list_skel(I)` is abbreviation for a list where all the cons-cells are known with elements instantiated as `I` and `out(I')` specifies a `free` variable which is instantiated with `I'` after successful evaluation. Both `in` and `out` are special cases, using `ground` as argument in their respective parametrized versions. The first mode of the predicate calculates the length of a list with known spine. The second mode produces a list with specified length, where all elements are left as free variables.
Additionally, mode and type systems synergise with each other, allowing for both more precise modes and more flexible type rules. The former is used in *Mercury* where instantiation descriptions (and modes by extension) are defined for some type, e.g. `[ground | free]` for a non-empty list of arbitrary length with fully instantiated first element. Mode information

is then used to decide the evaluation order of predicates, instead of the syntactical left-to-right order used in Prolog. The latter has been used in [DH88] to introduce subtypes into logic programming. It is discussed in more detail in section 4.4.
Usually modes are considered from a procedural viewpoint. Lee Naish provides a declarative interpretation in [Nai96a]. He proposes to use *constraint regular trees* to represent polymorphic modes. However, mode inference has not been fully developed, e.g. key algorithms like intersection and unification are still missing. Nonetheless, the author sees promising indications that "constraint regular trees with a suitably rich constraint language can form an excellent basis for mode analysis". [Nai96a, p.11]

4.3 Determinism

Determinism information specifies how many solutions exist for a predicate for different arguments. The determinism of a predicate depends on its mode and can be divided into four classes. "The four classes are those that have exactly one solution, those that have at most one solution, those that have at least one solution, and those that have any number of solutions." [SHCO95, p.5] Example 10 shows the mode for the `append` predicate, originally given in [SHCO95] and slightly extended.
Example 10:

```
:- pred append(list(A), list(A), list(A)).
:- mode append(in, in, in) is semidet.   % at most one solution
:- mode append(in, in, out) is det.      % exactly one solution
:- mode append(out, out, in) is multi.   % at least one solution
append([], Y, Y).
append([H|A], B, [H|C]) :- append(A, B, C).
```

The `append` predicate has a different number of solutions dependent on the modes. If all arguments are ground, the predicate might fail or succeed. If the first two arguments are ground, a solution always exists with the appended list as the third argument. However, if only the third argument is ground at least one solution always exists, e.g. the first argument is the empty list and the second a copy of the third list. More solutions are constructed by moving the first element of the second list to the end of the first list.
Prescriptive determinism information provides similar advantages to a prescriptive type system. It may help eliminating errors, e.g. identifying a failing predicate that should never fail. Furthermore, it may be used for optimizations, e.g. directly stopping after the first solution for a deterministic mode of a predicate.
Mercury also uses determinism information to ensure no backtracking happens for predicates with side effects. A predicate with IO must be deterministic, i.e. with exactly one solution. Alternatively, it can also be `cc_multi` where the predicate is guaranteed to succeed and execution (forcefully) stops after the first solution.
Zoltan Somogyi, Fergus Henderson and Thomas Conway note checking correctness of determinism declarations is undecidable in general. Nonetheless, fast approximate (conservative) solutions exist. The authors claim they work well in practice, and my experience with *Mercury* which implements the described determinism system supports this claim. For the rare cases the approximate solution is too general, *Mercury* also provides pragmas forcing the compiler to blindly trust the declared determinism.

4.4 Inclusion Polymorphism (Subtypes)

Inclusion Polymorphism, better known as subtypes, allows specifying subtype relations like $\sigma \leq \tau$ where all values (including variables) with type σ also have type τ. Since subtype

relations form a quasi order on types, in literature type systems including this feature are also referred to as order-sorted. One typical example for subtypes is the declaration `int` $\leq$ `float`, i.e. every integer value is also a floating point value. While certainly true in a mathematical sense, especially assuming arbitrarily large and precise numbers, programming languages rarely include this specific subtype relation due to different runtime properties and semantics. For example, consider the following two Prolog queries. Only the floating point version shows rounding inaccuracies.
Example 11:

```
?- X is 100000000000000000 + 2.
X = 100000000000000002.
?- X is 100000000000000000.0 + 2.0.
X = 1.0e+17.
```

Roland Dietrich and Frank Hagl suggest an extension to the MO type system, introducing the ability to declare subtype relations. They notice subtypes cannot be used anywhere a corresponding supertype can be used, linking the resulting typing of output arguments to the data flow of a predicate which can be provided by a mode system (for more detail about modes see section 4.2). They use three data flow modes for predicate arguments: *i* if the instantiation doesn't change (input arguments), *o* if it is a free variable at call time (output arguments) and *io* in every other case (inout arguments). *Input* arguments may be used with (an instance of) the specified type or one of its subtypes, *output* arguments with (an instance of) the specified type or one of its supertypes, and *inout* arguments only with (an instance of) the specified type. This difference between input and output position in subtype usage also exists in object-oriented programming languages, although there is a syntactical, and therefore more obvious separation between them.
Similar to the MO type system, type inference rules are given as a set of Horn clauses and adjusted to allow function symbols and variables to have more than one type. Additionally, a *variable typing condition* (VTC) is introduced which "guarantees that in well-typed clauses dataflow cannot happen from a supertype occurrence to a subtype occurrence" [DH88, p.86]. Assuming type and mode checking succeeds (including VTC), they show no ill-typed resolvent will be produced, thus the program won't "go wrong" in the sense of Mycroft-O'Keefe, without the need for runtime type or mode information.
Furthermore, they sketch how the implementation of a type checking algorithm could look like. Solving subtype relations poses the biggest challenge. Once polytypes are involved, it is impossible to decide whether one type is the subtype of another. They introduced conditional subtype relations (CSR) as method of formalising subtype relations depending on parameters. Type checking then involves showing whether a set of CSRs is satisfiable. Since the most general form of these CSRs are Horn clauses it is in general undecidable, but there are special cases where type checking becomes decidable. [Han91, p12]
Patricia Hill and Rodney Topor in [HT92] restrict subtype relations to type constructors of the same arity. In addition, for a subtype relation to hold they require monotonicity in their parameters. Consider for example K and L two m-ary type constructors with $K \leq L$, then $K(\sigma_1, \ldots, \sigma_m) \leq L(\tau_1, \ldots, \tau_m)$ holds only if $\forall i \in \{1, \ldots, m\}.\sigma_i \leq \tau_i$. The authors show type checking is decidable with these restrictions by providing a type inference algorithm and a proof for its termination. They continue by exploring several evaluation procedures for typed logic programs. These procedures are only sound if type information is present at runtime, executing runtime type checks. An important caveat was raised by Deransart and Smaus. They note some of the results have been shown to be faulty (Lemmas 1.1.7, 1.1.10, 1.2.7), affecting type systems which include subtypes [DS01]. Thus, the results in [HT92] have to be treated with care.
François Fages and Emmanuel Coquery provide a more general result in [FC01] than Hill and Topor, allowing subtype relations between type constructors of decreasing (or equal) arity. It

is for example possible to define a universe type $\top$ as a supertype of every type, but it is not possible to define `int_list` $\leq$ `list(A)`. The authors show the constraints produced by type checking can be solved in linear time, and those produced by type reconstruction in cubic time, assuming the types form a lattice. Therefore, type inference is again decidable under the described constraints. Again, runtime type checks are necessary to ensure soundness. Since they assume a *constraint logic programming* language they add them to program constraints, but translating the results to a logic programming language without constraints solving should pose no problem. One interesting property of the type inference algorithm is, it produces two types for predicates: the minimal type and a more permissive heuristic type that can include type variables.
In [Han91], Michael Hanus also considers an earlier work for functional programming by You-Chin Fuh and Prateek Mishra [FM88]. They consider the case where subtype relations are only possible between (different) basic types, or for the same type constructor with every parameter either being monotonic or anti-monotonic. Fuh and Mishra provide an decidable algorithm for solving subtype constraints under these conditions. Since these results are for functional programming, it is unknown whether in logic programming type information is necessary at runtime.

4.5 Higher Order Types

Higher-order programming allows defining predicates and functions in term of other predicates and functions, possibly unknown at compile time. Higher-order functions (and predicates by extension) allow to "express the essential part of an algorithm, while abstracting away the details that vary between different uses of that algorithm" [SHCO95, section 2.4], allowing code reuse where it previously wouldn't be possible.
Example 12:
The following predicate `map` originally presented in [SHCO95, section 2.4] transforms one list into another, by applying a (user-defined) function-like predicate to all its arguments.

```
:- pred map(pred(T1, T2), list(T1), list(T2)).
:- mode map(pred(in, out) is det, in, out) is det.
:- mode map(pred(in, in) is semidet, in, in) is semidet.

map(_Pred, [], []).
map(Pred, [X|Xs], [Y|Ys]) :-
      call(Pred, X, Y),
      map(Pred, Xs, Ys).
```

The problems of higher-order terms are twofold. First, according to [SHCO95] comparison of higher-order terms are undecidable. Second, traditional techniques to allow higher-order programming in untyped Prolog are ill-typed in polymorphic type systems strictly following the MO type system.
Continuing his work in [Han89a], Michael Hanus proposes a type system based on first order logic, but allowing higher-order programming in [Han89b]. This is achieved by introducing a constant λ`p` for every predicate `p`. They are linked by a predicate `applyN` where `N` is the arity of `p`. The `applyN`-predicates are polymorphic in their arguments, but the predicate `p` might not be. Therefore, defining `applyN` is only possible if the *head condition* is rejected as described in [Han89a].
Example 13:
The binary predicates `not` and `inc` together with their corresponding constants and the higher-order predicate `map` are presented. The definition of `apply2` is only possible, since the type system doesn't enforce the *head condition*. It was originally presented in [Han89b] and slightly adjusted to follow an extended *Mercury* syntax.

```
:- type bool ---> yes; no.
:- type nat ---> 0; s(nat).
:- type list(A) ---> []; [|](A, list(A)).

:- pred map(pred2(A, B), list(A), list(B)).
map(_P, [], []).
map(P, [E1|L1], [E2|L2]) :- apply2(P, E1, E2), map(P, L1, L2).

% this construct is not part of Mercury
% it mimics Haskell-GADTs to define the respective constants
:- type pred2(A, B) where [
    λnot: pred2(bool, bool),
    λinc: pred2(int, int)
].

:- pred not(bool, bool).
not(yes, no).
not(no, yes).
:- pred inc(nat, nat).
inc(N, s(N)).

% here, we link the λ-constants and corresponding predicates
% rejecting the head condition is required for defining apply2
:- pred apply2(pred2(A, B), A, B).
apply2(λnot, B1, B2) :- not(B1, B2).
apply2(λinc, I1, I2) :- inc(I1, I2).
```

With this definition of `map`, it is possible to transform every element of an input list with the specified predicate. This use case corresponds to the one in functional programming.

```
?- map(λnot, [yes, no, no], L).
L= [no, yes, yes].
```

Furthermore, it is possible to search for predicates which relate two lists, identifying it with its λ-constant. Since there is only a finite number of predicates, those explicitly defined in the program, the search for predicates is only undecidable if the predicates are undecidable themselves. Comparison is done purely based on the result of calling a predicate with a fixed set of arguments, not directly on higher-order terms.

```
?- map(P, [0], [s(0)]).
P = λinc.
```

Obviously, defining the `applyN` predicate clauses can easily be automated, removing a lot of the boilerplate code currently present in the system. Interestingly, the semantics of the langauge is still based on first-order logic. Higher-order objects are not directly used, but instead referred to by a name. Their application is defined through `applyN`-clauses, allowing to use them like higher-order values.
Finally, they provide optimization techniques, identifying conditions when type information is necessary at runtime. In cases they are not fulfilled, e.g. the MO type system including the *head condition*, runtime type information might be removed, resulting in a faster runtime behaviour.
Mercury takes a different approach. Here, higher-order terms are a direct part of the language. In addition to type information, higher-order terms also include mode and determinism information. Calling a higher-order predicate can be done with the builtin goal `call/N`,

higher-order functions with the expression `apply/N`. Syntax sugar exists to allow easier usage. Assuming a higher-order predicate `P` of arity 1 and `X` a value with the correct type and instantiation, it is possible to simply write `P(X)` instead of the slightly longer `call(P, X)`. One restriction placed upon higher-order types is they can't be polymorphic. They may contain type variables, but those have to be quantified outside of the higher-order type, e.g. in the type of a polymorphic `map` predicate. [SHCO95; Jef02]

4.6 Type Classes

Type classes provide a way to use ad-hoc polymorphism in a controlled manner. They are used with great success in functional programming languages like *Haskell*. Logic programming and functional programming "are closely connected since both paradigms are trying to solve the same problems with techniques that are very similar" [Fer97, p.1]. Therefore, translating their use to logic programming seems natural.
Type classes are always defined in terms of one or more parameters, and usually define predicate or function signatures. A common extension is the ability to declare a default implementation, allowing the user to overwrite it with e.g. a more performant one if desired. Types are declared to belong to a class by providing an instance, where explicit implementations are provided for all predicates and functions without default implementation. Using the syntax of *Mercury*, an example for a type class *semigroup* and an instance for lists of arbitrary type is given below.
Example 14:

```
:- typeclass semigroup(A) where [
     append(A::in, A::in, A::out) is det
].
:- instance semigroup(list(A)) where [
     append([], Y, Y).
     append([H|X], Y, [H|Z]) :- append(X, Y, Z).
].
```

Their functions and predicates can be used either for an explicit instance, e.g. `list(int)`, or a type variable with a constraint (an instance must be an instance of the type class). In the latter case, fulfilling the constraint is then required by the user of the current predicate. All constraints on (the parameters of) a predicate are called its context. In example 15, the type class version of `q` can be used with any instance of `pclass`, forwarding the constraint to the user of `q` through the context `pclass(T)`.
Example 15:
For this very constructed example, let `p` be an overloaded predicate defined for both types `pred(int)` and `pred(bool)`. Defining a predicate `q` using it would require a copy-and-paste implementation to replicate the overloading of `int` and `bool` arguments. In addition, this has to be repeated for every type we later decide to overload `p` with.

```
:- pred p(int::in) is det.
p(_).
:- pred p(bool::in) is det.
p(_).

:- pred q(int::in) is det.
q(X) :- p(X).
:- pred q(bool::in) is det.
q(X) :- p(X).
```

However, if `p` is defined as part of a type class we can define `q` for all instances of the type class, even user-defined ones, without any need for error-prone source code copying.

```
:- typeclass pclass(T) where [
    pred p(A::in) is det
].
:- instance pclass(int) where [
    p(_).
].
:- instance pclass(bool) where [
    p(_).
].

:- pred q(T) <= pclass(T).
q(T) :- p(T).
```

Antonio Fernández in [Fer97] defines an extension of type classes for a prescriptive language supporting parametric polymorphism and the ability to declare their own algebraic data types. The MO type system supports all these requirements, meaning any extension can be further enhanced with type classes in the described fashion.
It is possible to define a hierarchy of type classes declaring one class a subclass of another. The class of ordered types `Ord/1` for example is a subclass of all types with a equality relation `Eq/1`. For a type σ, declaring an instance `Ord(`σ`)` is only possible, if an instance for `Eq(`σ`)` exists as well. Furthermore, if a predicate has an `Ord(A)`-constraint for a parameter `A`, it is possible to use the equality-predicate from the `Eq` typeclass for values with type `A` without explicitly specifying a `Eq(A)`-constraint. Some restrictions on subtype relations are mentioned, e.g. the hierarchy should not be cyclic; for details the reader is pointed to an earlier work [FR96].
The author doesn't specify any information about decidability of instance resolution. *Haskell* normally uses a set of rules to ensure termination [GHC21, ch.6.8.8.3], so it is likely this is necessary in logic programming as well.
Moreover, Fernández proposes a generalization to the class system to constructor classes. This would allow to define type classes not only for types, but also for type constructors. Their motivating example is a typeclass containing a `map/3` predicate (in *Haskell*, the corresponding class is known as *Functor*) which can't be typed to a satisfactory level without constructor classes. In the article, the concept is limited to type constructors with 1 parameter, although the authors intended to study n-parametric classes in the future.
Enrique Martin-Martin in [Mar11] focuses on the implementation of type classes in logic programming. They notice some difficulties with multi-use of non-deterministic predicates, if type classes are implemented using a dictionary-passing scheme known from functional programming languages like *Haskell.* The problem is shown in figure 1 of [Mar11] and the associated discussion (end of p.1 to p.2) where some expected solutions are missing. Instead, they propose an implementation based on type indices as an implicit (inserted by the compiler) argument. Not only do they solve the described problem, they also report an increase in efficiency compared to a dictionary-passing implementation.
Mercury uses type classes, described in [Jef02]. It supports a hierarchy of subclasses and is implemented using a dictionary passing scheme. Instances are only allowed, when all parameters are "either a type with no arguments, or a polymorphic type whose arguments are all type variables" [Mer21, ch.10.8]. This ensures uniqueness of instance declarations for (a sequence of) types, and (according to the rules specified in [GHC21, ch.6.8.8.3]) guarantees termination of instance resolution. Later versions of *Mercury* also support functional dependencies, where one type of an instance uniquely determines another type [Mer21, ch.10.8]. Consider the class of (monomorphic) `collections` providing a `member` predicate for example.

Here, the collection uniquely determines the element type which can be encoded with a functional dependency. A functional dependency might help with type reconstruction, allowing to omit some otherwise required type declarations.
Often times laws are associated with a typeclass, e.g. instances to an ordering typeclass should only be defined for types forming a total order. However, this is no requirement and has no impact on the addition of type classes to a type system.

4.7 Existential Types

Existential types provide a way to introduce abstract types into the language. In contrast to specifying an abstract type through a module interface, using an existential type allows "multiple implementations of a given abstract type, and most importantly allows the construction of heterogeneous collections of different implementations of the same abstract type". [Jef02, p.63]
According to Luca Cardelli and Peter Wegner, not all existential types are useful [CW85], e.g. the type $\exists A.A$ cannot be manipulated, outside of passing it around, so it can't be used for any computation. They need to be sufficiently structured to be useful, e.g. a pair of a type and a predicate for further manipulation. It is hard to imagine useful existential types in a type system without higher-order types. In the *Mercury* language, such a structure might alternatively be provided by a type class constraint. At construction of the existential type, the compiler ensures a valid instance of the type class is used. Thus, after deconstruction type class predicates can be used for further manipulation. Appendix A uses existential types to store a typed predicate identifier with its applied arguments.
David Jeffrey introduced existential types to the logic programming language *Mercury* [Jef02]. They noticed in contrast to most other types, construction and deconstruction of existential types when thought of as functions have different types. For construction we can use a value of any type, it is therefore function with a universally qualified type variable. On the other hand, deconstruction produces a value of unknown type. Accordingly, the result type of the function has to be an existentially qualified type variable. Therefore, they provide a syntactic distinction by annotating all constructions with an existentially qualified component with the keyword new. Deconstruction syntax is left unchanged.
Example 16:

```
:- type to_int ---> some[T] f(T, pred(T, int)).

:- pred construct(to_int::out) is det.
construct(X) :- X = 'new␣f'(
    42,
    pred(A::in, B::out) is det :- B = A + 12
).

:- pred evaluate(to_int, int).
:- mode evaluate(
    in(f(ground, pred(A::in, B::out) is det)),
    out
) is det.
evaluate(X, R) :- X = f(V, P), P(V, R).
```

Here, the definition of an existential type, allowing conversion to an integer, as well as example predicates for construction and deconstruction (with direct usage) are shown. All examples were originally defined in [Jef02, p.67] and extended to full predicates.
Functional programming allows deconstruction through pattern matching in some form of

let expression. The result type of the body of the let expression is not allowed to contain any existentially qualified type variable. The type is not allowed to “escape” the let expression. In logic programming, the value can be used in the whole body of the predicate. The existentially qualified type is allowed to “escape”, with type soundness preserved through careful handling of higher-order expressions. Multiple deconstructions of the same type for example produce different existentially qualified type variables as they may be different types, thus the resulting values can’t be unified (rejected with a type error). Example 17 showcases this, where even unification of two different deconstructions of the same value is rejected.
Example 17:

```
:- pred dual_deconstruction(to_int::in).
dual_deconstruction(X) :- X=f(A, _), X=f(B, _), A=B.
% produces a type error at A=B
```

The authors continue to explore the combined use of existential types and type classes. Not only does it increase the freedom of the programmer, they also show it is possible to use many useful object-oriented designs. Due to the popularity of object-oriented programming they are very well understood, so the ability to use them allows us to “stand on the shoulder of giants”.
One caveat for existential types is a small runtime overhead, since type information has to be present at runtime. However, in most cases the added flexibility is worth the trade, considering it is only required exactly when existential types are involved instead of all the time like in object-oriented programming languages.

4.8 The Mercury Type System

Mercury was mentioned in many earlier sections, since it includes a combination of the respective described features. Like many prescriptive systems, its type system is based on the MO type system. It uses a strong mode and determinism system, guiding execution order and providing a controlled way to incorporate side effects (e.g. IO) into the language. The latter is achieved through uniqueness modes, threading an IO-token through the body of a predicate. However, the instantiation checker is not fully implemented resulting in some surprising error messages.
Example 18:

```
:- type test(A, B) ---> test(A, B).
:- inst test(A, B) for test/2 ---> test(A, B).

:- pred use_ground(A::in) is det.
use_ground(_).

:- pred invalid_inst(test(test(A, B), C)).
:- mode invalid_inst(in(test(test(ground, free), I))) is det.
invalid_inst(test(test(A, _B), _C)) :- use_ground(A).
```

This code should be accepted, since `A` is guaranteed to be ground by the specified mode. However, in *Mercury* version 20.06.01 it is rejected. Variable `A` is reported to have instantiatedness `free`, even though it should be `ground`.
Mercury supports higher-order types (section 4.5), including special higher-order modes including mode and determinism information. While they might include type variables, they can’t be polymorphic. Any type variable appearing in a higher-order type must be qualified by an outer scope, e.g. the type of a predicate.

Type classes can be defined for one or more types and might be defined as a subclass of another type class. It is possible to define functional dependencies between the parameters of a type class. Instances for a type class can be defined for types with a known type constructor, where all parameters are a (potentially constraint) type variable, including types without parameters. Instance may for example be defined for `int` and `list(T)`, but not for `T` and `list(int)`.
Existentially qualified type variables may be used in predicates, functions and data types. They can be constraint by type classes, however it is not possible to use both existentially and universal type variables in the same constraint.
Example 19:

```
:- typeclass some_class(T) where [].
:- instance some_class(int) where [].
:- typeclass multi_class(S, T) where [].
:- instance multi_class(int, int) where [].

:- all[A] (some[B] pred split(A, B) => some_class(B))
                                    <= some_class(A).
:- mode split(in, in) is semidet.
split(_, 14).

% :- all[A] some[B] pred mixed(A, B) => multi_class(A, B).
% :- mode mixed(in, in) is semidet.
% mixed(_, 14).

:- some[B] pred mono(int, B) => multi_class(int, B).
:- mode mono(in, in) is semidet.
mono(_, 14).
```

The methods of the type classes are not important for this example, therefore they don't contain any. The type variable `A` is universally, `B` is existentially qualified. Universal quantification is the default, but was specified explicitly for clarity sake. The predicate `split/2` shows a predicate using both existential and universal type variables, both constraint by a type class. On the other hand, `mixed/2` would be rejected, because it tries to impose a constraint on existential and universal type variables at the same time. As showcased by predicate `mono/2`, it is possible for constraints to mention existentially quantified type variables and ground types. If all ground types of a program are known, in combination with overloading this can be used to simulate constraints between universal and existential type variables. However, this is neither an elegant solution, nor does it scale very well.
Although the described limitations in implementation are annoying at times, the language and its type system are quite flexible. An example is provided in appendix A, where an implementation for a simple print meta interpreter for ground formulas is presented. An proof-of-concept implementation to include free parameters in goals and predicates is easily possible. It wasn't included, since it would require a lot of boilerplate code without requiring, and therefore without showing more features of the type system.
Experimental support is available for solver types, e.g. integer ranges. They get their name from constraint solving e.g. used in *constraint logic programming.* Newer versions of *Mercury* also provide experimental support for subtypes. No restriction was placed on the arity of super- and subtype, but it is not possible to declare *conditional subtype relations* described in [Han91]. It is possible this leads to the decidability problems described in section 4.4. Since both are experimental, I didn't try either of them.

4.9 Dependent Types

Dependent types allow specifying types that depend on a value. A common example are length-indexed vectors where the length is part of the type, e.g. `vector(3)` for vectors with 3 elements.

In his work to introduce dependent types into logic Programming [Pfe92], Frank Pfenning starts defining a dependently typed λ-calculus. He starts with a similar example of array-types carrying their respective size, providing a type for an array multiplication function which can only be called with correctly sized arrays. However, normal "function type construction $\rightarrow$ is not expressive enough for this purpose, so [he] introduce[s] the new notation $\Pi x : A.B$ for the type of a function whose argument x may be needed in the description of the type B". [Pfe92, p.289] He calls types constructed this way *dependent function type*.

Dependent types may require checking the equivalence of objects at the type level, e.g. `vector(2+3)` and `vector(3+2)` as a result type of two different calls to an dependently typed `append` function both describe the same type. The author points out equality checking of expressions in languages with recursion (e.g. functional languages) is undecidable, implying an undecidable and therefore unacceptable type system. Considering the example above for example, type checking with dependent types is equivalent to statically checking array bounds, which is known to be undecidable. Nonetheless, he claims the problem can be avoided in logic programming.

He continues by introducing the *LF logical framework* for dependent types in logic programming. It is based on encoding derivations of predicates in the proof calculus of natural deduction. Its methodology is described as adhering to the *judgments-as-types* principle. Afterwards, he presents an implementation of this framework with the programming language *Elf*.

Pfenning notes, unification in *Elf* is undecidable in the general case of the underlying data domain. His solution is to solve all "obvious" unification problems directly and leave everything else as *constraints* to be solved in a later step. In addition, *type reconstruction* is also undecidable. Again, he resorts to the constraint solving algorithm, using the fact type reconstruction can be reduced to unification. The author states in their experience, this works well in practice and is "not a bottle-neck of the system". [Pfe92, p.309]

Even though an implementation of the framework is shown, due to its nature of being the first to introduce dependent types to logic programming it is a very explorative and theoretical work.

In a later work, Frank Pfenning and Carsten Schürmann present a successor called *Twelf*, again based on the LF framework. It is described as "a tool for experimentation in the theory of programming languages and logics" [PS99, p.1]. The work provides a high-level description of the capabilities and some success stories of *Twelf*, showcasing the possibilities of a language using its type system.

CHAPTER 5

Unexplored Type Systems Features

Many forms of polymorphisms have been explored for logic programming. However, a typed language will always reject some algorithms possible in an untyped language. There will always be a desire for more flexible type systems, allowing to use these algorithms without giving up the benefits of a statically typed language.
On the other hand, since types also serve as a form of specification there is a need for very precise types, ultimately culminating in dependent types (see section 4.9). Type checking for dependent types often becomes undecidable, therefore the goal is usually to provide as precise types as possible while keeping type checking, and ideally type inference decidable.
Both desires can be summarised as a yearning for more type system features. This section mentions some features which to my knowledge haven't been explored in the context of logic programming.
Luca Cardelli and Peter Wegner present a functional language *Fun* in [CW85], with the purpose of discussing different forms of polymorphism. Interestingly, from their hierarchy in their Figure 2 [CW85, p.516] only *bounded existential* and *bounded quantifier*, i.e. the combination of existential types and subtypes, weren't explored for logic programming. It is possible, this would result in the same decidability problem as the combination of subtypes and parametric polymorphism does, possibly allowing similar solutions by restricting valid subtype relations.
Additionally, the combination of subtypes and type classes needs careful thought about allowed instances and instance resolution. Should subtypes only be allowed to define an instance to a type class if no supertype defines one, should multiple instances in the subtype hierarchy result in dynamic dispatch like it is used in object-oriented programming languages, or is another solution desirable? A type system including both must find an answer to this, and possibly other questions.
With regards to higher-order programming, so far only monomorphic types where explored. A convenient extension allows polymorphic higher-order types as well, a feature known as *RankNTypes* in *Haskell* [GHC21].
Another feature available in *Haskell* are generalized algebraic data types (GADT). They allow restricting constructors of a parametric type to a specific instance of the type.
Example 20:
Translating the example in the GHC user guide [GHC21], GADTs allow implementing a deterministic and well-typed *eval* predicate for the terms of integer expressions with zero-checking. The syntax for GADTs mimics those of type classes and functions in *Mercury*.

```
:- type term(A) where [
    lit: (int)=term(int),
    succ: (term(int))=term(int),
    is_zero: (term(int))=term(bool),
    if: (term(bool), term(A), term(A))=term(A)
].

:- pred eval(term(A)::in, A::out) is det.
eval(lit(I), I).
eval(succ(T), R) :- eval(T, P), R = P + 1.
eval(is_zero(T), R) :- (if eval(T, 0) then R=yes else R=no).
eval(if(yes, e1, _e2), R) :- eval(e1, R).
eval(if(no, _e1, e2), R) :- eval(e2, R).

:- pred is_if(term(bool), bool) is det.
is_if(is_zero(_T), no).
is_if(if(_b, _e1, _e2), yes).
```

A predicate for `term(bool)` then can only define clauses for `is_zero` and `if`. As a consequence, the `is_if` predicate defined above is deterministic; additional clauses for `lit` and `succ` would be ill-typed. Additionally, when pattern matching on e.g. the constructor `succ`, on the right hand side of the equation the type parameter `A` is known to be instantiated with the type `int`. Translated to logic programming, if `succ(T)` is used in the head of a predicate, we know in the body of the predicate `T` has type `term(int)`. This allows `R = P + 1` to be well-typed, since at this point both `R` and `P` are known to have type `int`, even though the `eval`-predicate is defined for terms with universally quantified parameter.
One can argue the same effect can be achieved by using several types. However, this would require a significant amount code duplication and boilerplate code. Interestingly, the example above is similar to those discussing the *head condition* (subsection 4.1.1). GADTs might provide a controlled way to achieve a similar effect like rejecting the *head condition*, without the associated repercussions.
It would be interesting to see for both concepts, whether the implementation can be translated to logic programming, possibly using similar techniques used when translating the features described in chapters 3 and 4.

CHAPTER 6

Conclusion

This Thesis has presented various type system features, together with type systems for logic programming implementing the respective feature. Overall, many forms of polymorphism have been explored in the context of logic programming.
Descriptive type systems approximate the meaning of formerly untyped programs with types as supersets of the success set of predicates. Their main success story lies in identifying unsatisfiable clauses. However, their use beyond that remains limited, since the inferred type often doesn't match the intend of the developer. [Nai92]
On the other hand, prescriptive type systems include types as part of the language, rejecting some ill-typed programs as meaningless. In logic programming, they are usually an extension to the type system introduced by Mycroft and O'Keefe [MO84] which is a translation of the ML type system [Mil78] featuring parametric polymorphism. Interestingly, in contrast to functional programming which often served as an inspiration many features require type information to be present at runtime.
Most works are of a theoretical nature, possibly providing algorithms required for an implementation and discussing their properties. Nonetheless, some implementations already exist in a quite usable state, even though all of them would benefit from additional developer time. Special attention was brought to the language *Mercury*. It has a flexible type system, allowing the use of many patterns known from *Prolog* or even other programming paradigms. Although the implementation is incomplete in some parts, it is possible to work around these holes in the implementation, resulting in a overall pleasant development experience.

APPENDIX A

A Mercury Meta Interpreter

In this appendix, a meta interpreter for ground *Mercury* programs, printing every evaluation step is presented. It doesn't short-circuit, always evaluating all clauses of a predicate, as well as all sub-formulas of disjunctions and conjunctions. The implementation should be regarded as a proof of concept, showcasing the possibility without worrying about an elegant implementation. It currently only supports predicates up to arity 2, with `int` and `string` as only allowed types for arguments. An extension to more argument types and predicates with higher arity is purely mechanical, but gets tedious very fast. An improved implementation might be easier to extent in that regard.
Allowing non-ground arguments is possible by extending the available abstract syntax tree. However, early experiments showed it requires a lot of extra code without providing much additional insight about the type system. All required features were already used in the current implementation. An extension was therefore omitted.

```
:- module program.

:- interface.

:- use_module io.
:- pred main(io.state::di, io.state::uo) is det.

:- implementation.

:- import_module map, list, bool, int, uint, string.
:- use_module exception.

% name types
:- type ident_nullary ---> ident_nullary(string).
:- type ident_unary(A) ---> ident_unary(string).
:- type ident_binary(A, B) ---> ident_binary(string).

:- typeclass ident(A) where [
    func name(A)=string,
    func ident(string)=A
```

```
 ].
:- instance ident(ident_nullary) where [
    name(ident_nullary(Name)) = Name,
    ident(Name) = ident_nullary(Name)
 ].
:- instance ident(ident_unary(A)) where [
    name(ident_unary(Name)) = Name,
    ident(Name) = ident_unary(Name)
 ].
:- instance ident(ident_binary(A, B)) where [
    name(ident_binary(Name)) = Name,
    ident(Name) = ident_binary(Name)
 ].

% type classes as a collection of requirements for argument types
:- typeclass argument(A)
   <= (
        unary_store(program, A), binary_store(program, A, A),
        binary_store(program, A, int),
        binary_store(program, int, A),
        binary_store(program, A, string),
        binary_store(program, string, A)
    ) where [].
:- instance argument(int) where [].
:- instance argument(string) where [].

% atomic formulas
% I would prefer to use a type parameter instead of program,
% but type constraints on existential types can only mention
% existentially qualified type parameters or ground types
:- type atom --->
    nullary(ident_nullary);
    some[A] unary(ident_unary(A), A) => argument(A);
    some[A, B] binary(ident_binary(A, B), A, B)
        => (
            argument(A), argument(B),
            binary_store(program, A, A),
            binary_store(program, B, B),
            binary_store(program, A, B),
            binary_store(program, B, A)
        ).

% goal type
:- type goal --->
    conjunction(goal, goal);
    disjunction(goal, goal);
    negation(goal);
    atomic(atom).

% types with one typed free variable (unary abstraction)
:- type var0(A) ---> var0.
:- type fv_atom(Var) --->
```

```
    fv_nullary(ident_nullary);
    some[A] fv_unary(ident_unary(A), A) => argument(A);
    fv_unary_var(ident_unary(Var), var0(Var));
    some[A, B] fv_binary(ident_binary(A, B), A, B)
        => (
            argument(A), argument(B),
            binary_store(program, A, A),
            binary_store(program, B, B),
            binary_store(program, A, B),
            binary_store(program, B, A)
        );
    % constraints only possible on existentially qualified
    % type variables and ground types
    % use explicit types instead,
    % put constraints on apply_unary function instead
    % some[B] fv_binary_var_fst(ident_binary(Var, B), Var, B)
    %     => binary_store(program, Var, B);
    fv_binary_var_int(ident_binary(Var, int), var0(Var), int);
    fv_binary_var_string(
        ident_binary(Var, string),
        var0(Var),
        string
    );
    % some[A] fv_binary_var_snd(ident_binary(Var, A), A, Var)
    %     => binary_store(program, A, Var);
    fv_binary_int_var(ident_binary(int, Var), int, var0(Var));
    fv_binary_string_var(
        ident_binary(string, Var),
        string,
        var0(Var)
    );
    fv_binary_var_both(
        ident_binary(Var, Var),
        var0(Var),
        var0(Var)
    ).

% types with one free type variable (unary abstraction)
:- type fv_goal(Var) --->
    fv_conjunction(fv_goal(Var), fv_goal(Var));
    fv_disjunction(fv_goal(Var), fv_goal(Var));
    fv_negation(fv_goal(Var));
    fv_atomic(fv_atom(Var)).

:- func apply_unary(fv_goal(A), A)=goal is det <= argument(A).
apply_unary(fv_conjunction(Goal0, Goal1), A)
    = conjunction(apply_unary(Goal0, A), apply_unary(Goal1, A)).
apply_unary(fv_disjunction(Goal0, Goal1), A)
    = disjunction(apply_unary(Goal0, A), apply_unary(Goal1, A)).
apply_unary(fv_negation(Goal), A)
    = negation(apply_unary(Goal, A)).
apply_unary(fv_atomic(fv_nullary(Nullary)), _A)
```

```
    = atomic(nullary(Nullary)).
apply_unary(fv_atomic(fv_unary(Unary, X)), _A)
    = atomic('new␣unary'(Unary, X)).
apply_unary(fv_atomic(fv_unary_var(Unary, var0)), A)
    = atomic('new␣unary'(Unary, A)).
apply_unary(fv_atomic(fv_binary(Binary, X, Y)), _A)
    = atomic('new␣binary'(Binary, X, Y)).
% apply_unary(fv_atomic(fv_binary_var_fst(Binary, var0, B)), A)
%     = atomic('new binary'(Binary, A, B)).
apply_unary(fv_atomic(fv_binary_var_int(Binary, var0, B)), A)
    = atomic('new␣binary'(Binary, A, B)).
apply_unary(fv_atomic(fv_binary_var_string(Binary, var0, B)), A)
    = atomic('new␣binary'(Binary, A, B)).
% apply_unary(fv_atomic(fv_binary_var_snd(Binary, A, var0)), B)
%     = atomic('new binary'(Binary, A, B)).
apply_unary(fv_atomic(fv_binary_int_var(Binary, A, var0)), B)
    = atomic('new␣binary'(Binary, A, B)).
apply_unary(fv_atomic(fv_binary_string_var(Binary, A, var0)), B)
    = atomic('new␣binary'(Binary, A, B)).
apply_unary(fv_atomic(fv_binary_var_both(Binary, var0, var0)), A)
    = atomic('new␣binary'(Binary, A, A)).

% types with up to 2 free variables (binary abstraction)
:- type var1(A) ---> var1.
:- type fv2_atom(Var0, Var1)--->
    fv2_nullary(ident_nullary);
    some[A] fv2_unary(ident_unary(A), A) => argument(A);
    fv2_unary_var0(ident_unary(Var0), var0(Var0));
    fv2_unary_var1(ident_unary(Var1), var1(Var1));
    some[A, B] fv2_binary(ident_binary(A, B), A, B)
        => (
            argument(A), argument(B),
            binary_store(program, A, A),
            binary_store(program, B, B),
            binary_store(program, A, B),
            binary_store(program, B, A)
        );
    % constraints only possible on existentially qualified
    % type variables and ground types
    % some[B] fv2_binary_var0_fst(
    %     ident_binary(Var0, B),
    %     var0(Var0),
    %     B
    % ) => binary_store(program, Var0, B);
    fv2_binary_var0_int(ident_binary(Var0, int), var0(Var0), int);
    fv2_binary_var0_string(
        ident_binary(Var0, string),
        var0(Var0),
        string
    );
    % some[A] fv2_binary_var0_snd(
    %     ident_binary(Var0, A),
    %       A,
    %       var0(Var0)
    % ) => binary_store(program, A, Var0);
    fv2_binary_int_var0(ident_binary(int, Var0), int, var0(Var0));
    fv2_binary_string_var0(
        ident_binary(string, Var0),
        string,
        var0(Var0)
    );
    fv2_binary_var0_both(
        ident_binary(Var0, Var0),
        var0(Var0),
        var0(Var0)
    );
    % some[B] fv2_binary_var1_fst(
    %       ident_binary(Var1, B),
    %       var1(Var1),
    %       B
    % ) => binary_store(program, Var1, B);
    % some[A] fv2_binary_var1_snd(
    %       ident_binary(Var1, A),
    %       A,
    %       var1(Var1)
    % ) => (binary_store(program, A, Var1);
    fv2_binary_var1_int(ident_binary(Var1, int), var1(Var1), int);
    fv2_binary_var1_string(
        ident_binary(Var1, string),
        var1(Var1),
        string
    );
    fv2_binary_int_var1(ident_binary(int, Var1), int, var1(Var1));
    fv2_binary_string_var1(
        ident_binary(string, Var1),
        string,
        var1(Var1)
    );
    fv2_binary_var1_both(
        ident_binary(Var1, Var1),
        var1(Var1),
        var1(Var1)
    );
    fv2_binary_var0_var1(
        ident_binary(Var0, Var1),
        var0(Var0),
        var1(Var1)
    );
    fv2_binary_var1_var0(
        ident_binary(Var1, Var0),
        var1(Var1),
        var0(Var0)
    ).
```

```
:- type fv2_goal(Var0, Var1) --->
    fv2_conjunction(fv2_goal(Var0, Var1), fv2_goal(Var0, Var1));
    fv2_disjunction(fv2_goal(Var0, Var1), fv2_goal(Var0, Var1));
    fv2_negation(fv2_goal(Var0, Var1));
    fv2_atomic(fv2_atom(Var0, Var1)).

:- func apply_binary(fv2_goal(A, B), A, B)=goal is det
    <= (
        argument(A), argument(B),
        binary_store(program, A, A), binary_store(program, B, B),
        binary_store(program, A, B), binary_store(program, B, A)
    ).
apply_binary(fv2_conjunction(Goal0, Goal1), A, B)
    = conjunction(
        apply_binary(Goal0, A, B),
        apply_binary(Goal1, A, B)
    ).
apply_binary(fv2_disjunction(Goal0, Goal1), A, B)
    = disjunction(
        apply_binary(Goal0, A, B),
        apply_binary(Goal1, A, B)
    ).
apply_binary(fv2_negation(Goal), A, B)
    = negation(apply_binary(Goal, A, B)).
apply_binary(fv2_atomic(fv2_nullary(Nullary)), _A, _B)
    = atomic(nullary(Nullary)).
apply_binary(fv2_atomic(fv2_unary(Unary, X)), _A, _B)
    = atomic('new␣unary'(Unary, X)).
apply_binary(fv2_atomic(fv2_unary_var0(Unary, var0)), A, _B)
    = atomic('new␣unary'(Unary, A)).
apply_binary(fv2_atomic(fv2_unary_var1(Unary, var1)), _A, B)
    = atomic('new␣unary'(Unary, B)).
apply_binary(fv2_atomic(fv2_binary(Binary, X, Y)), _A, _B)
    = atomic('new␣binary'(Binary, X, Y)).
% apply_binary(
%     fv2_atomic(fv2_binary_var0_fst(Binary, var0, B)),
%     A,
%     _B
% ) = atomic('new binary'(Binary, A, B)).
apply_binary(
    fv2_atomic(fv2_binary_var0_int(Binary, var0, Y)),
    X,
    _B
) = atomic('new␣binary'(Binary, X, Y)).
apply_binary(
    fv2_atomic(fv2_binary_var0_string(Binary, var0, Y)),
    X,
    _B
) = atomic('new␣binary'(Binary, X, Y)).
% apply_binary(
%     fv2_atomic(fv2_binary_var0_snd(Binary, A, var0)),
%     Y,
%       _B
% ) = atomic('new binary'(Binary, X, Y)).
apply_binary(
     fv2_atomic(fv2_binary_int_var0(Binary, X, var0)),
     Y,
     _B
) = atomic('new␣binary'(Binary, X, Y)).
apply_binary(
     fv2_atomic(fv2_binary_string_var0(Binary, X, var0)),
     Y,
     _B
) = atomic('new␣binary'(Binary, X, Y)).
apply_binary(
     fv2_atomic(fv2_binary_var0_both(Binary, var0, var0)),
     A,
     _B
) = atomic('new␣binary'(Binary, A, A)).
% apply_binary(
%      fv2_atomic(fv2_binary_var1_fst(Binary, var1, Y)),
%      _A,
%      X
% ) = atomic('new binary'(Binary, X, Y)).
apply_binary(
     fv2_atomic(fv2_binary_var1_int(Binary, var1, Y)),
     _A,
     X
) = atomic('new␣binary'(Binary, X, Y)).
apply_binary(
     fv2_atomic(fv2_binary_var1_string(Binary, var1, Y)),
     _A,
     X
) = atomic('new␣binary'(Binary, X, Y)).
% apply_binary(
%     fv2_atomic(fv2_binary_var1_snd(Binary, A, var1)),
%     _A,
%     B
% ) = atomic('new binary'(Binary, A, B)).
apply_binary(
     fv2_atomic(fv2_binary_int_var1(Binary, X, var1)),
     _A,
     Y
) = atomic('new␣binary'(Binary, X, Y)).
apply_binary(
     fv2_atomic(fv2_binary_string_var1(Binary, X, var1)),
     _A,
     Y
) = atomic('new␣binary'(Binary, X, Y)).
apply_binary(
     fv2_atomic(fv2_binary_var1_both(Binary, var1, var1)),
     _A,
     B
) = atomic('new␣binary'(Binary, B, B)).
```

```
apply_binary(
    fv2_atomic(fv2_binary_var0_var1(Binary, var0, var1)),
    A,
    B
) = atomic('new␣binary'(Binary, A, B)).
apply_binary(
    fv2_atomic(fv2_binary_var1_var0(Binary, var1, var0)),
    A,
    B
) = atomic('new␣binary'(Binary, B, A)).

% program (i.e. all clauses)
:- type program --->
    program(
        nullary::map(ident_nullary, list(goal)),
        unary_int::map(ident_unary(int), list(fv_goal(int))),
        unary_string
            ::map(ident_unary(string), list(fv_goal(string))),
        binary_int_int::map(
            ident_binary(int, int),
            list(fv2_goal(int, int))
        ),
        binary_string_string::map(
            ident_binary(string, string),
            list(fv2_goal(string, string))
        ),
        binary_string_int::map(
            ident_binary(string, int),
            list(fv2_goal(string, int))
        ),
        binary_int_string::map(
            ident_binary(int, string),
            list(fv2_goal(int, string))
        )
    ).

% custom clauses, failure means no custom clause
:- typeclass nullary_store(Program) where [
    func get_nullary(Program, ident_nullary)=list(goal) is semidet
].
:- typeclass unary_store(Program, A) <= builtin_unary(A) where [
    func get_unary(Program, ident_unary(A))
        =list(fv_goal(A)) is semidet
].
:- typeclass binary_store(Program, A, B)
        <= builtin_binary(A, B) where [
    func get_binary(Program, ident_binary(A, B))
        =list(fv2_goal(A, B)) is semidet
].

:- instance nullary_store(program) where [
    get_nullary(Program, Name) = search(nullary(Program), Name)
```

```
].

:- instance unary_store(program, int) where [
    get_unary(Program, Name) = search(unary_int(Program), Name)
].
:- instance unary_store(program, string) where [
    get_unary(Program, Name) = search(unary_string(Program), Name)
].

:- instance binary_store(program, int, int) where [
    get_binary(Program, Name)
        = search(binary_int_int(Program), Name)
].
:- instance binary_store(program, string, string) where [
    get_binary(Program, Name)
        = search(binary_string_string(Program), Name)
].
:- instance binary_store(program, string, int) where [
    get_binary(Program, Name)
        = search(binary_string_int(Program), Name)
].
:- instance binary_store(program, int, string) where [
    get_binary(Program, Name)
        = search(binary_int_string(Program), Name)
].

% builtin clauses
% if the predicate fails, no predicate of the name,
% arity and type was configured,
% otherwise it always succeeds,
% the result is returned in the bool-argument.
:- pred builtin_nullary(ident_nullary::in, bool::out) is semidet.
builtin_nullary(ident_nullary("true"), R)
    :- P=((pred) is semidet :- true), R=pred_to_bool(P).
builtin_nullary(ident_nullary("false"), R)
    :- P=((pred) is semidet :- false), R=pred_to_bool(P).
builtin_nullary(ident_nullary("fail"), R)
    :- P=((pred) is semidet :- fail), R=pred_to_bool(P).

:- typeclass builtin_unary(A) where [
    pred builtin_unary(ident_unary(A)::in, A::in, bool::out)
        is semidet
].
:- instance builtin_unary(int) where [
    builtin_unary(ident_unary("even"), N, R)
        :- R=pred_to_bool(even(N)),
    builtin_unary(ident_unary("odd"), N, R)
        :- R=pred_to_bool(even(N))
].
:- instance builtin_unary(string) where [
    builtin_unary(ident_unary("is_empty"), S, R)
        :- R=pred_to_bool(is_empty(S))
```

```
].

:- typeclass builtin_binary(A, B) where [
    pred builtin_binary(ident_binary(A, B), A, B, bool),
    mode builtin_binary(in, in, in, out) is semidet
].
:- instance builtin_binary(int, int) where [
    builtin_binary(ident_binary("="), A, B, R)
        :- R=pred_to_bool((pred) is semidet :- A=B),
    builtin_binary(ident_binary(">"), A, B, R)
        :- R=pred_to_bool((pred) is semidet :- A>B)
].
:- instance builtin_binary(string, string) where [
    builtin_binary(ident_binary("="), A, B, R)
        :- R=pred_to_bool((pred) is semidet :- A=B),
    builtin_binary(ident_binary("suffix"), A, B, R)
        :- R=pred_to_bool((pred) is semidet :- suffix(A, B))
].
:- instance builtin_binary(string, int) where [
    builtin_binary(ident_binary("length"), A, B, R)
        :- R=pred_to_bool((pred) is semidet :- length(A, B))
].
:- instance builtin_binary(int, string) where [
    builtin_binary(ident_binary(_Ident), _A, _B, _R) :- fail
].

% trace meta interpreter
% unknown identifiers (neither in program, nor configured builtin)
% throw a string-exception
% result is stored as a bool,
% to allow running IO (det or cc_multi required)
:- pred demo_trace(goal, program, bool, io.state, io.state).
:- mode demo_trace(in, in, out, di, uo) is det.
demo_trace(Goal, Program, Result, !IO)
    :- demo_trace(Goal, Program, Result, 0u, !IO).

:- pred demo_trace(goal, program, bool, uint, io.state, io.state).
:- mode demo_trace(in, in, out, in, di, uo) is det.
demo_trace(conjunction(A, B), Program, Result, Depth, !IO)
    :- io.write_string(nl_identation(Depth) ++ "AND(", !IO),
        demo_trace(A, Program, RA, Depth + 1u, !IO),
        io.write_string(nl_identation(Depth) ++ ",", !IO),
        demo_trace(B, Program, RB, Depth + 1u, !IO),
        Result=and(RA, RB),
        io.write_string(
            nl_identation(Depth) ++ ")␣=␣" ++ string(Result),
            !IO
        ).
demo_trace(disjunction(A, B), Program, Result, Depth, !IO)
    % this meta-interpreter will NOT shortcut on success of A!
    :- io.write_string(nl_identation(Depth) ++ "OR(", !IO),
        demo_trace(A, Program, RA, Depth + 1u, !IO),
        io.write_string(nl_identation(Depth) ++ ",", !IO),
        demo_trace(B, Program, RB, Depth + 1u, !IO),
        Result=or(RA, RB),
        io.write_string(
            nl_identation(Depth) ++ ")␣=␣" ++ string(Result),
            !IO
        ).
demo_trace(negation(Goal), Program, Result, Depth, !IO)
    :- io.write_string(nl_identation(Depth) ++ "NOT(", !IO),
        demo_trace(Goal, Program, R, Depth + 1u, !IO),
        Result=not(R),
        io.write_string(
            nl_identation(Depth) ++ ")␣=␣" ++ string(Result),
            !IO
        ).
demo_trace(atomic(nullary(Nullary)), Program, Result, Depth, !IO)
    :- if Goals=get_nullary(Program, Nullary)
        then
            io.write_string(
                nl_identation(Depth) ++ name(Nullary) ++ "␣:-␣[",
                !IO
            ),
            demo_trace_list(
                Goals,
                Program,
                Result,
                Depth + 1u,
                !IO
            ),
            io.write_string(
                nl_identation(Depth) ++ "]␣=␣" ++ string(Result),
                !IO
            )
    else if builtin_nullary(Nullary, R)
    then
        R=Result,
        Msg = "builtin␣" ++ name(Nullary)
            ++ "␣=␣" ++ string(Result),
        io.write_string(nl_identation(Depth) ++ Msg, !IO)
    else
        exception.throw(
            "Unknown␣nullary␣predicate:␣" ++ name(Nullary)
        ).
demo_trace(atomic(unary(Unary, A)), Program, Result, Depth, !IO)
    :- Description = name(Unary) ++ "(" ++ string(A) ++ ")",
    (if Fv_Goals=get_unary(Program, Unary)
    then
        io.write_string(
            nl_identation(Depth) ++ Description ++ "␣:-␣[",
            !IO
        ),
        Goals = map(
            func(Fv_Goal)=apply_unary(Fv_Goal, A),
            Fv_Goals
        ),
        demo_trace_list(
            Goals,
            Program,
            Result,
            Depth + 1u,
            !IO
        ),
        io.write_string(
            nl_identation(Depth) ++ "]␣=␣" ++ string(Result),
            !IO
        )
    else if builtin_unary(Unary, A, R)
        then
            R=Result,
            Msg = "builtin␣" ++ Description
                ++ "␣=␣" ++ string(Result),
            io.write_string(nl_identation(Depth) ++ Msg, !IO)
    else
        exception.throw(
            "Unknown␣unary␣predicate:␣" ++ name(Unary)
        )
    ).
demo_trace(
    atomic(binary(Binary, A, B)),
    Program,
    Result,
    Depth,
    !IO
) :- Description = name(Binary) ++ "(" ++ string(A)
        ++ ",␣" ++ string(B) ++ ")",
    (if Fv2_Goals=get_binary(Program, Binary)
    then
        io.write_string(
            nl_identation(Depth) ++ Description ++ "␣:-␣[",
            !IO
        ),
        Goals=map(
            func(Fv2_Goal)=apply_binary(Fv2_Goal, A, B),
            Fv2_Goals
        ),
        demo_trace_list(
            Goals,
            Program,
            Result,
            Depth + 1u,
            !IO
        ),
        io.write_string(
            nl_identation(Depth) ++ "]␣=␣" ++ string(Result),
            !IO
        )
    else if builtin_binary(Binary, A, B, R)
        then
            R=Result,
            Msg = "builtin␣" ++ Description
                ++"␣=␣" ++ string(Result),
            io.write_string(nl_identation(Depth) ++ Msg, !IO)
    else
        exception.throw(
            "Unknown␣billary␣predicate:␣" ++ name(Binary)
        )
    ).

:- pred demo_trace_list(
    list(goal)::in,
    program::in,
    bool::out,
    uint::in,
    io.state::di,
    io.state::uo
) is det.
demo_trace_list([], _Program, Result, _Depth, !IO) :- Result=no.
demo_trace_list([H | T], Program, Result, Depth, !IO) :-
    demo_trace(H, Program, RH, Depth, !IO),
    io.write_string(",", !IO),
    demo_trace_list(T, Program, RT, Depth, !IO),
    Result = or(RH, RT).

:- func identation(uint, string)=string is det.
identation(N, Start) = (
    if N = 0u
    then Start
    else identation(N - 1u, Start) ++ "␣␣"
).

:- func nl_identation(uint)=string is det.
nl_identation(N) = identation(N, "\n").

% :- pred main(io.state::di, io.state::uo) is det.
main(!IO) :-
    nullaries(Nullaries),
    unary_ints(UnaryInts),
    Program = program(
        Nullaries,
        UnaryInts,
        init,
        init,
        init,
        init,
        init
    ),
    Query = atomic(nullary(ident("p0"))),
    demo_trace(Query, Program, Result, !IO),
    io.write_string("\nResult␣=␣" ++ string(Result) ++ "\n", !IO).

:- pred nullaries(map(ident_nullary, list(goal))::out) is det.
nullaries(Map) :-
    P0 = [
            atomic(nullary(ident("fail"))),
            disjunction(
                atomic(nullary(ident("false"))),
                atomic('new␣unary'(ident("q1"), 3))
            ),
            negation(atomic(nullary(ident("true"))))
    ],
    Map = set(init, ident("p0"), P0).

:- pred unary_ints(map(ident_unary(int), list(fv_goal(int)))).
:- mode unary_ints(out) is det.
unary_ints(Map) :-
    Q1 = [
            fv_atomic(fv_binary_var_int(ident(">"), var0, 5)),
            fv_atomic('new␣fv_binary'(
                ident("="),
                "Hello,␣world!",
                "Hello,␣world!"
            ))
    ],
    Map = set(init, ident("q1"), Q1).
```

Bibliography

[BG94] Chitta Baral and Michael Gelfond, “Logic programming and knowledge representation”, *The Journal of Logic Programming*, vol. 19-20, pp. 73–148, 1994, Special Issue: Ten Years of Logic Programming, ISSN: 0743-1066. DOI: `https://doi.org/10.1016/0743-1066(94)90025-6`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/0743106694900256`.

[Chu40] Alonzo Church, “A formulation of the simple theory of types”, *The journal of symbolic logic*, vol. 5, no. 2, pp. 56–68, 1940.

[CW85] Luca Cardelli and Peter Wegner, “On understanding types, data abstraction, and polymorphism”, *ACM Computing Surveys (CSUR)*, vol. 17, no. 4, pp. 471–523, 1985.

[DH88] Roland Dietrich and Frank Hagl, “A polymorphic type system with subtypes for prolog”, in *ESOP ’88*, H. Ganzinger, Ed., Springer, Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 79–93, ISBN: 978-3-540-38941-5.

[DS01] Pierre Deransart and Jan-Georg Smaus, “Well-typed logic programs are not wrong”, in *Functional and Logic Programming*, Herbert Kuchen and Kazunori Ueda, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 280–295, ISBN: 978-3-540-44716-0.

[DZ92] Philip W. Dart and Justin Zobel, “A regular type language for logic programs”, in *Types in Logic Programming*, MIT Press, 1992, pp. 157–187.

[FC01] François Fages and Emmanuel Coquery, “Typing constraint logic programs”, *Theory and Practice of Logic Programming*, vol. 1, no. 6, pp. 751–777, 2001. DOI: `10.1017/S1471068401001120`.

[Fer97] Antonio Fernández-Leiva, “A type classes system for logic programming”, 1997.

[FM88] You-Chin Fuh and Prateek Mishra, “Type inference with subtypes”, in *European Symposium on Programming*, Springer, 1988, pp. 94–114.

[FR96] Antonio Fernández-Leiva and Blas C. Ruiz, “Un sistema de tipos bidireccional”, *Infome Técnico, Dpto. Lenguajes y Ciencias de la Computaci´on, Universidad de Málaga*, 1996.

[GHC21] GHC Team. “Ghc user’s guide, version 9.2.1”. (29th Nov. 2021), [Online]. Available: `https://downloads.haskell.org/~ghc/9.2.1/docs/html/users_guide/index.html`.

[GM84] Joseph A. Goguen and José Meseguer, "Equality, types, modules, and (why not?) generics for logic programming", *The Journal of Logic Programming*, vol. 1, no. 2, pp. 179–210, 1984, ISSN: 0743-1066. DOI: `https://doi.org/10.1016/0743-1066(84)90004-9`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/0743106684900049`.

[Han89a] Michael Hanus, "Horn clause programs with polymorphic types: Semantics and resolution", in *TAPSOFT '89*, J. Díaz and F. Orejas, Eds., Springer, Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 225–240, ISBN: 978-3-540-46118-0.

[Han89b] Michael Hanus, "Polymorphic higher-order programming in prolog", in *Proceedings of the Sixth International Conference on Logic Programming*, MIT Press Cambridge MA, 1989, pp. 382–397.

[Han91] Michael Hanus, "Parametric order-sorted types in logic programming", in *International Joint Conference on Theory and Practice of Software Development*, Springer, 1991, pp. 181–200.

[HJ92] Nevin Heintze and Joxan Jaffar, "Semantic types for logic programs", in *Types in Logic Programming*, MIT Press, 1992, pp. 141–155.

[HT92] Patricia M Hill and Rodney W Topor, "A semantics for typed logic programs.", in *Types in Logic Programming*, MIT Press, 1992, pp. 1–62.

[Jac92] Dean Jacobs, "A pragmatic view of types for logic program", in *Types in Logic Programming*, MIT Press, 1992, pp. 217–227.

[Jef02] David Jeffery, *Expressive type systems for logic programming languages*, 2002.

[Jon95] Mark P Jones, "A system of constructor classes: Overloading and implicit higher-order polymorphism", *Journal of functional programming*, vol. 5, no. 1, pp. 1–35, 1995.

[Llo87] John W. Lloyd, *Foundations of Logic Programming, 2nd Edition*. Springer, 1987, ISBN: 3-540-18199-7. DOI: `10.1007/978-3-642-83189-8`. [Online]. Available: `https://doi.org/10.1007/978-3-642-83189-8`.

[LR91] T. K. Lakshman and Uday S. Reddy, "Typed prolog: A semantic reconstruction of the mycroft-o'keefe type system.", in *ISLP*, vol. 91, 1991, pp. 202–217.

[LR96] Pascale Louvet and Olivier Ridoux, "Parametric polymorphism for typed prolog and λprolog", in *Programming Languages: Implementations, Logics, and Programs*, Herbert Kuchen and S. Doaitse Swierstra, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 47–61, ISBN: 978-3-540-70654-0.

[Mar11] Enrique Martin-Martin, "Type classes in functional logic programming", in *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, 2011, pp. 121–130.

[Mer21] The Mercury team. "The mercury language reference manual, version 20.06.01". (29th Nov. 2021), [Online]. Available: `https://mercurylang.org/information/doc-release/mercury_ref/index.html`.

[Mil78] Robin Milner, "A theory of type polymorphism in programming", *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[Mis84] Prateek Mishra, "Towards a theory of types in prolog", in *SLP*, 1984, pp. 289–298.

[MO84] Alan Mycroft and Richard A. O'Keefe, "A polymorphic type system for prolog", *Artificial Intelligence*, vol. 23, no. 3, pp. 295–307, 1984, ISSN: 0004-3702. DOI: `https://doi.org/10.1016/0004-3702(84)90017-1`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/0004370284900171`.

[Nai92] Lee Naish, “Types and the intended meaning of logic programs”, *Typed in Logic Programming*, pp. 189–216, 1992.

[Nai96a] Lee Naish, “A declarative view of modes.”, in *JICSLP*, 1996, pp. 185–199.

[Nai96b] Lee Naish, “Higher-order logic programming in prolog”, in *Proc. Workshop on Multi-Paradigm Logic Programming, JICSLP*, Citeseer, vol. 96, 1996, pp. 1–23.

[Pfe92] Frank Pfenning, “Dependent types in logic programming”, in *Types in Logic Programming*, MIT Press, 1992, pp. 285–311.

[PS99] Frank Pfenning and Carsten Schurmann, “System description: Twelf — a meta-logical framework for deductive systems”, in *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, Springer-Verlag LNAI, 1999, pp. 202–206.

[SHCO95] Zoltan Somogyi, Fergus Henderson, Thomas Conway and Richard O’Keefe, “Logic programming for the real world”, in *Proceedings of the ILPS*, Citeseer, vol. 95, 1995, pp. 83–94.

[Smo89] Gert Smolka, “Logic programming over polymorphically order-sorted types”, Ph.D. dissertation, Universitaet Kaiserslautern, 1989.

[Smo98] Gert Smolka, “Special issue on order-sorted rewriting: Foreword of the guest editor”, *Journal of Symbolic Computation*, vol. 25, no. 4, pp. 395–395, 1998.

[Wad89] Philip Wadler, “Theorems for free!”, in *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA ’89, Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 347–359, ISBN: 0897913280. DOI: `10.1145/99370.99404`. [Online]. Available: `https://doi.org/10.1145/99370.99404`.

[Wen97] Markus Wenzel, “Type classes and overloading in higher-order logic”, in *International Conference on Theorem Proving in Higher Order Logics*, Springer, 1997, pp. 307–322.

[WK20] Isabel Wingen and Philipp Körner, “Effectiveness of annotation-based static type inference”, *CoRR*, vol. abs/2008.12545, 2020. arXiv: `2008.12545`. [Online]. Available: `https://arxiv.org/abs/2008.12545`.

[Zap21] Carlo Zapponi. “Githut 2.0”. (12th Nov. 2021), [Online]. Available: `https://madnight.github.io/githut/#/pull_requests/2021/3`.

[ZY92] Joseph L. Zachary and Katherine A. Yelick, “Moded type systems to support abstraction”, in *Types in Logic Programming*, MIT Press, 1992, pp. 229–243.