Types for Grassroots Logic Programs

Abstract

Grassroots Logic Programs (GLP) is a concurrent logic programming language in which logic variables are partitioned into complementary readers and writers: an assignment to a variable is produced at most once via its sole writer, consumed at most once via its sole reader, and may contain additional readers and/or writers. This enables the concise expression of rich multidirectional communication modalities.

Building on "Logic Programs as Types for Logic Programs" (LICS'91), which defined LP types as regular sets of paths over derivable ground atoms, we develop a moded type system for GLP. Modes capture directionality of communication—whether subterms were consumed from or produced to the environment—enabling the typing of all partial computations, including those that deadlock, fail, or not terminate. GLP types are regular sets of moded paths; well-typing captures both covariance (outputs match the type) and contravariance (all typed inputs are accepted).

We introduce *type complementarity*, a relation between types that determines when their interaction is safe. In standard subtyping, contravariance emerges from recursive rule application; in session type duality, protocols must match exactly. Type complementarity differs: it builds inclusion-direction-flipping directly into the relation, reversing the inclusion obligation at each mode complement encountered. This enables asymmetric client/server interaction where a client uses a subset of operations while accepting all responses—a form of safe composition that neither standard subtyping nor session type duality can express.

ACM Reference Format:

. 2026. Types for Grassroots Logic Programs. In . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Grassroots Logic Programs (GLP) is an expressive concurrent logic programming language that partitions logic variables into complementary *readers* and *writers*, and restricts variables in goals and clauses to have at most a single occurrence. The result eschews unification in favour of simple term matching and is reminiscent of both linear logic's resource sensitivity and the futures/promises paradigm: during a computation, an assertion on the assignment of a value to a variable is produced at most once, via the sole writer (the promise), and consumed at most once, via the sole complementary reader (the future).

In a distributed multiagent setting, an assignment to a logic variable results in a single asynchronous message from its producer to its consumer. Since messages may contain both readers and writers, GLP allows the concise expression of rich multidirectional communication modalities and dynamic network reconfiguration—patterns that are central to modern concurrent and distributed systems.

The challenge of ensuring correctness in such programs motivates a type system that can express and verify the intended communication behaviour. We build on the "Logic Programs as Types for Logic Programs" approach of Frühwirth and Shapiro [6], in which a type for Logic Programs (LP) is defined as a regular set of *paths*—sequences of functor/argument-index pairs that describe the structure of terms. An LP is well-typed if this regular set includes the path abstraction of its ground-atom semantics: intuitively, every derivable ground atom has a structure permitted by the type.

To type GLP programs, we must go beyond standard LP semantics in two ways. First, we augment paths with *modes* that capture whether each subterm was consumed from the environment or produced to the environment. This is essential because GLP programs engage in bidirectional communication, and correctness depends on matching the direction of data flow. Second, we adopt a clausal semantics that captures partial derivations, enabling us to type programs that may deadlock, fail, or not terminate.

GLP types are thus regular sets of moded paths. Well-typing is defined to capture both *covariance*—all values produced by the program conform to the type—and *contravariance*—the program accepts all inputs permitted by the type. We establish the connection between well-typing and GLP semantics via a soundness theorem.

Type Complementarity. A key contribution of this paper is the *type complementarity* relation, which determines when the interaction between two types is safe. In standard type theory, contravariance arises from the recursive application of subtyping rules: when checking $S_1 \to S_2 <: T_1 \to T_2$, the subtyping direction reverses for the argument types [4, 15]. The direction flip is a consequence of applying the rules, not a primitive of the relation itself.

Type complementarity takes a different approach: the inclusion-direction-flipping is built directly into the relation. When walking two type structures simultaneously, encountering a mode complement (?) in either type reverses the inclusion obligation at that position. This continues recursively, so that doubly-complemented positions return to the original direction—mirroring how contravariant positions compose in standard subtyping, but as a definitional primitive rather than an emergent property.

This also differs from session type duality [3, 10, 24], which is an involution mapping each session type to its exact complement: output becomes input, select becomes branch. Duality ensures that dual endpoints have perfectly matching protocols, but it cannot express asymmetric interaction where one side uses a subset of the other's capabilities. Session type subtyping [8] addresses protocol refinement but operates on session type judgments with explicit input/output modalities, whereas GLP types are regular sets of moded paths.

Type complementarity captures a specific form of safe interaction: a client may use a *subset* of a server's operations, but for each

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM

https://doi.org/10.1145/nnnnnnn.nnnnnnn

operation used, must accept *all* possible responses. The recursive direction-flipping ensures that this asymmetry is checked correctly at every level of the type structure.

Concept	LP	GLP	Session Types
Semantics	Ground atoms	Moded atoms	Channel traces
Types	Path sets	Moded path sets	Judgments
Duality	—	Mode (?)	Protocol ($\overline{S}$)
Variance	—	Moded paths	Subtyping rules
Composition	Inclusion	Complementarity	Duality + subtyping

Paper Structure. Section 2 recalls Logic Programs, defining syntax, operational semantics via transition systems, and LP types as regular sets of paths. Section 3 introduces GLP, which partitions variables into readers/writers and eschews unification in favour of term matching. Section 4 defines moded types and well-typing for GLP programs, and introduces type complementarity. Section 5 establishes the connection between well-typing and GLP semantics. Section 6 discusses related work, and Section 7 concludes.

2 Logic Programs

We recall standard Logic Programs (LP) notions of syntax, most-general unifier (mgu), and semantics via goal reduction.

2.1 Syntax

We employ the standard LP notions of variables, constants, terms, clauses, procedures, and programs.

Definition 2.1 (Logic Programs Syntax). *We employ standard LP notions. Let $\mathcal{V}$ denote the set of **variables** (identifiers beginning with uppercase). A **term** is a variable, a constant (numbers, strings, or the empty list* `[]`*), or a compound term $f(T_1, \ldots, T_n)$ with functor f and subterms T_i. Let $\mathcal{T}$ denote the set of all terms. We use standard list notation:* `[X|Xs]` *for a list cell,* `[X1,...,Xn]` *for finite lists. A term is **ground** if it contains no variables.*

*A **goal** is a multiset of atoms; the empty goal is written* `true`*. A **clause** A `:-` B has head atom A and body goal B; a **unit clause** has empty body. A **logic program** is a finite set of clauses; clauses for the same predicate form a **procedure**. Let $\mathcal{G}(P)$ denote the set of goals over program P.*

Example 2.2 (Append). *The quintessential logic program for list concatenation is the following procedure, which has two clauses:*

```
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
append([], Ys, Ys).
```

Logically, a clause A `:-` B is a universally-quantified implication in which B implies A, and a program is a conjunction of its clauses. By convention, we use plural variable names like `Xs` *to denote a list of* `X`*'s.*

2.2 Operational Semantics

A *substitution* σ is an idempotent function $\sigma : \mathcal{V} \to \mathcal{T}$, namely a mapping from variables to terms applied to a fixed point. By convention, $\sigma(x) = x\sigma$. Let Σ denote the set of all substitutions. We assume the standard notions of instance, ground, renaming, renaming apart, unifier, and most-general unifier (mgu). We assume a fixed renaming-apart function, so that the result of renaming T' apart from T is well defined.

Definition 2.3 (LP Goal/Clause Reduction). *Given an LP goal A and clause C, with H `:-` B being the result of renaming C apart from A, the **LP reduction** of A with C **succeeds with** (B, σ) if A and H have an mgu σ.*

We define the operational semantics of LP and of GLP via transition systems; in both, set relations and operations refer to multisets.

Definition 2.4 (Transition System). *A **transition system** is a tuple $TS = (C, c_0, T)$ where:*

- *C is an arbitrary set of **configurations***
- *$c_0 \in C$ is a designated **initial configuration***
- *$T \subseteq C \times C$ is a **transition relation**. A transition $(c, c') \in T$ is also written as $c \to c' \in T$.*

*A transition $c \to c' \in T$ is **enabled** from configuration c. A configuration c is **terminal** if no transitions are enabled from c. A **computation** is a (finite or infinite) sequence of configurations where for each two consecutive configurations (c, c') in the sequence, $c \to c' \in T$. A **run** is a computation starting from c_0, which is **complete** if it is infinite or ends in a terminal configuration.*

Definition 2.5 (Logic Programs Transition System). *A transition system $LP(P) = (C, c_0, T)$ is a **Logic Programs transition system** for a logic program P and initial goal $G_0 \in \mathcal{G}(P)$, if $C = \mathcal{G}(P) \times \Sigma$, $c_0 = (G_0, \emptyset)$, and T is the set of all transitions $(G, \sigma) \to (G', \sigma') \in (\mathcal{G}(P) \times \Sigma)^2$ such that for some atom $A \in G$ and clause $C \in P$ the LP reduction of A with C succeeds with $(B, \hat{\sigma})$, $G' = (G \setminus \{A\} \cup B)\hat{\sigma}$, and $\sigma' = \sigma \circ \hat{\sigma}$*

The following notion of a proper run ensures that variable names are not re-used; in implementation terms this is an assumption on the integrity of the memory-management/garbage-collection system.

Definition 2.6 (Proper and Successful Run, Outcome). *A run $\rho : (G_0, \sigma_0) \to \cdots \to (G_n, \sigma_n)$ of $LP(P)$ is **proper** if for any $1 \le i < n$, a variable that occurs in G_{i+1} but not in G_i also does not occur in any G_j, $j < i$. If proper, the **outcome** of ρ is $(G_0$ `:-` $G_n)\sigma_n$. Such a run is **successful** if $G_n = \emptyset$.*

The following proposition justifies calling a proper LP run a *derivation*, and a complete proper run ending in the empty goal a *successful derivation*.

Proposition 2.7 (LP Computation is Deduction). *The outcome $(G_0$ `:-` $G_n)\sigma$ of a proper run $\rho : (G_0, \sigma_0) \to \cdots \to (G_n, \sigma_n)$ of $LP(P)$ is a logical consequence of P.*

Note that the LP transition system does not require the initial goal G_0 to be atomic.

2.3 Denotational Semantics

The $LP(P)$ transition system allows defining several denotational semantic notions for a program P:

(1) The *clause semantics* of P is the set of all outcomes of all proper runs of $LP(P)$ with an initial most-general atomic goal (arguments are distinct variables). It is closely related to the fully-abstract compositional semantics of LP [7].

(2) The *atom semantics* of P is the set of all outcomes of all successful derivations of $LP(P)$ with an initial most-general atomic goal.
(3) The *ground atom semantics* of P is the standard model-theoretic semantics of logic programs. It is the set of ground instances of the atom semantics of P over the Herbrand universe of P.

2.4 Types

We associate with each logic term T a *term tree*: a labelled tree with vertices labelled with the functor/arity for compound terms and constants/variables for leaves in T, and edges representing the subterm relation and labelled with the integer argument index of the subterm within the term.

So, the term T = `append([X|Xs],Ys,[X|Zs])` is represented by the term tree:

```
append
    1: "."/2
        1: X
        2: Xs
    2: Ys
    3: "."/2
        1: X
        2: Zs
```

The *paths* of a term T, $paths(T)$, is the set of labelled paths in its term tree. For the example above, $paths(T)$ is the set:

```
append  --1--> "."/2 --1--> X.
append  --1--> "."/2 --2--> Xs.
append  --2--> Ys.
append  --3--> "."/2 --1--> X.
append  --3--> "."/2 --2--> Zs.
```

In [6], a type for LP is defined as a regular set of paths, and a program is well-typed if the type includes all paths of all atoms in its ground-atom semantics. Types are naturally specified using well-founded BNF rules and procedure type declarations, for example:

```
List ::= [] ; [_|List].

procedure append(List,List,List).
```

with _ denoting any path in any ground term in the Herbrand universe of the program. Note that LP well-typing is *covariant*: it guarantees that all outputs are within the type, but does not constrain inputs. Thus `append([],c,c)` is in the ground-atom semantics and well-typed by `append(_,_,_)`.

The append program above is well-typed by the type thus defined. We note, however, that append is also well-typed with any of the following declarations:

```
procedure append(_,_,_).
procedure append(List,List,_).
procedure append(_,_,List).
```

etc. So, providing useful type declarations is up to the programmer or program designer: the more informative and tight the type declaration, the more useful is the statement that the program is well-typed. Many extensions for this notion of LP types are possible and needed, in particular polymorphic types, discussed in [25].

3 Grassroots Logic Programs

Grassroots Logic Programs (GLP) extend LP by (1) adding a paired *reader* $X?$ to every "ordinary" logic variable X, now called a *writer* (2) restricting variables in goals and clauses to have at most a single occurrence (SO) (3) requiring that a variable occurs in a clause iff its paired variable also occurs in it (single-reader single-write, SRSW). The result eschews unification in favour of simple term matching, is linear-logic-like, and is futures/promises-like: each assignment $X := T$ of a term T to a variable X is produced at most once, via the sole writer (promise) X, and consumed at most once, via its sole paired reader (future) $X?$.

3.1 GLP Variables: Readers and Writers

Definition 3.1 (GLP Variables). *Recall that $\mathcal{V}$ is the set of LP variables, henceforth called **writers**. Define $\mathcal{V}? = \{X? \mid X \in \mathcal{V}\}$, called **readers**. The set of all GLP variables is $\hat{\mathcal{V}} = \mathcal{V} \cup \mathcal{V}?$. A writer X and its reader $X?$ form a **variable pair**.*

GLP terms, goals, and clauses are as in LP except that they are defined over the variables in $\hat{\mathcal{V}}$. We use $\mathcal{G}_?(P)$ to denote the set of goals over $\hat{\mathcal{V}}$ restricted to the vocabulary of P. Primitives types are handled with *guard predicates*.

Definition 3.2 (Single-Occurrence (SO) Invariant, Single-Reader/Single-Writer (SRSW) Restriction). *A goal or clause satisfies the **single-occurrence (SO) invariant** if every variable occurs in it at most once. A clause C satisfies the **single-reader/single-writer (SRSW) syntactic restriction** if it satisfies SO and, furthermore, a variable occurs in C iff its paired variable also occurs in C.*

As we shall see (Proposition 3.9), the SO invariant is maintained by the SRSW restriction: reducing a goal satisfying SO with a clause satisfying SRSW results in a goal satisfying SO. The goal of the SRSW requirement is to prevent multiple writer occurrences racing to bind a variable. However, the type of a variable may be determined to be ground by several guards (e.g `guard(X)` or `X? < Y?`), and hence may not include writers, now or in the future. In such a case the requirement is relaxed and any number of readers in a clause are allowed. In Typed GLP, the condition can be further relaxed if the type of a variable is constant (number or string).

Example 3.3 (Merge). *Consider the quintessential concurrent logic program for fairly merging two streams, written in GLP:*

```
merge([X|Xs], Ys, [X?|Zs?]) :- merge(Ys?, Xs?, Zs).
merge(Xs, [Y|Ys], [Y?|Zs?]) :- merge(Xs?, Ys?, Zs).
merge(Xs, [], Xs?).
merge([], Ys, Ys?).
```

and the goal `merge([1,2,3|Xs?],[a,b|Ys?],Zs)` *and note that both satisfy SO and each clause satisfies SRSW.*

3.2 Operational Semantics

Definition 3.4 (Writers Substitution, Assignment, Readers Substitution and Counterpart). *A GLP **writer assignment** is a term of the form $X := T$, $X \in \mathcal{V}$, $T \notin \mathcal{V}$, satisfying SO. Similarly, a GLP **reader assignment** is a term of the form $X? := T$, $X? \in \mathcal{V}?$, $T \notin \mathcal{V}$, satisfying SO. A **writers (readers) substitution** σ is the substitution implied by a set of writer (reader) assignments that jointly satisfy SO. Given a writers assignment $X := T$, its **readers***

counterpart *is* $X? := T$*, and given a writers substitution* σ*, its* ***readers counterpart*** $\sigma?$ *is the readers substitution defined by* $X?\sigma? = X\sigma$*.*

Definition 3.5 (GLP Renaming, Renaming Apart). *A* ***GLP renaming*** *is a substitution* $\rho : \hat{\mathcal{V}} \to \hat{\mathcal{V}}$ *such that for each* $X \in \mathcal{V}$*:* $X\rho \in \mathcal{V}$ *and* $X?\rho = (X\rho)?$*. Two GLP terms have a variable in common if for some writer* $X \in \mathcal{V}$*, either* X *or* $X?$ *occurs in both. A GLP renaming* σ *renames* T' ***apart from*** T *if* $T'\sigma$ *and* T *have no variable in common.*

Definition 3.6 (GLP Goal/Clause Reduction). *Given GLP goal* A *and clause* C*, with* $H :- B$ *being the result of the GLP renaming of* C *apart from* A*, the* ***GLP reduction*** *of* A *with* C ***succeeds with result*** (B, σ) *if* A *and* H *have a writer mgu.*

Definition 3.7 (GLP Transition System). *Given a GLP program* P*, an* ***asynchronous resolvent*** *over* P *is a pair* (G, σ) *where* $G \in \mathcal{G}_?(P)$ *and* σ *is a readers substitution.*

A transition system $GLP = (\mathcal{C}, c_0, \mathcal{T})$ *is a* ***GLP transition system*** *over* P *and initial goal* G_0 *satisfying SO if:*

1. $\mathcal{C}$ *is the set of all asynchronous resolvents over* P
2. $c_0 = (G_0, \emptyset)$
3. $\mathcal{T}$ *is the set of all transitions* $(G, \sigma) \to (G', \sigma')$ *satisfying either:*
   (a) ***Reduce:*** *there exists unit goal* $A \in G$ *such that* $C \in P$ *is the first clause for which the GLP reduction of* A *with* C *succeeds with result* $(B, \hat{\sigma})$*,* $G' = (G \setminus \{A\} \cup B)\hat{\sigma}$*, and* $\sigma' = \sigma \circ \hat{\sigma}?$
   (b) ***Communicate:*** $\hat{\sigma} = \{X := T\} \in \sigma$*,* $X? \in G$*,* $G' = G\hat{\sigma}?$*, and* $\sigma' = \sigma$

GLP Reduce is different from LP in (1) the use of a writer mgu instead of a regular mgu and (2) the choice of the first applicable clause instead of any clause. The first is the fundamental use of GLP readers for synchronization. The second compromises on the or-nondeterministic of LP to allow the writing of fair concurrent programs, such as fair merge above. Note that or-nondeterminism is not completely eliminated, as different scheduling of arrival of bindings on the two input streams of `merge` may result in different orders in its outputs stream. The GLP Communicate rule realises the use of reader/writer pairs for asynchronous communication: It applies an assignment to a reader after it has been applied to its paired writer.

Abstractly, the key differences between LP and GLP relate to monotonicity: In LP, if a goal cannot be reduced, it will never be reduced. In GLP, a goal that cannot be reduced now may be reduced in the future, due to GLP's use of paired logic variables for communication and synchronization: If A and H have an mgu that writes on a reader $X? \in A$, and therefore have no writer mgu at present, it may be possible that another goal that has X will reduce, assigning X, and later $X?$, to a value that will allow A and H to have a writer mgu. Conversely, in LP, if a goal A can be reduced now with some clause $H :- B$, with a regular mgu of A and H, it may not be reducible in the future due to variables that A shares with other goals being assigned values by other goal reductions, preventing unification between the instantiated A and H. In GLP, if a goal A can be reduced now (with a writers mgu), it can always be reduced in the future, as the SO invariant ensures that no other goal can assign any writer in A.

Implementation-wise, if a GLP goal A cannot be reduced now, but there is a readers substitution σ such that $A\sigma$ can be reduced, such readers are identified, the goal A *suspends* on these readers, and is rescheduled for reduction once any of them is assigned.

Despite these differences, GLP can adopt the same notion of successful run and outcome of LP (Definition 2.6), and have the same notion of logic consequence as LP. Let $/?$ be an operator that replaces every reader by its paired writer.

Proposition 3.8 (GLP Computation is Deduction). *Let* $(G_0 :- G_n)\sigma$ *be the outcome of a proper run* $\rho : (G_0, \sigma_0) \to \cdots \to (G_n, \sigma_n)$ *of* $GLP(P)$*. Then* $(G_0 :- G_n)\sigma/?$ *is a logical consequence of* $P/?$*.*

We note two additional safety properties of GLP runs.

Proposition 3.9 (SO Preservation). *If the initial goal* G_0 *satisfies SO, then every goal in the GLP run satisfies SO.*

Proposition 3.10 (Monotonicity). *In any GLP run, if unit goal* A *can reduce with clause* C *at step* i*, then either an instance of* A *has been reduced by step* $j > i$*, or an instance of* A *can still reduce with* C *at step* j*.*

Monotonicity is a consequence of the single-assignment discipline: a successful writes mgu match cannot be invalidated by subsequent assignments.

3.3 Term Matching Eschews Unification

If two terms T_1 and T_2 that jointly satisfy SO are unifiable with an mgu σ, then σ maps any variable in T_1 to a subterm of T_2 and vice versa. Hence, the SO invariant of GLP allows eschewing unification in favour of *term matching* that performs joint term-tree traversal and collects variable assignments along the way, as follows.

Definition 3.11 (Term Matching). *Given two terms* T_1 *and* T_2 *that jointly satisfy SO, their* ***term matching*** *proceeds via the joint traversal of the term-trees of* T_1 *and* T_2*, consulting the following table at each pair of joint vertices, where* X_1, X_2 *denote writers,* $X_1?, X_2?$ *denote readers, and* f/n *denotes a non-variable term, a constant (strings, numbers, functors/predicate names) when* $n = 0$ *and a compound term when* $n > 0$*:*

$T_1 \backslash T_2$	*Writer* X_2	*Reader* $X_2?$	*Term* f_2/n_2
Writer X_1	*fail*	$X_1 := X_2?$	$X_1 := T_2$
Reader $X_1?$	$X_2 := X_1?$	*fail*	*fail**
Term f_1/n_2	$X_2 := T_1$	*fail*	*fail if* $f_1 \neq f_2$ *or* $n_1 \neq n_2$

The writer mgu is the union of all writer assignments if no fail *was encountered.*

Remark 3.12. *In an actual implementation, assuming* T_1 *is a goal term and* T_2 *a head term, the* fail* *case of* $X_1?$ *and* T_2 *would add* $X_1?$ *to the set of readers the goal would suspend upon.*

4 Typed GLP

While GLP is an untyped language, with types being an addition, we introduce Typed GLP including the notion of types and well-typing. The GLP type system supports correct programming of multidirectional communication modalities.

4.1 Typed GLP Programs

LP types are concerned with the values logic variables may take. In GLP, we are also concerned whether these values were produced or consumed during the computation. Thus, a GLP type also indicates whether a term, or a subtype, is produced or consumed. By convention, we define types from the perspective of their producer, which implicitly defines its complement—the same type but from the perspective of its consumer.

GLP types are also naturally specified using BNF rules and procedure type declarations, with a twist: They include the mode complementation operator ?, with every type definition implicitly defining also its complement. For example, the stream producer type:

```
Stream ::= [] ; [_|Stream].
```

implicitly defines the complementary stream consumer type `Stream?`. If it were to be specified explicitly, it would be written as:

```
Stream? ::= []? ; [_?|Stream?]?.
```

with primitive types _ denoting any produced term and _? any consumed term.

For each type T, complementation defines the dual type $T?$ with a corresponding dual automaton. The automaton for $T?$ is obtained from the automaton for T by:

(1) Replacing each state S with its complement state $S?$
(2) Replacing each mode annotation: ↑ becomes ↓, and ↓ becomes ↑

This defines complementation as an involution: $(T?)? = T$, since flipping states and modes twice returns to the original automaton. The primitive types _ and _? are complements of each other, as are `Integer` and `Integer?`, and `String` and `String?`.

A type that does not include a complementation (for example `Stream`) is an *output type*, and its complement (for example `Stream?`) is an *input type*. Otherwise it is an *interactive type*.

Typed GLP includes a type declaration for each procedure. For example, the `merge` program above could include the procedure declaration:

```
procedure merge(Stream?,Stream?,Stream).
```

which is a syntactically-pleasing alternative to the formal definition of a goal with the procedure's predicate being but a typed term:

```
Procedure ::= merge(Stream?,Stream?,Stream).
```

This equivalence is definitional: in the DFA constructed from a typed GLP program, each procedure declaration introduces a state with transitions to its argument type states. For `merge(Stream?,Stream?,Stream)` the DFA state `merge/3` has three outgoing transitions labeled with argument positions 1, 2, and 3, leading to states `Stream?`, `Stream?`, and `Stream` respectively. Each argument is checked against the automaton for its declared type: arguments 1 and 2 use the `Stream?` automaton, while argument 3 uses the `Stream` automaton.

We impose standard restrictions on GLP type definitions so that they correspond to a DFA. The primitive types are formalized as follows:

```
Integer  ::= ... ; -1 ; 0 ; 1 ; 2 ; ...
Real     ::= ... ; -1.0 ; 0.0 ; 3.14159 ; 2.5e10 ; ...
Number   ::= Integer ; Real.
String   ::= ... ; a ; b ; ... ; foo ; bar ; ...
Constant ::= Number ; String.
```

where each numeric and string literal is conceptually a distinct alternative. Integer literals (no decimal point or exponent) and real literals (with decimal point or exponent notation) are syntactically disjoint, so `Number ::= Integer ; Real` is deterministic. Similarly, numeric and string literals are disjoint, so `Constant ::= Number ; String` is deterministic.

The DFA has two states per defined type name—one for the type and one for its complement (e.g., `Stream` and `Stream?`, `CounterCall` and `CounterCall?`)—plus one state per procedure (e.g., `merge/3`). The system-defined states are `Integer`, `Integer?`, `Real`, `Real?`, `Number`, `Number?`, `String`, `String?`, _, and _?, where each pair consists of a type and its complement.

The states _ and _? are final states: _ accepts any produced term (a writer), while _? accepts any consumed term (a reader). These are complements of each other.

The states `Integer`, `Real`, and `String` have conceptually infinite outgoing transitions—one for each literal of the respective type—each leading to a final state. The state `Number` has transitions to both `Integer` and `Real`, determined by the syntactic form of the literal. The complement states `Integer?`, `Real?`, `Number?`, and `String?` have the same structure with modes flipped. In practice, the implementation checks type membership rather than enumerating transitions.

Transitions are labeled with functor, arity, argument position, and mode. Procedure states have transitions to their argument type states, labeled by argument position. Type states have transitions based on their BNF alternatives. Constants appearing in type alternatives (such as `[]` in the definition of `Stream`) are transition labels leading to a final state.

Example 4.1 (Complement Automata). *For the type definition* `Stream ::= [] ; [_Stream]`*, the automata are:*

Stream automaton *(producer view):*

- *State* `Stream` *with transitions:*
  - `[]` → *final state*
  - `[]`*|(2,1):*↑ → _
  - `[]`*|(2,2):*↑ → `Stream`

Stream? automaton *(consumer view, complement):*

- *State* `Stream?` *with transitions:*
  - `[]` → *final state*
  - `[]`*|(2,1):*↓ → _?
  - `[]`*|(2,2):*↓ → `Stream?`

Every aspect is flipped: states (`Stream` ↔ `Stream?`*,* _ ↔ _?*) and modes (*↑ ↔ ↓*).*

The BNF must be deterministic: alternatives must be distinguishable by their top-level functor or, for primitive types, by disjoint type membership. The definition `Constant ::= Integer ; String` is legal because integers and strings are syntactically disjoint. However, the following declarations are illegal because their alternatives overlap:

```
Any ::= _ ; _?.       % overlapping: both accept all terms
AnyOne ::= 1 ; 1?.    % overlapping: 1 matches both alternatives
Ambiguous ::= _ ; Integer.  % overlapping: integers match both
```

Type aliases are also prohibited: a type definition must introduce new structure, not merely rename an existing type or primitive. The following are illegal:

```
Output ::= _.            % alias for primitive
Input ::= _?.            % alias for primitive
MyList ::= List.         % alias for defined type
MyStream ::= Stream?.   % alias for complement of defined type
```

Each type definition must have at least one alternative that is a constant, a structured term (functor with arguments), an empty list `[]`, a list cons `[HT]`, or a difference list. Primitives (`_`, `_?`) and type references (`T`, `T?`) may only appear as arguments within structured alternatives, not as top-level alternatives by themselves.

Definition 4.2 (Typed GLP Program). *A **typed GLP program** $P = (Cs, D)$ has GLP clauses Cs and a GLP type D defining the type of every procedure in Cs.*

4.2 Typed GLP Programming Examples

Stream merging. Combining the elements above provides a typed GLP `merge` program:

```
Stream ::= [] ; [_|Stream].

procedure merge(Stream?,Stream?,Stream).
merge([X|Xs], Ys, [X?|Zs?]) :- merge(Ys?, Xs?, Zs).
merge(Xs, [Y|Ys], [Y?|Zs?]) :- merge(Xs?, Ys?, Zs).
merge(Xs, [], Xs?).
merge([], Ys, Ys?).
```

We will see later that this typed `merge` program is indeed well-typed. Similarly to LP types, it is also well-typed with any of the following weaker type declarations:

```
procedure merge(_?,_?,_).
procedure merge(Stream?,Stream?,_).
procedure merge(_?,_?,Stream).
```

etc., but is not well-typed with the following declarations:

```
procedure merge(Stream,Stream,Stream).
procedure merge(Stream?,Stream?,Stream?).
procedure merge(_,_,_).
procedure merge(_?,_?,_?).
```

etc.

So, as in LP, providing useful type declarations is up to the programmer. To ease the specification of types, we include parametrised types, a limited form of type polymorphism. A parametrised type declaration of `merge` could look like:

```
Stream(X) ::= [] ; [X|Stream(X)].

procedure merge(Stream(X)?,Stream(X)?,Stream(X)).
```

Monitors with hollow messages. Next, we show an interactive type with mode complementation. Consider a monitor process, maintaining a local state and serving requests to update the state and read it. It can serve arbitrarily many clients through a network of merge processes. Monitor queries use a programming technique called a *hollow message*—a message from the producer to the consumer that includes a writer, which inverts the direction of communication and allows the message consumer to respond to its producer. Note that the program relaxes the SRSW requirement for consumed integers.

```
CounterCall::= add ; clear ; read(Integer?).

procedure monitor(Stream(CounterCall)?)
monitor(In) :- monitor(0,In?)

procedure monitor(Integer?,Stream(CounterCall)?)
monitor(N,[add|In]) :- N1 := N? +1, monitor(N1?,In?).
monitor(N,[clear|In]) :- monitor(0,In?).
monitor(N,[read(N?)|In]) :- monitor(N?,In?).
```

Bounded-buffer communication with hollow streams. In the output `Stream` type the producer of the stream also produces its elements and in its complement the consumer consumes both the stream and its elements. The interactive `HollowStream` is produced with hollow elements, namely writers, to be filled by its consumer:

```
HollowStream ::= [] ; [_?|HollowStream] .
```

The following program demonstrates the use of a `HollowStream` for bounded-buffer communication. In this example the `consumer` of a stream of values controls the pace of their production by producing a hollow stream, which is consumed and filled with values, one element at a time, by the `producer` of the values. The buffer in this example is bounded to 2 hollow slots.

```
HollowIntegers ::= [Integer?|HollowIntegers] ; [].

procedure consumer(HollowIntegers)
consumer([X1, X2, X3 | Xs?]) :-
    integer(X1?) | consumer([X2?, X3? | Xs]).

procedure producer(Integer?,HollowIntegers?)
producer(N, [N? | Xs]) :-
    number(N?) | N1 := N? + 1, producer(N1?, Xs?).
```

A goal that initialised buffered communication could look like:

```
consumer([X1?, X2? | Xs]), producer(1, [X1, X2 | Xs?]).
```

We note that the same effect could be achieved by more sophisticated, albeit more cryptic, parametric declarations:

```
procedure consumer(Stream(Integer?))

procedure consumer(Stream(Integer?)?)
```

Cooperative stream construction. Consider a stream produced cooperatively by two processes, which sporadically relegate control of stream production among them.

```
CoopStream ::=
    [Integer|CoopStream] ; [switch|CoopStream]? ; [].

procedure read(Integer?,CoopStream?).
read(N,[X|Xs]) :- integer(X?) | read(N?,Xs?). % keep reading
read(N,[switch|Xs?]) :- write(N?,Xs). % start writing


procedure write(Integer?,CoopStream).

write(N,Xs?) :- integer(N) | write(N?,N?,Xs).
```

```
procedure write(Integer?,Integer?,CoopStream).
write(N,0,[switch|Xs]) :- read(N?,Xs?).
    % offer to switch and start reading
write(N,K,[N?|Xs?]) :-
    K > 0 | K1 := K?-1, write(N?,K1?,Xs).
    % keep writing
```

A goal that initialised this cooperative stream production could look like:

```
write(3,Xs), read(4,Xs?).
```

The result would be a stream of three 3's followed by four 4's, with `switch` in between, repeatedly, each subsequence produced by the corresponding process and consumed by the other.

Abstract data types: Difference-list and bidirectional channels. We show how complex data types can be treated abstractly. Lists can be represented as the difference between two streams, allowing constant-time list concatenation.

```
DiffList ::=  Stream? \ Stream.

procedure dl_append(DiffList?, DiffList?, DiffList).
dl_append(A\B?, B\C?, A?\C).
```

A bidirectional channel has two streams, one in each direction.

```
Channel(X) ::= ch(Stream(X)?, Stream(X)).

procedure new_channel(Channel(X), Channel(X)).
new_channel(ch(Xs?, Ys), ch(Ys?, Xs)).

procedure send(X,Channel(X)?, Channel(X)).
send(X, ch(In, [X?|Out?]), ch(In?, Out)).

procedure receive(X, Channel(X)?, Channel(X)).
receive(X?, ch([X|In], Out?), ch(In?, Out)).
```

GLP supports such abstract data typed via define guards, which are unfolded (partially-evaluated) at compile time.

4.3 Well-Typing

The basic semantic object that captures the behaviour of a GLP program is the moded term, which specifies whether each subterm was produced or consumed during the computation. Next we define this notion and when it is well-typed.

Definition 4.3 (Moded Term, Complement). *Given a GLP term T, a **moded term** T' corresponding to T is the result of adding one of two **mode annotations**, **consume** ↓ or **produce** ↑, to T and to every non-variable subterm of T. Given a moded term T, its **complement** $T?$ is obtained from T by flipping every mode annotation and replacing every variable by its paired variable.*

For example, consider the term:

```
merge([3|Xs?],Ys?,[3|Zs])
```

and a corresponding moded term:

```
↓merge(↓[↓3|Xs?],Ys?,↑[↑3|Zs])
```

A moded term can be equivalently represented as a *moded term-tree*, which is an LP term tree with pair-labelled edges: the integer argument index of the subterm within the term and a mode annotation. A term tree has an incoming edge to the root and and explicit consumed/produced annotations for readers/writers. Here is the moded term tree of the `merge` moded term presented above:

```
(0,↓): merge/3
    (1,↓): "."/2
        (1,↓): 3
        (2,↓): Xs?
    (2,↓): Ys?
    (3,↑): "."/2
        (1,↑): 3
        (2,↑): Zs
```

We can then collect all moded paths of a moded term tree:

```
(0,↓) --> merge/3 --(1,↓)--> "."/2 --(1,↓)--> 3
(0,↓) --> merge/3 --(1,↓)--> "."/2 --(2,↓)--> Xs?
(0,↓) --> merge/3 --(2,↓)--> Ys?
(0,↓) --> merge/3 --(3,↑)--> "."/2 --(1,↑)--> 3
(0,↓) --> merge/3 --(3,↑)--> "."/2 --(2,↑)--> Zs
```

Remark 4.4 (Structural Mode vs. Variable Mode). *The mode annotations in a moded term tree are* structural*—they describe the direction of data flow at each position, as determined by the type context. Variables do not carry mode annotations themselves. Instead, each variable has an* implicit mode*: readers have implicit mode consume (↓), and writers have implicit mode produce (↑).*

For a moded term to be well-typed, each variable's implicit mode must be consistent *with the structural mode of its position:*

- *A reader* `X?` *(implicit mode: ↓) may appear at a position with structural mode ↓*
- *A writer* `X` *(implicit mode: ↑) may appear at a position with structural mode ↑*

Definition **??** *formalizes this consistency requirement. The structural mode is inherited from the enclosing context and propagates through the term structure; the implicit mode is intrinsic to the variable's reader/writer form. These are distinct concepts that must agree at variable positions.*

For a moded term T, $paths(T)$ denote the set of moded paths of its moded term tree. A path that commences with a consume (produce) annotation is called an *input* (*output*) path. A moded term is *consumed* (*produced*) if all its mode annotations are consume (produce), and *I/O* if it is ↓-annotated with at most one mode-inversion to ↑ on any of its paths.

A defined GLP type D also defines a regular set of moded paths, denoted $paths(D)$, in which every symbol corresponds to a type and the last symbol is a primitive type. However, the vocabulary of the two languages is not identical. Specifically, type paths end with primitive types, whereas moded term paths end with primitive terms—constants or variables.

The table below summarises how primitive terms in a moded term path correspond to types in a type path. Each type state has a complement state; the appropriate one is used based on the declared argument type. When a type uses `Integer` or `String`, any value of that primitive type is accepted. When a type alternative specifies an exact constant (e.g., `[]` in `Stream` or `0` in a counter type), the corresponding term must match that constant exactly; the constant serves as a transition label to a final state. Moreover, a term path and a type path may be consistent even if one is a prefix of the other: If the term path ends with a reader and the corresponding type is

Term	State	Complement	Interpretation
`X`, writer	`_`	`_?`	any produced term
`X?`, reader	`_?`	`_`	any consumed term
`42`, integer	`Integer`	`Integer?`	any integer literal
`3.14`, real	`Real`	`Real?`	any real literal
numeric literal	`Number`	`Number?`	any numeric literal
`foo`, string	`String`	`String?`	any string literal
`[]`, constant	—	—	exact match required

consumed; or a writer and the term path is produced; or if the term path is primitive input `_?` and the corresponding term is consumed; or if the term path is primitive output `_` and the corresponding term is produced. Hence:

Definition 4.5 (Consistent Paths). *Let x be a moded term path and y a GLP type path. Then x and y are **consistent** if:*

1. *They are of equal length and identical except for their last symbols, which are consistent.*
2. *Else, if x is a prefix of y except for its last symbol that is:*
   (a) *a reader `X?` and the mode of the corresponding type symbol is consume ↓, or*
   (b) *a writer `X` and the mode of the corresponding type symbol is produce ↑,*
3. *Else, if y is a prefix of x except for its last symbol that is:*
   (a) *`_?` and the mode of the corresponding term symbol is consume ↓, or*
   (b) *`_` and the mode of the corresponding term symbol is produce ↑*

Example 4.6 (Consistent Paths). *Consider the first* `merge` *clause with type declaration* `merge(Stream?,Stream?,Stream)`*:*

```
merge([X|Xs], Ys, [X?|Zs?]) :- merge(Ys?, Xs?, Zs).
```

We construct a moded head H' (see Definition 4.8 below) and check path consistency. For the first argument, the type path is:

```
(0,↓) --> merge --(1,↓)--> Stream? --(1,↓)--> "."/2 --(1,↓)--> _?
```

The corresponding term path in H' is:

```
(0,↓) --> merge --(1,↓)--> "."/2 --(1,↓)--> X?
```

These paths are consistent by case 2(a): the term path is a prefix of the type path, ending in reader `X?`, and the corresponding type symbol has mode consume ↓. The variable `X?` is assigned type `_?` (consumed).

For the third argument, the type path is:

```
(0,↓) --> merge --(3,↑)--> Stream --(1,↑)-->
    "."/2 --(1,↑)--> _
```

The corresponding term path in H' is:

```
(0,↓) --> merge --(3,↑)--> "."/2 --(1,↑)--> X
```

These are consistent by case 2(b): the term path ends in writer `X`, and the corresponding type symbol has mode produce ↑. The variable `X` is assigned type `_` (produced).

The types of `X?` and `X` are `_?` and `_`, which are complements, as required.

Definition 4.7 (Well-Typed Moded Term). *A moded term T is **well-typed** by a GLP type D if for each term path $x \in paths(T)$ there is a consistent path $y \in paths(D)$, and if for every pair of variables in T, their types as determined by D are complementary.*

Head modes are most confusing, for two reasons: First, in GLP (prior to imposing type constraints) the same head term can be used as both consumer and producer, depending on the goal. Hence, to verify that a head is well-typed, modes consistent with the type have to be successfully to it. Moreover, goal/head unification can initially consume a goal term, and upon reaching a goal writer can produce a value to it. So the modes head can be initially produce, or initially consume and then produce, but once produced cannot flip back to consuming. Hence the head must be I/O moded.

Second, the role of head variables is inverted to what is expected: A head writer X serves as input, to be bound by some goal term, with the paired reader `X?`, whether it occurs in the head or the body of the clause, being simultaneously bound to the same value. A head reader `X?` serves as *output*, either as part of a produced term that is bound to a goal writer, or being directly bound to some goal writer `Y`. The paired writer `X` in the body can subsequently be assigned a value, and consequently assigning `X?` and hence the goal variable `Y` previously bound to `X?` to the same value. The following definition aims to capture all these aspects:

Definition 4.8 (Moded Head). *Given a head H, a **moded head** H' is obtained by (1) constructing an I/O-moded term corresponding to H, then (2) for each variable, if its form does not match its position's structural mode, replacing it with its paired variable. Specifically, a variable at a position with structural mode ↓ should be a reader, and a variable at a position with structural mode ↑ should be a writer.*

Example 4.9 (Moded Head). *Consider the first* `merge` *clause head:*

```
H = merge([X|Xs], Ys, [X?|Zs?])
```

with type `merge(Stream?,Stream?,Stream)`*.*

Step 1: Construct an I/O-moded term. The type tells us arguments 1 and 2 are consumed (`Stream?`) and argument 3 is produced (`Stream`):

```
↓merge(↓[↓X|Xs], Ys, ↑[↑X?|Zs?])
```

Step 2: Replace each variable by its paired variable:

```
H' = ↓merge(↓[↓X?|Xs?], Ys?, ↑[↑X|Zs])
```

Step 2 ensures each variable's form matches its position's structural mode. For simple input/output types, this typically means replacing all variables: head writers at input positions become readers (serving as input—bound by the goal), while head readers at output positions become writers (serving as output—will be bound by the body). For interactive types with internal mode complementation, some variables may already have the correct form and require no change.

Remark 4.10 (Mode Correspondence). *Let H' be a moded head constructed per Definition 4.8 from head H and type declaration D. For any term path $x \in paths(H')$ ending at position p, and corresponding type path $y \in paths(D)$ at the same position p, the structural mode at p in x equals the mode at p in y.*

This follows directly from the construction: Definition 4.8 assigns modes to each position in H' based on the argument types in D. Input arguments (T?) receive mode ↓, output arguments (T) receive mode ↑, and nested positions inherit modes according to the type structure.

The same property holds for body atoms: each body atom A is constructed as a produced moded term with mode ↑ throughout, matching the type's expectation that body atoms are produced.

*Consequently, when checking path consistency (Definition **??**) for a moded head or body atom, the "mode of the corresponding type*

symbol" can be read directly from the term path—no separate type path traversal is required for mode information.

With these notions, we can define when a clause is well-typed. The definition has three components:

Definition 4.11 (Well-typed, input accepting clause). *Let $C = (H :- B)$ be a GLP clause and D a GLP type for all its procedures. Then C is **well-typed** by D if:*

1. *There is a moded head H' corresponding to H that is well-typed by D.*
2. *For each atom $A \in B$, the produced moded term A' corresponding to A is well-typed by D.*
3. *Every pair of variables that occur in C are assigned complementary types by D.*

*In addition, C **accepts** an input path $x \in paths(D)$ if H' has a path consistent with x.*

Example 4.12 (Well-typed Clause). *We verify the first* `merge` *clause is well-typed:*

```
merge([X|Xs], Ys, [X?|Zs?]) :- merge(Ys?, Xs?, Zs).
```

Condition 1 (Head well-typed): *The moded head from the previous example is:*

```
H' = ↓merge(↓[↓X?|Xs?], Ys?, ↑[↑X|Zs])
```

Each path in H' is consistent with a path in $paths(D)$, as shown above.

Condition 2 (Body atoms well-typed): *The body atom* `merge(Ys?, Xs?, Zs)` *as a produced moded term is:*

```
↑merge(Ys?, Xs?, Zs)
```

The paths are consistent with the type `merge(Stream?,Stream?,Stream)`*:* `Ys?` *and* `Xs?` *are readers at consumed positions (*`Stream?`*), and* `Zs` *is a writer at a produced position (*`Stream`*).*

Condition 3 (Complementary variable types): *The variable pairs and their types are:*

- `X`/`X?`*:* `X` *has type _ (produced),* `X?` *has type _? (consumed)—complements.*
- `Xs`/`Xs?`*:* `Xs` *has type* `Stream` *(from head arg 1 tail),* `Xs?` *has type* `Stream?` *(from body arg 2)—complements.*
- `Ys`/`Ys?`*:* `Ys` *has type* `Stream?` *(from head arg 2),* `Ys?` *has type* `Stream` *(from body arg 1)—complements.*
- `Zs`/`Zs?`*:* `Zs` *has type* `Stream` *(from body arg 3),* `Zs?` *has type* `Stream?` *(from head arg 3 tail)—complements.*

All conditions are satisfied, so the clause is well-typed.

Definition 4.13 (Well-typed GLP program). *A typed GLP program $P = (Cs, D)$ is **well-typed** if:*

1. ***Covariance:** Every clause $C \in Cs$ is well-typed by D.*
2. ***Contravariance:** Every input path in every procedure type in D has a clause $C \in Cs$ that accepts it.*

4.4 Type Complementarity

Well-typing requires that paired variables have complementary types: if `X` has type T then `X?` must have type $T?$. But this requirement can be relaxed when the programmer accepts that a reader may encounter fewer alternatives than the type permits.

Consider a file system monitor that handles `read`, `write`, and `delete` operations:

```
FileOp ::= read(Path?,Content)
         ; write(Path?,Content?)
         ; delete(Path?).

procedure fs_monitor(Stream(FileOp)?).
```

A read-only client wishes to interact with this monitor, but only issues `read` operations:

```
ReadOp ::= read(Path?,Content).

procedure ro_client(Stream(ReadOp)).
```

Connecting these directly violates the standard complementarity requirement: the monitor expects `Stream(FileOp)?` but the client produces `Stream(ReadOp)`, and `ReadOp` ≠ `FileOp?`. Yet this interaction is safe: the client produces a subset of what the monitor accepts, and the monitor's responses (the `Content` in `read` operations) are exactly what the client expects.

This motivates *type complementarity*, a relation between types that captures when their interaction is safe despite not being exact complements.

Definition 4.14 (Type Complementarity). *Let S and T be GLP types. We say S is **complementary to** T, written $S \trianglelefteq T$, if for every moded path in $paths(S)$, one of the following holds at each position along the path:*

1. *At positions with mode produce ↑: the alternative at that position in S is among the alternatives at the corresponding position in T.*
2. *At positions with mode consume ↓: the alternative at that position in T is among the alternatives at the corresponding position in S.*

In other words, at each mode complement encountered along a path, the inclusion obligation reverses: producers must stay within their declared alternatives (covariance), while consumers must accept all alternatives the other party might produce (contravariance).

Example 4.15 (Type Complementarity Analysis). *We verify* `ReadOp` $\trianglelefteq$ `FileOp?`.

The type `ReadOp` *has paths through the single alternative* `read(Path?,Content)`*. Consider the path to the first argument:*

```
(0,↑) --> ReadOp --(1,↑)--> read/2 --(1,↓)--> Path?
```

At the root (mode ↑), `ReadOp` *produces* `read`*. The complement* `FileOp?` *at this position (mode ↓) consumes any of* `{read, write, delete}`*. Since* `read` ∈ `{read, write, delete}`*, condition 1 is satisfied.*

At argument position 1 (mode ↓ after passing through the functor), the client consumes `Path?`*. The monitor at this position produces* `Path?`*. Both match exactly.*

For the second argument `Content`*:*

```
(0,↑) --> ReadOp --(1,↑)--> read/2 --(2,↑)--> Content
```

The client produces a writer expecting `Content`*; the monitor (at mode ↓) will produce* `Content` *into it. The types match exactly.*

All paths satisfy the complementarity conditions, so `ReadOp` $\trianglelefteq$ `FileOp?`.

Definition 4.16 (Complementarily-Typed Variables). *Let X and $X?$ be a variable pair in a GLP clause, with X assigned type S and $X?$ assigned type T. The pair is **complementarily typed** if $S \trianglelefteq T$.*

Type complementarity generalises exact complementation: if $S = T?$, then trivially $S \trianglelefteq T$ since every alternative in $T?$ corresponds exactly to an alternative in T with modes flipped. The relation is reflexive and transitive but not symmetric: `ReadOp` $\trianglelefteq$ `FileOp?` but `FileOp?` $\not\trianglelefteq$ `ReadOp` (the monitor might produce `write` or `delete`, which the read-only client cannot handle).

Remark 4.17 (Not Closed Under Complementation). *Type complementarity is not closed under complementation: $S \trianglelefteq T$ does not imply $S? \trianglelefteq T?$. This is because complementation flips all modes, which inverts all the inclusion obligations. In our example,* `ReadOp` $\trianglelefteq$ `FileOp?` *but* `ReadOp?` $\not\trianglelefteq$ `FileOp`*: a consumer expecting any* `FileOp` *cannot be satisfied by a producer restricted to* `ReadOp`.

Next, we establish the connection between the semantics of a well-typed program and its type.

5 GLP Semantics and Well-Typing

Similarly to LP, we can derive denotation semantic notions for the type system to be an abstraction thereof. However, the semantic objects we employ—moded terms—are richer than in LP, since we wish them to capture:

(1) **Behaviours of partial runs**, so that a well-typed program would well-behave even in runs that eventually fail or deadlock, or not terminate.
(2) **Interaction with the environment**, namely whether values to variables were produced or consumed during a run. This would allow the type system to express both covariance—all values produced are in the type, and contravariance—any input within the type may be consumed by the program.

First, we define how a substitution applied to an initial input or output term defines a moded term.

Definition 5.1 (Substitution-induced Moded Term). *Given a moded term T and a substitution σ, the **moded term** $T\sigma$ is obtained from T by repeating the following to completion: For each reader $X? \in T \cap \mathrm{dom}\sigma$, replace $X?$ by the consumed term $X?\sigma$. For each writer $X \in T \cap \mathrm{dom}\sigma$, replace X by the produced term $X\sigma$.*

Example 5.2 (Substitution-induced Moded Term). *Consider the produced moded term $T =\uparrow$ `merge(Xs?,Ys?,Zs)` and substitution $\sigma =$ {`Zs := [1|Zs1]`}. Applying σ to T:*

- *Writer* `Zs` $\in T \cap \mathrm{dom}\,\sigma$*, so replace* `Zs` *by the produced term* `[1|Zs1]`

The result is $T\sigma =\uparrow$ `merge(Xs?,Ys?,`$\uparrow$ `[`$\uparrow$ `1|Zs1])`, where the substituted subterm inherits the produce mode from the position.

Next, we define the moded outcome of a run.

Definition 5.3 (Moded-atoms outcome and moded-resolvent of a run; their well-typing). *Given a GLP run of a typed GLP program $P = (Cs, D)$ with outcome $(G0 : -G, \sigma)$, let $G0'$ be the produced moded term corresponding to $G0$. The **moded-atoms outcome** of the run are the moded-terms $H\sigma$ for every atom $H \in G0'$. The **moded resolvent** of the run is the set of produced moded atoms G', with each $A' \in G'$ corresponding to an atom $A \in G$. We say that the initial goal $G0$ and the resolvent G are well-typed if $G0'$ and G', respectively, are well-typed by D.*

For example, consider the `merge` program above, the two programs below:

```
copy([X|Xs],[X?|Ys?]) :- copy(Xs?,Ys).
copy([],[]).
```

and the goal:

```
copy([1,2,3|Xs?],Xs1),
copy([a,b|Ys?],Ys1),
merge(Xs1?,Ys1?,Zs).
```

A complete run of this goal and program would end with an outcome similar to:

```
(((copy([1,2,3|Xs?],Xs1),
    copy([a,b|Ys?],Ys1),
    merge(Xs1?,Ys1?,Zs)) :-
     copy(Xs?,Xs2), copy(Ys?,Ys2), merge(Xs2?,Ys2?,Zs1))
{Xs1 := [1,2,3|Xs2?],
    Ys1 := [a,b|Ys2?],
    Zs := [1,a,2,b,3|Zs1]})
```

And the moded goals will be:

```
copy(↓[↓1,↓2,↓3|Xs?], ↑[↑1,↑2,↑3|Xs2])
copy(↓[↓a,↓b|Ys?],↑[↑a,↑b|Ys2])
merge(↓[↓1,↓2,↓3|Xs2?],↓[↓a,↓b|Ys2?],↑[↑1,↑a,↑2,↑b,↑3|Zs1]))
```

And moded resolvent will be:

```
↑copy(Xs?,Xs),
↑copy(Ys?,Ys2),
↑merge(Xs2?,Ys2?,Zs1)
```

Theorem 5.4 (Well-Typing of Outcomes). *Given a GLP run of a well-typed GLP program with outcome $(G0 : -G, \sigma)$, if $G0$ is well-typed then the moded-atoms outcome of the run and the resolvent G are well-typed.*

Proof sketch. By induction on the length of the run. The base case (empty run) is immediate: $G0' = G'$ and $\sigma = \emptyset$.

For the inductive step, consider a single reduction of goal A with clause $H : -B$. By well-typing of the clause (Definition 4.11):

(1) The moded head H' is well-typed by D
(2) Each body atom in B is well-typed as a produced moded term
(3) Variable pairs have complementary types

The writer mgu $\hat{\sigma}$ matches A with H, binding writers to terms. Since A (from a well-typed goal) and H' (from a well-typed clause) are both well-typed, and the substitution respects the mode structure, the resulting moded terms $A\hat{\sigma}$ and $B\hat{\sigma}$ remain well-typed. The complementary types condition ensures that when a writer is bound, its paired reader receives a value of compatible type.

By contravariance, every input path in D has an accepting clause, ensuring that well-typed inputs can always be consumed by some clause. Thus well-typing is preserved through each reduction step. □

Definition 5.5 (Moded-Atom Semantics). *The **moded-atom semantics** of a typed GLP program P is the set of all moded-atoms outcomes of every run of P with a well-typed initial goal.*

Corollary 5.6 (Well-Typing of Semantics). *The moded-atom semantics of a well-typed GLP program is well-typed.*

Proof. Immediate from Theorem 5.4: every element of the moded-atom semantics is a moded-atoms outcome of some run with a well-typed initial goal, hence is well-typed. □

6 Related Work

Concurrent Logic Programming. GLP belongs to the family of concurrent logic programming (CLP) languages that emerged in the 1980s: Concurrent Prolog [17], GHC [20], and PARLOG [5]. These languages interpret goals as concurrent processes communicating through shared logical variables, using committed-choice execution with guarded clauses.

A key evolution was *flattening*: restricting guards to primitive tests only. Flat Concurrent Prolog (FCP) [12] and Flat GHC [20] demonstrated that flat guards suffice for practical parallel programming while dramatically simplifying semantics and implementation. Shapiro's comprehensive survey [18] documents this family and its design space.

GLP can be understood as **Flat Concurrent Prolog with the Single-Reader Single-Writer (SRSW) constraint added**. FCP introduced read-only annotations (?) distinguishing readers from writers of shared variables, enabling dataflow synchronization. However, read-only unification proved semantically problematic: Levi and Palamidessi [11] showed it is order-dependent, and Mierowsky et al. documented non-modularity issues. GHC dispensed with read-only annotations entirely, relying on guard suspension semantics.

GLP's SRSW constraint—requiring that each variable has exactly one writer and one reader occurrence—resolves these difficulties by ensuring that (1) no races occur on variable binding, and (2) term matching suffices, eschewing unification entirely. The result is a cleaner semantic foundation while preserving the expressiveness of stream-based concurrent programming.

Modes in Concurrent Logic Programming. Mode systems for CLP have a rich history. PARLOG used mode declarations at the predicate level, with input modes enforcing one-way matching where clause variables can be instantiated but goal variables cannot be bound.

Ueda's work on moded Flat GHC [21, 23] is most directly relevant to GLP. His mode system assigns polarity to every variable occurrence: positive for input/read capability, negative for output/write capability. The *well-modedness* property guarantees each variable is written exactly once—a single-writer constraint matching half of GLP's SRSW requirement. Ueda's subsequent linearity analysis [22] identifies variables read exactly once, enabling compile-time garbage collection for single-reader structures.

GLP enforces both single-reader and single-writer universally as a syntactic restriction, whereas Ueda's system guarantees single-writer with single-reader as an optional refinement. This stronger restriction enables GLP's simpler execution model. Our contribution is a *type system* for this discipline, building on LP types rather than mode inference.

Types for Logic Programs. The type system for GLP builds on the Frühwirth-Shapiro framework [6], which defines LP types as regular sets of paths and characterizes well-typing via path abstraction of the ground-atom semantics. The companion paper [26] establishes the semantic foundations through tuple-distributive sets. Our moded types extend this framework by augmenting paths with mode annotations that capture the distinction between consumed and produced subterms.

Earlier work on LP types includes Mycroft and O'Keefe's prescriptive polymorphic type system [14] and Mishra's foundational work on tuple-distributivity [13]. Mercury [19] provides a comprehensive practical system separating type checking, mode checking, and determinism inference. Our work differs in integrating modes directly into the type structure rather than treating them as a separate analysis.

Linear Logic and Futures. GLP's SRSW discipline is reminiscent of linear logic's [9] resource sensitivity: each variable binding is produced at most once and consumed at most once. A GLP reader/writer pair is operationally a future or promise [1]: the writer creates a promise for a value, the reader awaits fulfillment. Unlike typical futures, GLP futures can carry logical terms containing further reader/writer pairs, enabling recursive communication structures.

Session types [10, 24] use linear logic to type communication protocols. GLP's reader/writer pairs can be viewed as degenerate session types: a single-message protocol with one send and one receive. The key difference is that GLP types are defined as regular sets of moded paths, inheriting the automata-theoretic foundations of LP types, rather than as session type judgments derived from linear logic sequents.

Type Complementarity and Session Subtyping. Session type theory distinguishes two notions: *duality* and *subtyping*. Duality [10] is an involution $S \mapsto \overline{S}$ that swaps input and output modalities while leaving message types unchanged; dual endpoints have perfectly matching protocols. Subtyping [8] is a preorder for safe substitution: $S <: T$ if a channel of type S can be used where T is expected. Crucially, session subtyping is contravariant in input types and covariant in output types—the variance arises from applying subtyping rules recursively through the session structure.

Type complementarity differs from both. Unlike duality, it is not an involution and permits asymmetric interaction: a client may use a subset of a server's operations. Unlike session subtyping, where variance emerges from rule application on judgments with explicit input/output modalities, type complementarity builds the direction-flipping directly into the relation itself. When traversing two GLP types simultaneously, encountering a mode complement reverses the inclusion obligation at that position. This is a definitional primitive, not a derived property.

The practical consequence is that type complementarity can express safe compositions that session type duality cannot (asymmetric subsets) using a mechanism distinct from session subtyping (moded paths rather than polarized judgments). Systems like SILL_S [2] and CLASS [16] extend session types with shared state, addressing related concerns through different mechanisms: explicit acquire/release protocols rather than mode-directed inclusion reversal.

7 Conclusion

We have developed a type system for Grassroots Logic Programs (GLP), a concurrent programming language that partitions logic variables into complementary readers and writers, restricts each to a single occurrence, and eschews unification in favour of term

matching. This design ensures that each variable binding is produced at most once and consumed at most once.

Our *moded types* extend LP types [6] with mode annotations that capture the distinction between values produced by the program and values consumed from the environment. Types are regular sets of moded paths, specified via BNF rules with a mode complementation operator $(\cdot)?$ that uniformly relates producer and consumer views of the same communication channel.

Well-typing is defined to capture both covariance—all values produced conform to the type—and contravariance—the program accepts all inputs permitted by the type. Theorem 5.4 establishes that well-typing is preserved through GLP computation: if a program and initial goal are well-typed, then all moded-atoms outcomes and resolvents remain well-typed.

We introduced *type complementarity*, a relation between types that determines when their interaction is safe even when types are not exact complements. Unlike standard subtyping where contravariance emerges from recursive rule application, and unlike session type duality which requires exact protocol matching, type complementarity builds inclusion-direction-flipping directly into the relation. This enables asymmetric client/server interactions—such as a read-only client interacting with a full-access file system monitor—that neither standard subtyping nor session type duality can express.

Future Work. Several directions remain for future investigation:

- *Polymorphic moded types*: Extending the framework to handle parametric polymorphism, following [26].
- *Implementation*: Integrating moded type checking into the GLP compiler pipeline.
- *Gradual typing*: Supporting partially-typed programs where some predicates have moded type declarations and others do not.
- *Inference*: Developing algorithms for inferring moded types from unannotated programs.
- *Multiparty complementarity*: Extending type complementarity to handle multiple clients with different capability subsets interacting with a shared server.

References

[1] Henry G. Baker and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59. ACM, 1977.

[2] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):37:1–37:29, 2017.

[3] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR)*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010.

[4] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–523, 1985.

[5] Keith Clark and Steve Gregory. Parlog: parallel programming in logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(1):1–49, 1986.

[6] Thom Frühwirth, Ehud Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 300–309. IEEE Computer Society, 1991.

[7] Haim Gaifman and Ehud Shapiro. Fully abstract compositional semantics for logic programs. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 134–142, 1989.

[8] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

[9] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[10] Kohei Honda. Types for dyadic interaction. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR)*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[11] Giorgio Levi and Catuscia Palamidessi. The declarative semantics of read-only annotations in logic programming. *Proceedings of the 1985 Symposium on Logic Programming*, pages 212–220, 1985.

[12] C. Mierowsky, S. Taylor, E. Shapiro, J. Levy, and M. Safra. On the implementation of flat concurrent prolog. *Proceedings of the 1985 Symposium on Logic Programming*, pages 276–286, 1985.

[13] Prateek Mishra. Towards a theory of types in prolog. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 289–298. IEEE, 1984.

[14] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. In *Proceedings of the 1983 Logic Programming Workshop*, pages 107–122. Universidade Nova de Lisboa, 1984.

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

[16] Pedro Rocha and Luís Caires. Propositions-as-types and shared state. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–30, 2021.

[17] Ehud Shapiro. A subset of concurrent prolog and its interpreter. *ICOT Technical Report, TR-003*, 1983.

[18] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys (CSUR)*, 21(3):413–510, 1989.

[19] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of mercury: An efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[20] Kazunori Ueda. Guarded horn clauses. In *Logic Programming '85*, volume 221 of *Lecture Notes in Computer Science*, pages 168–179. Springer, 1986.

[21] Kazunori Ueda. Moded flat ghc and its message-oriented implementation technique. *New Generation Computing*, 12(4):337–368, 1994.

[22] Kazunori Ueda. Resource-passing concurrent programming. *Proceedings of TACS 2001*, pages 95–126, 2001.

[23] Kazunori Ueda and Masao Morita. I/o mode analysis in concurrent logic programming. In *Proceedings of the International Symposium on Theory and Practice of Parallel Programming*, pages 356–368. Springer, 1995.

[24] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 273–286. ACM, 2012.

[25] E YARDENI, T FRUEHWIRTH, and E SHAPIRO. Polymorphically typed logic programs. In *Logic Programming: Proceedings of the Eighth International Conference*, pages 379–393. MIT Press, 1991.

[26] Eyal Yardeni and Ehud Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–153, 1991.