Semantics-Based Analysis for Optimizing Compilation of Concurrent Programs

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree
of

Doctor of Information and Computer Science

Graduate School of Science and Engineering
Waseda University

December 2005

Norio Kato

Keywords: Concurrent Logic Programming, Static Analysis, Concurrent Programming, Hierarchical Graph Rewriting, Optimization.

Abstract

Concurrent programming languages provide a paradigm for describing programs with complicated communication networks in a clear way. They support the creation of multiple computational units called *processes* that run concurrently in a program together with the mechanism for communication between processes, thereby simplifying the programming of synchronization and communication. Among them, concurrent languages that support fine-grained processes are suitable for formal specification of concurrent systems.

A runtime system of a fine-grained concurrent language, however, tends to suffer from overheads of primitive operations needed for handling concurrency. Although there is a lot of work on static analysis and optimization techniques for removing these overheads, comprehensive research on the safety of applying multiple optimization techniques has been limited in its scope. For instance, no frameworks have existed that can justify the correctness of static scheduling of concurrent processes *and* memory reuse. The point is that some issues of optimization cannot be fully expressed as program transformation on the original concurrent language. When these issues are not described formally, the optimization must be carried out on the responsibility of implementors of an optimizing compiler and hence there is no proof that the optimization is really correct.

The objective of this dissertation is to clarify (a) how to perform semantics-based runtime system optimization for concurrent programming languages and (b) how to give theoretical justification to such optimization. To this end, we take two concrete target languages—the concurrent logic programming language and the concurrent graph rewriting language LMNtal (pronounced as *elemental*)—and discuss for these languages several methods to perform static analysis and runtime system optimization based on the analysis.

In this dissertation, several semantics-based program analysis techniques are proposed for optimizing compilation. For concurrent logic programs, the following will be explained: the safety of moving synchronization points as semantics-preserving program transformation for process scheduling, sequentiality analysis and code generation as a justified framework of applying multiple optimization techniques, and occurs-check analysis under cooperative

modings for runtime system optimization. After that, we will discuss process structure analysis for the concurrent graph rewriting language LMNtal by introducing a type system useful for runtime system optimization.

Acknowledgments

I thank my supervisor Prof. Kazunori Ueda for encouraging me to publish this dissertation. Of course, there has been more than encouragement. His motivative suggestion of interesting research topics in the field of analysis and optimizing compilation of concurrent languages was essential for completing each chapter of this dissertation. The author also thanks the thesis committee members, Prof. Katsuhiko Kakehi, Prof. Shigeki Goto, et al., for their patient support to improve the thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Concurrent programming languages provide a paradigm for describing programs with complicated communication networks in a clear way. They support the creation of multiple computational units called *processes* that run concurrently in a program and provide the mechanism for communication between processes, thereby simplifying the programming of synchronization and communication. In particular, we are interested in concurrent languages designed in a close relationship with a formally defined language model since concurrent programs written in such languages provide formal specification of the concurrent systems that can be analyzed precisely. Concurrent languages that support fine-grained processes are suitable for this purpose since they enable us to uniformly describe computation as interaction between processes. These languages include process calculi such as the pi-calculus, lazy functional programming languages, functional languages with future primitives, concurrent logic languages, and concurrent object-oriented languages. Moreover, since concurrent languages offer a framework for describing parallel and distributed systems in a cleaner semantics, they are expected to provide foundations for parallel and distributed programming languages where more and more study is being required lately both in theory and in practice.

A runtime system of a fine-grained concurrent language on sequential machines, however, tends to suffer from overheads of primitive operations needed for handling concurrency, which is not the case with a simple concurrent extension of a sequential language by threads because most of the

primitive operations are designed in a sequential setting. There is a lot of work on static analysis and optimization techniques for removing these overheads, including schedule analysis and data representation optimization (see for example the papers [13, 14, 23]). However, comprehensive research on the safety of applying multiple optimization techniques so far has been limited in its scope. For instance, no frameworks have existed that can justify the correctness of static scheduling of concurrent processes *and* memory reuse. As a more specific example, the paper [15] discusses the safety of multiple program transformation techniques, which as a whole introduces a justified unfold/fold transformation system for concurrent constraint programs, but the scope of the optimization covered by their work is limited in that it does not focus on data representation. The point is that some issues of optimization, including static scheduling and memory reuse, cannot be fully expressed as program transformation on the original concurrent language. When these issues are not described formally, the optimization must be carried out on the responsibility of implementors of an optimizing compiler and hence there is no proof that the optimization is really correct.

1.2 Objectives

The objective of this dissertation is to clarify (a) how to perform semantics-based runtime system optimization for concurrent programming languages and (b) how to give theoretical justification to such optimization. To this end, we take two concrete target languages—the concurrent logic programming language and the concurrent graph rewriting language LMNtal (pronounced as *elemental*)—and discuss for these languages several methods to perform static analysis and runtime system optimization based on the analysis.

Concurrent logic programming is oriented to concurrent symbol manipulation. In particular, it allows us to describe in a flexible way complex message protocols among concurrent processes as well as dynamic process networks found in a distributed system. All of these characteristics come from the expressive power of logic variables used for representing communication channels among concurrent processes. Moreover, the single-assignment property of logic variables enables us to perform theoretical analysis of concurrent logic programs elegantly.

1.3 Contributions

The contribution of this dissertation is as follows. First, we provide the results on three topics on static analysis of concurrent logic programs:

- the safety of the motion of synchronization points,
- sequentiality analysis, and
- occurs-check analysis under cooperative modings.

After that, we will discuss:

- process structure analysis for the concurrent language LMNtal.

Details of each of these topics are explained in the following subsections.

1.3.1 On the Safety of Moving Synchronization Points

In concurrent logic programming languages, processes synchronize by means of input and output on a global store. Any output from a process is accumulated in the store while any input to a process is looked up in the store. Most processes in concurrent logic programs or more generally in concurrent constraint programs have a *restartability* property, meaning that once they can proceed they can also proceed at any future time. Hence, the execution of a process that requires some input in order to produce any output may be deferred until the necessary input becomes available. This kind of static scheduling helps us to determine the order of computation within a process and hence can be applied to optimizing compilation that reduces runtime overheads. The program transformation that achieves this kind of scheduling is called *the motion of synchronization points*. In order to implement this, it is necessary to analyze what input is required by a process for that process to produce any output. In the nondeterministic paradigm like concurrent logic programming, it might seem hard to analyze this precisely.

In this work, we present an abstract interpretation method to find out which of the variables contained in a process should be instantiated (i.e., assigned a value) prior to its execution. For any *good* process, namely a process without the possibilities of divergence (infinite computation) and failure (inconsistent output), we have proved that the semantics of the process does not change after moving synchronization points.

The main contribution of this work is on the definition of the semantics suitable for justifying program transformation. There are two results. Firstly, the denotational semantics of a process we will define has a direct correspondence with the operational semantics, which facilitates the proofs of process equivalence. Secondly, we define the denotational semantics so that it takes account of not only the interaction sequences of input and output on the store but also termination, divergence, and failure, which reflects our observation that doing so is desirable for the purpose of justifying the program transformation. These results will be explained in Chapter 3.

Another contribution is the construction of the abstract domain suitable for analyzing the motion of synchronization points. The abstraction of the constraint store is the set of possibly instantiated variables occurring in a goal. We will also give a theoretical account on what happens if our algorithm is applied to a process with divergence or failure. These results will be explained in Chapter 4.

1.3.2 Sequentiality Analysis for Concurrent Logic Programs

In this work, we present a bottom-up method of extracting those fragments of concurrent logic programs that can be executed sequentially, and propose a framework of optimizing compilation of concurrent logic programs that uses sequential intermediate code generated through the extraction of sequentiality. The extraction of sequentiality is directed by the inference of an *interface* of a process, where an interface represents a possible behavior of a process under some class of input. Using interfaces, we can systematically analyze a specialized way of the execution of given processes.

Although the specialization of an agent by sequentialization using the notion of interfaces could be axiomatized as a type system not in terms of the operational semantics, this work proposes formalizing an interface of an agent in terms of the operational semantics, which enables us to directly justify the inference of interfaces that takes place in the program analysis. The way of formalizing types in this way for justifying process specialization is one of the contributions of this work.

Our framework of optimizing compilation proposes that sequentialization be performed first to generate sequential intermediate code, and then other optimization including source-level transformation, such as copy propagation,

and implementation-level optimization, such as tag elimination and update-in-place optimization, be performed on the generated sequential code. If a formal semantics of the sequential language is given, we can prove the correctness of the implementation-level optimization techniques that cannot be formally proved by source-level analysis on the original concurrent program.

Although the formalism of interfaces exploits the constraint-based communication feature of concurrent logic programming, our method can also be applied to extracting sequentiality in other fine-grained concurrent languages including functional languages with futures.

1.3.3 Occurs-Check Analysis under Cooperative Modings

In this work, we present a mode-based static occurs-check algorithm that can handle logic programs with bidirectional communication between goals. Occurs-check is a task in unification that determines whether the unification generates an infinite structure. This check is required for justifying the soundness of a runtime system of logic programming as a prover of first-order logic as well as for guaranteeing the termination of recursive predicate calls.

Our algorithm attempts to prove the NSTO (Not Subject To Occurs-check) property of a given program, which is a sufficient condition for guaranteeing that the program does not build any infinite structures in the program execution and hence runtime occurs-check can be safely omitted. The central part of our analysis is formulated as a constraint satisfaction problem on mode information. Modes express the direction of information flow. We assume that the program is cooperatively moded, that is, every variable in a program clause has exactly one output occurrence, which is a relatively reasonable requirement in the concurrent logic programming language without atomic tells. We exploit this assumption to prove the correctness of our analysis. Our algorithm requires that the head of every program clause is linear, namely, every variable occurs at most once in a clause head. We explain how to transform a cooperatively-moded concurrent logic program into a linear-headed program so that our algorithm is applicable.

In this work, the program execution is abstracted as a *connection graph*, which expresses the equality between variables that have been introduced so far. Every infinite structure is represented as a *cycle* in the connection graph and each cycle is initiated by an *initial graph*, which is a subset of a

connection graph and expresses the multiple body occurrences of variables in a clause. Our algorithm attempts to ensure that there is no cycle by reducing every initial graph to the empty graph. We can reduce initial graphs by strengthening the mode information with asymptotic equality constraints.

The main contribution of this work is that our algorithm can prove the NSTO property of a program with bidirectional communication between goals, which cannot be handled by existing analysis. Technically, this result relies on the formalism of mode information originated from the paper [34] which takes account of not only top-level mode information of predicate arguments but also *deeper* mode information, namely mode information inside the terms passed as predicate arguments.

1.3.4 Process Structure Analysis for LMNtal

This work explains how to apply the above-mentioned mode analysis for concurrent logic programs that handles deeper mode information [34] to a hierarchical graph rewriting language, LMNtal, in order to obtain a type system useful for optimizing compilation. The type system introduced expresses several static properties on process structures formed by graph nodes, including the direction of information flow, and what kind of graphs can be connected to certain graph nodes. Since LMNtal is a relatively simple language in the sense that it has no syntactic distinction between processes and data—both of these are just represented as graph nodes called *atoms*—we need to reconstruct the distinction between them so that the type system has practical usefulness. To this end, we introduce a typing scheme that makes this distinction explicit, which is one of the novelties of this work.

The main contribution of this work is to present a useful type system for a graph rewriting language where the types are reconstructed during type inference, rather than they are assumed to be given prior to type checking. In our type system, a programmer only needs to classify atoms into active atoms and data atoms in order to obtain by inference a type graph that describes complex process structures in the program.

LMNtal provides a feature called *membranes* that allows us to group atoms in a graph structure and to localize computation to each membrane. Membranes can be arbitrarily nested so that they form a hierarchical graph. Another contribution of this work is to type a nested graph.

Properties of the type system, including type safety, will be shown.

1.4 Overview of the Thesis

The rest of this dissertation is organized as follows.

Chapter 2 defines a concurrent logic programming language we will work on throughout the dissertation. The operational semantics of the concurrent logic programming is given by means of that of CCP (concurrent constraint programming), which is a generalization of concurrent logic programming. To do this, the syntax and semantics of CCP will also be explained. Chapter 3 formalizes a denotational semantics of CCP, which is required for the theoretical discussion on the safety of program transformation. Chapter 4 explains the program transformation for concurrent logic programs called the motion of synchronization points, introduces a fixed-point abstract computation algorithm for obtaining the program transformation, and then justifies its correctness in terms of the denotational semantics. Chapter 5 explains sequentiality analysis for concurrent logic programs and its application to bottom-up code generation.

Chapter 6 introduces a moding system of concurrent logic programs and shows an algorithm for static occurs-check under cooperative modings.

Chapter 7 explains process structure analysis for the concurrent graph rewriting language LMNtal.

Chapter 8 gives concluding remarks.

Chapter 2

Concurrent Logic Programs

This chapter defines a concurrent logic programming (CLP) language we will work on throughout this dissertation (except for Chapter 7 where another concurrent language based on hierarchical graph rewriting, LMNtal, is dealt with). In concurrent logic programming, computation is essentially a series of asks and tells of constraints with a constraint store that is strengthened monotonically in the course of program execution. We adopt the formalism of concurrent constraint programming (CCP) [32], which is a generalization of concurrent logic programming, to describe the behavior of a process of concurrent logic programming, for CCP has a mathematical characteristics more suitable for proof description than concurrent logic programming [9].

2.1 Syntax of CLP (Concurrent Logic Programming)

To begin with, we introduce the syntax of concurrent logic programming (CLP).

Definition 2.1 The syntax of CLP is defined in Figure 2.1. There, *terms* and *variables* are defined the same as those in first-order logic. *Goals* are parallel composition of unification goals $s = t$ and predicate calls $p(t)$ where s and t are terms and p is taken from a known set of *predicate symbols*. We assume that parallel composition is commutative and associative.

A *program* P of CLP is defined as a set of rewrite rules $(\, p(s) \ :\!- \ e \mid G \,)$ called *clauses* where $p(s)$, e, and G are referred to as their head, guard, and

Term	s, t	$::= X \mid f(t_1, \ldots, t_n)$	
Goals	G	$::= s = t$	(unification)
		$\mid\ p(t)$	(predicate call)
		$\mid\ G \parallel G$	(parallel composition)
Program	P	$::= \mid P \,.\, P \mid (\, p(s)$ `:-` e `|` $G\,)$	(clauses)
Askable Constraints			
	e	$::= e \wedge e \mid s = t \mid$ `wait(`X`)` $\mid$ `int(`X`)` $\mid s$ `>=` $t \mid s$ `<` $t \mid \ldots$	

Figure 2.1: Syntax of concurrent logic programming

body, respectively. □

A unification goal $s = t$ is to *bind* variables in s or t with terms so that s and t express the same term. For example, `X = 1` binds to `X` the term `1`.

A clause rewrites predicate calls to goals. The head of a clause works as a template of pattern matching and the guard specifies an additional condition for rewriting with the clause. A constraint that can be written in a guard is called an *askable constraint*. Askable constraints must at least include 'equality constraints' of the form $s = t$ specifying that the terms s and t are the same. Chapter 4 requires that askable constraints also include the *synchronization constraint* `wait(`X`)` specifying that the variable X is bound to a non-variable term, in order to represent the motion of synchronization points. Askable constraints can also include other conditional tests such as integer comparison (e.g. s `>=` t) and runtime typechecking (e.g. `int(`X`)`).

In the concrete syntax, we will use commas (`,`) instead of $\parallel$ to describe parallel composition, and we may abbreviate $t = t$ to `true`.

Example. Figure 2.2 shows a concurrent logic program. The two processes `intlist(1,N,S)` and `sum(S,X0,X)` proceed concurrently, that is, in parallel or by interleaving, in this program. The final result of a goal `stair(10,0,X)` should be `X = 45`. □

Informal Operational Semantics.

In the following paragraphs, we will introduce an informal operational semantics of this language in order to give an idea of how it works. The formal

```
stair(N,X0,X)   :- true  | intlist(1,N,S), sum(S,X0,X).
intlist(K0,N,S) :- K0>=N | S=[].
intlist(K0,N,S) :- K0< N | S=[K0|S1], K:=K0+1, intlist(K,N,S1).
sum([],   X0,X) :- true  | X=X0.
sum([E|S],X0,X) :- true  | X1:=X0+E, sum(S,X1,X).
```

Figure 2.2: A concurrent logic program

definition will be given by means of concurrent constraint programming that will be defined later.

Before that, we define substitutions as usual.

Definition 2.2 A *substitution* is a mapping from variables to terms. For a substitution σ and a syntactic object A (such as a term or goals), $A\sigma$ denotes the result of application of σ to A. For substitutions σ and θ, $\sigma\theta$ stands for the substitution satisfying $A(\sigma\theta) = (A\sigma)\theta$ for all A where $=$ is the syntactic equality. □

In concurrent logic programming, the ordering of goals has no meaning. A goal can be reduced whenever possible.

In the informal operational semantics, the store is expressed as a substitution. The configuration of the informal reduction system is a pair $\langle G, \sigma \rangle$ of goals G and a substitution σ. The initial store is an empty substitution. We have only two reduction rules depending on the goal to be reduced.

A goal $p(t)$ reduces as follows:

$$\langle p(t) \parallel G, \sigma \rangle \longrightarrow \langle A\theta \parallel G, \sigma \rangle$$

where there exist a clause $(\, p(s) \texttt{ :- } e \texttt{ | } A \,)$ and a substitution θ such that both of $s\theta = t$ and $e\theta$ hold in the pre-defined constraint system of askable constraints. Here, θ must replace for every variable occurring in (e, A) and not in s by a fresh, distinct variable.

Notice that we require $s\theta = t$ rather than $s\theta = t\theta$ that would be the case in other logic programming languages including Prolog. This reflects the setting that the head of a clause works as a template of pattern matching and does not produce any output.

On the other hand, a goal $s = t$ reduces as follows:

$$\langle s = t \parallel G, \sigma \rangle \longrightarrow \langle G\theta, \sigma\theta \rangle$$

where θ is an mgu of s and t. The reader can see that the store accumulates the effect of the unification goals reduced so far. Recall that for terms s and t we say that a substitution θ is an *mgu* (most general unifier) of s and t if and only if $s\theta = t\theta$ and for each σ such that $s\sigma = t\sigma$ there exists τ such that $\sigma = \theta\tau$.

There are two cases where no reductions are possible. When some goals remain but none of them can be reduced, the computation is in deadlock. On the other hand, when all goals are reduced, the computation *terminates* and the store contains the binding information on all the variables that have occurred so far. Since most of them are local variables and in reality of no interest, we should consider the restriction of the store to a variable set of interest.

Why We Will Use CCP.

While the informal semantics explained above is concise and suitable for symbolic, rapid prototype implementation, it is not suitable for analyzing various properties of concurrent processes including the properties on interaction between goals and termination. This is mainly because the semantics provides no means to group the distributed components of a process, that is, subprocesses and local channels, originally written as only one or a few predicate calls in the program text. Another reason is that a rewriting of a predicate call is composed of too many tasks—namely, predicate definition lookup, head matching, guard testing, variable renaming, fresh variable introduction, and replacement of a call with a clause body—which makes a formal proof difficult to grasp. Moreover, we must sooner or later define a constraint system of askable constraints in order to formulate the precise semantics of guards because most of interesting askable constraints, including `wait(`X`)` we will use in Chapter 4, cannot be directly represented as a substitution.

To give a formal operational semantics of concurrent logic programming, we shall use the CCP (concurrent constraint programming) language, for CCP has a mathematical characteristics more suitable for proof description than concurrent logic programming. We define the operational semantics of concurrent logic programming through that of CCP. The following sections define the CCP language.

Notes on Atomic Tells.

In this dissertation, we will not allow a so-called *atomic tell* operation. An atomic tell of a constraint e_0 performs a consistency check of the constraint e_0 to the store before proceeding the rest of computation. A clause with an atomic tell is commonly written as ($p(s)$:- $e : e_0 \mid A$). Here, e is a rewriting condition (same as in the original syntax of a clause) and e_0 is a comma-separated list of equality constraints executed as an atomic tell. If e_0 is consistent with the store, e_0 is added to the store atomically with the test and then the rewriting takes place; otherwise the rewriting does not take place. Since atomic tells are known to make the program semantics complicated [30], we have decided to restrict ourselves to only *eventual tells* (i.e., unification goals). This decision will be exploited in Theorem 2.1 in this chapter, which pertains to process scheduling, and in its applications in Chapters 4 and 5. On the other hand, the decision is not exploited in the other parts including Chapter 6, which presents a static occurs-check algorithm.

2.2 Constraints

In this section, we define the syntax and semantics of constraints. In the CCP formalism, constraints are used to represent a store state in place of substitutions. Unifications to be told to the store as well as askable constraints to be asked of the store are also represented as constraints. Intuitively, a constraint is a logical formula that describes certain information on variables such as binding information.

2.2.1 Lattice Structure of Constraints

Section 2.2.1 originates from the paper [32]. Recall that $(S, \leq)$ is a lattice if $\leq$ is a partial order over S with a top element and a bottom element, and that $\leq$ is a partial order if it enjoys reflexivity, antisymmetry, and transitivity.

Definition 2.3 Let $(Con, \leq)$ be a lattice with the bottom element *true* and the top element *false*. We call an element of Con a *constraint*. □

In the following, we introduce several assumptions on constraints.

Assumption 2.1 We assume that constraints can be interpreted as logical formulas and that the least upper bound $c \sqcup d$ of two constraints c and d is interpreted as their conjunction $c \wedge d$. □

By the above assumption, we can say that $c \leq d$, or equivalently $c \wedge d = d$, means that d contains more information than c (in short, d entails c). The constraint *true* represents no constraints while *false* represents inconsistency.

Assumption 2.2 We assume that the sets of *variables* and *terms* of first-order logic are given. The set of all variables is denoted by Var. We assume there is a special variable denoted by α. For any term t, we denote by $Vars(t)$ the set of variables occurring in t. □

We reserve the variable α to represent the formal parameter in a predicate definition. In fact, this makes sense because we will later define the language so that every predicate takes exactly one argument. The choice of the name of α is derived from the initial letter of 'argument'.

2.2.2 Pathed Variables

Next, we extend the definition of terms to obtain a new syntactic category which we call pathed terms.

Definition 2.4 A *pathed term* t' is defined by the following syntax:

$$\begin{array}{lll} \textit{Pathed Variables} & X' ::= X \mid X' \cdot i \\ \textit{Pathed Terms} & t' ::= X' \mid f(t'_1, \ldots, t'_n) \end{array}$$

where $X \in Var$ and $i \geq 1$ and $n \geq 0$. X' is called a *pathed variable*. □

The objective of extending terms with pathed variables is to give a concise, formal description of the meaning of unification between first-order logic terms. The pathed variable $X' \cdot i$ is intended to be interpreted as the i-th argument of (the term that has been unified with) the pathed variable X'.

Assumption 2.3 We assume that every constraint can be represented syntactically as a finite conjunction of pathed terms. □

Since we have already assumed that every constraint can be interpreted as a logical formula, it follows that a finite conjunction of pathed terms that represents a constraint must be interpreted as a logical formula. The interpretation of pathed variables has already been explained above. The objective of this assumption is to define the effect of unification by means of syntactic substitution, which will be explained in Section 2.2.4.

The syntactic representation of a constraint as a finite conjunction of pathed terms is not unique. Two representations of the same constraint may even have different sets of variables occurring in them. Given a syntactic representation c' of a constraint c, we shall say that a pathed variable X *syntactically occurs in* the constraint c if and only if X occurs in c'. Hence, in saying that a (pathed) variable syntactically occurs in a constraint, we assume a syntactic representation of that constraint.

2.2.3 Equality Constraints

Next, we define equality constraints that express the effect of unification.

Assumption 2.4 For any pathed variables X and Y, we assume that the constraint $X = Y$ is in *Con* representing the equality between them. We also assume that these constraints satisfy the following:

(U1) $(X = X) = \mathit{true}$
(U2) $(X = Y) = (Y = X)$
(U3) $c[Y/X] \leq c \wedge (X = Y)$
(U4) $(X \cdot i = Y \cdot i) \leq (X = Y)$

where X and Y are pathed variables, $c \in \mathit{Con}$, and $[Y/X]$ syntactically replaces every X by Y. □

Assumption 2.5 For any function symbol f and an integer $n \geq 0$, we call the pair of these, written as f/n, a *functor*. For any pathed variable X, function symbol f, and an integer $n \geq 0$, we assume there exists the constraint $\texttt{func}(X, f, n)$ in *Con* representing that the pathed variable X is *bound* to the functor f/n.

We will also assume that there exists the constraint $\texttt{wait}(X)$ in *Con* such that for all functors f/n it holds that $\texttt{wait}(X) \leq \texttt{func}(X, f, n)$. □

Now we can define equality constraints between pathed terms.

Definition 2.5 For pathed terms s and t, the *equality constraint* $s = t$ is defined inductively as follows:

$$\begin{array}{ll} \text{(U5)} & (X = f(t_1, \ldots, t_n)) = \mathtt{func}(X, f, n) \wedge \bigwedge_{i=1}^{n} (X \cdot i = t_i) \\ \text{(U6)} & (f(t_1, \ldots, t_n) = X) = \mathtt{func}(X, f, n) \wedge \bigwedge_{i=1}^{n} (X \cdot i = t_i) \\ \text{(U7)} & (f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)) = \bigwedge_{i=1}^{n} (s_i = t_i) \\ \text{(U8)} & (f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n)) = \mathit{false} \text{ if } f/m \not\equiv g/n \end{array}$$

where X is a pathed variable, and s_i and t_j are pathed terms. □

Equality constraints can be better seen as tellable constraints:

Definition 2.6 A constraint is a *tellable constraint* if and only if it can be represented in the following syntax:

$$\textit{Tellable Constraints} \quad c ::= c \wedge c \mid X = Y \mid \mathtt{func}(X, f, n)$$

where X and Y are pathed variables and f/n is a functor. We denote by Con_0 the set of all the tellable constraints. □

The name of tellable constraints reflects that these are the only constraints that can be told in concurrent *logic* programming.

The store c is strengthened to $e \wedge c$ by a tell of e. When the store becomes *false*, we consider that the computation *fails*. Failure can be thought of as a program error like division by zero.

2.2.4 Hiding Operator

Finally, we introduce a family of operators for restricting the scopes of variables. These operators are borrowed from the formalization of cylindric constraint systems [10, 32], which are algebraic systems for modeling variable hiding and parameter passing where these operators are called as cylindrification operators.

Assumption 2.6 For each variable X we assume that the hiding operator $\exists_X : Con \to Con$ is defined satisfying:

1. $\exists_X(c) \leq c$
2. $c \leq d$ implies $\exists_X(c) \leq \exists_X(d)$

3. $\exists_X(\exists_Y(c)) = \exists_Y(\exists_X(c))$
4. $\exists_X(c \wedge \exists_X(d)) = \exists_X(c) \wedge \exists_X(d)$
5. $\exists_X(\mathit{false}) = \mathit{false}$

□

Intuitively, $\exists_X(c)$ represents the partial constraint of c that forgets the information on X. In addition, we require the following assumption.

Assumption 2.7 The hiding operators enjoy the following properties:

(E1)	$\exists_X(X = t) = \mathit{true}$	if X does not occur syntactically in t
(E2)	$\exists_X(c) = c$	if X does not occur syntactically in c

for any variable X, any pathed term t, and any constraint c. □

(E1) states that the operator $\exists_X$ must fully hide the information on X. (E2) states that the operator $\exists_X$ must not hide any information on a variable other than X. By (E2), we have $\exists_X(c[Y/X]) = c[Y/X]$.

Example.

$$\exists_{\mathtt{A}}(\mathtt{A = f(B)}) = \mathit{true}$$
$$\exists_{\mathtt{B}}(\mathtt{A = f(B)}) = (\mathtt{A = f(\_)})$$
$$\exists_{\mathtt{C}}(\mathtt{A = f(B)}) = (\mathtt{A = f(B)})$$
$$\exists_{\mathtt{X}}(\mathtt{Y = f(X,X \cdot 4)}) = \mathtt{func(Y,f,2)} \wedge (\mathtt{Y \cdot 1 \cdot 4 = Y \cdot 2}).$$

In the example, `_` stands for an anonymous variable, which is formally defined as $C[\_] \stackrel{\text{def}}{=} \exists_X(C[X])$ where $C[\cdot]$ is a context with a *single* hole in a pathed-term position and X does not occur syntactically in $C[\cdot]$. □

The set of tellable constraints, Con_0, is closed under hiding operators.

Equality constraints and hiding operators together can model parameter passing. The following proposition demonstrates how this is done. The proposition states that the information on the variable X in a constraint c is fully transmitted to the formal parameter Y that is fresh, namely $\exists_Y c = c$.

Proposition 2.1 $\exists_X(c \wedge (X = Y)) = c[Y/X]$ if $\exists_Y c = c$.

Proof. Since

$$\begin{aligned}\exists_X(c \wedge (X = Y)) &= \exists_X(c \wedge (X = Y) \wedge (Y = X)) \\ &\geq \exists_X(c[Y/X] \wedge (Y = X)) \\ &\geq \exists_X(c[Y/X][X/Y] \wedge (X = Y)) \\ &= \exists_X(c \wedge (X = Y))\end{aligned}$$

we have

$$\begin{aligned}\exists_X(c \wedge (X = Y)) &= \exists_X(c[Y/X] \wedge (Y = X)) \\ &= c[Y/X] \wedge \exists_X(Y = X) \\ &= c[Y/X].\end{aligned}$$

□

Cylindric constraint systems provide diagonal elements, in place of equality constraints, for modeling parameter passing and variable substitution. They call the equality constraint $X = Y$ between variables X and Y a diagonal element and write it by d_{XY}. Diagonal elements clarify the mathematical structure of variable renamings, We have decided not to obey the entire formalization of cylindric constraint systems that includes diagonal elements because our direct approach of defining substitution by means of (U3) seems more suitable for our purpose of justifying program analysis.

Definition 2.7 (Constrained Variables) For $c \in \mathit{Con}$, we define

$$\mathit{var}(c) \stackrel{\text{def}}{=} \{X \in \mathit{Var} \mid c \neq \exists_X(c)\}$$

if $c \neq \mathit{false}$, and $\mathit{var}(\mathit{false}) \stackrel{\text{def}}{=} \mathit{Var}$. □

The set $\mathit{var}(c)$ represents the set of variables *constrained* in c.

Example.

$$\begin{aligned}\mathit{var}(\mathtt{E = 1}) &= \{\mathtt{E}\} \\ \mathit{var}(\mathtt{A = f(B)}) &= \{\mathtt{A}, \mathtt{B}\} \\ \mathit{var}(\mathtt{A = f(B)} \wedge \mathtt{C = D}) &= \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\} \\ \mathit{var}(\mathtt{X = X}) &= \mathit{var}(\mathit{true}) = \{\}\,.\end{aligned}$$

We have $\mathtt{B} \in \mathit{var}(\mathtt{A = f(B)})$ since $(\mathtt{A = f(B)} \wedge \mathtt{B = 4}) \neq (\exists_{\mathtt{B}}(\mathtt{A = f(B)}) \wedge \mathtt{B = 4})$. □

Proposition 2.2 For any $c \neq \mathit{false}$, we have $\mathit{var}(\exists_X c) \subseteq \mathit{var}(c) \setminus \{X\}$.

Proof. Let $Y \notin var(c) \setminus \{X\}$. If $Y \notin var(c)$, we have $\exists_Y \exists_X c = \exists_X \exists_Y c = \exists_X c$, which means $Y \notin var(\exists_X c)$. In the other case, Y is equal to X and hence we have $Y \notin var(\exists_X c)$ because $\exists_Y \exists_X c = \exists_X \exists_X c = \exists_X c$. □

Here is a typical example where equality does not hold:

$$var(\exists_{\mathtt{X}}(\mathtt{X = Y})) = var(true) = \{\} \neq \{\mathtt{Y}\} = var(\mathtt{X = Y}) \setminus \{\mathtt{X}\}\,.$$

Some constraints in *Con* may be non-tellable. For example, we may want to have `int(`X`)` in *Con* representing that X is bound to some integer. Moreover, we can even have a disjunction of two constraints expressed as a constraint. In order to allow arbitrary non-tellable constraints, however, we will require extra assumptions on hiding operators.

In this dissertation, we postulate the following assumption.

Assumption 2.8 $c \wedge d \neq false$ implies $var(c \wedge d) \subseteq var(c) \cup var(d)$.

This assumption states that a constraint has the information about which variables it has an influence on.

This assumption will be strengthened in Section 4.2.3 (devoted to moving synchronization points) by:

$$var(c \wedge d) = var(c) \cup var(d).$$

With this strengthening, we have $c \leq d \Rightarrow var(c) \subseteq var(d)$ and disjunctions over different sets of variables such as $((\mathtt{X = 1}) \vee (\mathtt{Y = 2}))$ cannot be described as a member of *Con*. Note however that disjunctions over the exactly same set of variables such as `int(X)` can still be a member of *Con*.

2.3 Syntax of CCP (Concurrent Constraint Programming)

Having formalized the constraint system, we are going to define our CCP language. In this section, we will define the syntax of CCP.

Given the set *Pred* of predicate symbols, we define the syntactic class of *agents* as in Figure 2.3. We denote by *Agents* the set of all agents.

The agent $\mathbf{tell}(c)$ adds constraint c to the store, while $\sum_{i=1}^{n} \mathbf{ask}(e_i) \rightarrow A_i$ nondeterministically waits for the store to entail some e_i and then behaves as A_i. The agent $A_1 \parallel A_2$ is parallel composition while **stop** is the terminated

$$
\begin{array}{lll}
\textit{Agents} & A & ::= \ \mathbf{tell}(c) \ \mid \ \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i \\
& & \mid \ A \parallel A \ \mid \ \mathbf{stop} \ \mid \ \exists X \langle A, c \rangle \ \mid \ p(X)
\end{array}
$$

where $c \in \mathit{Con}$, $n \geq 1$, $p \in \mathit{Pred}$ and X is a variable other than α.

Figure 2.3: Syntax of CCP agents

agent. $\exists X \langle A, c \rangle$ is the agent A that introduces X as a local variable and c as internal store that keeps local information on the variable X, as well as shared information on the other variables.

We abbreviate $\exists X \langle A, \mathit{true} \rangle$ to $\exists X A$.

Definition 2.8 For each agent A, we inductively define the set $\mathit{Vars}(A)$ as follows:

$$
\begin{array}{lcl}
\mathit{Vars}(\mathbf{stop}) & \stackrel{\text{def}}{=} & \{\} \\
\mathit{Vars}(\mathbf{tell}(e)) & \stackrel{\text{def}}{=} & \mathit{var}(e) \\
\mathit{Vars}(p(X)) & \stackrel{\text{def}}{=} & \{X\} \\
\mathit{Vars}(\sum_{i=1}^{n} \mathbf{ask}(e_i) \rightarrow A_i) & \stackrel{\text{def}}{=} & \bigcup_{i=1}^{n} \mathit{var}(e_i) \cup \mathit{Vars}(A_i) \\
\mathit{Vars}(A \parallel B) & \stackrel{\text{def}}{=} & \mathit{Vars}(A) \cup \mathit{Vars}(B) \\
\mathit{Vars}(\exists X \langle B, d \rangle) & \stackrel{\text{def}}{=} & (\, \mathit{Vars}(B) \cup \mathit{var}(d) \,) \setminus \{X\} \,.
\end{array}
$$

Moreover, for any pair $\langle A, c \rangle$ of an agent A and a constraint c, we define $\mathit{Vars}\langle A, c \rangle = \mathit{Vars}(A) \cup \mathit{var}(c)$. □

Intuitively, $\mathit{Vars}(A)$ is the set of variables occurring 'free' in the agent A. We shall not interpret the set-theoretic meaning of $\mathit{Vars}(A)$ for an agent A that contains the inconsistency constraint *false*.

Finally, we can define a program of CCP. A *program* of CCP is a mapping *Prog* from predicate symbols to *Agents* such that $\mathit{Vars}(\mathit{Prog}(p)) \subseteq \{\alpha\}$ for every p. The condition part expresses that a predicate definition should not contain free variables other than the formal parameter α.

2.4 Translating CLP into CCP

We now define a translating function $\mathcal{T}$ from CLP (concurrent logic programming) goals to CCP (concurrent constraint programming) agents as follows:

- $\mathcal{T}[\![s = t]\!] = \mathbf{tell}(s = t)$,
- $\mathcal{T}[\![p(t)]\!] = \exists G\langle p(G), G = t\rangle$ where G is a variable not occurring in t and other than α, and
- $\mathcal{T}[\![G_1 \parallel G_2]\!] = \mathcal{T}[\![G_1]\!] \parallel \mathcal{T}[\![G_2]\!]$.

Having defined this, we can translate a CLP program P into a CCP program *Prog* defined by:

$$Prog(p) = \sum_{(p(s)\,\texttt{:-}\,e\,\mid\,A)\in P} \mathbf{ask}(\overline{\exists}_\alpha((\alpha = s) \wedge e)) \rightarrow \overline{\exists}\alpha(\mathbf{tell}(\alpha = s) \parallel \mathcal{T}[\![A]\!])$$

where $\overline{\exists}\alpha(A)$ abbreviates $\exists X_1 \ldots \exists X_n(A)$ where $\{X_1, \ldots, X_n\}$ is the set of all the variables other than α that syntactically occur in the agent A. In addition, $\overline{\exists}_\alpha(c)$ abbreviates $\exists_{X_1} \ldots \exists_{X_n}(c)$ where $\{X_1, \ldots, X_n\}$ is the set of all the variables other than α that syntactically occur in the constraint c. We assume that every askable constraint e is in *Con*. For simplicity, we also assume that there is at least one clause for each predicate.

Example. The `intlist` predicate in Figure 2.2 is translated into CCP as follows:

$$\begin{array}{l}
Prog(\texttt{intlist}) = \\
\quad \mathbf{ask}(\exists_{\texttt{K0}}\exists_{\texttt{N}}\exists_{\texttt{L}}(\alpha = (\texttt{K0}, \texttt{N}, \texttt{L}) \wedge \texttt{K0} < \texttt{N})) \\
\qquad \rightarrow \exists\texttt{K0}\exists\texttt{N}\exists\texttt{L}\exists\texttt{K}\exists\texttt{L1}(\ \mathbf{tell}(\alpha = (\texttt{K0}, \texttt{N}, \texttt{L})) \parallel \mathbf{tell}(\texttt{L} = \texttt{[K0|L1]}) \\
\qquad\qquad \parallel \exists\texttt{G}\langle \texttt{:=(G)}, \texttt{G} = (\texttt{K0}, \texttt{1}, \texttt{K})\rangle \\
\qquad\qquad \parallel \exists\texttt{G}\langle \texttt{intlist(G)}, \texttt{G} = (\texttt{K}, \texttt{N}, \texttt{L1})\rangle\ ) \\
+\ \mathbf{ask}(\exists_{\texttt{K0}}\exists_{\texttt{N}}\exists_{\texttt{L}}(\alpha = (\texttt{K0}, \texttt{N}, \texttt{L}) \wedge \texttt{K0} >= \texttt{N})) \\
\qquad \rightarrow \exists\texttt{K0}\exists\texttt{N}\exists\texttt{L}(\ \mathbf{tell}(\alpha = (\texttt{K0}, \texttt{N}, \texttt{L})) \parallel \mathbf{tell}(\texttt{L=[]})\ )
\end{array}$$

2.5 Operational Semantics of CCP

In this section, we define the operational semantics of CCP.

Given a program *Prog*, we define the operational semantics as the smallest set $\longrightarrow \subseteq Proc \times Proc$ that enjoys the rules in Figure 2.4, where *Proc* is the set of *processes* defined by $Proc \stackrel{\text{def}}{=} Agents \times Con$.

$$
\begin{array}{ll}
\text{(R1)} & \langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{stop}, c \wedge d\rangle \\
\text{(R2)} & \langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle A_j, d\rangle \ \text{ if } c_j \leq d \\
\text{(R3)} & \langle p(X), d\rangle \longrightarrow \langle \exists\alpha \langle Prog(p),\ \alpha = X\rangle, d\rangle \\
\text{(R4)} & \dfrac{\langle A, c \wedge \exists_X d\rangle \longrightarrow \langle A', c'\rangle}{\langle \exists X \langle A, c\rangle, d\rangle \longrightarrow \langle \exists X \langle A', c'\rangle, d \wedge \exists_X c'\rangle} \\
\text{(R5)} & \dfrac{\langle A, d\rangle \longrightarrow \langle A', d'\rangle}{\langle A \parallel B, d\rangle \longrightarrow \langle A' \parallel B, d'\rangle} \\
\text{(R5}') & \dfrac{\langle B, d\rangle \longrightarrow \langle B', d'\rangle}{\langle A \parallel B, d\rangle \longrightarrow \langle A \parallel B', d'\rangle} \\
\text{(R6)} & \langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, d\rangle \text{ if } \forall i\, (c_i \wedge d = \mathit{false})
\end{array}
$$

Figure 2.4: Operational semantics of CCP

(R3) expresses the commencement of a goal $p(X)$ with a new store deployed. (R4) refreshes internal store as well as shared store for each reduction of the subagent.

In this dissertation, we employ an operational semantics such that every store in the configuration is updated after each reduction. Thanks to this, we can formulate the restartability of agents concisely as in Theorem 2.1.

(R6) expresses the computation in the situation of *reduction failure* where no legitimate reduction is possible for an **ask** agent. The consequences of (R6) and its variants are summarized as follows:

- With (R6), reduction failure is observed as divergence (infinite computation). This means that reduction failure cannot be distinguished from divergence.
- If (R6) is removed, reduction failure is observed as suspension. This means that reduction failure cannot be distinguished from suspension. This is the semantics adopted by the theoretical papers [10, 32].
- If the right-hand side of (R6) is replaced with $\langle \sum_{i=1}^{n} \mathbf{ask}(e_i) \rightarrow A_i, \mathit{false}\rangle$, reduction failure is observed as failure. This means that reduction failure cannot be distinguished from the failure by inconsistent tells.

The choice among these is independent of the rest of this dissertation.

Let $\longrightarrow^*$ denote the reflexive and transitive closure of $\longrightarrow$. At least in principle, a concurrent process has nothing to do with how many reductions other concurrent processes perform (as long as they are finite reductions). Thus, $\longrightarrow^*$ plays an important role in formalizing a denotational semantics of processes in Chapter 3.

Proposition 2.3 (Monotonicity) $\langle A, c\rangle \longrightarrow \langle A', c'\rangle$ implies $c \leq c'$.

Proof. By structural induction on $\longrightarrow$. □

2.6 External Input

In this section, we introduce a notation for expressing an agent that receives external input. The need of this notation is due to the local store scattered around the syntax of an agent. Nonetheless, this explicit treatment of external input turns out to be useful in stating the restartability theorem in the next section. The notation of external input was devised by the author for the first time.

Definition 2.9 (External Input to Agents) Let A be an agent and $a \in Con$ a constraint. We inductively define the agent Aa, called as the result of external input of a to A, as follows:

$$\begin{aligned}
\mathbf{stop}\, a &\stackrel{\text{def}}{=} \mathbf{stop} \\
\mathbf{tell}(e)\, a &\stackrel{\text{def}}{=} \mathbf{tell}(e) \\
p(X)\, a &\stackrel{\text{def}}{=} p(X) \\
(\textstyle\sum_{i=1}^{n} \mathbf{ask}(e_i) \rightarrow A_i)\, a &\stackrel{\text{def}}{=} \textstyle\sum_{i=1}^{n} \mathbf{ask}(e_i) \rightarrow A_i \\
(A \parallel B)\, a &\stackrel{\text{def}}{=} (A\, a) \parallel (B\, a) \\
(\exists X \langle B, d\rangle)\, a &\stackrel{\text{def}}{=} \exists X \langle B(d \wedge \exists_X a), d \wedge \exists_X a\rangle
\end{aligned}$$

□

External input to an agent models the communication from other agents, which results in just updating every local store scattered within the agent.

Let A and A' be agents satisfying $Ac = A'c$ where c is a constraint. The agents A and A' are syntactically the same except for the content of the

local stores they contain. Moreover, the processes $\langle A, c\rangle$ and $\langle A', c\rangle$ have the operational behavior. In this case, we will say that the agents A and A' are *syntactically the same under* the store c.

Example. The two agents $\exists \mathtt{X}\langle \mathtt{p(X)}, \mathtt{X = Y}\rangle$ and $\exists \mathtt{X}\langle \mathtt{p(X)},\ \mathtt{X = Y} \wedge \mathtt{Y = 1}\rangle$ are syntactically the same under the store $\mathtt{Y = 1}$ since we have $\exists \mathtt{X}\langle \mathtt{p(X)}, \mathtt{X = Y}\rangle(\mathtt{Y = 1}) = \exists \mathtt{X}\langle \mathtt{p(X)},\ \mathtt{X = Y} \wedge \mathtt{Y = 1}\rangle(\mathtt{Y = 1})$. □

Definition 2.10 (External Input to Processes) We extend the notation of external input to processes as follows:

$$\langle A, c\rangle a \stackrel{\text{def}}{=} \langle A(c \wedge a), c \wedge a\rangle.$$

□

Intuitively, a process $\langle A, c\rangle$ is updated to $\langle A, c\rangle a$ on receiving a constraint a. We will write $(\exists X\langle B, d\rangle)a$ as $\exists X\langle B, d\rangle a$. Note that $\exists X\langle B, d\rangle a = \exists X(\langle B, d\rangle \exists_X a)$.

Proposition 2.4 $f \leq g$ implies $Afg = Ag$.

Proof. By structural induction on agents. The only nontrivial case is the case of $\exists X\langle B, d\rangle$ and a proof is shown below.

$$\begin{aligned}
& \exists X\langle B, d\rangle fg \\
= \ & \exists X(\langle B, d\rangle \exists_X f)g \\
= \ & \exists X\langle B(d \wedge \exists_X f), d \wedge \exists_X f\rangle g \\
= \ & \exists X\langle B(d \wedge \exists_X f)(d \wedge \exists_X f \wedge \exists_X g), d \wedge \exists_X f \wedge \exists_X g\rangle \\
= \ & \exists X\langle B(d \wedge \exists_X f \wedge \exists_X g), d \wedge \exists_X f \wedge \exists_X g\rangle \\
= \ & \exists X\langle B(d \wedge \exists_X g), d \wedge \exists_X g\rangle \\
= \ & \exists X\langle B, d\rangle g.
\end{aligned}$$

□

As a corollary to this, we have $\langle A, c\rangle fg = \langle A, c\rangle g$ if $f \leq g$. Moreover, we can prove that $f \geq g$ implies $Afg = Af$ and $\langle A, c\rangle fg = \langle A, c\rangle f$ in almost the same way.

2.7 Restartability of Agents

We will prove a basic theorem on the restartability of agents. Intuitively, the theorem guarantees that once an agent becomes reducible, it is reducible forever. The result of this theorem is essential for proving the correctness of program transformation explained in Chapter 4. It is worth noting that this theorem relies on the absence of atomic tell operations.

Theorem 2.1 (Restartability) Assume $\langle A, c\rangle \longrightarrow \langle A', c'\rangle$. For every agent $\hat{A}$ and constraint a satisfying $\hat{A}(c \wedge a) = A(c \wedge a)$, there exists an agent $\hat{A}'$ such that $\langle \hat{A}, c \wedge a\rangle \longrightarrow \langle \hat{A}', c' \wedge a\rangle$ and $\hat{A}'(c' \wedge a) = A'(c' \wedge a)$.

Note. The reader can first consider the special case where $\hat{A} = A$, in which case the statement reads as this: "Assume $\langle A, c\rangle \longrightarrow \langle A', c'\rangle$. For every constraint a, there exists an agent $\hat{A}'$ such that $\langle A, c \wedge a\rangle \longrightarrow \langle \hat{A}', c' \wedge a\rangle$ and $\hat{A}'(c' \wedge a) = A'(c' \wedge a)$." The agents $\hat{A}'$ and A' are syntactically the same under the store $c' \wedge a$ but not always identical. The exact reason of not always having $\hat{A}' = A'$ is that, by the rule (R4), the local store within $\hat{A}'$ contains more information than the corresponding local store within A'. Precisely speaking, whereas $c' \wedge a$ can be distributed inside $\hat{A}'$ at the reduction, only c' can be distributed inside A' at the reduction. Since we have assumed the external input of a, we should only want to know whether $\langle \hat{A}', c' \wedge a\rangle a = \langle A', c'\rangle a$, namely $\hat{A}'(c' \wedge a) = A'(c' \wedge a)$.

Proof. By structural induction on $\longrightarrow$.

- Case $\langle \mathbf{tell}(e), c\rangle \longrightarrow \langle \mathbf{stop}, e \wedge c\rangle$.
  Note that $\hat{A} = A$. We have $\langle \mathbf{tell}(e), c \wedge a\rangle \longrightarrow \langle \mathbf{stop}, e \wedge c \wedge a\rangle$.
- Case $\langle p(X), c\rangle \longrightarrow \langle \exists\alpha\langle \mathit{Prog}(p),\ \alpha = X\rangle, c\rangle$.
  Note that $\hat{A} = A$. We have $\langle p(X), c \wedge a\rangle \longrightarrow \langle \exists\alpha\langle \mathit{Prog}(p),\ \alpha = X\rangle, c \wedge a\rangle$.
- Case $\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, c\rangle \longrightarrow \langle A_j, c\rangle$ where $c_j \leq c$.
  Note that $\hat{A} = A$. Since $c_j \leq c \leq c \wedge a$, we have
  $$\langle \textstyle\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_j, c \wedge a\rangle \longrightarrow \langle A_j, c \wedge a\rangle.$$

- Case $\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i, c\rangle \longrightarrow \langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i, c\rangle$ where $\forall i(c_i \wedge c = \mathit{false})$.

  Note that $\hat{A} = A$. We have $\mathit{false} = c_i \wedge c \leq c_i \wedge c \wedge a$.

- Case $\langle \exists X \langle B, d\rangle, c\rangle \longrightarrow \langle \exists X \langle B', d'\rangle, c \wedge \exists_X d'\rangle$.

  By definition of external input, there exist some $\hat{B}$ and $\hat{d}$ such that $\hat{A} = \exists X \langle \hat{B}, \hat{d}\rangle$ and $\hat{d} \wedge \exists_X(c \wedge a) = d \wedge \exists_X(c \wedge a)$ and $\hat{B}(d \wedge \exists_X(c \wedge a)) = B(d \wedge \exists_X(c \wedge a))$.

  By induction hypothesis for $\langle B, d \wedge \exists_X c\rangle \longrightarrow \langle B', d'\rangle$ applied to $\exists_X(c \wedge a)$, there exists $\hat{B}'$ such that $\langle \hat{B}, d \wedge \exists_X(c \wedge a)\rangle \longrightarrow \langle \hat{B}', d' \wedge \exists_X(c \wedge a)\rangle$ and $\hat{B}'(d' \wedge \exists_X(c \wedge a)) = B'(d' \wedge \exists_X(c \wedge a))$. Since $\exists_X c \leq \exists_X(c \wedge a)$, we have $\hat{d} \wedge \exists_X(c \wedge a) = d \wedge \exists_X(c \wedge a)$ and therefore we have $\langle \hat{B}, \hat{d} \wedge \exists_X(c \wedge a)\rangle \longrightarrow \langle \hat{B}', d' \wedge \exists_X(c \wedge a)\rangle$. Hence, we have

$$\begin{aligned}
& \langle \exists X \langle \hat{B}, \hat{d}\rangle, c \wedge a\rangle \\
\longrightarrow\ & \langle \exists X \langle \hat{B}', d' \wedge \exists_X(c \wedge a)\rangle, c \wedge a \wedge \exists_X(d' \wedge \exists_X(c \wedge a))\rangle \\
=\ & \langle \exists X \langle \hat{B}', d' \wedge \exists_X(c \wedge a)\rangle, c \wedge a \wedge \exists_X d' \wedge \exists_X(c \wedge a)\rangle \\
=\ & \langle \exists X \langle \hat{B}', d' \wedge \exists_X(c \wedge a)\rangle, c \wedge \exists_X d' \wedge a\rangle.
\end{aligned}$$

  Moreover, we have

$$\begin{aligned}
& \exists X \langle \hat{B}', d' \wedge \exists_X(c \wedge a)\rangle(c \wedge \exists_X d' \wedge a) \\
=\ & \exists X(\langle \hat{B}', d' \wedge \exists_X(c \wedge a)\rangle \exists_X(c \wedge \exists_X d' \wedge a)) \\
=\ & \exists X(\langle \hat{B}', d' \wedge \exists_X(c \wedge a)\rangle(\exists_X d' \wedge \exists_X(c \wedge a))) \\
=\ & \exists X \langle \hat{B}'(d' \wedge \exists_X(c \wedge a)), d' \wedge \exists_X(c \wedge a)\rangle \\
=\ & \exists X \langle B'(d' \wedge \exists_X(c \wedge a)), d' \wedge \exists_X(c \wedge a)\rangle \\
=\ & \exists X(\langle B', d'\rangle(\exists_X d' \wedge \exists_X(c \wedge a))) \\
=\ & \exists X(\langle B', d'\rangle \exists_X(c \wedge \exists_X d' \wedge a)) \\
=\ & \exists X \langle B', d'\rangle(c \wedge \exists_X d' \wedge a).
\end{aligned}$$

- Case $\langle B_1 \parallel B_2, c\rangle \longrightarrow \langle B_1' \parallel B_2, c'\rangle$.

  There exist $\hat{B}_1$ and $\hat{B}_2$ such that $\hat{B} = \hat{B}_1 \parallel \hat{B}_2$ and $\hat{B}_1(c \wedge a) = B_1(c \wedge a)$ and $\hat{B}_2(c \wedge a) = B_2(c \wedge a)$. By induction hypothesis for $\langle B_1, c\rangle \longrightarrow \langle B_1', c'\rangle$, there exists $\hat{B}_1'$ such that $\langle \hat{B}_1, c \wedge a\rangle \longrightarrow \langle \hat{B}_1', c' \wedge a\rangle$ and $\hat{B}_1'(c' \wedge a) = B_1'(c' \wedge a)$. Hence we have $\langle \hat{B}_1 \parallel \hat{B}_2, c \wedge a\rangle \longrightarrow \langle \hat{B}_1' \parallel \hat{B}_2, c' \wedge a\rangle$.

  Moreover, since $\hat{B}_2(c' \wedge a) = \hat{B}_2(c \wedge a)(c' \wedge a) = B_2(c \wedge a)(c' \wedge a) = B_2(c' \wedge a)$, we have $(\hat{B}_1' \parallel \hat{B}_2)(c' \wedge a) = \hat{B}_1'(c' \wedge a) \parallel \hat{B}_2(c' \wedge a) = B_1'(c' \wedge a) \parallel B_2(c' \wedge a) = (B_1' \parallel B_2)(c' \wedge a)$.

The case of $\langle B_1 \parallel B_2, c\rangle \longrightarrow \langle B_1 \parallel B_2', c'\rangle$ is similar. □

Example. Consider the agent $\exists \mathtt{X}\langle \mathbf{tell}(\mathtt{Y = f(X)}), \mathit{true}\rangle$. For the reduction $\langle \exists \mathtt{X}\langle \mathbf{tell}(\mathtt{Y = f(X)}), \mathit{true}\rangle, \mathit{true}\rangle \longrightarrow \langle \exists \mathtt{X}\langle \mathbf{stop}, \mathtt{Y = f(X)}\rangle, \mathtt{Y = f(\_)}\rangle$, we have $\langle \exists \mathtt{X}\langle \mathbf{tell}(\mathtt{Y = f(X)}), \mathit{true}\rangle, \mathtt{Z = 1}\rangle \longrightarrow \langle \exists \mathtt{X}\langle \mathbf{stop}, \mathtt{Y = f(X)} \wedge \mathtt{Z = 1}\rangle, \mathtt{Y = f(\_)} \wedge \mathtt{Z = 1}\rangle$ and that $\exists \mathtt{X}\langle \mathbf{stop}, \mathtt{Y = f(X)}\rangle(\mathtt{Y = f(\_)} \wedge \mathtt{Z = 1}) = \exists \mathtt{X}\langle \mathbf{stop}, \mathtt{Y = f(X)} \wedge \mathtt{Z = 1}\rangle = \exists \mathtt{X}\langle \mathbf{stop}, \mathtt{Y = f(X)} \wedge \mathtt{Z = 1}\rangle(\mathtt{Y = f(\_)} \wedge \mathtt{Z = 1})$. This says that we can defer the execution of $\exists \mathtt{X}\langle \mathbf{tell}(\mathtt{Y = f(X)}), \mathit{true}\rangle$ until the store becomes $\mathtt{Z = 1}$. □

Chapter 3

Denotational Semantics

Although a compiler should generate object code which behaves as specified by the input program, the exact meaning of the correctness of a compiler depends on the definition of behavior. In a nondeterministic setting, it is sometimes needed to alter the behavior of the object code in order to perform particular optimization. A denotational semantics maps a process to a denotation (a certain mathematical object) representing the behavior of that process, thereby enabling the discussion of relationships between different behaviors. In this chapter, we build a denotational semantics of processes based on observational equivalence. To do this, we first give the definition of the observables of a process and then define the denotational semantics of processes that is fully abstract, meaning that no contexts exist that can distinguish two processes having the same denotation.

3.1 The Observables

We formalize the observables of a process by all the phenomena we want to regard as its whole behavior.

Definition 3.1 (Observables) Given a program *Prog*, we define the *observables* of a process $\langle A, c\rangle$, written as $\mathcal{O}[\![\langle A, c\rangle]\!]_{Prog}$, as follows:

$$\begin{aligned}\mathcal{O}[\![\langle A, c\rangle]\!]_{Prog} \stackrel{\text{def}}{=}\ & \{d\cdot\bot \mid \langle A, c\rangle \longrightarrow^* \langle A', c'\rangle,\ c \le d \le c',\ d \in Con\} \\ \cup\ & \{c' \cdot dd \mid \langle A, c\rangle \longrightarrow^* \langle A', c'\rangle \not\longrightarrow,\ A' \notin Stop\} \\ \cup\ & \{c' \cdot tt \mid \langle A, c\rangle \longrightarrow^* \langle A', c'\rangle,\ A' \in Stop\} \\ \cup\ & \{c' \cdot \infty \mid \langle A, c\rangle \longrightarrow^* \langle A', c'\rangle \longrightarrow^\infty\}\end{aligned} \tag{3.1}$$

where $\langle A, c\rangle \longrightarrow^{\infty}$ means the existence of infinite computation beginning with $\langle A, c\rangle$, in which case we say $\langle A, c\rangle$ *diverges*, and *Stop* is the set of agents containing no asks, tells, or calls. Formally, the set *Stop* of the terminated agents is defined as follows:

$$S \ ::= \ \mathbf{stop} \ \mid \ S \parallel S \ \mid \ \exists X \langle S, c\rangle$$

where X is a variable and $c \in Con$. □

The $d \cdot \bot$ in $\mathcal{O}[\![\langle A, c\rangle]\!]_{Prog}$ represents the possibility that the agent A at the store c may cause the store to entail d. On the other hand, $c' \cdot dd$, $c' \cdot tt$, and $c' \cdot \infty$ represent the possibility that A at the store c may make the store c' and then suspend, terminate, and diverge, respectively.

The reason of having partial information $d \cdot \bot$ in addition to the total information $c' \cdot \bot$ in the observables is to reflect the setting that tells are eventual, not atomic. The outcome of this definition is that we can freely convert several tells to their equivalent composite and vice versa without worrying about the change of the program semantics, which greatly simplifies the justification of these kinds of program transformation.

We may abbreviate $\mathcal{O}[\![\langle A, \mathit{true}\rangle]\!]_{Prog}$ to $\mathcal{O}[\![A]\!]$.

Example.

$$\begin{aligned}
\mathcal{O}[\![\langle \mathbf{tell}(c \wedge d), \mathit{true}\rangle]\!]_{Prog} &= \{e \cdot \bot \mid e \leq \underline{cd}\} \cup \{\underline{cd} \cdot tt\} \\
\mathcal{O}[\![\langle \mathbf{tell}(c) \parallel (\mathbf{ask}(c) \rightarrow \mathbf{tell}(d)), \mathit{true}\rangle]\!]_{Prog} &= \{e \cdot \bot \mid e \leq \underline{cd}\} \cup \{\underline{cd} \cdot tt\} \\
\mathcal{O}[\![\langle \mathbf{ask}(c) \rightarrow \mathbf{tell}(d), \mathit{true}\rangle]\!]_{Prog} &= \{\mathit{true} \cdot \bot\} \cup \{\mathit{true} \cdot dd\} \\
\mathcal{O}[\![\langle \mathbf{ask}(c) \rightarrow \mathbf{tell}(d), c\rangle]\!]_{Prog} &= \{e \cdot \bot \mid c \leq e \leq \underline{cd}\} \cup \{\underline{cd} \cdot tt\}
\end{aligned}$$

where c and d are constraints and $\underline{cd}$ abbreviates $c \wedge d$. We can see that the observables of the first two agents coincide. It reflects the fact that there is no contexts that distinguish between them. □

It is known that collecting observables $\mathcal{O}[\![\langle C[A], \mathit{true}\rangle]\!]_{Prog}$ for all contexts $C[\cdot]$ generates a compositional denotational semantics of the agent A under the program *Prog* [10, 32].

Observable Output

In executing a program, suspension of a process can be observed only if every process is suspended, in which case we say that the computation is in *deadlock*. It encourages us to define the notion of observable output:

Definition 3.2 (Observable Output) We define the *observable output* of a process $\langle A, c\rangle$ by the set $\mathcal{O}[\![\langle A, c\rangle]\!]_{Prog} \cap ObsOut(c)$ where

$$ObsOut(c) = \{d \cdot \bot \mid d > c\} \cup \{false \cdot \bot\} \cup \{d \cdot tt \mid d \geq c\} \cup \{d \cdot \infty \mid d \geq c\}\,.$$

We say that a process has observable output if and only if its observable output is not empty. □

Intuitively, a process has observable output if it can do something other than suspension—that is, telling something new to the store, making the store inconsistent, doing termination, or going to divergence.

3.2 Observational Equivalence

Next, we will define observational equivalence. An intuitive definition will be given first. The formal definition will be given at the end of this section.

Intuitively, two processes are said to be *observationally equivalent* if and only if their sets of possible sequences of input/output coincide, where *input* is an ask of a constraint and *output* is either a tell, suspension, termination, or divergence. Notice that the output other than a tell corresponds to the set $\{dd, tt, \infty\}$ appearing in the definition of the observables.

We are interested in an observational equivalence that is a congruence, i.e., preserved under parallel composition and local variable introduction. In this section, we will see that our definition of the observables of processes has enough information to make the resulting observational equivalence a congruence. Finally, we will prove that our denotational semantics is equivalent to the quotient set of all processes by the observational equivalence relation.

The key point in defining a denotational semantics of CCP is that local choices (also known as indeterminism or don't-care nondeterminism) cannot be observed directly from other processes since every local choice takes place asynchronously from the outside world. This is not the case with CCS (Calculus of Communicating Systems) [28] or CSP (Communicating Sequential Processes) [18] where communication is synchronous. Readers are referred to the paper [10] for the details of these issues.

We first formalize the input/output sequences of a process.

Definition 3.3 (Interaction Sequences) Let $\langle G_0, b_0\rangle$ be a process and n a non-negative interger. We call a finite sequence $(a_1, G_1, b_1), \ldots, (a_n, G_n, b_n)$

in $(D \times \mathit{Agents} \times D)^*$ satisfying $b_0 \leq a_1 \leq b_1 \leq \ldots \leq a_n \leq b_n$ and

$$\langle G_{i-1}, a_i \rangle \longrightarrow^* \langle G_i, b_i \rangle,\ i = 1, \ldots, n \tag{3.2}$$

an *interaction sequence* of $\langle G_0, b_0 \rangle$ and refer to n as its *length*. We shall write an expression in the format of (3.2) to speak of an interaction sequence. □

We should mention that each process $\langle G_0, b_0 \rangle$ has an interaction sequence of length 0.

A *step-by-step interaction sequence* is a special kind of an interaction sequence of the form:

$$\langle G_{i-1}, a_i \rangle \longrightarrow \langle G_i, b_i \rangle,\ i = 1, \ldots, n.$$

Definition 3.4 (Observational Subsumption) Let $\langle G_0, b_0 \rangle$ and $\langle G'_0, b_0 \rangle$ be processes. We say $\langle G_0, b_0 \rangle$ is *observationally subsumed* by $\langle G'_0, b_0 \rangle$ if and only if the following holds, in which case we write $\langle G_0, b_0 \rangle \leq \langle G'_0, b_0 \rangle$:

For any interaction sequence

$$x : \langle G_{i-1}, a_i \rangle \longrightarrow^* \langle G_i, b_i \rangle,\ i = 1, \ldots, n$$

of $\langle G_0, b_0 \rangle$, there exists an interaction sequence

$$y : \langle G'_{i-1}, a_i \rangle \longrightarrow^* \langle G'_i, b_i \rangle,\ i = 1, \ldots, n$$

of $\langle G'_0, b_0 \rangle$ satisfying the following:

(div) If $\langle G_n, b_n \rangle \longrightarrow^\infty$ then $\langle G'_n, b_n \rangle \longrightarrow^\infty$.
(end) If $\langle G_n, b_n \rangle \longrightarrow^* \surd$ then $\langle G'_n, b_n \rangle \longrightarrow^* \surd$.

We define that $\langle A, c \rangle \longrightarrow^* \surd$ means the existence of some $S \in \mathit{Stop}$ such that $\langle A, c \rangle \longrightarrow^* \langle S, c \rangle$. □

The reason of introducing $\langle A, c \rangle \longrightarrow^* \surd$ rather than simply saying $A \in \mathit{Stop}$ is that we don't want to make a distinction between $\langle \mathbf{tell}(e), c \rangle$ and $\langle \mathbf{stop}, c \rangle$ where $e \leq c$. In fact, they are distinguishable only if the exact number of reductions is taken into account in defining the equivalence. Such an equivalence, however, is not suitable for our purpose of justifying program transformation.

It can be easily shown that we can obtain the same definition of $\leq$ by restricting x to only step-by-step interaction sequences.

When $\langle G_0, b_0\rangle \leq \langle G'_0, b_0\rangle$, every operational characteristic observable in $\langle G_0, b_0\rangle$ is also observed in $\langle G'_0, b_0\rangle$. When $\langle G_0, b_0\rangle \leq \langle G'_0, b_0\rangle$ and $\langle G'_0, b_0\rangle \leq \langle G_0, b_0\rangle$, they have the same set of operational characteristics and are thought to be *observationally equivalent* to each other. Let us employ this as our definition of the observational equivalence.

3.3 Denotations

In this section, we define a denotational semantics of processes so that the correspondence with the operational semantics is induced straightforwardly. This correspondence is important for the justification of the program analysis. The denotation we are defining is due to the author but can be related with the one defined in the paper [10] that is also based on a trace semantics with explicit treatment of divergence.

We begin with the definition of traces of interaction sequences.

Definition 3.5 (Traces) For every interaction sequence

$$x : \langle G_{i-1}, a_i\rangle \longrightarrow^* \langle G_i, b_i\rangle,\ i = 1, \ldots, n$$

of $\langle G_0, b_0\rangle$, we define its *normal trace* by

$$\langle \mathit{true}, b_0\rangle\langle a_1, b_1\rangle\langle a_2, b_2\rangle \ldots \langle a_n, b_n\rangle \cdot \bot.$$

Moreover, we define that x has a *divergent trace*

$$\langle \mathit{true}, b_0\rangle\langle a_1, b_1\rangle\langle a_2, b_2\rangle \ldots \langle a_n, b_n\rangle \cdot \infty$$

if and only if $\langle G_n, b_n\rangle \longrightarrow^{\infty}$ holds.

Furthermore, we define that x has a *terminated trace*

$$\langle \mathit{true}, b_0\rangle\langle a_1, b_1\rangle\langle a_2, b_2\rangle \ldots \langle a_n, b_n\rangle \cdot tt$$

if and only if $\langle G_n, b_n\rangle \longrightarrow^* \surd$ holds. □

Next, we define the set of denotations. Each denotation represents the semantics of a certain process.

Definition 3.6 (Denotation) A subset E of

$$\{\langle \mathit{true}, b_0\rangle\langle a_1, b_1\rangle \ldots \langle a_n, b_n\rangle \cdot m \,|\, n \geq 0,\ b_0 \leq a_1 \leq \ldots \leq b_n,\ m \in \{\bot, \infty, tt\}\}$$

is called a *denotation* of a process if the following closure conditions are met:

1. there is a unique constraint b_0 such that $\langle true, b_0\rangle \cdot \bot \in E$
2. $t\langle a, b\rangle t' \cdot m \in E$ implies $t\langle a, b\rangle\langle b, b\rangle t' \cdot m \in E$
3. $\langle true, c\rangle t\, t' \cdot m \in E$ implies $\langle true, c\rangle t \cdot \bot \in E$
4. $\langle true, c\rangle t\, t' \cdot m \in E$ implies $\langle true, c\rangle t' \cdot m \in E$
5. $\langle true, c\rangle t\langle a_1, b_1\rangle \ldots \langle a_n, b_n\rangle \cdot m \in E$ implies
   $\langle true, c\rangle t\langle a_1 \wedge e, b_1 \wedge e\rangle \ldots \langle a_n \wedge e, b_n \wedge e\rangle \cdot m \in E$
6. $\forall n \geq 0\ \exists b_1 < \ldots < b_n (\, t\langle a_0, b_0\rangle\langle b_0, b_1\rangle \ldots \langle b_{n-1}, b_n\rangle \cdot \bot \in E\,)$ implies
   $t\langle a_0, b_0\rangle \cdot \infty \in E$

where t and t' stand for finite sequences of pairs and e is a constraint. □

(1) represents the uniqueness of the initail store. (2) represents that reductions are delayable. (3) represents that the trace is a time sequence. (4) and (5) represent the restartability of a process. (6) represents that a process with infinite output diverges.

We will show that the denotational semantics of a process $\langle A, c\rangle$ can be obtained as the set of (all of the three kinds of) traces of all the interaction sequences of $\langle A, c\rangle$. We denote this set by $\mathcal{D}[\![\langle A, c\rangle]\!]$. Our denotational semantics can fully handle nondeterministic features of a process as the semantics defined in the papers [33, 32, 10]. This is manifested by the following proposition.

Proposition 3.1 (Equivalence to Operational Semantics)

$$\langle G_0, b_0\rangle \leq \langle G_0', b_0\rangle \Leftrightarrow \mathcal{D}[\![\langle G_0, b_0\rangle]\!] \subseteq \mathcal{D}[\![\langle G_0', b_0\rangle]\!].$$

Proof.

⇒) Obvious.

⇐) It can be easily shown that $\langle G_0, b_0\rangle \not\leq \langle G_0', b_0\rangle$ implies $\mathcal{D}[\![\langle G_0, b_0\rangle]\!] \not\subseteq \mathcal{D}[\![\langle G_0', b_0\rangle]\!]$. □

As a corollary of this proposition, the observational equivalence is a necessary and sufficient condition for the coincidence of denotations.

3.4 Properties on Processes

In this section, we prove several properties on processes we will use later. This section can be skipped in the first reading, for they are only needed for describing proofs in the next chapter.

Proposition 3.2 (Tell Construction) If $\langle A, c\rangle \longrightarrow \langle A', c'\rangle$ then there exists some e such that $var(e) \subseteq Vars(A)$ and $c' = e \wedge c$.

Proof. By structural induction on $\longrightarrow$.

- Case $\langle \mathbf{tell}(e), c\rangle \longrightarrow \langle \mathbf{stop}, e \wedge c\rangle$.

  Note that $var(e) = Vars(\mathbf{tell}(e))$.

- Case $\langle p(X), c\rangle \longrightarrow \langle \exists\alpha\langle Prog(p),\ \alpha = X\rangle, c\rangle$.

  Let $e = true$.

- Case $\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, c\rangle \longrightarrow \langle A_j, c\rangle$ where $c_j \leq c$.

  Let $e = true$.

- Case $\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, c\rangle \longrightarrow \langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, c\rangle$ where $\forall i(c_i \wedge c = false)$.

  Let $e = true$.

- Case $\langle \exists X\langle B, d\rangle, c\rangle \longrightarrow \langle \exists X\langle B', d'\rangle, c \wedge \exists_X d'\rangle$.

  By induction hypothesis for $\langle B, d \wedge \exists_X c\rangle \longrightarrow \langle B', d'\rangle$, there exists some f such that $var(f) \subseteq Vars(B)$ and $d' = f \wedge d \wedge \exists_X c$. Therefore, $c \wedge \exists_X d' = c \wedge \exists_X(f \wedge d \wedge \exists_X c) = c \wedge \exists_X(f \wedge d) \wedge \exists_X c = \exists_X(f \wedge d) \wedge c$ and $var(\exists_X(f \wedge d)) \subseteq (var(f) \cup var(d)) \setminus \{X\} \subseteq (Vars(B) \cup var(d)) \setminus \{X\} \subseteq Vars(\exists X\langle B, d\rangle)$.

- Case $\langle B_1 \parallel B_2, c\rangle \longrightarrow \langle B_1' \parallel B_2, c'\rangle$.

  By induction hypothesis for $\langle B_1, c\rangle \longrightarrow \langle B_1', c'\rangle$, there exists some e such that $var(e) \subseteq Vars(B_1) \subseteq Vars(B_1 \parallel B_2)$ and $c' = e \wedge c$.

  The case of $\langle B_1 \parallel B_2, c\rangle \longrightarrow \langle B_1 \parallel B_2', c'\rangle$ is similar. □

Proposition 3.3 If $\langle A, c\rangle \longrightarrow^* \langle A', c'\rangle$ then there exists some e such that $var(e) \subseteq Vars(A)$ and $c' = e \wedge c$.

Proof. Corollary of the previous proposition.

Proposition 3.4 $\langle A, c\rangle \longrightarrow \langle A', c'\rangle$ implies $\mathit{Vars}\langle A', c'\rangle \subseteq \mathit{Vars}\langle A, c\rangle$.

Proof. By structural induction on $\longrightarrow$.

- Case $\langle \mathbf{tell}(e), c\rangle \longrightarrow \langle \mathbf{stop}, e \wedge c\rangle$.
  $\mathit{Vars}\langle \mathbf{stop}, e \wedge c\rangle = \mathit{var}(e \wedge c) \subseteq \mathit{var}(e) \cup \mathit{var}(c) = \mathit{Vars}\langle \mathbf{tell}(e), c\rangle$.

- Case $\langle p(X), c\rangle \longrightarrow \langle \exists\alpha\langle \mathit{Prog}(p),\ \alpha = X\rangle, c\rangle$.

$$\begin{array}{rcl} & & \mathit{Vars}\langle \exists\alpha\langle \mathit{Prog}(p),\ \alpha = X\rangle, c\rangle \\ & = & (\ \mathit{Vars}(\mathit{Prog}(p)) \cup \mathit{var}(\alpha = X)\ ) \setminus \{\alpha\} \cup \mathit{var}(c) \\ & \subseteq & (\{\alpha\} \cup \{\alpha, X\}) \setminus \{\alpha\} \cup \mathit{var}(c) \\ & = & \{X\} \cup \mathit{var}(c) \ = \ \mathit{Vars}\langle p(X), c\rangle. \end{array}$$

- Case $\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, c\rangle \longrightarrow \langle A_j, c\rangle$.

$$\begin{array}{rcl} \mathit{Vars}\langle A_j, c\rangle & = & \mathit{Vars}(A_j) \cup \mathit{var}(c) \\ & \subseteq & (\ \bigcup_{i=1}^{n} \mathit{var}(c_i) \cup \mathit{Vars}(A_i)\ ) \cup \mathit{var}(c) \\ & = & \mathit{Vars}\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, c\rangle \end{array}$$

- Case $\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, c\rangle \longrightarrow \langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i, c\rangle$.
  Obvious.

- Case $\langle \exists X\langle B, d\rangle, c\rangle \longrightarrow \langle \exists X\langle B', d'\rangle, c \wedge \exists_X d'\rangle$.
  By induction hypothesis for $\langle B, d \wedge \exists_X c\rangle \longrightarrow \langle B', d'\rangle$, it is the case that $\mathit{Vars}\langle B', d'\rangle \subseteq \mathit{Vars}\langle B, d \wedge \exists_X c\rangle$. Therefore, we have

$$\begin{array}{rcl} & & \mathit{Vars}\langle \exists X\langle B', d'\rangle, c \wedge \exists_X d'\rangle \\ & = & \mathit{Vars}\langle B', d'\rangle \setminus \{X\} \cup \mathit{var}(c \wedge \exists_X d') \\ & \subseteq & \mathit{Vars}\langle B', d'\rangle \setminus \{X\} \cup \mathit{var}(c) \cup (\mathit{var}(d') \setminus \{X\}) \\ & = & \mathit{Vars}\langle B', d'\rangle \setminus \{X\} \cup \mathit{var}(c) \\ & \subseteq & \mathit{Vars}\langle B, d \wedge \exists_X c\rangle \setminus \{X\} \cup \mathit{var}(c) \\ & = & (\ \mathit{Vars}(B) \cup \mathit{var}(d \wedge \exists_X c)\ ) \setminus \{X\} \cup \mathit{var}(c) \\ & \subseteq & (\ \mathit{Vars}(B) \cup \mathit{var}(d) \cup (\mathit{var}(c) \setminus \{X\})\ ) \setminus \{X\} \cup \mathit{var}(c) \\ & = & (\ \mathit{Vars}(B) \cup \mathit{var}(d)\ ) \setminus \{X\} \cup \mathit{var}(c) \\ & = & \mathit{Vars}(\exists X\langle B, d\rangle) \cup \mathit{var}(c) \\ & = & \mathit{Vars}\langle \exists X\langle B, d\rangle, c\rangle. \end{array}$$

- Case $\langle B_1 \parallel B_2, c\rangle \longrightarrow \langle B_1' \parallel B_2, c'\rangle$.

  By induction hypothesis for $\langle B_1, c\rangle \longrightarrow \langle B_1', c'\rangle$, we have $\mathit{Vars}\langle B_1', c'\rangle \subseteq \mathit{Vars}\langle B_1, c\rangle$. Therefore, we have

$$\begin{array}{rcl} \mathit{Vars}\langle B_1' \parallel B_2, c'\rangle & = & \mathit{Vars}(B_1') \cup \mathit{Vars}(B_2) \cup \mathit{var}(c') \\ & = & \mathit{Vars}\langle B_1', c'\rangle \cup \mathit{Vars}(B_2) \\ & \subseteq & \mathit{Vars}\langle B_1, c\rangle \cup \mathit{Vars}(B_2) \\ & = & \mathit{Vars}(B_1) \cup \mathit{Vars}(B_2) \cup \mathit{var}(c) \\ & = & \mathit{Vars}\langle B_1 \parallel B_2, c\rangle. \end{array}$$

  The case of $\langle B_1 \parallel B_2, c\rangle \longrightarrow \langle B_1 \parallel B_2', c'\rangle$ is similar. □

Proposition 3.5 Let $\langle A, c\rangle$ be a process and $\hat{A}$ an agent satisfying $\hat{A}c = Ac$ and $Y \notin \mathit{Vars}(\hat{A})$. For every process $\langle A', c'\rangle$ such that

$$\langle A, c\rangle \longrightarrow \langle A', c'\rangle,$$

there exists an agent $\hat{A}'$ such that $\hat{A}'c' = A'c'$ and $Y \notin \mathit{Vars}(\hat{A}')$ and

$$\langle \hat{A}, \exists_Y c\rangle \longrightarrow \langle \hat{A}', \exists_Y c'\rangle.$$

Proof. Note that by Theorem 2.1 applied to $\langle A, c\rangle \longrightarrow \langle A', c'\rangle$ with $(Ac)c = Ac$, there exists an agent E such that $\langle Ac, c\rangle \longrightarrow \langle E, c'\rangle$ and $Ec' = A'c'$. Therefore, by Theorem 2.1 applied to $\langle Ac, c\rangle \longrightarrow \langle E, c'\rangle$ with $\hat{A}c = (Ac)c$, there exists an agent $\hat{E}$ such that $\langle \hat{A}, c\rangle \longrightarrow \langle \hat{E}, c'\rangle$ and $\hat{E}c' = Ec' = A'c'$.

By structural induction on $\longrightarrow$ for $\langle A, c\rangle \longrightarrow \langle A', c'\rangle$.

- Case $\langle \mathbf{tell}(e), c\rangle \longrightarrow \langle \mathbf{stop}, e \wedge c\rangle$.

  We have $\langle \mathbf{tell}(e), \exists_Y c\rangle \longrightarrow \langle \mathbf{stop}, e \wedge \exists_Y c\rangle$. Since $Y \notin \mathit{var}(e)$, we have $\exists_Y e = e$ and hence $\exists_Y(e \wedge c) = e \wedge \exists_Y c$.

- Case $\langle p(X), c\rangle \longrightarrow \langle \exists\alpha\langle \mathit{Prog}(p),\ \alpha = X\rangle, c\rangle$.

  We have $\langle p(X), \exists_Y c\rangle \longrightarrow \langle \exists\alpha\langle \mathit{Prog}(p),\ \alpha = X\rangle, \exists_Y c\rangle$.

- Case $\langle \sum_{i=1}^n \mathbf{ask}(c_i) \to A_i, c\rangle \longrightarrow \langle A_j, c\rangle$ where $c_j \leq c$.

  Since $Y \notin \mathit{var}(c_j) \subseteq \mathit{Vars}(\sum_{i=1}^n \mathbf{ask}(c_i) \to A_i)$, we have $c_j = \exists_Y c_j \leq \exists_Y c$ and hence $\langle \sum_{i=1}^n \mathbf{ask}(c_i) \to A_i, \exists_Y c\rangle \longrightarrow \langle A_j, \exists_Y c\rangle$.

- Case $\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i, c\rangle \longrightarrow \langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i, c\rangle$ where $\forall i(c_i \wedge c = \mathit{false})$.

  Let i be any of $1, \ldots, n$. Since $Y \notin \mathit{var}(c_i) \subseteq \mathit{Vars}(\sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i)$, we have $c_i = \exists_Y c_i$ and hence $c_i \wedge \exists_Y c = \exists_Y(c_i \wedge c) = \exists_Y \mathit{false} = \mathit{false}$.

  It follows that $\langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i, \exists_Y c\rangle \longrightarrow \langle \sum_{i=1}^{n} \mathbf{ask}(c_i) \to A_i, \exists_Y c\rangle$.

- Case $\langle \exists X \langle B, d\rangle, c\rangle \longrightarrow \langle \exists X \langle B', d'\rangle, c \wedge \exists_X d'\rangle$.

  By definition of external input, we have $\hat{A} = \exists X \langle \hat{B}, \hat{d}\rangle$ and $\hat{d} \wedge \exists_X c = d \wedge \exists_X c$ and $\hat{B}(d \wedge \exists_X c) = B(d \wedge \exists_X c)$. We must show the assumption $Y \notin \mathit{Vars}(\hat{B})$ in order to use the induction hypothesis for $\langle B, d \wedge \exists_X c\rangle \longrightarrow \langle B', d'\rangle$.

  We first consider the case where Y is other than X. In this case, we have $Y \notin \mathit{Vars}(\hat{B})$. Hence by induction hypothesis there exists $\hat{B}'$ such that

  $$\langle \hat{B}, \exists_Y (d \wedge \exists_X c)\rangle \longrightarrow \langle \hat{B}', \exists_Y d'\rangle$$

  and $Y \notin \mathit{Vars}(\hat{B}')$ and $\hat{B}' d' = B' d'$. Since $\exists_Y \hat{d} = \hat{d}$, this means that

  $$\langle \hat{B}, \hat{d} \wedge \exists_X \exists_Y c\rangle \longrightarrow \langle \hat{B}', \exists_Y d'\rangle$$

  and thus we have

  $$\langle \exists X \langle \hat{B}, \hat{d}\rangle, \exists_Y c\rangle \longrightarrow \langle \exists X \langle \hat{B}', \exists_Y d'\rangle, \exists_Y c \wedge \exists_X \exists_Y d'\rangle.$$

  All what we need is to prove:

  1. $\exists_Y (c \wedge \exists_X d') = \exists_Y c \wedge \exists_X \exists_Y d'$,
  2. $\exists_Y d' \wedge \exists_X (c \wedge \exists_X d') = d' \wedge \exists_X (c \wedge \exists_X d')$,
  3. $\hat{B}'(d' \wedge \exists_X (c \wedge \exists_X d')) = B'(d' \wedge \exists_X (c \wedge \exists_X d'))$, and
  4. $Y \notin \mathit{Vars}(\exists X \langle \hat{B}', \exists_Y d'\rangle)$.

  Proving statement 4 is easy.

  Equation 3 holds because $\hat{B}' d' = B' d'$.

  Equation 2 states that $\exists_Y d' \wedge \exists_X c \wedge \exists_X d' = d' \wedge \exists_X c$. We prove this by showing that $\exists_Y d' \wedge \exists_X c = d'$. By Proposition 3.2 applied to $\langle \hat{B}, d \wedge \exists_X c\rangle \longrightarrow \langle -, d'\rangle$, there exists some f such that $\mathit{var}(f) \subseteq$

$Vars(\hat{B})$ and $d' = f \wedge d \wedge \exists_X c$. Since $\exists_Y f = f$ and $\exists_Y \hat{d} = \hat{d}$ and $\hat{d} \wedge \exists_X c = d \wedge \exists_X c$, we have

$$\begin{aligned} \exists_Y d' \wedge \exists_X c &= \exists_Y (f \wedge d \wedge \exists_X c) \wedge \exists_X c \\ &= \exists_Y (f \wedge \hat{d} \wedge \exists_X c) \wedge \exists_X c \\ &= f \wedge \hat{d} \wedge \exists_Y \exists_X c \wedge \exists_X c \\ &= f \wedge d \wedge \exists_X c \\ &= d'. \end{aligned}$$

These proved that $\exists X \langle \hat{B}', \exists_Y d' \rangle (c \wedge \exists_X d') = \exists X \langle B', d' \rangle (c \wedge \exists_X d')$.

Finally, equation 1 holds since $\exists_Y (c \wedge \exists_X d') = \exists_Y (c \wedge \exists_X (\exists_Y d' \wedge \exists_X c)) = \exists_Y (c \wedge \exists_X \exists_Y d' \wedge \exists_X c) = \exists_Y (c \wedge \exists_X \exists_Y d') = \exists_Y c \wedge \exists_X \exists_Y d'$.

Consider the case where X is equal to Y. Recall that $\langle B, d \wedge \exists_X c \rangle \longrightarrow \langle B', d' \rangle$ and $\hat{B}(d \wedge \exists_X c) = B(d \wedge \exists_X c)$. By Theorem 2.1, there exists $\hat{B}'$ such that $\langle \hat{B}, d \wedge \exists_X c \rangle \longrightarrow \langle \hat{B}', d' \rangle$ and $\hat{B}' d' = B' d'$. Hence we have $\langle \exists X \langle \hat{B}, d \rangle, \exists_X c \rangle \longrightarrow \langle \exists X \langle \hat{B}', d' \rangle, \exists_X c \wedge \exists_X d' \rangle$. Note that $\exists_X c \wedge \exists_X d' = \exists_X (c \wedge \exists_X d')$.

- Case $\langle B_1 \parallel B_2, c \rangle \longrightarrow \langle B_1' \parallel B_2, c' \rangle$.

  By definition of external input, we have $\hat{A} = \hat{B}_1 \parallel \hat{B}_2$ and $\hat{B}_1 c = B_1 c$ and $\hat{B}_2 c = B_2 c$. We also have $Y \notin Vars(\hat{B}_1) \cup Vars(\hat{B}_2)$. By induction hypothesis for $\langle B_1, c \rangle \longrightarrow \langle B_1', c' \rangle$, there exists $\hat{B}_1'$ such that $\langle \hat{B}_1, \exists_Y c \rangle \longrightarrow \langle \hat{B}_1', \exists_Y c' \rangle$ and $\hat{B}_1' c' = B_1' c'$ and $Y \notin Vars(\hat{B}_1')$. Hence, we have $\langle \hat{B}_1 \parallel \hat{B}_2, \exists_Y c \rangle \longrightarrow \langle \hat{B}_1' \parallel \hat{B}_2, \exists_Y c' \rangle$.

  Moreover, we have $(\hat{B}_1' \parallel \hat{B}_2) c' = \hat{B}_1' c' \parallel \hat{B}_2 c' = B_1' c' \parallel B_2 c' = (B_1' \parallel B_2) c'$.

  Finally, we have $Y \notin Vars(\hat{B}_1') \cup Vars(\hat{B}_2) = Vars(\hat{B}_1' \parallel \hat{B}_2)$.

  The case of $\langle B_1 \parallel B_2, c \rangle \longrightarrow \langle B_1 \parallel B_2', c' \rangle$ is similar. □

This proposition states that an agent asks and tells through only those variables contained in itself. This together with the condition $Vars(Prog(p)) \subseteq \{\alpha\}$ for predicate p means that clauses have no access to global variables.

Proposition 3.6 Let $\langle G_0, b_0 \rangle$ be a process and $Y \notin Vars(G_0)$. For every step-by-step interaction sequence

$$\langle G_{i-1}, a_i \rangle \longrightarrow \langle G_i, b_i \rangle, \ i = 1, \ldots, n$$

of $\langle G_0, b_0\rangle$, there exists an interaction sequence

$$\langle \hat{G}_{i-1}, \exists_Y a_i\rangle \longrightarrow \langle \hat{G}_i, \exists_Y b_i\rangle,\ i = 1, \ldots, n$$

of $\langle G_0, \exists_Y b_0\rangle$ such that $\hat{G}_i a_i = G_i a_i$ for $i = 1, \ldots, n$ where $\hat{G}_0 = G_0$.

Moreover, if $\langle G_n, b_n\rangle \longrightarrow^{\infty}$ then one of such $\hat{G}_n$ satisfies $\langle \hat{G}_n, \exists_Y b_n\rangle \longrightarrow^{\infty}$.

Proof. Corollary of Proposition 3.5. □

Proposition 3.7 Let $\langle A, c\rangle$ be a process and $Y \notin \mathit{Vars}(A)$. If $\langle A, c\rangle \longrightarrow^* \surd$, then for every agent $\hat{A}$ satisfying $\hat{A}c = Ac$ we have $\langle \hat{A}, \exists_Y c\rangle \longrightarrow^* \surd$.

Proof. By assumption, there exists an $S \in \mathit{Stop}$ such that $\langle A, c\rangle \longrightarrow^* \langle S, c\rangle$. By Proposition 3.5 there exists some $\hat{S}$ such that $\langle \hat{A}, \exists_Y c\rangle \longrightarrow^* \langle \hat{S}, \exists_Y c\rangle$ and $\hat{S}c = Sc$. By definition of external input, it holds that $\hat{S} \in \mathit{Stop}$. □

Finally, we have the following theorem.

Theorem 3.1 $\langle \mathit{true}, b_0\rangle\langle a_1, b_1\rangle \ldots \langle a_n, b_n\rangle m \in \mathcal{D}[\![\langle G_0, b_0\rangle]\!]$ implies that $\langle \mathit{true}, \exists_Y b_0\rangle\langle \exists_Y a_1, \exists_Y b_1\rangle \ldots \langle \exists_Y a_n, \exists_Y b_n\rangle m \in \mathcal{D}[\![\langle G_0, \exists_Y b_0\rangle]\!]$ if $Y \notin \mathit{Vars}(G_0)$.

Proof. By Propositions 3.6 and 3.7. □

Intuitively, this theorem states that every agent performs asks or tells of constraints on only those variables which occur syntactically in that agent.

Chapter 4

On the Safety of Moving Synchronization Points

This chapter shows a program transformation technique called the motion of synchronization points, its implementation algorithm, and a proof of its correctness. The algorithm is based on fixed-point abstract interpretation. We will discuss in which sense the transformation is correct. In particular, we prove that the transformation does not introduce deadlocks. In the discussion of the correctness of the program transformation, we refer to the denotational semantics introduced in Chapter 3.

4.1 Overview

This section introduces the motion of synchronization points and briefly explains how to prove its correctness in concurrent logic programming.

4.1.1 Moving Synchronization Points

In order to execute concurrent programs on sequential machines, control transfer between processes is required. The time point when the control transfers from one process to another is called a synchronization point. Transferring control to a process that is not ready for computation results in spawning child processes that all suspend without observable output, which can be an overhead in the runtime system. This overhead can be eliminated by delaying the transfer of control until a sufficient amount of information needed

by the process to produce some results becomes ready. This optimization is called the motion of synchronization points.

Delaying computation too much, however, would introduce deadlocks to the program. This chapter clarifies how to give a theoretical justification of the safety of the motion.

In concurrent constraint programming, moving synchronization points amounts to lifting up the ask operations. Since the asks should be lifted up beyond predicate calls to whichever appropriate subagents within the whole program, the formalization of a program transformation algorithm for the full syntax of concurrent constraint programming could be troublesome. So, we restrict ourselves to concurrent logic programming since the transformation takes place only at the guard parts of program clauses. In concurrent logic programming, the transformation is expressed as the strengthening of the constraint in the guards.

4.1.2 Example of Moving Synchronization Points

The following program shows a definition of the predicate `sigma`. In this program, the call `sigma(K,N,Tmp,S)` computes the value of $\texttt{Tmp} + \sum_{i=\texttt{K}}^{\texttt{N}-1} 1/i^2$ and binds it to `S`. Here, `$:=` is the built-in predicate that computes the right-hand side expression and binds the result as a floating point value to the left-hand side variable. It suspends until all the variables in the right-hand side are bound to non-variable terms.

```
sigma(K,N,Tmp,S) :- K=<N | S $:= Tmp.
sigma(K,N,Tmp,S) :- K >N |
        FK $:= float(K),
        K1 := K+1,
        Tmp1 $:= 1.0/(FK*FK)+Tmp,
        sigma(K1,N,Tmp1,S).
```

Our optimization will translate this program into the following one:

```
sigma(K,N,Tmp,S) :- K=<N | S $:= Tmp.
sigma(K,N,Tmp,S) :- K >N, wait(Tmp) |
        FK $:= float(K),
        K1 := K+1,
        Tmp1 $:= 1.0/(FK*FK)+Tmp,
        sigma(K1,N,Tmp1,S).
```

Recall that the constraint `wait(Tmp)` specifies that this rule waits until `Tmp` is bound to a non-variable term. This transformation will reduce the number of suspended predicate calls to `$:=` that introduce overheads.

Although these two predicate definitions seem to have the same response for any call, it is not easy to give a precise account of this fact. In this chapter, we prove that the behavior of these programs coincide in terms of the denotational semantics defined in Chapter 3.

4.1.3 Basic Strategy

If a process can perform some output, there is a context that proceeds depending on that output. For instance, in the above `sigma` example, `S` can be fed to another predicate call that depends on `S`. Hence, we cannot in general defer the execution of `sigma` when it can bind something to `S`.

In concurrent constraint programming, output that can be observed by a process is limited to tells to the constraint store. On the other hand, a runtime system of concurrent constraint programming can observe from the program more than just constraint tells. Consider the following program.

```
countdown(K) :- K=<0 | true.
countdown(K) :- K >0 | K1 := K-1, countdown(K1).
```

Any call to `countdown` does not perform any constraint tells, but deferring its reduction forever will introduce deadlock. In order to discuss this issue precisely, we need a denotational semantics that takes termination into account. This issue was not considered in the work of Demand Transformation Analysis [16] where only constraint tells were formulated.

In this chapter, we discuss how our program transformation is correct in terms of the observable output of processes. Recall that we have defined the observable output of a process to include termination, failure, and divergence in addition to constraint tells.

4.2 Abstract Store Space

In order to formalize our abstract interpretation algorithm, we introduce a unary abstraction operator on constraints that computes partial information. We also introduce a few additional assumptions to the store space *Con* so that the abstraction is correct.

4.2.1 Abstraction Operator

First of all, we must have an operator for making abstraction of the constraint store. At the same time, we want to specify the class of possible constraints that can be added to the guards of clauses in the motion of synchronization points. Hence, we assume an operator on constraints, written by β, that computes certain partial information of a constraint given as its argument which can also serve as the abstraction of the original constraint.

Assumption 4.1 There exists a mapping $\beta : \mathit{Con} \to \mathit{Con}$ enjoying the following:

(B1) $\beta(c) \leq c$

(B2) $c \leq d$ implies $\beta(c) \leq \beta(d)$

(B3) $\beta(\exists_X(c)) = \exists_X(\beta(c))$

(B4) $\beta(c \wedge d) = \beta(\beta(\exists_{var(d)\setminus var(c)}(c \wedge d)) \wedge d)$ if $d \neq \mathit{false}$

(B5) $c \wedge \beta(d) = \mathit{false}$ implies $c = \mathit{false}$ if $\beta(d) \neq \mathit{false}$

□

(B5) states that any $\beta(d) \neq \mathit{false}$ does not introduce inconsistency.

Given such a mapping β, the set $\{\beta(c) \mid c \in \mathit{Con}\}$ forms an abstract store space where for each $c \in \mathit{Con}$, $\beta(c)$ represents some information that increases according to the program execution.

Proposition 4.1 Assume $c \wedge d \neq \mathit{false}$ and $var(c) \cap var(d) = \{\}$. We have $\beta(c \wedge d) = \beta(c) \wedge \beta(d)$.

Proof.

$$\begin{aligned}
\beta(c \wedge d) &= \beta(\beta(\exists_{var(d)\setminus var(c)}(c \wedge d)) \wedge d) \\
&= \beta(\beta(\exists_{var(d)}(\exists_{var(d)}(c) \wedge d)) \wedge d) \\
&= \beta(\exists_{var(d)}(\beta(c)) \wedge d) \\
&= \beta(\beta(c) \wedge \beta(d)) \\
&\leq \beta(c) \wedge \beta(d) \leq \beta(c \wedge d).
\end{aligned}$$

□

4.2.2 Instantiation Information

Our program transformation amounts to adding to the guards of clauses constraints of the form `wait(`X`)` defined in Section 2.2.3. Hence, we will restrict β as follows:

Definition 4.1 Let c be a constraint. A variable X is said to be *instantiated* in c if and only if $\texttt{wait(}X\texttt{)} \leq c$. We denote by $bound(c)$ the set of all the variables instantiated in c and call the set the *instantiation information* of c:

$$bound(c) \stackrel{\text{def}}{=} \{X \in Var \mid \texttt{wait(}X\texttt{)} \leq c\}\,.$$

We will then let

$$\beta(c) = \bigwedge_{X \in bound(c)} \texttt{wait(}X\texttt{)}.$$

□

Note that this definition satisfies the conditions in Assumption 4.1.
We have $bound(c) \subseteq bound(d)$ if and only if $\beta(c) \leq \beta(d)$.
We also have $bound(c) = var(\beta(c))$.

Example.

$$\begin{aligned} bound(\texttt{A=B}) &= var(true) = \{\} \\ bound(\texttt{A=f(B)}) &= var(\texttt{wait(A)}) = \{\texttt{A}\} \\ bound(\texttt{A=f(B)} \wedge \texttt{C=D} \wedge \texttt{E=1}) &= var(\texttt{wait(A)} \wedge \texttt{wait(E)}) = \{\texttt{A}, \texttt{E}\}\,. \end{aligned}$$

□

4.2.3 Additional Assumptions on Store

Our plan is to make abstraction of a given constraint c with $var(c)$ and to compute the instantiation information of c from $var(c)$. To make abstraction of a tell work correctly, we need the following assumption:

Assumption 4.2 $c \wedge d \neq false$ implies $var(c \wedge d) = var(c) \cup var(d)$. □

As mentioned in Section 2.2.4, this assumption makes *var* monotonic. Moreover, we have $bound(c) \subseteq var(c)$ because of $\beta(c) \leq c$.

Finally, we will employ the following assumption in order to ensure that the existence of constraint output can be known only from the constrained variable set, i.e. $var(c)$:

Assumption 4.3 $var(c) = \{\}$ implies $c = true$. □

4.3 Required Variable Sets

Having defined *bound*, we can now describe the notion of a required variable set of a goal G. It is the set of all the variables required to be instantiated in the input to G in order that G performs some observable output.

Definition 4.2 (Required Variable Sets) For every goal G of concurrent logic programming, we define the *required variable set* of G, written as $req(G)$, as follows:

$$
\begin{aligned}
&req(G) = \bigcap \{V \subseteq Var \mid says(G,V) \neq \{\}\} \\
&says(G,V) \\
&\quad = \{X \in var(c') \mid bound(c) \subseteq V \\
&\qquad\qquad \text{and } \langle \mathcal{T}[\![G]\!], c\rangle \longrightarrow^* \langle A', c'\rangle \text{ and } c \neq c' \neq false\} \\
&\quad \cup \{end \mid bound(c) \subseteq V \text{ and } \langle \mathcal{T}[\![G]\!], c\rangle \longrightarrow^* \langle S, c'\rangle \text{ and } S \in Stop\} \\
&\quad \cup \{fail \mid bound(c) \subseteq V \text{ and } \langle \mathcal{T}[\![G]\!], c\rangle \longrightarrow^* \langle A', false\rangle\} \\
&\quad \cup \{div \mid bound(c) \subseteq V \text{ and } \langle \mathcal{T}[\![G]\!], c\rangle \longrightarrow^\infty\}\,.
\end{aligned}
$$

□

We have a few words to these formulas. The first term in $says(G,V)$, a set of variables, represents the set union of the $var(c')$'s for all c' satisfying the condition part. The last term in $says(G,V)$ becomes $\{div\}$ if there exists some c such that $bound(c) \subseteq V$ and $\langle \mathcal{T}[\![G]\!], c\rangle \longrightarrow^\infty$, and otherwise it becomes the empty set. We have that $V \subseteq W$ implies $says(G,V) \subseteq says(G,W)$.

Let $c \in Con$. If $req(G)$ contains some variable not instantiated in c, then the process $\langle \mathcal{T}[\![G]\!], c\rangle$ never performs any observable output.

The set $says(G,V)$ expresses the set of variables through which the process $\langle \mathcal{T}[\![G]\!], c\rangle$ may perform observable output where $bound(c) \subseteq V$. The symbol *end* expresses the possibility of termination. Similarly, the symbols *fail* and *div* represent the possibilities of failure and divergence, respectively. These three symbols can be handled in principle in the same way as ordinary variables.

If $says(G,V)$ is the empty set, then we can safely defer the execution of the process $\langle \mathcal{T}[\![G]\!], c\rangle$ where $bound(c) \subseteq V$ since it has no possibilities of performing observable output.

A fixed-point computation algorithm will be presented later that computes a set $a(G,V)$ satisfying $says(G,V) \neq \{\} \Rightarrow a(G,V) \neq \{\}$. As we will see, we can obtain a subset of $req(G)$ from $a(G,V)$.

4.4 Formalization of Moving Synchronization Points

Assume that we have computed a subset of $req(G)$ for each G. We formalize the motion of synchronization points and prove that the motion does not alter the denotational semantics of the program agents.

4.4.1 Program Transformation

Given a concurrent logic program P, we introduce the transformed program, denoted by $P\$$, as follows. For each

$$(\ p(s) \ :\!- \ e \mid B \ ) \in P$$

we postulate

$$(\ p'(s) \ :\!- \ \beta(q) \wedge e \mid B\$ \ ) \in P\$$$

where $q \in Con$ and $bound(\beta(q)) \subseteq req(p(s)) \cap var(s)$ and p' is a new predicate symbol unique to p.

For each agent A in concurrent logic programming or concurrent constraint programming, we define $A\$$ as the agent A with every predicate call $p(X)$ replaced with $p'(X)$.

As long as the above condition is met, we can choose arbitrary q. For instance, we may want to let $\beta(q) = true$ for those clauses to which we do not perform our program transformation.

4.4.2 Safety of Transformation

In this subsection, we prove that moving synchronization points by adding $\beta(q)$ as explained above does not alter the denotational semantics.

The following proposition states that any reduction in a transformed program has a corresponding reduction in the original program.

Proposition 4.2 In the program $P \cup P\$$, the following holds:

$$\langle A\$, c\rangle \longrightarrow \langle A'\$, c'\rangle \text{ implies } \langle A, c\rangle \longrightarrow \langle A', c'\rangle.$$

Proof. By structural induction on $\longrightarrow$. The only nontrivial case is when A is of the form $\sum_{i=1}^{n} \mathbf{ask}(f_i) \to A_i$. We have two cases.

In the case where

$$\langle \textstyle\sum_{i=1}^{n} \mathbf{ask}(\overline{\exists}_\alpha((\alpha = s_i) \wedge e_i \wedge \beta(q_i))) \to A_i\$, c\rangle \longrightarrow \langle A_j\$, c\rangle,$$

we have $\langle A, c\rangle \longrightarrow \langle A_j, c\rangle$ since $\overline{\exists}_\alpha((\alpha = s_i) \wedge e_j) \leq \overline{\exists}_\alpha((\alpha = s_i) \wedge e_j \wedge \beta(q_j)) \leq c$.

In the other case, we have

$$\begin{array}{rl} & \langle \sum_{i=1}^{n} \mathbf{ask}(\overline{\exists}_\alpha((\alpha = s_i) \wedge e_i \wedge \beta(q_i))) \to A_i\$, c\rangle \\ \longrightarrow & \langle \sum_{i=1}^{n} \mathbf{ask}(\overline{\exists}_\alpha((\alpha = s_i) \wedge e_i \wedge \beta(q_i))) \to A_i\$, c\rangle. \end{array}$$

and $\forall i(\overline{\exists}_\alpha((\alpha = s_i) \wedge e_i \wedge \beta(q_i)) \wedge c = \mathit{false})$. To prove $\langle A, c\rangle \longrightarrow \langle A, c\rangle$, we must show that $\overline{\exists}_\alpha((\alpha = s_i) \wedge e_i \wedge \beta(q_i)) \wedge c = \mathit{false}$ implies $\overline{\exists}_\alpha((\alpha = s_i) \wedge e_i) \wedge c = \mathit{false}$ for all i, and this is done using the property (B5) of β. □

The following is the main theorem in this chapter. It states that any computation in a program can be simulated in the transformed program.

Theorem 4.1 For the program $P \cup P\$$, we have $\langle \mathcal{T}[\![G]\!], c\rangle \leq \langle \mathcal{T}[\![G]\!]\$, c\rangle$.

Proof. The key point of the proof is to show that whenever an ask agent performs some observable output, the corresponding guard strengthening $\beta(q)$ is always satisfied after a finite number of reductions.

We will prove that $\langle A, c\rangle \leq \langle A\$, c\rangle$ for those agents A such that every **ask** agent occurring within A is declared as $\mathit{Prog}(p)$ with some predicate symbol p. These agents form a class of agents that is preserved under reductions because $\mathit{Prog}(p)$ itself is in this class. Moreover, every $\mathcal{T}[\![G]\!]$ is also contained in this class. Hence, it is sufficient to consider only this class of agents.

To prove the theorem, we first construct for every interaction sequence

$$x \ : \ \langle G_{i-1}, a_i\rangle \longrightarrow \langle G_i, b_i\rangle, \ i = 1, \ldots, n$$

of $\langle G_0, b_0\rangle$, an interaction sequence

$$y \ : \ \langle G'_{i-1}, a_i\rangle \longrightarrow^* \langle G'_i, b_i\rangle, \ i = 1, \ldots, n$$

of $\langle G_0\$, b_0\rangle$.

This is done by mathematical induction on the length n for all G_0's at a time, and then by structural induction on each G_0.

Let $n \geq 1$. We proceed depending on the top-level construct of G_0:

- Case $\mathbf{tell}(e)$.

  We have $\langle \mathbf{tell}(e)\$, b_0\rangle = \langle \mathbf{tell}(e), b_0\rangle \longrightarrow \langle \mathbf{stop}, e \wedge b_0\rangle$.

- Case $\exists X\langle H_0, c_0\rangle$.

  The interaction sequence x is of the form

  $$\langle \exists X\langle H_{i-1}, c_{i-1}\rangle, a_i\rangle \longrightarrow \langle \exists X\langle H_i, c_i\rangle, b_i\rangle, \quad i = 1, \ldots, n$$

  where

  $$\langle H_{i-1}, c_{i-1} \wedge \exists_X a_i\rangle \longrightarrow \langle H_i, c_i\rangle, \quad i = 1, \ldots, n$$

  and

  $$b_i = a_i \wedge \exists_X c_i, \quad i = 1, \ldots, n.$$

  By induction hypothesis there exists an interaction sequence

  $$\langle H'_{i-1}, c_{i-1} \wedge \exists_X a_i\rangle \longrightarrow^* \langle H'_i, c_i\rangle, \quad i = 1, \ldots, n$$

  of $\langle H_0\$, c_0\rangle$. This leads to

  $$\begin{array}{rl} & \langle \exists X\langle H'_{i-1}, c_{i-1}\rangle, a_i\rangle \\ \longrightarrow^* & \langle \exists X\langle H'_i, c_i\rangle, a_i \wedge \exists_X c_i\rangle \\ = & \langle \exists X\langle H'_i, c_i\rangle, b_i\rangle, \quad i = 1, \ldots, n \end{array}$$

  and this is an interaction sequence of $\langle \exists X\langle H_0, c_0\rangle, b_0\rangle\$$.

- Case $p(X)$.

  Since $p(X)$ has exactly one outgoing reduction of the form

  $$\langle p(X), a_1\rangle \longrightarrow \langle \exists\alpha\langle \mathit{Prog}(p),\ \alpha = X\rangle, a_1\rangle,$$

  we have

  $$\langle p'(X), a_1\rangle \longrightarrow \langle \exists\alpha\langle \mathit{Prog}(p'),\ \alpha = X\rangle, a_1\rangle.$$

  Since $\mathit{Prog}(p') = \mathit{Prog}(p)\$$, we can make use of induction hypothesis on length.

- Case $\sum_{i=1}^{n} \mathbf{ask}(f_i) \rightarrow A_i$.

  We determine k as follows:

  - If $\langle G_n, b_n\rangle \longrightarrow^{\infty}$, there exists the smallest positive integer k satisfying $\langle G_k, b_k\rangle \longrightarrow^{\infty}$. We call this k by k_1 if any.

– If $\langle G_n, b_n\rangle \longrightarrow^* \surd$, there exists the smallest positive integer k satisfying $\langle G_k, b_k\rangle \longrightarrow^* \surd$. We call this k by k_2 if any.
– There may exist the smallest positive integer k satisfying $a_k \neq b_k$. We call this k by k_3 if any.

If none of the above applies, for every $i = 1, \ldots, n$ it holds that $a_i = b_i$. In this case, letting $G'_i = G_0\$$ for $i = 1, \ldots, n$ is sufficient. (k is not set but the proof is done). Otherwise, at least one of the above applies. So, let k be the smallest of k_1, k_2, and k_3.

Let $V = var(a_k) \setminus \{\alpha\}$.

The first reduction in x is of the form

$$\langle \textstyle\sum_{i=1}^{n} \mathbf{ask}(f_i) \rightarrow A_i, a_1\rangle \longrightarrow \langle A_j, a_1\rangle.$$

By assumption, there must be a predicate symbol p such that

$$\begin{array}{l} \mathit{Prog}(p) \ = G_0 = \sum_{i=1}^{n} \mathbf{ask}(\overline{\exists}_\alpha((\alpha = s_i) \wedge e_i)) \rightarrow A_i, \ \text{ and} \\ \mathit{Prog}(p') = G'_0 = \sum_{i=1}^{n} \mathbf{ask}(\overline{\exists}_\alpha((\alpha = s_i) \wedge e_i \wedge \beta(q_i))) \rightarrow A_i\$. \end{array}$$

Since $\overline{\exists}_\alpha((\alpha = s_j) \wedge e_j) \leq a_k$, we have $\overline{\exists}_\alpha((\alpha = s_j) \wedge e_j) \leq \exists_V a_k$. We also have that $\exists_{\mathit{Vars}(s_j)}(\alpha = s_j) \leq \exists_V a_k$.

By Proposition 3.6, there exists a step-by-step interaction sequence

$$\langle \hat{G}_{i-1}, \exists_V a_i\rangle \longrightarrow \langle \hat{G}_i, \exists_V b_i\rangle, \quad i = 1, \ldots, n$$

such that $\hat{G}_i a_i = G_i a_i$ for $i = 1, \ldots, n$ where $\hat{G}_0 = G_0$. We also have that $\langle \hat{G}_n, \exists_V b_n\rangle \longrightarrow^\infty$ if $\langle G_n, b_n\rangle \longrightarrow^\infty$.

Let

$$\begin{array}{l} g = \exists_\alpha(\alpha = s_j \wedge \exists_V a_k), \\ c = g \wedge (G = s_j) = \exists_\alpha(G = s_j \wedge \alpha = s_j \wedge \exists_V a_k), \ \text{and} \\ c' = c \wedge \exists_\alpha(\alpha = G \wedge c \wedge \exists_V b_k) = \exists_\alpha(\alpha = G \wedge c \wedge \exists_V b_k) \end{array}$$

where G is a variable not in $var(a_k) \cup \{\alpha\} \cup \mathit{Vars}(s_j)$.

By Theorem 2.1, there exists an agent E_k satisfying

$$\langle G_0, (\alpha = G) \wedge c\rangle \longrightarrow^* \langle E_k, \alpha = G \wedge c \wedge \exists_V b_k\rangle$$

and $E_k(\alpha = G \wedge c \wedge \exists_V b_k) = G_k(\alpha = G \wedge c \wedge \exists_V b_k)$.

Therefore, we have

$$\begin{array}{l} \langle \exists \alpha \langle G_0, \alpha = G \wedge c \rangle, c \rangle \\ \longrightarrow^* \langle \exists \alpha \langle E_k, \alpha = G \wedge c \wedge \exists_V b_k \rangle, c' \rangle \end{array}$$

and

$$\begin{array}{rl} & \langle \mathcal{T}[\![p(s_j)]\!], g \rangle \\ = & \langle \exists G \langle p(G), G = s_j \rangle, g \rangle \\ \longrightarrow & \langle \exists G \langle \exists \alpha \langle G_0, \alpha = G \rangle, c \rangle, g \rangle \\ \longrightarrow^* & \langle \exists G \langle \exists \alpha \langle E_k, \alpha = G \wedge c \wedge \exists_V b_k \rangle, c' \rangle, g \wedge \exists_G c' \rangle. \end{array}$$

Recall that we have at least one of the following:

1. $\langle G_k, b_k \rangle \longrightarrow^\infty$ holds.
2. $\langle G_k, b_k \rangle \longrightarrow^* \surd$ holds.
3. $a_k \neq b_k$ holds.

We will prove $says(p(s_j), bound(g)) \neq \{\}$ for each of these cases.

- Case 1.
  Since $\langle G_k, \exists_V b_k \rangle \longrightarrow^\infty$, we have $\langle E_k, \alpha = G \wedge c \wedge \exists_V b_k \rangle \longrightarrow^\infty$ and therefore

  $$\langle \exists G \langle \exists \alpha \langle E_k, \alpha = G \wedge c \wedge \exists_V b_k \rangle, c' \rangle, g \wedge \exists_G c' \rangle \longrightarrow^\infty.$$

  And hence we have $div \in says(p(s_j), bound(g))$.
- Case 2.
  By Proposition 3.7 we have that $\langle G_k, \exists_V b_k \rangle \longrightarrow \surd$. Hence, we have $\langle E_k, \alpha = G \wedge c \wedge \exists_V b_k \rangle \longrightarrow^* \surd$ and therefore

  $$\langle \exists G \langle \exists \alpha \langle E_k, \alpha = G \wedge c \wedge \exists_V b_k \rangle, c' \rangle, g \wedge \exists_G c' \rangle \longrightarrow^* \surd.$$

  And hence we have $end \in says(p(s_j), bound(g))$.
- Case 3.
  By definition of *Prog*, we have that $Vars(G_0) \subseteq \{\alpha\}$. By Proposition 3.2, there exists some f such that $var(f) \subseteq Vars(G_0)$ and $b_k = f \wedge a_k$. Thus, we have $\exists_V f = f$ and therefore $\exists_V b_k = f \wedge \exists_V a_k$. If $f \leq \exists_V a_k$ then we have $f \leq \exists_V a_k \leq a_k$ and this is not consistent with $a_k \neq b_k$. Therefore, we have that $\exists_V a_k \neq \exists_V b_k$.

Let us assume, hypothetically, that $\exists_G c' \leq g$. Since $\exists_G c' = \exists_\alpha(\alpha = s_j \wedge g \wedge \exists_V b_k) \geq \exists_\alpha g = g$, we have $\exists_G c' = g$. This means, however, that $\exists_V a_k = \exists_V b_k$ since

$$\begin{array}{rl} & \exists_{Vars(s_j)}(\alpha = s_j \wedge g) \\ = & \exists_{Vars(s_j)}(\alpha = s_j \wedge \exists_\alpha(\alpha = s_j \wedge \exists_V a_k)) \\ = & \exists_{Vars(s_j)}(\alpha = s_j \wedge \exists_G(\alpha = G \wedge \exists_\alpha(\alpha = G \wedge \exists_V a_k))) \\ = & \exists_{Vars(s_j)}(\alpha = s_j \wedge \exists_V a_k) \\ = & \exists_{Vars(s_j)}(\alpha = s_j) \wedge \exists_V a_k \\ = & \exists_V a_k \end{array}$$

and

$$\begin{array}{rl} & \exists_{Vars(s_j)}(\alpha = s_j \wedge \exists_G c') \\ = & \exists_{Vars(s_j)}(\alpha = s_j \wedge \exists_G \exists_\alpha(\alpha = G \wedge g \wedge G = s_j \wedge \exists_V b_k)) \\ = & \exists_{Vars(s_j)} \exists_G(\alpha = s_j \wedge \exists_\alpha(\alpha = G \wedge g \wedge G = s_j \wedge \exists_V b_k)) \\ = & \exists_{Vars(s_j)} \exists_G(\alpha = s_j \wedge G = s_j \wedge \exists_\alpha(\alpha = G \wedge g \wedge \exists_V b_k)) \\ = & \exists_{Vars(s_j)} \exists_G(\alpha = s_j \wedge \alpha = G \wedge \exists_\alpha(\alpha = G \wedge g \wedge \exists_V b_k)) \\ = & \exists_{Vars(s_j)}(\alpha = s_j \wedge \exists_G(\alpha = G \wedge \exists_\alpha(\alpha = G \wedge g \wedge \exists_V b_k))) \\ = & \exists_{Vars(s_j)}(\alpha = s_j \wedge g \wedge \exists_V b_k) \\ = & \exists_{Vars(s_j)}(\alpha = s_j \wedge g) \wedge \exists_V b_k \\ = & \exists_V a_k \wedge \exists_V b_k \\ = & \exists_V b_k. \end{array}$$

This contradicts the assumption.

Thus, $g \wedge \exists_G c' \neq g$. Therefore, we have $says(p(s_j), bound(g)) \neq \{\}$.

Since $says(p(s_j), bound(g)) \neq \{\}$ implies $req(p(s_j)) \subseteq bound(g)$, we have $bound(\beta(q_j)) \subseteq req(p(s_j)) \subseteq bound(g)$ and thus we have $\beta(q_j) \leq \beta(g) \leq g$. Since

$$\begin{array}{l} \overline{\exists}_\alpha(\alpha = s_j \wedge e_j) \leq \exists_V a_k \\ \exists_\alpha(\alpha = s_j \wedge \overline{\exists}_\alpha(\alpha = s_j \wedge e_j)) \leq \exists_\alpha(\alpha = s_j \wedge \exists_V a_k) \\ \exists_G(G = s_j \wedge \exists_\alpha(\alpha = G \wedge \overline{\exists}_\alpha(\alpha = G \wedge e_j))) \leq g \\ \exists_G(G = s_j \wedge e_j) \leq g \\ e_j \leq g, \end{array}$$

we have $e_j \wedge \beta(q_j) \leq g$ and therefore

$$\overline{\exists}_\alpha(\alpha = s_j \wedge e_j \wedge \beta(q_j)) \leq \overline{\exists}_\alpha(\alpha = s_j \wedge g) = \exists_V a_k \leq a_k.$$

Thus, we have that

$$\langle G'_0, a_k\rangle \longrightarrow \langle G_1\$, a_k\rangle.$$

Let $(d_i, d'_i) = (a_k, a_k)$ for $i = 2, \ldots, k-1$ and $(d_i, d'_i) = (a_i, b_i)$ for $i = k, \ldots, n$. By Theorem 2.1 applied to $\langle G_{i-1}, a_i\rangle \longrightarrow \langle G_i, b_i\rangle,\ i = 2, \ldots, n$, there exists $\langle H_{i-1}, d_i\rangle \longrightarrow \langle H_i, d'_i\rangle,\ i = 2, \ldots, n$ such that $H_i d'_i = G_i d'_i$ for $i = 2, \ldots, n$ where $H_1 = G_1$. By induction hypothesis on length, there exists $\langle H'_{i-1}, d_i\rangle \longrightarrow^* \langle H'_i, d'_i\rangle,\ i = 2, \ldots, n$ where $H'_1 = H_1\$$. This means we have $\langle G'_0, a_i\rangle \longrightarrow^* \langle G'_0, b_i\rangle,\ i = 1, \ldots, k-1$ and $\langle G'_0, a_k\rangle \longrightarrow^* \langle H'_k, b_k\rangle$ and $\langle H'_{i-1}, a_i\rangle \longrightarrow^* \langle H'_i, b_i\rangle,\ i = k+1, \ldots, n$.

- Case $D_0 \parallel E_0$.

  By definition of $\longrightarrow$, we have

  $$\langle D_{i-1} \parallel E_{i-1}, a_i\rangle \longrightarrow \langle D_i \parallel E_i, b_i\rangle, \quad i = 1, \ldots, n$$

  where

  $$\begin{array}{l}\langle D_{i-1}, a_i\rangle \longrightarrow \langle D_i, b_i\rangle \ \text{ and } \ E_{i-1} = E_i, \quad i \in L, \ \text{ and} \\ \langle E_{i-1}, a_i\rangle \longrightarrow \langle E_i, b_i\rangle \ \text{ and } \ D_{i-1} = D_i, \quad i \in \{1, \ldots, n\} \setminus L.\end{array}$$

  By induction hypothesis on G_0, there exist two interaction sequences:

  $$\begin{array}{l}\langle D'_{i-1}, a_i\rangle \longrightarrow^* \langle D'_i, b_i\rangle, \quad i \in L, \\ \langle E'_{i-1}, a_i\rangle \longrightarrow^* \langle E'_i, b_i\rangle, \quad i \in \{1, \ldots, n\} \setminus L\end{array}$$

  where $D'_0 \parallel E'_0 = G_0\$$ and

  $$\begin{array}{l}D'_{i-1} = D'_i, \quad i \in \{1, \ldots, n\} \setminus L, \\ E'_{i-1} = E'_i, \quad i \in L.\end{array}$$

  Therefore, we have

  $$\langle D'_{i-1} \parallel E'_{i-1}, a_i\rangle \longrightarrow^* \langle D'_i \parallel E'_i, b_i\rangle, \quad i = 1, \ldots, n.$$

Now, we have constructed for each $\langle G_{i-1}, a_i\rangle \longrightarrow^* \langle G_i, b_i\rangle,\ i = 1, \ldots, n$, $\langle G'_{i-1}, a_i\rangle \longrightarrow^* \langle G'_i, b_i\rangle,\ i = 1, \ldots, n$. Notice that the above proof uniquely determines for each interaction sequence

$$\langle G_{i-1}, a_i\rangle \longrightarrow \langle G_i, b_i\rangle, \quad i = 1, \ldots, m$$

with $m \leq n$, the sequence

$$\langle G'_{i-1}, a_i \rangle \longrightarrow^* \langle G'_i, b_i \rangle, \quad i = 1, \ldots, m.$$

We can easily see that G'_i can be represented as a modification of $G_i\$$ where agents in $G_i\$$ may be replaced with asks.

Next, we prove that $\langle G_n, b_n \rangle \longrightarrow^* {}^{\surd}$ implies $\langle G'_n, b_n \rangle \longrightarrow^* {}^{\surd}$. To do this, we assume $G_n \in \mathit{Stop}$ and prove $G'_n \in \mathit{Stop}$. Note that G'_n cannot contain $p(X)$ or $\mathbf{tell}(e)$ because, if so, they must also be contained in G_n, which is not the case. Assume, hypothetically, that G'_n contains an ask agent $A\$$. Then, there exists some k such that the ask agent A is reduced within $\langle G_{k-1}, a_k \rangle \longrightarrow \langle G_k, b_k \rangle$ because otherwise G_n would contain the ask A. By the way, there exists the smallest integer m such that within $\langle G_{m-1}, a_m \rangle \longrightarrow \langle G_m, b_m \rangle$ the descendant of A performs some observable output, which may or may not be termination. Hence, according to the construction of G'_m, we have had the ask $A\$$ reduced within $\langle G'_{m-1}, a_m \rangle \longrightarrow^* \langle G'_m, b_m \rangle$, which contradicts to the hypothesis. Thus, it holds that $G'_n \in \mathit{Stop}$.

Finally, we prove that $\langle G_n, b_n \rangle \longrightarrow^\infty$ implies $\langle G'_n, b_n \rangle \longrightarrow^\infty$. We have

$$\langle G_{i-1}, a_i \rangle \longrightarrow \langle G_i, b_i \rangle, \quad i \geq 1 \tag{4.1}$$

where $a_i = b_{i-1}$ for all $i > n$. Notice that we have constructed this:

$$\langle G'_{i-1}, a_i \rangle \longrightarrow^* \langle G'_i, b_i \rangle, \quad i \geq 1.$$

Consider a minimal subagent A of G_n that performs infinite computation in (4.1) without receiving external input to A after $i = n$. If this computation produces an infinite number of different local store values at A, by the construction of G'_i's we have that $\langle G'_n, b_n \rangle \longrightarrow^\infty$. Otherwise, there exists some k such that G_{k-1} contains an ask B that performs infinite computation in (4.1) without receiving external input to B, without producing any constraints, and B is reduced within $\langle G_{k-1}, a_k \rangle \longrightarrow \langle G_k, b_k \rangle$. By construction of G'_i's, we have that $B\$$ is reduced within $\langle G'_{k-1}, a_k \rangle \longrightarrow \langle G'_k, b_k \rangle$. This means that we have $\langle G'_n, b_n \rangle \longrightarrow^\infty$. □

Finally, we have the following theorem which states the correctness of our program transformation.

Theorem 4.2 For the program $P \cup P\$$, $\mathcal{D}[\![\langle \mathcal{T}[\![G]\!], c \rangle]\!] = \mathcal{D}[\![\langle \mathcal{T}[\![G]\!]\$, c \rangle]\!]$ holds.

Proof. By Proposition 4.2 and Theorem 4.1. □

As a corollary, replacing all $p(t)$'s by $p'(t)$'s at a time does not alter the denotation of any process.

4.5 Practical Algorithm

4.5.1 The Algorithm

In the previous section, we have defined *req* taking divergence and failure into account. However, it is not advisable to compute the possibility of failure or divergence by means of abstract interpretation since abstraction loses information needed for the precise analysis of failure or divergence. On the other hand, most programs we want to perform optimizing compilation do not exhibit failure or divergence. Therefore, we introduce the following definition, *req*′, disregarding the possibility of divergence and failure:

$$
\begin{aligned}
&\mathit{req}'(G) = \bigcap \{V \subseteq \mathit{Var} \mid \mathit{says}'(G,V) \neq \{\}\} \\
&\mathit{says}'(G,V) \\
&\quad = \{X \in \mathit{var}(c') \mid \mathit{bound}(c) \subseteq V \\
&\qquad\qquad \text{and } \langle \mathcal{T}[\![G]\!], c\rangle \longrightarrow^* \langle A', c'\rangle \text{ and } c \neq c' \neq \mathit{false}\} \\
&\quad \cup \{\mathit{end} \mid \mathit{bound}(c) \subseteq V \text{ and } \langle \mathcal{T}[\![G]\!], c\rangle \longrightarrow^* \langle S, c'\rangle \text{ and } S \in \mathit{Stop}\}\,.
\end{aligned}
$$

For every G and V, it is the case that $\mathit{says}(G,V) \supseteq \mathit{says}'(G,V)$, and therefore $\mathit{req}(G) \subseteq \mathit{req}'(G)$. Note that $\mathit{req}'(G) = \mathit{req}(G)$ holds when the process $\langle \mathcal{T}[\![G]\!], c\rangle$ (where $c \neq \mathit{false}$) neither diverges nor fails since in this case the last two terms in the definition of *says* becomes empty.

It is easy to compute a superset of $\mathit{says}'(G,V)$ by fixed-point computation. The computation is done by the abstract interpretation which takes for each $c \in \mathit{Con}$ a finite set $\mathit{var}(c) \subseteq \mathit{Var}$ as its abstract store.

We will give a intuitive view of the algorithm. The algorithm in the form of formulas will be given later.

For every clause $(\ p(s)\ \texttt{:-}\ e \mid B\ ) \in P$ and every variable set $V \subseteq \mathit{var}(s)$, we compute a set $a(p(s),V)$ satisfying $\mathit{says}'(p(s),V) \neq \{\} \Rightarrow a(p(s),V) \neq \{\}$ as the least fixed point of the abstract store. The initial abstract store for $a(p(s),V)$ is the empty set. The k-th iteration of computing $a(p(s),V)$ generates those elements which are contained in $\mathit{says}'(p(s),V)$ and correspond to reduction sequences whose predicate calls are limited to the depth k.

Abstract computation for parallel composition is performed by local fixed-point computation using $a(p(s), V)$ computed so far. The computation of $a(p(s), V)$ can be done lazily to avoid irrelevant computation.

This fixed-point computation produces a subset of $req'(G)$.

4.5.2 Reforming a Process

The fixed-point method explained above preserves the denotational semantics of a process only if the process does not either fail or diverge. In the following, we will explain what is guaranteed if the process may fail or diverge.

Recall the definition of traces in Section 3.5.

Definition 4.3 We call a trace that ends with $\langle a, \mathit{false}\rangle \cdot \bot$ a *failed trace*. We say that $\langle A, c\rangle$ is *reformed* into $\langle A', c\rangle$ if and only if:

- every trace in $\mathcal{D}[\![\langle A', c\rangle]\!]$ that is failed or divergent is in $\mathcal{D}[\![\langle A, c\rangle]\!]$, and
- every trace in $\mathcal{D}[\![\langle A, c\rangle]\!]$ not failed or divergent is in $\mathcal{D}[\![\langle A', c\rangle]\!]$. □

It can be proved that if we use $req'(p(s))$ in place of $req(p(s))$ in the construction of $P\$$, we obtain a reformed process. This means that the semantics of processes is preserved if the processes neither diverge nor fail, and otherwise the processes are reformed in the sense that while divergence and failure may be removed, other normal computation is preserved.

4.6 Fixed-Point Algorithm

In this section, we will detail the fixed-point computation explained in Section 4.5.1.

4.6.1 Variable Translation

In order to make abstraction of parameter passing in a predicate call, we introduce a special notation which we call variable translation, and then define two operations on sets of variables, using this notation. The two operations defined are input variable translation, which expresses parameter passing from caller to callee, and output variable translation, expressing that from callee to caller.

For terms s and t, we call the notation $\{t \mapsto s\}$ the *variable translation* from t to s. We shall typically use σ to denote some variable translation. The intention is that we use $\{t \mapsto s\}$ for expressing the parameter passing mechanism that takes place when a predicate call $p(t)$ is matched with a clause head $p(s)$ for some predicate symbol p.

In order to explain the use of a variable translation $\{t \mapsto s\}$, we adopt the following syntactic convention. For any subterm u of s, we shall write u' to mean the term u with every variable X occurring in u replaced with its corresponding variable X' that is unique to X and does not occur in $var(t)$.

Notice that $(t = s') \neq \mathit{false}$ means that the call $p(t)$ can match with the clause head $p(s)$. In writing $\{t \mapsto s\}$, we shall assume that $(t = s') \neq \mathit{false}$. Now, we can decompose the equation $t = s'$ into a conjunction of equations by iteratively replacing $f(t_1, \ldots, t_n) = f(s'_1, \ldots, s'_n)$ with $t_1 = s'_1 \wedge \ldots \wedge t_n = s'_n$. Let

$$X_1 = s'_1 \wedge \ldots \wedge X_m = s'_m \wedge t_1 = Y'_1 \wedge \ldots \wedge t_n = Y'_n \tag{4.2}$$

be a conjunction obtained by fully decomposing $t = s'$ and then by reordering the conjuncts arbitrarily so that $X_i \in \mathit{Var}$ and $t_j \notin \mathit{Var}$ for all i and j. We have that $X_i \in var(t)$ and $Y_j \in var(s)$.

Let U be the set of all the variables that occur more than once in the left-hand side of the conjunction (4.2). We have $U \subseteq var(t)$. Since these variables impose constraints to some of the variables in the clause head, we need to consider them in order to have correct abstraction of the execution.

Here, we introduce two operations for a given $\sigma = \{t \mapsto s\}$ together with the conjunction (4.2) and the set U:

Definition 4.4 For $V \subseteq var(t)$, we define the *input variable translation* of V by σ, written as $V\sigma$, by

$$V\sigma \stackrel{\text{def}}{=} \{Y_1, \ldots, Y_n\} \cup \bigcup_{X_i \in U \cup V} var(s_i).$$

For $W \subseteq var(s)$, we define the *output variable translation* of W by σ, written as $W\overline{\sigma^{-1}}$, by

$$W\overline{\sigma^{-1}} \stackrel{\text{def}}{=} \{X_i \mid var(s_i) \cap W \neq \{\}\} \cup \bigcup_{Y_j \in W} var(t_j).$$

□

As we have said, $\sigma = \{t \mapsto s\}$ is used for expressing parameter passing in a call $p(t)$ matched with a clause head $p(s)$ for some p.

$V\sigma$ contains all the variables in $var(s)$ that can be constrained when V contains all the variables constrained in the call. A variable Y in the clause head $p(s)$ has three possible sources of being constrained by the caller. The first case is that Y is some Y_j, in which case Y is instantiated with a non-variable term t_j prior to the execution of the callee. The second case is that Y occurs in some s_j and the caller variable X_j to which s_j is bound is specified as possibly constrained, that is, $X_j \in V$. In this case, Y may be constrained since Y is a subterm of s_j that is constrained. The third case is that Y occurs in some s_j and there exists some variable Z occurring in the right-hand side of a conjunct the left-hand side of which contains the variable X_j, that is, $X_j \in U$. In this case, Y may be constrained in terms of Z.

On the other hand, $W\overline{\sigma^{-1}}$ contains all the variables in $var(t)$ that can be constrained when W contains all the variables constrained in the clause head. A variable X in the call $p(t)$ has two possible sources of being constrained by the callee. The first case is that X is some X_i and its corresponding head term s_i contains a variable Y specified as possibly constrained, that is, $Y \in W$. The second case is that X occurs in some t_j and the variable Y_j to which t_j is bound is specified as possibly constrained, that is, $Y_j \in W$.

Both of these variable translation are monotonic with respect to set inclusion.

Example. Let $\sigma = \{(\texttt{A,B,f(g(B,C),1)}) \mapsto (\texttt{h(X),Y,f(Z,W)})\}$.

We can use this σ to make abstraction of parameter passing in a call `p(A,B,f(g(B,C),1))` matched with a clause head `p(h(X),Y,f(Z,W))`.

By decomposing the equation

$$(\texttt{A,B,f(g(B,C),1)}) = (\texttt{h(X),Y,f(Z,W)}),$$

we obtain the conjunction

$$\texttt{A} = \texttt{h(X)} \wedge \texttt{B} = \texttt{Y} \wedge \texttt{g(B,C)} = \texttt{Z} \wedge \texttt{1} = \texttt{W}.$$

Thus, we have the following:

$$\begin{aligned} \{\}\,\sigma &= \{\texttt{Y}, \texttt{Z}, \texttt{W}\} \\ \{\texttt{R}\}\,\overline{\sigma^{-1}} &= \{\} \\ \{\texttt{Z}\}\,\overline{\sigma^{-1}} &= \{\texttt{B}, \texttt{C}\} \end{aligned}$$

- $a_k(s = t, V) \stackrel{\text{def}}{=} var(s = t) \cup \{end\}$
- $a_0(p(t), V) \stackrel{\text{def}}{=} \{\}$
- $a_{k+1}(p(t), V) \stackrel{\text{def}}{=} \{\ X \in a_k(B, V\sigma)\overline{\sigma^{-1}} \mid$
  $(\ p(s)\ \texttt{:-}\ e \mid B\ ) \in P$
  and $\sigma = \{(t, end) \mapsto (s, end)\}$
  and $bound(\exists_\alpha(\alpha = t \wedge \exists_{Vars(s)}(\alpha = s))) \subseteq V$
  and $bound(e) \subseteq V\sigma\ \}$
- $a_k(G_1 \| \ldots \| G_n, V) \stackrel{\text{def}}{=} \text{fix}\, \lambda T.\, T \cup \bigsqcup_{i=1}^{n} a_k(G_i, V \cup T)$
  where for sets V and W we define that
  $$V \sqcup W \stackrel{\text{def}}{=} \begin{cases} V \cup W & \text{if } end \in V \cap W \\ (V \cup W) \setminus \{end\} & \text{if } end \notin V \cap W \end{cases}$$

Figure 4.1: Iterative formulas for computing fixed points

The set U of the left-hand side variables in the conjunction that occur more than once is $\{\texttt{B}\}$. Hence, the variables `Y` and `Z`, both occurring in the right-hand sides of conjuncts containing `B`, are always constrained—we have `Z·1 = Y`. This explains $\texttt{Y}, \texttt{Z} \in \{\}\,\sigma$. In addition, `W` is always constrained, which explains $\texttt{W} \in \{\}\,\sigma$. Since `R` is none of `X`, `Y`, `Z`, `W` (i.e., `R` is a local variable of the callee), any output at `R` cannot be observed from the caller. On the other hand, output at `Z` can be observed through `B` or `C`. □

The objective of having different forms of variable translation between input and output is to make precise abstraction of the operational semantics where a clause head works as a template and does not produce any constraints. Roughly speaking, output variable translation is smaller than input variable translation. In fact, we have $W\overline{\{t \mapsto s\}^{-1}} \subseteq W\,\{s \mapsto t\}$. Using $\{s \mapsto t\}$ instead of $\overline{\{t \mapsto s\}^{-1}}$ in the analysis would result in non-variable terms occurring in a clause head considered as possibly producing some output, which could lower the precision of the analysis.

4.6.2 Iterative Formulas

The iterative formulas for computing fixed points are shown in Figure 4.1. We assume that *end* is treated as a special variable that does not occur elsewhere.

The set $a_k(G, V)$ is an abstraction of the reduction

$$\langle G, c\rangle \longrightarrow^* \langle G', c'\rangle,\ \mathit{bound}(c) \subseteq V \text{ and } c \neq c' \neq \mathit{false}.$$

Precisely speaking, $a_k(G, V)$ stands for the set of variables through which observable output may be told in the execution where the depth of predicate calls is limited to k. The set $a_{k+1}(p(t), V)$ is nonempty if some process performs observable output starting from the abstract store $V\sigma$, thereby it abstracts the execution of a predicate call. The computation of parallel composition is abstracted by local fixed-point computation that does not keep track of the history of internal computation and iteratively feedbacks the possible instantiation information to the components.

Since $a_k(G, V) \subseteq a_{k+1}(G, V)$, we will eventually obtain the least solution of the simultaneous equations on $a(G, V)$'s obtained from Figure 4.1 by dropping subscripts k and $k+1$ in the formulas. One can prove that the solution $a(p(s), V)$ satisfies that

$$\mathit{says}'(p(s), V) \neq \{\} \ \Rightarrow\ a(p(s), V) \neq \{\}\,.$$

4.7 Example of Fixed-Point Computation

In this section, we apply the fixed-point computation explained in Section 4.6 to an example program and perform program transformation of the motion of synchronization points.

Consider the following program.

```
c([],A)             :- true | true.
c([tell(Ans)|S],A) :- true | Ans:=A,  c(S,A).
c([add(N)|S],A)    :- true | A1:=A+N, c(S,A1).
```

We shall write the i-th clause of the above program as $(G_i$ `:-` true `|` $B_i)$. We assume that `c` $\in \mathit{Pred}$ and that `c` expects as its argument certain terms of a special functor representing pairs, encoding the virtual arity 2 of `c`.

The computation of $a_k(B_i, V)$ is summarized in Figure 4.1.

As we can see, we can reach the least fixed point with $k = 3$ and then we obtain the subsets of $\mathit{req}'(G_i)$ as follows:

$\mathit{req}'(G_1) \supseteq \bigcap_{(a(B_1,V)\neq\{\})} V = \{\}$
$\mathit{req}'(G_2) \supseteq \bigcap_{(a(B_2,V)\neq\{\})} V = \{\mathtt{A}\}$
$\mathit{req}'(G_3) \supseteq \bigcap_{(a(B_3,V)\setminus\{\mathtt{A1}\}\neq\{\})} V = \{\mathtt{N}, \mathtt{S}, \mathtt{A}\}$

This says that we can transform the program as follows.

Table 4.1: Fixed-point computation of $a(B_i, V)$.

(a) $a_1(B_i, V)$

B	V	$a_1(B, V)$
B_1	$\{\}, \{\texttt{A}\}$	$\{end\}$
B_2	$\{\}, \{\texttt{Ans}\}, \{\texttt{S}\}, \{\texttt{S}, \texttt{Ans}\}$	$\{\}$
	$\{\texttt{A}\}, \{\texttt{A}, \texttt{Ans}\}$	$\{\texttt{Ans}\}$
	$\{\texttt{S}, \texttt{A}\}, \{\texttt{S}, \texttt{Ans}, \texttt{A}\}$	$\{\texttt{Ans}, end\}$
B_3	$\{\}, \{\texttt{N}\}, \{\texttt{A}\}$	$\{\}$
	$\{\texttt{N}, \texttt{A}\}$	$\{\texttt{A1}\}$
	$\{\texttt{S}\}, \{\texttt{N}, \texttt{S}\}, \{\texttt{S}, \texttt{A}\}$	$\{\}$
	$\{\texttt{N}, \texttt{S}, \texttt{A}\}$	$\{\texttt{A1}, end\}$

(b) $a_2(B_i, V)$

B	V	$a_2(B, V)$
B_1	$\{\}, \{\texttt{A}\}$	$\{end\}$
B_2	$\{\}, \{\texttt{Ans}\}, \{\texttt{S}\}, \{\texttt{S}, \texttt{Ans}\}$	$\{\}$
	$\{\texttt{A}\}, \{\texttt{A}, \texttt{Ans}\}$	$\{\texttt{Ans}\}$
	$\{\texttt{S}, \texttt{A}\}, \{\texttt{S}, \texttt{Ans}, \texttt{A}\}$	$\{\texttt{Ans}, \texttt{S}, end\}$
B_3	$\{\}, \{\texttt{N}\}, \{\texttt{A}\}$	$\{\}$
	$\{\texttt{N}, \texttt{A}\}$	$\{\texttt{A1}\}$
	$\{\texttt{S}\}, \{\texttt{N}, \texttt{S}\}, \{\texttt{S}, \texttt{A}\}$	$\{\}$
	$\{\texttt{N}, \texttt{S}, \texttt{A}\}$	$\{\texttt{S}, \texttt{A1}, end\}$

(c) $a_3(B_i, V)$

B	V	$a_3(B, V)$
B_1	$\{\}, \{\texttt{A}\}$	$\{end\}$
B_2	$\{\}, \{\texttt{Ans}\}, \{\texttt{S}\}, \{\texttt{S}, \texttt{Ans}\}$	$\{\}$
	$\{\texttt{A}\}, \{\texttt{A}, \texttt{Ans}\}$	$\{\texttt{Ans}\}$
	$\{\texttt{S}, \texttt{A}\}, \{\texttt{S}, \texttt{Ans}, \texttt{A}\}$	$\{\texttt{Ans}, \texttt{S}, end\}$
B_3	$\{\}, \{\texttt{N}\}, \{\texttt{A}\}$	$\{\}$
	$\{\texttt{N}, \texttt{A}\}$	$\{\texttt{A1}\}$
	$\{\texttt{S}\}, \{\texttt{N}, \texttt{S}\}, \{\texttt{S}, \texttt{A}\}$	$\{\}$
	$\{\texttt{N}, \texttt{S}, \texttt{A}\}$	$\{\texttt{S}, \texttt{A1}, end\}$

```
c([],A)             :- true                      | true.
c([tell(Ans)|S],A) :- wait(A)                    | Ans:=A, c(S,A).
c([add(N)|S],A)    :- wait(N),wait(S),wait(A)| A1:=A+N,c(S,A1).
```

Notice that four `wait` constraints are added to the guard, which suspends until the top-level of its argument is determined.

4.8 Conclusion

We have given a method for computing a set of variables for which we can move synchronization points by program transformation. The construction of the abstract domain suitable for the analysis of this program transformation was presented. We have also shown the theoretical account for the safety of the program transformation. The safety guaranteed here is that the semantics of processes is preserved if the processes neither diverge nor fail, and otherwise the processes are reformed in the sense that while divergence and failure may be removed, other normal computation is preserved. The denotational semantics used for formalizing the safety is based on interaction sequences which in turn are based on the operational semantics.

4.8.1 Related Work

The denotational semantics of concurrent logic programs that is based on the set of traces is originated from the paper [33] on program transformation of GHC (Guarded Horn Clauses). GHC is a concurrent logic programming language without atomic tells and is essentially the same as the concurrent logic programming language defined in Chapter 2.

Related work on the motion of synchronization points in concurrent logic or concurrent constraint programs includes Demand Transformation Analysis [16]. It defines an abstract domain that can represent recursive data structures and introduces a fixed-point method that computes for each demand for some specific output the required input needed for obtaining that output. In their work, however, only functional behaviors of processes are considered, that is, only the relationship between input constraints and output constraints is formalized. It means that moving synchronization points based on solely their method can introduce deadlocks to a program since their denotational semantics does not take termination into account. Our work, on the other hand, gives a denotational semantics that can handle termination

and divergence and then discusses the safety of the motion of synchronization points. It should also be noted that our work does not require that the program is well moded, whereas their work requires mode information that specifies whether each argument of a predicate is input or output.

4.8.2 Additional Remarks

We have implemented a prototype version of the program transformation tool for KLIC programs. KLIC is a language system based on GHC. We have written this prototype tool itself in KLIC. It turned out, however, that exploiting sequentiality in the program was more important to achieve good optimization effect than simply inserting `wait` constraints to the guard. To address this issue in a way that is based on the program semantics, we will introduce in the next chapter the framework of sequentiality analysis and code derivation directed by the sequentiality analysis. Nonetheless, optimization by program transformation, including the motion of synchronization points, has the advantage of being independent of any particular implementation of the language system and is suitable for theoretical study.

Chapter 5

Sequentiality Analysis for Concurrent Logic Programs

5.1 Introduction

5.1.1 Background

Fine-grained concurrent languages such as concurrent logic programming languages allow us to describe parallel computation in a natural fashion. In general, a runtime system for these languages tends to suffer from the overhead of context switching and dynamic data manipulation. However, there are cases where these overheads can be removed. Some fragments of a program can be executed more efficiently by compiling them into sequential code [14]. Moreover, compilation into sequential code enables us to perform various low-level optimization techniques for reducing runtime overheads substantially.

As an example, consider the program in Figure 5.1 copied from Figure 2.2. Suppose we can infer that `intlist` and `sum` can be reduced alternately. Then, an optimizing compiler can perform elimination of suspension checks and suspension itself, tag elimination (or unboxing; see [25, 26]), and heap usage reduction—local messages between calls need not always be allocated in the heap.

In principle, such optimization requires dataflow analysis that decides whether an agent can be executed without suspension under a given input constraint. Previous work on such optimization includes dependence analysis between goals [23], suspension analysis based on abstract interpretation [6], and demand transformation analysis [16]. But they all suffer from nonde-

```
stair(N,X0,X)   :- true  | intlist(1,N,S), sum(S,X0,X).
intlist(K0,N,S) :- K0>=N | S=[].
intlist(K0,N,S) :- K0< N | S=[K0|S1], K:=K0+1, intlist(K,N,S1).
sum([],   X0,X) :- true  | X=X0.
sum([E|S],X0,X) :- true  | X1:=X0+E, sum(S,X1,X).
```

Figure 5.1: A concurrent logic program

terminism in the order of reductions and are not able to extract sufficient sequentiality for achieving the optimization effect. For instance, consider the predicate `stair` in our example. Since the goal `stair(0,X0,X)` tells `X0=X`, the predicate `stair` does not depend on its second argument, that is, `stair` can tell something even if the second argument is not instantiated. We know, however, that this tell can often be deferred until the second argument `X0` is instantiated. This is because realistic programs with a goal `stair(N,X0,X)` would not ask the constraint `X0=X` but simply waits for the instantiation of `X`.

One may think that assuming the mode of usage of a predicate may simplify things. Note, however, that well-modedness itself does not ensure that the second argument is instantiated whenever `stair` is called. This means that `X0=X` can still be told with `X0` uninstantiated.

Compiled code of `stair` not specialized for the case where the second argument has received an integer value will incur the suspension overhead and may well lose opportunities for tag elimination.

5.1.2 Our Framework

Our challenge is to formalize the sequentialization of concurrent programs, and to apply the formalism to the construction of an optimizing compiler whose correctness can be formally justified in terms of program semantics. In this dissertation, we propose a framework for constructing an optimizing compiler for concurrent languages that

1. extracts sequentiality by sequentiality analysis,
2. generates sequential intermediate code that is specialized for the extracted sequential usage, and

3. performs various code optimization including tag elimination and update-in-place.

Our sequentiality analysis is directed by the inference of *interfaces* of agents. Roughly speaking, each interface of an agent tells us, given a class of input constraints, what class of constraints the agent can tell without suspension and as what agent the residual agent behaves.

In this dissertation, we will explain sequentiality analysis and intermediate code generation. The formalization and implementation of particular optimization techniques toward the intermediate code is an important issue but not the scope of this dissertation.

We have chosen concurrent logic languages to explain our framework, for it features constraint-based communication, which enables us to formalize our analysis in a concise way. However, a part of our framework can be applied to the optimizing compilation of other fine-grained concurrent languages, such as the pi-calculus, to specialize programs by sequentialization.

The rest of this chapter is organized as follows. Section 5.2 formalizes an interface in terms of the operational semantics. Sections 5.3 and 5.4 explain our framework of optimizing compilation that generates intermediate code directed by the bottom-up analysis using interfaces. Section 5.5 mentions the optimization of the generated intermediate code. Section 5.6 shows some related work, and Section 5.7 concludes.

5.2 Interfaces

Given the operational semantics, we can express concisely the behavior of an agent under a given class of input. In this section, we introduce the notion called *interfaces* of an agent to formalize such properties. Using interfaces, we can systematically analyze a specialized way of the execution of agents, which in turn guarantees the correctness of the generated intermediate code.

As before, the formalization is done taking concurrent constraint programming language as the target language. In this chapter, we prefer reducing the number of local variables occurring in an agent, by replacing local variables with pathed variables of the form $X' \cdot i$ introduced in Section 2.2.2, so that we can analyze constraints more precisely. For example, we shall translate the program in Figure 5.1 into Figure 5.2.

In this chapter, we shall use the following abbreviations. Firstly, for every predicate symbol p, we abbreviate $Prog(p)$ to p. Next, for a variable G

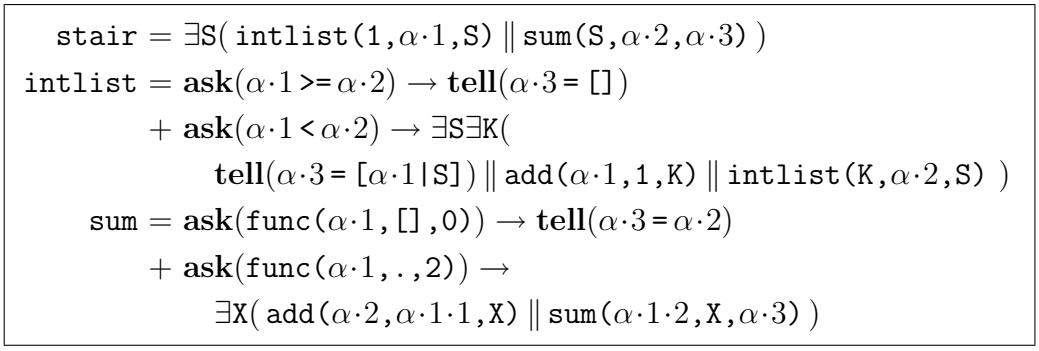


Figure 5.2: Program translated from Figure 5.1

not occurring in pathed terms $t_1, \ldots, t_m$, we write $\exists G\langle p(G), G = (t_1, \ldots, t_n)\rangle$, namely $\mathcal{T}[\![p(t_1, \ldots, t_n)]\!]$, as $p(t_1, \ldots, t_n)$. We call this *term abbreviation*.

We need some preliminaries before giving the formalization of interfaces.

5.2.1 Result of Local Choices

A compiler and a sequential runtime system of concurrent programs are allowed to remove local choices statically. In concurrent logic programs, local choices exist as nondeterministic choice of clauses and do not exist within the syntax of a goal. On the other hand, in concurrent constraint programs, local choices do exist within the syntax of an agent, specifically at **ask** subagents, as nondeterministic branches. Hence, we can formalize static removal of local choices as a binary relation on agents.

We will define a reflexive and transitive relation on agents to formalize safe transformation of agents, including static removal of local choices as well as semantics-preserving transformation.

Definition 5.1 (Local Choices) Given a program *Prog*, we say that the agent A' is a *result of local choices* of A if and only if $\mathcal{O}[\![\langle C[A], \mathit{true}\rangle]\!]_{\mathit{Prog}} \supseteq \mathcal{O}[\![\langle C[A'], \mathit{true}\rangle]\!]_{\mathit{Prog}}$ holds for any context $C[\cdot]$. In such a case, we will write $A \succeq A'$. □

The relationship $A \succeq A'$ states that the agent A can be transformed into the agent A'. Note that this relationship formulates not only removal of local choices but also every semantics-preserving transformation. Moreover,

the relationship $\mathbf{ask}(c) \to A \succeq \mathbf{ask}(c) \to A'$ means that A can be transformed into A' under a store that entails c, since this relationship entails that $\mathcal{O}[\![\langle C[A], c\rangle]\!]_{Prog} \supseteq \mathcal{O}[\![\langle C[A'], c\rangle]\!]_{Prog}$ for any context $C[\cdot]$.

5.2.2 Upward-Closed Sets of Tellable Constraints

Next, we define the class of constraint sets used as input and output in the formalization of interfaces; they are upward-closed sets of tellable constraints. Recall that Con_0 denotes the set of tellable constraints defined in Section 2.2.3.

First, we define $\uparrow c \stackrel{\text{def}}{=} \{d \in Con_0 \mid d \geq c\}$. We say a set S of tellable constraints is *upward-closed* if $S = \bigcup_{c \in S} \uparrow c$ holds.

Every constraint in $\uparrow c$ entails c and is a possible state of the constraint store defined by the operational semantics. Moreover, upward-closed sets of tellable constraints are closed under set union and set intersection. The intersection corresponds to conjunction in the following sense: $\uparrow c \cap \uparrow d = \uparrow (c \wedge d)$. Hence, we will use an upward-closed set of tellable constraints as an abstract store in our interface analysis. We typically use $\vec{c}, \vec{d}, \ldots$ to denote upward-closed sets of tellable constraints; note that $\vec{\ }$ is not an operator. Intuitively, $\vec{c} \supseteq \vec{d}$ can be read as $\vec{d}$ implies $\vec{c}$.

Next, let us introduce a *type constructor*, an operator that maps a variable to an upward-closed set of tellable constraints. For instance, assuming that `int(`X`)` is a constraint representing X is bound to an integer, we define the type constructor i by $i(X) \stackrel{\text{def}}{=} \uparrow$`int(`$X$`)`. Likewise, we define $n(X) \stackrel{\text{def}}{=} \uparrow$`func(`$X$`,[],0)` for *nil*, and $c(X) \stackrel{\text{def}}{=} \uparrow$`func(`$X$`,.,2)` for *cons*.

We have `int(X)` in Con but not in Con_0. This is a subtle point in the formalization and can be ignored in practice. Having two sets simplifies the treatment of guards.

Next, we extend the hiding operator $\exists_X$ to an upward-closed set of tellable constraints in a natural way:

$$\exists_X(\textstyle\bigcup_{i \in I} \uparrow c_i) \stackrel{\text{def}}{=} \bigcup_{i \in I} \uparrow \exists_X c_i.$$

For example, we have $\exists_{\mathtt{X}}(i(\mathtt{X}{\cdot}1) \cap c(\alpha{\cdot}2)) = c(\alpha{\cdot}2)$.

Finally, for variables X and Y, we define the operator $\{X \mapsto Y\}$ on upward-closed sets of tellable constraints as follows:

$$(\textstyle\bigcup_{i \in I} \uparrow c_i)\,\{X \mapsto Y\} \stackrel{\text{def}}{=} \bigcup_{i \in I} \uparrow ((\overline{\exists}_X c_i)[Y/X])$$

$$
\begin{array}{lll}
\textit{Interface} & E ::= & \vec{c} \to \exists X_1 \ldots \exists X_k \sum_{j=1}^{m} (\vec{c_j} \gg E_j) \\
& | & A \\
& | & E \wedge E \\
& | & E \parallel E
\end{array}
$$

where $k \geq 0$, $m \geq 1$, X is a variable, A an agent, and $\vec{c}$ is an upward-closed set of tellable constraints.

Figure 5.3: Syntax of interfaces

where $\overline{\exists}_X(c) = \exists_{X_1} \ldots \exists_{X_n}(c)$ and $\{X_1, \ldots, X_n\}$ is the set of variables syntactically occurring in c and different from X. This operator can be used to represent the variable renaming on a predicate call by letting X be the actual parameter and Y the formal parameter α. For example, we have $(i(\mathtt{X} \cdot 1) \cap c(\alpha \cdot 2) \cap i(\mathtt{Y} \cdot 3))\,\{\mathtt{X} \mapsto \alpha\} = i(\alpha \cdot 1)$, stating that the first argument of a call is bound to an integer.

5.2.3 Formalization of Interfaces

Having defined preliminaries, we now formalize the notion of interfaces. The formalization consists of two definitions: the syntax of interface, and the relation $\geq$ on interfaces.

The syntactic class of *interfaces* is defined in Figure 5.3. The following is the intended meaning of an interface F when we think of the relationship $B \geq F$ for a given agent B. The interface $\vec{c} \to \vec{d} \gg E$ represents the property that, for any input store $c \in \vec{c}$, (a) the agent can make the store evolve into some $d \in \vec{d}$ without suspension and then behave as E or (b) the agent can diverge. More generally, $\vec{c} \to \exists X_1 \ldots \exists X_k \sum_{j=1}^{m} (\vec{d_j} \gg E_j)$ represents the property that for any input store $c \in \vec{c}$ there exists some j such that after declaring new variables $X_1, \ldots, X_k$ the agent can make the store evolve into some $d \in \vec{d_j}$ without suspension and then behave as E_j unless the agent diverges. Here, the choice of j is performed nondeterministically by the agent itself and cannot be controlled from outside. For any agent A, the interface A, which we call an *agent interface*, represents the property that the agent can behave as A. $E_1 \wedge E_2$ represents the property that the agent can behave as E_1 and also as E_2. In $E_1 \wedge E_2$, the choice between E_1 and E_2 can be controlled from outside. $E_1 \parallel E_2$ represents the property that the agent can

$\geq \subseteq \mathit{Agents} \times \mathit{Interface}$ ::

(I1) $$\frac{B \in \mathit{Agents} \quad A \succeq B}{A \geq B}$$

(I2) $$\frac{A \geq E_1 \quad A \geq E_2}{A \geq E_1 \wedge E_2}$$

(I3) $$\frac{B_1 \in \mathit{Agents} \quad B_2 \in \mathit{Agents} \quad A \succeq (B_1 \parallel B_2) \quad B_1 \geq E_1 \quad B_2 \geq E_2}{A \geq E_1 \parallel E_2}$$

(I4) $$\frac{\begin{array}{l}\forall c \in \vec{c}\,(\ \langle A, c\rangle \longrightarrow^{\infty} \text{ or} \\ \qquad \exists j \in \{1, \ldots, m\}\ \exists d \in \vec{d_j}\ \exists B \geq E_j \\ \qquad\quad (\mathbf{ask}(c) \rightarrow A \ \succeq\ \mathbf{ask}(c) \rightarrow \exists X_1 \ldots \exists X_k(\mathbf{tell}(d) \parallel B))\ )\end{array}}{A \geq \vec{c} \rightarrow \exists X_1 \ldots \exists X_k \textstyle\sum_{j=1}^{m}(\vec{d_j} \gg E_j)}$$

$\geq \subseteq (\mathit{Interface} \setminus \mathit{Agents}) \times \mathit{Interface}$ ::

(I5) $$\frac{\forall A \in \mathit{Agents}\,(A \geq E \ \text{ implies } \ A \geq E')}{E \geq E'}$$

Figure 5.4: Definition of the relation $\geq$ on interfaces

behave as the parallel composition of the two agents each of which behaves as E_1 and E_2, respectively.

After defining the syntax of interfaces, the relation $\geq$ on interfaces is defined inductively as in Figure 5.4. The relation $\geq$ is firstly defined as a relation between agents and interfaces, by (I1) through (I4), in terms of the operational semantics. Then, the definition (I5) makes $\geq$ a relation on interfaces that satisfies the reflexive and transitive laws. The rule (I5) embeds agents into interfaces.

5.3 Interface Analysis

In this section, we describe our bottom-up method to analyze sequentiality based on the call graph of predicates. The analysis is formulated as the inference of interfaces.

In the analysis, we must analyze interfaces not of a goal but of the

predicate itself. Hence, as we have said, we abbreviate $Prog(p)$ to p and analyze this agent. We assume that every pathed variable $X \cdot i$ that syntactically occurs in an agent $Prog(p)$ is guarded by some $\mathbf{ask}(c)$ such that $\mathtt{func}(X, f, n) \leq c$ with some f and $n \geq i$ if X is different from α.[1] It is also assumed that any call to p occurs in a term-abbreviated form having the arity determined by p.

5.3.1 Linear Interfaces

In our interface analysis, we are interested in constructing the following two forms of interfaces.

We call an interface of the form:

$$\bigwedge_{i \in I}(\vec{c_i} \rightarrow \exists G \sum_{j \in J(i)}(\vec{d_{i,j}} \gg q_{i,j}(G))),$$

where $I \neq \{\}$, a *linear interface*. The union $\bigcup_{i \in I} \vec{c_i}$ is called the *input assumption* of this linear interface. Intuitively, a linear interface represents the possibility that the agent can be reduced to a single call $q_{i,j}(G)$ when its input assumption is given. Here, we introduce a trick. We assume $Prog(\mathtt{halt}) = \mathbf{stop}$ in order to express the termination as a call. `halt` takes one argument but simply discards it. We will abbreviate $\exists G(\vec{d} \gg \mathtt{halt}(G))$ to $\vec{d}$ if $\vec{d} = \exists_G \vec{d}$.

As a special case of linear interfaces, we call an interface of the form:

$$\vec{c} \rightarrow \vec{d}$$

a *sequential interface*. These two forms of interfaces, linear interfaces and sequential interfaces, are used to formulate our bottom-up interface analysis.

Most of built-in predicates have their own sequential interfaces. For instance, `add` that performs integer addition enjoys $\mathtt{add} \geq i(\alpha \cdot 1) \cap i(\alpha \cdot 2) \rightarrow i(\alpha \cdot 3)$.

The $q_{i,j}(G)$ is called a *tail call*. Although the interface analysis is directed by interfaces, we will also need to have access to the original, term-abbreviated form of each tail call in order to perform the process interleaving analysis. Our process interleaving analysis that makes use of interfaces will be explained in Section 5.3.5.

[1] The condition X is other than α is not crucial; it only makes example programs shorter. This is thanks to the assumption that every call has an appropriate arity.

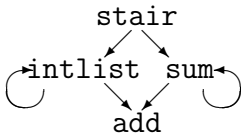


Figure 5.5: A call graph

5.3.2 Bottom-up Analysis of Predicates

The bottom-up interface analysis is performed in the following steps:

1. Build a call graph, namely, a directed graph whose nodes are predicates and whose arcs represent the caller-callee relationship between predicates (draw an arc from caller to callee).

2. Choose a node without outgoing arcs, remove it from the graph and try to find a linear interface of the corresponding predicate. This step is repeated until every node has outgoing arcs.

3. Choose a strongly connected component, which represents (mutually) recursive predicates, remove it from the graph and try to find their linear interfaces. Go back to step 2 if any node remains.

Each linear interface analysis of a predicate can be performed using known linear interfaces of the predicates it calls. The details will be explained soon.

When the interfaces of the predicates it calls cannot be used in the bottom-up analysis—this can happen when the analyzer is not powerful enough or the input program contains a deadlock—we say the analysis fails for this predicate and abandon the analysis for this predicate. This will result in the failure of the analysis of a predicate that may eventually call this predicate. For these predicates, we will generate 'general' code that may spawn many goals.

Example. The call graph of the program in Figure 5.2 is shown in Figure 5.5. We first remove the node `add` without any outgoing arcs. Since `add` is a built-in predicate, its interface is known prior to the analysis. Then, we try to find interfaces of `intlist` and of `sum`, and then of `stair`.

5.3.3 Bottom-up Analysis of Agents

We describe how to compute a linear interface of an agent. This can be performed systematically in a bottom-up manner using structural recursion. The analysis directs intermediate code generation which is explained in Section 5.4.

Inferring interfaces relies on abstract interpretation. We use an upward-closed set of tellable constraints as an abstract store that represents (1) *dynamic type information*, including recursive data types, and (2) *alias information*, that is, information on the unification between pathed variables.

Now, we explain how our interface analysis proceeds for agents. Receiving a pair of an agent and an *initial input assumption* $\vec{a}$, the analysis recursively computes a linear interface for each subagent. As we have said, the analysis may fail, which takes place in recursive calls (it becomes a tail call candidate) and other cases such as deadlocks. For parallel composition agents, static scheduling for the agents is also performed. The analysis is performed as follows:

$p(X)$: If a linear interface

$$\bigwedge_{i\in I}(\vec{c_i} \to \exists G \sum_{j\in J(i)}(\vec{d_{i,j}} \gg q_{i,j}(G)))$$

has been computed for p where G is other than X, return

$$\bigwedge_{i\in I}(\vec{c_i}\,\{\alpha \mapsto X\} \to \exists G \sum_{j\in J(i)}(\vec{d_{i,j}}\,\{\alpha \mapsto X\} \gg q_{i,j}(G))).$$

Otherwise, the analysis fails. We can safely rename G in order to resolve a name collision with X. Note that p can be a built-in predicate such as `add`.

tell(c)**:** We may return $\uparrow \mathit{true} \to \uparrow c$. However, we can be more specific. The agent **tell**(c) can be decomposed into a sequence of agents of the forms **tell**(`func(`X`,`f`,`n`)`) and **tell**$(X = Y)$. For the former agent, return $\uparrow \mathit{true} \to \uparrow$`func(`$X$`,`$f$`,`$n$`)`. For the latter agent, we have five interfaces to be returned depending on the instantiatedness and groundness information in $\vec{a}$:

$$\begin{cases} \tau(X) \to \tau(Y) & \text{if } X \text{ is ground,} \\ \tau(Y) \to \tau(X) & \text{if } Y \text{ is ground,} \\ \tau(X) \to \tau(Y) \cap \uparrow(X = Y) & \text{if } X \text{ is instantiated but not ground,} \\ \tau(Y) \to \tau(X) \cap \uparrow(X = Y) & \text{if } Y \text{ is instantiated but not ground,} \\ \uparrow \mathit{true} \to \uparrow(X = Y) & \text{otherwise} \end{cases}$$

where τ is a type constructor that describes the dynamic type information on either X or Y, entailed by every constraint in $\vec{a}$.

A pathed term is said to be *ground* if it contains no pathed variables. A pathed variable X is said to be *ground* in a constraint c if there exists a ground term t such that $(X = t) \leq c$.

$\exists X \langle A, c \rangle$**:** If $c \neq \mathit{true}$, retry assuming that the agent is $\exists X \langle \mathbf{tell}(c) \parallel A, \mathit{true} \rangle$. Hence, we assume that $c = \mathit{true}$. We first compute a linear interface $\bigwedge_{i \in I}(\vec{c_i} \rightarrow \exists G \sum_{j \in J(i)}(\vec{d_{i,j}} \gg q_{i,j}(G)))$ of A with the initial input assumption $\exists_X \vec{a}$. Then, if $\exists_X \vec{c_i} = \vec{c_i}$ holds for each i, return $\bigwedge_{i \in I}(\vec{c_i} \rightarrow \exists G \sum_{j \in J(i)}(\exists_X \vec{d_{i,j}} \gg q_{i,j}(G)))$. Otherwise, the analysis fails.

In the simplest case where we have computed a linear interface of A of the form $\vec{c} \rightarrow \exists G(\vec{d} \gg q(G))$ such that $\exists_X \vec{c} = \vec{c}$, the above amounts to returning $\vec{c} \rightarrow \exists G(\exists_X \vec{d} \gg q(G))$.

$\sum_{h \in H} \mathbf{ask}(b_h) \rightarrow A_h$**:** First, we compute a subset H' of the branches H we should consider. If $\uparrow b_h \supseteq \vec{a}$ holds for some $h \in H$, in which case b_h is always entailed, let $H' = \{h\}$ so that the local choice is eliminated. Otherwise, let $H' = \{h \in H \mid \vec{a} \cap \uparrow b_h \neq \uparrow \mathit{false}\}$, the set of all the possible branches consistent with $\vec{a}$. Then, for each $h \in H'$, we compute a linear interface $\bigwedge_{i \in I(h)}(\vec{c}_{h,i} \rightarrow \exists G \sum_{j \in J(h,i)}(\vec{d}_{h,i,j} \gg q_{h,i,j}(G)))$ of A_h with the initial input assumption $\vec{a} \cap \uparrow b_h$. Return their conjunction (use $\bigwedge$). We fail if $H' = \{\}$.

In the simplest case where H is a singleton of $\{h\}$ and $\vec{a} \cap \uparrow b_h \neq \uparrow \mathit{false}$, the above amounts to computing a linear interface of A_h with the initial input assumption $\vec{a} \cap \uparrow b_h$. Observe that the constraint b_h is extracted from the agent and accumulated into what is returned. This resembles the type inference of lambda abstraction in typed functional languages.

$A_1 \parallel \ldots \parallel A_n$**:** We use abstract interpretation based on dynamic type information to schedule subagents (i.e., A_i's). Subagents not scheduled as a tail call will be executed according to their sequential interfaces.

The algorithm for scheduling proceeds in the following steps. It will return a linear interface of the form $\vec{c} \rightarrow \exists G(\vec{d} \gg q(G))$ if the subagents are successfully scheduled. In the description of the algorithm, we use $\vec{c}$ and $\vec{d}$ as assignable variables, both of which are initialized with the initial input assumption $\vec{a}$. The variable $\vec{c}$ remembers the input

assumption to be returned while the variable $\vec{d}$ expresses the current abstract store in the algorithm.

1. For each agent A_i, try to find a linear interface, and then a sequential interface (see Section 5.3.4), using $\vec{d}$ as the initial input assumption.
2. Choose an agent A_i with a sequential interface $\vec{c_i} \rightarrow \vec{d_i}$ whose input assumption $\vec{c_i}$ is entailed by the current abstract store $\vec{d}$, that is, $\vec{c_i} \supseteq \vec{d}$. We should choose an agent with an interface that contains no alias information, if any. Remove that agent from the composition, and then strengthen the current abstract store $\vec{d}$ by intersecting $\vec{d}$ with $\vec{d_i}$. This step is repeated as many times as applicable. If all the agents are removed, return $\vec{c} \rightarrow \vec{d}$.
3. If no agent can be removed under the current abstract store, compute again linear interfaces for the remaining agents using the current abstract store as the initial input assumption.
4. If we have more than one agent with linear interfaces, we try to interleave them by the method we will explain in Section 5.3.5. If successful, we have an interleaved agent with a linear interface.
5. If we have exactly one agent with a linear interface, try to find a sequential interface $\vec{c_i} \rightarrow \vec{d_i}$ of it. If successful, remove that agent from the composition. Let $\vec{e}$ be a 'difference' of the input assumption $\vec{c_i}$ from the current abstract store $\vec{d}$, which means that $\vec{e}$ is an arbitrary upward-closed set of tellable constraints that satisfies $\vec{c_i} \supseteq \vec{d} \cap \vec{e}$. We should choose $\vec{e}$ that has as little information as possible. How to compute $\vec{e}$ is delegated to the implementation of the algorithm. Lift up this difference $\vec{e}$ to the input assumption of the interface to be returned, namely $\vec{c}$, which is done by intersecting $\vec{c}$ with $\vec{e}$. After that, update the current abstract store $\vec{d}$, which is done by intersecting $\vec{d}$ with $\vec{d_i} \cap \vec{e}$.
6. If exactly one term-abbreviated call remains, say $q(t)$, we will choose it as the tail call. Let G be a variable satisfying $\exists_G \vec{d} = \vec{d}$ and not occurring in t. Compute an interface $\vec{c_0} \rightarrow \vec{d_0}$ of $\mathbf{tell}(G = t)$ with the initial input assumption $\vec{d}$. Since we have $q(t) \geq \vec{c_0} \rightarrow \exists G(\vec{d_0} \gg q(G))$, return $\vec{c} \cap \vec{c_0} \rightarrow \exists G(\vec{d} \cap \vec{d_0} \gg q(G))$.
7. The analysis fails when this step is reached.

Most agents without recursion or with a single tail-recursive call can be sequentialized with this algorithm. To deal with other forms of recursive predicates, other techniques such as conversion to tail-recursive forms [2] should be used in conjunction with our method.

Subagents may return alias information in output. By applying it to the remaining subagents, some pathed variables may be represented by means of α, thus possibly eliminating the number of variables used.

It should be straightforward to prove by induction that every agent has the interface that is returned by the method.

Example. Let L be the agent

$$\mathbf{tell}(\alpha{\cdot}3 = [\alpha{\cdot}1 \mid \mathtt{S}]) \parallel \mathtt{add}(\alpha{\cdot}1, 1, \mathtt{K}) \parallel \mathtt{intlist}(\mathtt{K}, \alpha{\cdot}2, \mathtt{S})$$

found in *Prog*(`intlist`). We show the interface analysis for the agent L with the initial input assumption $i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2)$, which comes from $\alpha{\cdot}1 < \alpha{\cdot}2$.

Step 2 requires us to find interfaces of each subagents. Let us first find an interface of $\mathtt{add}(\alpha{\cdot}1, 1, \mathtt{K})$, namely $\exists \mathtt{G} \langle \mathtt{add}(\mathtt{G}), \mathtt{G} = (\alpha{\cdot}1, 1, \mathtt{K}) \rangle$. To do this, we should find an interface of $\exists \mathtt{G}(\mathbf{tell}(\mathtt{G} = (\alpha{\cdot}1, 1, \mathtt{K})) \parallel \mathtt{add}(\mathtt{G}))$. Since we know $\mathtt{add} \geq i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \rightarrow i(\alpha{\cdot}3)$, we have $\mathtt{add}(\mathtt{G}) \geq i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \rightarrow i(\mathtt{G}{\cdot}3)$. Since we have $\mathbf{tell}(\mathtt{G} = (\alpha{\cdot}1, 1, \mathtt{K})) \geq i(\alpha{\cdot}1) \rightarrow i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \cap \uparrow(\mathtt{G}{\cdot}3 = \mathtt{K})$, we have $(\mathbf{tell}(\mathtt{G} = (\alpha{\cdot}1, 1, \mathtt{K})) \parallel \mathtt{add}(\mathtt{G})) \geq i(\alpha{\cdot}1) \rightarrow i(\mathtt{K}) \cap i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \cap \uparrow(\mathtt{G}{\cdot}3 = \mathtt{K})$, thus $\mathtt{add}(\alpha{\cdot}1, 1, \mathtt{K}) \geq i(\alpha{\cdot}1) \rightarrow i(\mathtt{K})$. Seeing that this interface has no alias information and that its input assumption $i(\alpha{\cdot}1)$ is subsumed by the current abstract store $i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2)$, we decide to schedule $\mathtt{add}(\alpha{\cdot}1, 1, \mathtt{K})$ first. By this scheduling, the abstract store is updated to $i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \cap i(\mathtt{K})$.

We will then schedule $\mathbf{tell}(\alpha{\cdot}3 = [\alpha{\cdot}1 \mid \mathtt{S}])$, for which we obtain $i(\alpha{\cdot}1) \rightarrow c(\alpha{\cdot}3) \cap i(\alpha{\cdot}3{\cdot}1) \cap \uparrow(\mathtt{S} = \alpha{\cdot}3{\cdot}2)$. By this scheduling, the abstract store is updated to $i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \cap c(\alpha{\cdot}3) \cap i(\alpha{\cdot}3{\cdot}1) \cap i(\mathtt{K}) \cap \uparrow(\mathtt{S} = \alpha{\cdot}3{\cdot}2)$. The alias information $\uparrow(\mathtt{S} = \alpha{\cdot}3{\cdot}2)$ is applied to the remaining call $\mathtt{intlist}(\mathtt{K}, \alpha{\cdot}2, \mathtt{S})$, replacing it with $\mathtt{intlist}(\mathtt{K}, \alpha{\cdot}2, \alpha{\cdot}3{\cdot}2)$ so that the number of variables other than α occurring in the call will be reduced.

Now, it is easy to infer the relation $L \geq i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \rightarrow \exists \mathtt{G}(\ c(\alpha{\cdot}3) \cap i(\alpha{\cdot}3{\cdot}1) \cap i(\mathtt{K}) \cap i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \cap \uparrow(\mathtt{S} = \alpha{\cdot}3{\cdot}2) \cap \uparrow(\mathtt{G}{\cdot}3 = \alpha{\cdot}3{\cdot}2) \gg \mathtt{intlist}(\mathtt{G})\ )$.

Likewise, we can derive that $\exists \mathtt{S} \exists \mathtt{K}(L) \geq i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \rightarrow \exists \mathtt{G}(\ c(\alpha{\cdot}3) \cap i(\alpha{\cdot}3{\cdot}1) \cap i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \cap \uparrow(\mathtt{G}{\cdot}3 = \alpha{\cdot}3{\cdot}2) \gg \mathtt{intlist}(\mathtt{G})\ )$.

Finally, we have

$$\begin{aligned}\texttt{intlist} \;\geq\; & i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \rightarrow \exists \texttt{G}(\; n(\alpha{\cdot}3) \\ & + (\; c(\alpha{\cdot}3) \cap i(\alpha{\cdot}3{\cdot}1) \cap i(\texttt{G}{\cdot}1) \cap i(\texttt{G}{\cdot}2) \cap \uparrow(\texttt{G}{\cdot}3 = \alpha{\cdot}3{\cdot}2) \\ & \quad \gg \texttt{intlist}(\texttt{G})\;)).\end{aligned}$$

Similarly, we have

$$\begin{aligned}\texttt{sum} \;\geq\; & n(\alpha{\cdot}1) \rightarrow \uparrow(\alpha{\cdot}3 = \alpha{\cdot}2) \\ \wedge\; & c(\alpha{\cdot}1) \cap i(\alpha{\cdot}1{\cdot}1) \cap i(\alpha{\cdot}2) \rightarrow \exists \texttt{G}(\\ & \quad \uparrow(\texttt{G}{\cdot}1 = \alpha{\cdot}1{\cdot}2) \cap i(\texttt{G}{\cdot}2) \cap \uparrow(\texttt{G}{\cdot}3 = \alpha{\cdot}3) \gg \texttt{sum}(\texttt{G})\;).\end{aligned}$$

□

5.3.4 Inferring Sequential Interfaces

We mention how to infer sequential interfaces from linear interfaces. Essentially, it consists of two steps:

1. find an upward-closed set of input constraints sufficient to execute the agent without suspension—usually the execution terminates but may diverge—and

2. compute an upward-closed set of output constraints the agent can perform on termination.

Each of the above steps needs to compute a fixed point of simultaneous equations translated from linear interfaces of predicates that form a strongly connected component in the call graph. The basic idea of doing this can be found, for example, in the paper [16]. To implement this, we must compute a fixed point directly from recursive equations. Basically, this can be done by computing least fixed points. However, some elaboration will be needed to handle dynamic type information in output constraints.

5.3.5 Process Interleaving

We must also consider a method of interleaving parallel composition, and here is how to utilize the interface information to interleave producer and consumer processes. The method is basically unfold/fold transformation [33]

but is directed by the interfaces of the two agents, which enables us to directly justify the specialization of the parallel composition. Although we will explain only the simplest case, it gives us a good insight into our method.

For the sake of simplicity of the explanation, we assume that the argument of every call is a sequence of ground terms and distinct pathed variables. This assumption can always be made true by introducing temporary variables, though doing so may obscure alias information and lower the precision of the analysis.

1. Let P and Q be agents to be interleaved. We assume that we have computed a linear interface of P (and Q, respectively) that has exactly one tail call to p (and q). We also assume that we have computed a linear interface of the predicate p (and q, respectively) that has exactly one tail call to p (and q) which corresponds to the agent $p(X_1,\ldots,X_m)$ (and $q(Y_1,\ldots,Y_n)$) in the original, term-abbreviated form.

2. For each shared variable S between P and Q, we put an assumption that the paths where S occurs are always aliased. Let $(Y'_1,\ldots,Y'_{n'})$ be the list obtained by removing every X_i from $(Y_1,\ldots,Y_n)$. Every Y_j is either some $Y'_{j'}$ or some X_i. Let θ be the syntactic substitution which maps every $\alpha{\cdot}j$ to $\alpha{\cdot}(m+j')$ if Y_j is $Y'_{j'}$, and every $\alpha{\cdot}j$ to $\alpha{\cdot}i$ if Y_j is X_i. Then, introduce a new predicate q' defined by $\mathit{Prog}(q') = \mathit{Prog}(q)\theta$. We have $q' \geq F\theta$ if $q \geq F$.

3. Introduce a new predicate r defined by $\mathit{Prog}(r) = \mathit{Prog}(p) \parallel \mathit{Prog}(q')$. Then, try to find a linear interface of r from those of p and q', by replacing each pair of tail calls to p and q' by a tail call to r.

Example. Let us interleave the two predicate calls in `intlist(1,`$\alpha{\cdot}1$`,S)` $\parallel$ `sum(S,`$\alpha{\cdot}2$`,`$\alpha{\cdot}3$`)` in `stair`. Assume that we have computed linear interfaces of `intlist` and `stair` as in the previous example.

Since `S` is a shared variable, the third argument of `intlist` and the first argument of `sum` are assumed to be always aliased in the tail calls. Accordingly, we introduce predicates by $\mathit{Prog}(\texttt{sum}') = \mathit{Prog}(\texttt{sum})[\alpha{\cdot}3/\alpha{\cdot}1, \alpha{\cdot}4/\alpha{\cdot}2, \alpha{\cdot}5/\alpha{\cdot}3]$ and $\mathit{Prog}(r) = \mathit{Prog}(\texttt{intlist}) \parallel \mathit{Prog}(\texttt{sum}')$. Finally, we can find a linear interface of r from the linear interfaces of `intlist` and `sum`:

$$
\begin{aligned}
r \geq i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \cap i(\alpha{\cdot}4) \rightarrow \exists \mathtt{G}(\\
\quad (\ n(\alpha{\cdot}3) \cap i(\alpha{\cdot}5) \ ) \\
\quad + (\ c(\alpha{\cdot}3) \cap i(\alpha{\cdot}3{\cdot}1) \cap i(\mathtt{G}{\cdot}1) \cap i(\mathtt{G}{\cdot}2) \cap i(\mathtt{G}{\cdot}4) \\
\quad\quad \cap \uparrow(\mathtt{G}{\cdot}3 = \alpha{\cdot}3{\cdot}2) \cap \uparrow(\mathtt{G}{\cdot}5 = \alpha{\cdot}5) \gg r(\mathtt{G}) \ )).
\end{aligned}
$$

5.4 Code Generation

In this section, we define an intermediate language and explain the code generation in our framework.

5.4.1 Definition of Intermediate Code

Figure 5.6 summarizes our low-level, sequential intermediate code. There, *Atom* and *Variable* correspond to function symbols and variable names, respectively. This subsection defines the intermediate language and can be skipped in the first reading, for it is not the subject of this chapter.

The intermediate language explicitly manipulates the memory. The memory consists of cells. Here, we assume a pointer-tag implementation of this intermediate language, that is, each *initialized* cell is tagged with one of FUNC, FUNCREF, or REF and a cell tagged with FUNC contains atomic data (a functor) while a cell tagged with FUNCREF or REF contains a pointer that points to a structure or a cell, respectively. A cell tagged with REF works either as a reference to another cell, or as an uninstantiated variable if the cell points to itself.

A *path* refers to a cell that implements (i.e., contains the content of) a pathed variable. The syntax of a path is defined in Figure 5.6. A path that is a variable X refers to the cell that implements X. The path X`-`i refers to the cell containing the i-th element of the structure referenced by the path X, which is to be located at offset i from the cell X points to.

The explanations for the other instructions follow.

- "`var` X" acquires a new cell that has a reference to itself with REF tag, and assigns it to the location specified by X.
- "`alloc` X`,`n" acquires $n+1$ contiguous, uninitialized cells and assigns the reference to its first cell with FUNCREF tag to the location specified by X.

- "`copy` X`,`Y" assigns the content of the location specified by Y to the location specified by X.
- "`func` X`,`f`,`n" assigns the atomic value 'f/n' with FUNC tag to the location specified by X.
- "`atom` X`,`f" is shorthand for "`func` X`,`f`,0`".
- "`prim` X`,`t" computes the primitive expression t and assigns the result content to the location specified by X. A typical example of a primitive expression is `iadd(Y,Z)`, which performs integer addition.
- "`test` c `[`A`]` B" performs an entailment check of the constraint c with the current store. A typical example of a constraint is `int(A-1)`, which is the translation of `int(`$\alpha \cdot 1$`)` to this intermediate language. In our implementation scheme, the store is in fact implemented by the cells corresponding to the paths occurring in c. If the entailment is observed, the code B is executed. If the entailment is *not yet* observed, which includes the case where c is inconsistent with the store, the alternative code A is executed.

The instruction modifier `t` means that the consistency of the output by the instruction is recursively checked with the current content of the destination path (`t` stands for 'tell'). The recursive case happens only in `tcopy`, which exactly performs general unification.

Instructions are preceded by a jump label. A label $p\_\mathcal{I}(\vec{a})$ describes the entry point of the predicate p with the input assumption $\vec{a}$. We assume $\mathcal{I}(\vec{a})$ generates an *Info* text that is supposed to describe the dynamic type information on the formal parameter α in $\vec{a}$. For example, the code implementing the built-in predicate `add` can have the input assumption $i(\alpha \cdot 1) \cap i(\alpha \cdot 2)$ and it will be preceded by the label `add_i1i2`. Let us require that the input assumption specified in each label argument of an instruction (for instance in the argument of a `goto` instruction) is guaranteed to be always entailed. Proofs of the entailment can be automatically extracted from the linear interfaces we have computed for the agents.

A label can be in another form. A label that is just a predicate name p is used for the entry point of the general (i.e., unspecialized) code for p. A label of the form $p\_\mathcal{I}(\vec{c})\_\mathcal{I}(\vec{d})$ can be used for explicitly expressing $p \geq \vec{c} \rightarrow \vec{d}$. We

will omit the definition details of *Info* for they are not the subject of this work.

The distinguished variable `A` is used as the actual argument register throughout the code. Filling in `A` followed by a jump (`goto`) to a label is the translation of a tail call to this intermediate language. The scopes of variables other than `A` are confined between two labels.

The path `A` corresponds to the formal parameter variable α. The content of `A` is assumed to be a FUNCREF reference to a *non-shared* structure, that is, a structure referenced only by one cell. Any other structure, including the one referenced by `A-`i, may be shared unless the corresponding path has the type modifier `d` in the label (`d` stands for 'destructive'). Type modifiers are used only for optimization purposes.

The explanations for the other instructions follow.

- "`spawn` l`,`G" enqueues to the *process pool* a new goal `goal(`l`,`G`)` whose entry point is l and whose actual argument is pointed to by G.
- "`halt`" means that the current process has been terminated. Another process should be picked up from the process pool and be executed.
- "`fail`" means that the whole computation fails.
- "`hook` *PathList*`,`l" hooks a goal `goal(`l`,A)` to each path, which must be initialized, in the list *PathList*. Every tell involving some path in *PathList* causes the hooked goal to be spawned to the process pool.
- "`goto` l" jumps to the label l.
- "`call` l`,`G" is a non-suspending subroutine call to the label l. This instruction is prepared for expressing a built-in call.

The following stuff is prepared for optimization purposes.

- "`arg` X`,`Y`,`K" assigns the reference to the K-th argument of the structure pointed to by Y to the location specified by X.
- "`deref` X`,`Y" fully dereferences a chain of REF pointers in the content of the cell Y and then assigns the content of the found cell to the location specified by X.

```
Path   ::= Variable | Path-Integer
Label  ::= Atom[_Info[_Info]]
Info   ::= (Tycon Integer+)+
Tycon  ::= [d|x]( c|e|f|i|n|u|w | (l|v) Tycon
               | z Atom Integer )
Term   ::= Path | Atom(Term_1, ..., Term_n)
Entry  ::= Label: newline Inst
Inst   ::= Inst    newline Inst
         | mtest Term [ Inst_alt ] Inst_ok
         | mprim [*]Path, Term
         | mcopy [*]Path, Path_src
         | mfunc [*]Path, Atom, Integer
         | matom [*]Path, Atom
         | var   [*]Path
         | box   [*]Path
         | unbox [*]Path
         | alloc [*]Path, Integer
         | [x]arg [*]Path, Path_src, Term_ofst
         | deref Path, Path_ref
         | call  Label, Path
         | spawn Label, Path
         | goto  Label
         | fail
         | halt
         | hook  Paths, Label
m      ::= |x|t
```

Figure 5.6: An intermediate language

- The `*` that prefixes a destination path specifies that the destination path is dereferenced by one level. For example, `atom *A-3,[]` assigns a nil to the location referenced by `A-3`, not to the location of `A-3`. Such explicit management of pointer dereferences enables several low-level optimization including uninitialized variable optimization developed in sequential logic languages like Prolog.

- "`unbox` X" and "`box` X" convert the content of X to its unboxed and boxed value, respectively. The general status is being boxed. Once unboxed, any `arg`, `copy`, `prim` or `test` instruction relevant to that path must be prefixed with the instruction modifier `x`, and the type modifier `x` must also be used for that path in a label (`x` stands for 'unboxed').

The formal semantics of the code, which is required to justify code optimization formally, is omitted in this dissertation.

5.4.2 Code Generation Directed by Interface Analysis

In this subsection, we explain how to generate intermediate code directed by interface analysis. We assume that we have computed a linear interface of the agent $Prog(p)$ for a predicate p. Let $\vec{a}$ be the input assumption of the given linear interface. The code generation is defined by structural induction on the syntax of the agent as follows:

$p(X)$**:** If we have found a sequential interface of p whose input assumption is entailed by $\vec{a}$, we must also have the corresponding code B for p. Built-in predicates fall under this case. In this case, we perform inline expansion of B to generate the code for $p(X)$. Return the code B with `A` replaced by X and other variable names replaced by fresh variable names. Although inline expansion may cause code explosion, in most cases code can be significantly compacted by subsequent code specialization.

If no sequential interfaces have been found, we should return `spawn` $p\_\mathcal{I}(\vec{a}\,\{X \mapsto \alpha\}),X$. If the goal is a tail call, it may be immediately rewritten to `copy A,`X followed by `goto` $p\_\mathcal{I}(\vec{a}\,\{X \mapsto \alpha\})$.

Note that p can be a built-in predicate. For example, we have the code for `add` corresponding to $i(\alpha{\cdot}1) \cap i(\alpha{\cdot}2) \rightarrow i(\alpha{\cdot}3)$ as follows:

```
add_i1i2:
  tprim A-3,iadd(A-1,A-2)
  halt
```

tell(c)**:** This case is almost the same as the case of a built-in predicate call explained above. As in the interface analysis, we can decompose **tell**(c) into a list of agents of the forms **tell**(X `=` f`(`t_1`,...,`t_n`)`) and **tell**(X `=` Y).

We first consider the case where X has been initialized.

- For **tell**(X `=` Y) where $\uparrow$`func(`Y`,`f`,0)` $\supseteq \vec{a}$, return `tatom` $\mathcal{P}(X)$`,`f.
- For **tell**(X `=` Y) of the other cases, return `tcopy` $\mathcal{P}(X)$`,`$\mathcal{P}(Y)$.
- For **tell**(X `=` f`(`t_1`,...,`t_n`)`), return the following code:

  ```
  alloc W,n
  func  W-0,f,n
  code for tell(W·1 = t1) assuming W·1 is uninitialized
    ⋮
  code for tell(W·n = tn) assuming W·n is uninitialized
  tcopy P(X),W
  ```

  where W is a fresh variable.

We assume that $\mathcal{P}$ translates pathed variables into paths: $\mathcal{P}(\alpha) =$ `A`, $\mathcal{P}(X) = X$ for a variable X other than α, and $\mathcal{P}(X \cdot i) = \mathcal{P}(X)$`-`$i$.

When X has not been initialized, we modify the above three arrays of code by removing the `t` instruction modifier from the last instruction of each array and by replacing W by $\mathcal{P}(X)$.

Finally, add a `halt` instruction to the tail of the code to be returned.

$\exists X\langle A, c\rangle$**:** For $c \neq$ *true*, return the code for the agent $\exists X\langle$**tell**$(c) \parallel A,$ *true*$\rangle$. For $c =$ *true*, we firstly build the code B for A. Then, return the code `new` X followed by B with all X replaced by a fresh variable name.

$\sum_{h=1}^{m}$**ask**$(c_h) \rightarrow A_h$**:** Choose an h according to the interface. Return a `test` instruction that performs the entailment check of c_h (a portion of which may be statically done using $\vec{a}$) followed by the code for A_h. The code argument of the `test` instruction, which is executed when the check does not pass, is the code for the remaining branches. If the check is found to pass always, we can simply return the code for A_h.

```
intlist_i1i2:
  test ilt(A-1,A-2) [
    tatom A-3,[]
    halt
  ]
  var   S
  var   K
  tprim K,iadd(A-1,1)
  alloc G,2
  func  G-0,.,2
  copy  G-1,A-1
  copy  G-2,S
  tcopy A-3,G
  alloc H,3
  copy  H-1,K
  copy  H-2,A-2
  copy  H-3,S
  copy  A,H
  goto  intlist_i1i2
```

(a) Before Optimization

```
intlist_i1i2:
  test ilt(A-1,A-2) [
    tatom A-3,[]
    halt
  ]
  alloc G,2
  func  G-0,.,2
  copy  G-1,A-1
  var   G-2
  tcopy A-3,G
  prim  A-1,iadd(A-1,1)
  copy  A-3,G-2
  goto  intlist_i1i2
```

(b) After Optimization

Figure 5.7: Generated intermediate code

$A_1 \parallel \ldots \parallel A_n$**:** Return the code for the subagents concatenated in the scheduled order. The `halt` instructions between subagents must be removed.

For efficiency reasons of the code optimization, we should maintain a list of unconstrained paths in the above code generation procedure. Such a list enables us to early remove many `t` instruction modifiers and `var` instructions.

Figure 5.7 (a) shows the generated code for `intlist`.

5.5 Code Optimization

We will briefly mention the optimization on the sequential intermediate code though it is not the subject of this work.

Optimization in our framework consists of two stages: one is to generate sequential intermediate code, and the other is to optimize the generated code. The former is primarily concerned with process scheduling, together with local choice elimination, for suspension avoidance and process fusion, and is

directed by the sequentiality analysis. The latter relates to both the source-level optimization such as copy propagation and the implementation-level optimization including tag elimination and update-in-place. Our framework can straightforwardly justify the safety of these kinds of low-level optimization using the semantics of the intermediate language. We stress that such justification cannot be done systematically through source-level optimization in the original concurrent language.

We should mention that other analysis frameworks that justify particular optimization techniques can be peacefully incorporated into our framework to build a more powerful and efficient compiler. For instance, the linearity analysis [37] can be incorporated so as to statically guarantee the safety of update-in-place optimization. Mode analysis [34] can statically determine which interface should be used for each tell of a unification.

Example. The code shown in Figure 5.7 (a) can be optimized into Figure 5.7 (b). General (i.e., unspecialized) code for `stair` after moderate optimization can be like Figure 5.8 (a), which can be specialized and optimized as in Figure 5.8 (b). We can see that the specialization by sequentialization has significant importance in accelerating the code optimization. Further low-level optimization, such as tag elimination and uninitialized variable optimization, can be applied to the code but is not shown because it is out of the scope of this dissertation.

5.6 Related Work

Van Roy [39] demonstrates an optimizing compilation framework for Prolog that uses low-level intermediate language, rather than WAM (Warren's Abstract Machine), to make code specialization more effective. The optimization techniques explained there can be systematically applied to concurrent logic programs only if sequentiality is extracted from the program. Our sequentiality analysis contributes to the sequentialization phase in an optimizing compilation framework for concurrent logic languages (and other fine-grained concurrent languages).

Related work on optimizing compilers for fine-grained concurrent languages include [12], [13], and [31].

Debray [12] describes a sequentializing compiler for the concurrent constraint language Janus. It shares many concepts with our work including

```
stair:
  alloc G,3
  var   G-1
  copy  G-2,A-2
  copy  G-3,A-3
  spawn sum,G
  copy  A-2,A-1
  copy  A-3,G-1
  atom  A-1,1
  goto  intlist

sum:
  test wait(A-1) [
    hook [A-1],sum
    halt
  ]
  test func(A-1,.,2)[
    test func(A-1,[],0)[
      fail
    ]
    tcopy A-3,A-2
    halt
  ]
  alloc G,3
  copy  G-1,A-1-2
  var   G-2
  copy  G-3,A-3
  spawn sum,G
  copy  A11,A-1-1
  copy  A-1,A-2
  copy  A-2,A11
  copy  A-3,G-2
  goto  add
```

(a) General Code

```
stair_i1i2:
  copy A-4,A-2
  copy A-2,A-1
  copy A-5,A-3
  var  A-3
  atom A-1,1
  goto r_i1i2i4

r_i1i2i4:
  test ilt(A-1,A-2) [
    tatom A-3,[]
    tcopy A-5,A-4
    halt
  ]
  prim A-4,iadd(A-4,A-1)
  var  A-3
  prim A-1,iadd(A-1,1)
  goto r_i1i2i4
```

(b) Specialized Code

Figure 5.8: General code vs. specialized code

the extraction of sequentiality of predicates using instantiatedness analysis. However, since their compiler uses Prolog as its sequential intermediate language, optimization on memory management is completely delegated to an underlying Prolog compiler. It means that the implementation techniques unique to fine-grained concurrent language that require the analysis of process interleaving cannot be achieved.

Debray et al. [13] explain top-down non-suspension analysis and its application to an optimizing compiler for Janus that uses C as its sequential intermediate language. Fixed-point dataflow analysis, however, becomes complicated in the presence of complex message flow, which is the *raison d'être* of concurrent logic programming. We believe that bottom-up analysis can compute fixed points with better precision and modularity, and can smoothly connect the extracted sequentiality to the runtime system.

Overton [31] discusses mode analysis and optimizing compilation for the moded and typed concurrent logic language Mercury. Mercury provides several kinds of declarations useful for optimizing compilation, among which the instantiatedness declaration roughly corresponds to our sequential interfaces. The author thinks that the notion of linear interfaces can enhance their declarations by enabling an appropriate treatment of reactive goals. It should also be noted that our framework itself does not require that the program is moded.

5.7 Conclusion and Future Work

We have presented a framework for extracting sequentiality in concurrent logic programs and for generating corresponding sequential intermediate code. The proposed framework is based on bottom-up analysis using interfaces that formalize non-suspending fragments of agents, and hence can justify the intermediate code it generates.

A major advantage of our framework is that the bottom-up analysis of interfaces ensures the correctness of the intermediate code generated at the same time during the interface analysis.

Although the specialization of an agent by sequentialization using the notion of interfaces could be axiomatized as a type system without relying on the relation $\succeq$ which is defined in terms of the operational semantics, we have proposed formalizing an interface of an agent in terms of the operational semantics, which enables us to directly justify the inference of interfaces that

takes place in the program analysis. The way of formalizing types in this way for justifying process specialization is one of the contributions of this work.

Future work includes a definition of the formal semantics of the intermediate language, which enables us to guarantee the correctness of various intermediate code optimization. Although this dissertation itself does not aim to present efficient implementation, it is also important to implement and evaluate an optimizing compiler for concurrent logic programs based on our framework.

Chapter 6

Occurs-Check Analysis under Cooperative Modings

6.1 Introduction

6.1.1 Background

Occurs-check is a task in unification that tests whether the unification generates an infinite structure. For example, the unification `X = f(X)` does not pass the occurs-check since this unification requires that `X` be bound to the infinite data structure `f(f(f(...)))`. Infinite structures are not allowed to exist if we need to justify the soundness of a runtime system of logic programming as a prover of first-order logic. Furthermore, they can also be a cause of unexpected infinite computation since they can be passed to recursive predicates that are only intended to terminate on finite input. Nonetheless, since runtime occurs-check is costly, most language systems of logic programming intentionally omit the test unless explicitly specified and grant unification to be successful even if it does not pass the test. The omission of occurs-check also incurs extra overheads in the runtime system itself for manipulating potential infinite structures. Thus, the study of static occurs-check is asked for.

6.1.2 Related Work

Static occurs-check has two approaches [4, 8]: occurs-check reduction, and NSTO (Not Subject To Occurs-check) tests. The former tries to find those

unification operations performed in a program that require no occurs-check. On the other hand, the latter attempts to guarantee that a program generates no infinite structures. The occurs-check reduction approach aims to improve the efficiency of program execution while admitting infinite structures in the program. On the other hand, the NSTO approach mainly attempts to guarantee the soundness and termination of a program.

Finite-tree analysis [4] belongs to the former approach; it finds out program variables that will never hold infinite structures and guarantees the omission of occurs-check in the related unification. Since their analysis is based on abstract interpretation for obtaining correct and accurate results, it is difficult to apply their method to general logic programming languages that have no predetermined reduction strategies, including concurrent logic programming languages that are equipped with suspension and resumption of goals.

For such languages, we must perform some analysis that does not depend on any specific reduction strategy. In fact, instead of abstract interpretation, analysis methods based on some static information of a program, such as input/output modes of usage, are considered to be more suitable.

The existing methods of mode-based approach, however, cannot guarantee that programs with bidirectional communication, for example the concurrent logic program shown in Figure 6.1, do not produce infinite structures in any finite time. The reason is that they all have exploited only top-level mode information of each argument of a predicate. [3, 8]

6.1.3 Our Method

This chapter proposes an algorithm that ensures the NSTO property of a well-moded logic program. The algorithm exploits precise mode information of the program in the sense explained in the paper [34]. A mode inference algorithm for concurrent logic programs adapted from the one in the above paper is also presented, so that our NSTO test can be implemented in practice.

Our NSTO test attempts to assert that each clause is not a cause of infinite structures. Each clause is tested whether the mode information related to that clause can be strengthened without introducing inconsistency so that a particular characteristic described using the notion of 'asymptotic equality' is observed. We will explain how infinite structures are related to asymptotic equality and prove that our algorithm is correct.

```
:- print(Hs0), Hs0 = [1|Hs1],
        mul(2,Hs0,L2), mul(3,Hs0,L3), mul(5,Hs0,L5),
        merge(L2,L3,L23), merge(L23,L5,Hs1).
merge([A|As1],[B|Bs1],Ms) :- A < B |
        Ms = [A|Ms1], merge(As1,[B|Bs1],Ms1).
merge([A|As1],[B|Bs1],Ms) :- A > B |
        Ms = [B|Ms1], merge([A|As1],Bs1,Ms1).
merge([A|As1],[B|Bs1],Ms) :- A=:=B |
        Ms = [A|Ms1], merge(As1,    Bs1,Ms1).
mul(N,As0,Ms) :- As0 = [A|As1] |
        Ms = [M|Ms1], M := N*A, mul(N,As1,Ms1).
```

Figure 6.1: Program with bidirectional communication

The rest of this chapter is organized as follows. Section 6.2 formalizes the NSTO property. Section 6.3 reduces the property to the problem on *initial graphs*. Section 6.4 explains how to use mode information in checking that initial graphs do not generate cycles. Section 6.5 shows an analysis example. Section 6.6 discusses the limitations of our analysis, and Section 6.7 concludes.

6.2 Problem Formalization

This section defines an abstract transition system and formalizes the NSTO property. In principle, any logic language, including the one defined in Chapter 2, can be embedded into this abstract transition system. This means that a logic program is NSTO if the embedded abstract program is NSTO.

6.2.1 Terms and Paths

Here, we define notations used in this chapter.

Definition 6.1 We assume that the set *Var* of *variables* and the set *Term* of *terms* of first-order logic are given. We denote by $Vars(t)$ the set of variables occurring in term t. We write $s \equiv t$ to mean that terms s and t are syntactically equal. A term is said to be *linear* if and only if it does not contain duplicate variables. □

Definition 6.2 We denote by *Func* the set of signatures f/n, pairs of a function symbol f and a non-negative integer n, used for constructing terms from variables. We call each element of *Func* a *functor*. We define by $\mathit{Path} \stackrel{\text{def}}{=} \{\langle f/n, i\rangle \mid f/n \in \mathit{Func},\, i \in 1..n\}^*$ the set of *paths*. We write ϵ for the empty path and $p \bowtie q \stackrel{\text{def}}{\Leftrightarrow} p = \epsilon \vee q = \epsilon$. For integers m and n, we write $m..n$ to denote the set $\{m, m+1, \ldots, n\}$. □

We use a path to describe a specific term position within a term. For example, the path $\langle \texttt{f}/3, 1\rangle$ represents the first argument of a term constructed with the function symbol `f` of arity 3. The empty path ϵ represents a term itself.

In this work, we will use function symbols also as predicate symbols in order to simplify the formalization. It means that a goal is represented as a term. Hence, a nonempty path also describes a specific term position within a goal. For instance, the path $\langle \texttt{merge}/3, 1\rangle$ represents the first argument of a call to the predicate `merge` of arity 3.

Definition 6.3 Let $\bot \notin \mathit{Term}$. For every path p, we define the function $\overrightarrow{p} \colon \mathit{Term} \to \mathit{Term} \cup \{\bot\}$ as follows:

$$\begin{cases} \overrightarrow{\epsilon}(t) & \stackrel{\text{def}}{=} t \\ \overrightarrow{\langle f/n, i\rangle p}(f(t_1, \ldots, t_n)) & \stackrel{\text{def}}{=} \overrightarrow{p}(t_i) \\ \overrightarrow{\langle f/n, i\rangle p}(t) & \stackrel{\text{def}}{=} \bot \qquad \text{otherwise.} \end{cases}$$

□

6.2.2 Equation Sets

Next, we formalize alias information accumulated during program execution as equation sets. An *alias* refers to the unification between two term positions within two variables. For example, the unification `Y = f(X,a)` makes `X` and the first argument of `Y` be an alias. We will formalize an alias as an equation over equation targets.

Definition 6.4 For every pair of a variable X and a path p, we call Xp an *equation target*. We mean by $Xp \equiv Yq$ the syntactic equality of the two equation targets Xp and Yq. For equation targets Xp and Yq, we call $Xp = Yq$ an *equation*. □

Definition 6.5 For an equation set C and a variable X, we denote by $\exists_X(C)$ the set of those equations in C which do not contain X. □

Example. $\exists_{\mathtt{Y}}(\{\mathtt{X}=\mathtt{X}, \mathtt{X}=\mathtt{Y}, \mathtt{Y}\langle \mathtt{f}/3, 2\rangle = \mathtt{Z}\}) = \{\mathtt{X}=\mathtt{X}\}$. □

The definition of $\exists_X(C)$ makes sense only if C is closed by some notion of transitivity. For example, the input to $\exists_{\mathtt{Y}}(\cdot)$ in the above example should have contained $\mathtt{X}\langle \mathtt{f}/3, 2\rangle = \mathtt{Z}$ to obtain a meaningful result.

Definition 6.6 For $n \geq 0$ we call a dot-separated sequence of positive integers $i_1.i_2.\ldots.i_n$ a *goal identifier* (typically written as w), and $\overline{i_1.i_2.\ldots.i_n}$ a *head identifier*, and each of them a *node identifier* (typically written as u). For goal identifier w, we define $\overline{\overline{w}} \stackrel{\text{def}}{=} w$. We assume that we have a one-to-one map $u \mapsto \langle u \rangle$ from node identifiers to variables. We denote by ε_0 the empty goal identifier.

For every variable X, we define $|X| \stackrel{\text{def}}{=} \langle w \rangle$ if there exists a goal identifier w such that $X \equiv \langle \overline{w} \rangle$; otherwise we define $|X| \stackrel{\text{def}}{=} X$. □

We must explain the intention of the above definition. We use a goal identifier to express a particular goal in the program execution; the integer sequence denotes the history of subgoal indices within the clauses. Similarly, we use a head identifier to express a particular head matched in the program execution. The mapping $\langle \cdot \rangle$ is a device for encoding node identifiers as variables so that we can use them in an equation set.

We can think of $|\cdot|$ as a function that returns the 'absolute value' of the specified node identifier. For example, we have that $|\langle\overline{1.3}\rangle| \equiv |\langle 1.3\rangle| \equiv \langle 1.3\rangle$.

Example. The equation $(\langle\overline{1.3}\rangle\langle \mathtt{p}/3, 1\rangle = \langle 1.3.2\rangle\langle \mathtt{q}/2, 2\rangle)$ can express the alias introduced by the variable `A` in a clause `p(A,B,C) :- p(X,B,C), q(X,A)` used in the reduction of a call to `p`. The goal identifier 1.3 expresses that this call to `p` was the third body goal of the clause used in the reduction of the first body goal of the clause used in the reduction of the initial goal. We consider that the empty goal identifier ε_0 expresses the initial goal. □

Definition 6.7 We define the equation set ID and the functions Sat, $\mathcal{S}$, $-\star$ on equation sets as follows:

$$
\begin{array}{rcl}
Sat(C) & \stackrel{\text{def}}{=} & \bigcup_{k=0}^{\infty} \mathcal{S}^k(ID \cup C\star) \\
\mathcal{S}^0(C) & \stackrel{\text{def}}{=} & C \\
\mathcal{S}^{k+1}(C) & \stackrel{\text{def}}{=} & \mathcal{S}(\mathcal{S}^k(C)) \qquad \text{for all } k \geq 0 \\
ID & \stackrel{\text{def}}{=} & \{Xp = Xp \mid X \in Var,\, p \in Path\} \\
C\star & \stackrel{\text{def}}{=} & \{Yq = Xp \mid (Xp = Yq) \in C\} \cup C \\
\mathcal{S}(C) & \stackrel{\text{def}}{=} & C \cup \{e * f \mid e \in C,\, f \in C,\, e * f \neq \bot\}
\end{array}
$$

where

$$\begin{cases} (Xp{=}Yq) * (Y'r{=}Zs) \stackrel{\text{def}}{=} (Xpq' = Zsr') & \text{if } |Y|qq' \equiv |Y'|rr' \text{ and } q' \bowtie r' \\ e * f \stackrel{\text{def}}{=} \bot & \text{otherwise.} \end{cases}$$

□

The equation set ID contains no information. The set $C\star$ is the symmetric closure of the equation set C. The equation $(Xp = Yq) * (Y'r = Zs)$ represents the application of the 'transitivity' law to these equations, whose result is defined when there exists q' such that $|Y|qq' \equiv |Y'|r$, in this case $(Xpq' = Yqq')$ and $(Y'r = Zs)$ give $(Xpq' = Zs)$, or when there exists r' such that $|Y|q \equiv |Y'|rr'$, in this case $(Xp = Yq)$ and $(Y'rr' = Zsr')$ give $(Xp = Zsr')$. The equation set $Sat(C)$ is obtained by closing an equation set C with reflexivity, symmetry and transitivity.

Definition 6.8 An equation of the form $(Xp = Ypq)$ that satisfies $|X| \equiv |Y|$ and $q \neq \epsilon$ is called an *infinite equation*. We say that an equation set C *has an infinite structure* if there exists some infinite equation in $Sat(C)$; otherwise we write $\mathrm{fin}(C)$. □

6.2.3 Abstract Transition System

This subsection defines an abstract transition system in terms of which we formalize the NSTO property. The system to be defined accumulates every alias information and, on the other hand, abstracts away any information on instantiation of variables and hence any consistency checks (functor matching tests) performed in ordinary logic programming languages. This setting ensures that the transition system we will define is the abstraction of the original language.

Definition 6.9 For $n \geq 0$ and terms $h, b_1, \ldots, b_n$, we call $(h \leftarrow b_1, \ldots, b_n)$ a *clause*. The term h is called its *head* and each b_i is called its *body*. A set of clauses whose heads are all linear is called a *program*. □

For simplicity, we have formalized a goal (an atomic formula) as a term.

Unification goals are considered to be polymorphic, that is, each unification goal in a program has its own mode of usage. To formalize this, we will

subscript each unification in the program with a distinct positive integer k and treat them as different predicates, though their semantics are the same. To this end, we define as follows.

Definition 6.10 For $k \geq 1$, we assume there exists a function symbol written as $=_k$. Let $U_k \stackrel{\text{def}}{=} (X =_k X)$ where X is some arbitrary variable. A clause of the form $(U_k \leftarrow)$ is called a *unification clause*. □

Next, we define configurations of our abstract transition system.

We refer to a pair $w : b$ of a goal identifier w and a term b as a *labeled goal*. A labeled goal uniquely determines a goal that appears in the program execution. A configuration of the abstract transition system is a pair of a set of labeled goals and an equation set that accumulates alias information. We give two formulas for describing alias information as follows.

Definition 6.11 For every clause w and a term c, we define two equation sets as follows:

$$\begin{aligned}
\mathcal{G}(w, (h \leftarrow b_1, \ldots, b_n)) &\stackrel{\text{def}}{=} \{\langle \overline{w} \rangle \langle =_k /2, 1 \rangle = \langle \overline{w} \rangle \langle =_k /2, 2 \rangle \mid h \equiv U_k\} \star \\
&\cup \{\langle w.i \rangle p = \langle \overline{w} \rangle q \mid \overrightarrow{p}(b_i) \equiv \overrightarrow{q}(h) \in \mathit{Var}\} \star \\
&\cup \mathcal{T}(w, (h \leftarrow b_1, \ldots, b_n)), \\
\mathcal{T}(w, (h \leftarrow b_1, \ldots, b_n)) &\stackrel{\text{def}}{=} \{\langle w.i \rangle p = \langle w.j \rangle q \mid \overrightarrow{p}(b_i) \equiv \overrightarrow{q}(b_j) \in \mathit{Var}\} \setminus \mathit{ID}\,.
\end{aligned}$$

□

The formula $\mathcal{G}(w, c)$ defines the equation set generated in a reduction of a w-labeled goal, say $w : b$, with the clause c. Each of the three parts in the definition of $\mathcal{G}$ describes multiple occurrences of variables between head-head, head-body, and body-body positions, respectively. Multiple occurrences of variables between heads take place only for unification clauses. The formula $\mathcal{T}(w, c)$ describes the multiple occurrences between body goals.

We define the second component of a configuration as an equation set of the form $\bigcup_{i=1}^{n} \mathcal{G}(w_i, c_i)$.

Finally, the abstract transition system is defined as follows.

Definition 6.12 For program $\boldsymbol{P}$, we define the abstract transition system $\longrightarrow$ as follows:

$$\begin{aligned}
&\text{(R1)} \quad \langle \{w : a\} \cup Q,\, G \rangle \longrightarrow \langle \{w.i : b_i \mid i \in 1..n\} \cup Q,\, G \cup \mathcal{G}(w, c) \rangle \\
&\qquad\qquad \text{if } \; c = (h \leftarrow b_1, \ldots, b_n) \in \boldsymbol{P} \\
&\text{(R2)} \quad \langle \{w : (s =_k t)\} \cup Q,\, G \rangle \longrightarrow \langle Q,\, G \cup \mathcal{G}(w, (U_k \leftarrow)) \rangle
\end{aligned}$$

□

Definition 6.13 (NSTO) A program $\boldsymbol{P}$ is said to be NSTO if and only if:

$$\forall Q, G(\, \langle \{\varepsilon_0 : \mathrm{main}\}, \{\} \rangle \longrightarrow \langle Q, G \rangle \ \text{ implies } \ \mathrm{fin}(G)\,).$$

□

Notes on the Linear Head Condition.

Since the existence of clauses that are not linear headed, including unification clauses ($U_k \leftarrow$), is apparently one of the fundamental sources of infinite structures, the linear head condition in the definition of programs may seem too restrictive. However, assuming the cooperativeness condition we will define later, one can transform clauses so that every head of a program clause is linear. Intuitively, cooperativeness means that each variable in a clause has exactly one output occurrence.

Let us forget for now about the input/output mode of usage and the cooperativeness. Logically, we can make a program linear headed by rewriting every multiple occurrence of a variable in a clause head into unification goals in the body. For example, we can transform

```
p(A,A) :- body(A,A).
```

into

```
p(A,B) :- A =1 B, body(A,A).
```

The real issue is that the resulting program is possibly not well moded in a moding system even if the original program is well moded. Before proceeding to the case of our moding system, we introduce a taxonomy for unification.

Each unification that takes place at runtime is either active or passive: *active* one unifies a variable with a term while *passive* one unifies two non-variable terms. Performing passive unification may require unification between corresponding subterms, each of which is again either active or passive.

In Section 6.4, in order to perform mode inference, we will require the following two assumptions. Firstly, we will assume that whether each unification is active or passive is determined statically. Secondly, we will assume that every unification goal is active unification. Then, every passive unification should be represented in terms of head matching, not as a unification goal. For example, if the `A`'s in the head are intended to express passive unification, we should transform

```
p(A,A) :- body(A,A).
```

into

```
p(A,B) :- A ==1 B, body(A,A).
```

together with the clause

f`(A1,...,A`n`)` $==_1$ f`(A1,...,A`n`) :- .`

for each possible functor f/n where $==_1$ is a fresh name for a predicate.

In general, performing such transformation appropriately is difficult to automate. Nonetheless, we will show in Section 6.5.1 how to perform this transformation automatically by assuming the cooperativeness condition.

6.3 Reduction to Initial Graphs

In this section, we will reduce the NSTO property to a property of 'initial graphs' of the program. An initial graph will be defined as an equation set $\mathcal{T}(w, c)$ in Section 6.3.3.

6.3.1 Chains

Definition 6.14 (Chains) A *chain* is a nonempty sequence of semicolon-separated pairs of an equation and a path of the form

$$\langle\langle v_0\rangle q_0 = \langle u_1\rangle p_1; r_1\rangle \ldots \langle\langle v_{n-1}\rangle q_{n-1} = \langle u_n\rangle p_n; r_n\rangle$$

such that

$$\forall i \in 2..n(\, |\langle u_{i-1}\rangle| p_{i-1} r_{i-1} \equiv |\langle v_{i-1}\rangle| q_{i-1} r_i \,)$$

holds. We call n its *length*.

For any chain H, we define $E(H) \stackrel{\text{def}}{=} \{\langle v_{i-1}\rangle q_{i-1} = \langle u_i\rangle p_i \mid i \in 1..n\}$ and the *object equation* of H by $O(H) \stackrel{\text{def}}{=} (\langle v_0\rangle q_0 r_1 = \langle u_n\rangle p_n r_n)$. □

Theorem 6.1 Let $|Xp = Yq| \stackrel{\text{def}}{=} (|X|p = |Y|q)$. For every equation set G and equation $e \in Sat(G) \setminus ID$, there exists a chain H such that $|O(H)| = |e|$ and $E(H) \subseteq G\star$ hold.

Proof. We define that $\pi_L(Xp = Yq) \stackrel{\text{def}}{=} Xp$ and $\pi_R(Xp = Yq) \stackrel{\text{def}}{=} Yq$. We also define that $\langle e_i; r_i \rangle_{i=1}^n \cdot s \stackrel{\text{def}}{=} \langle e_i; r_i s \rangle_{i=1}^n$ and that $(Xp = Yq) \cdot s \stackrel{\text{def}}{=} (Xps = Yqs)$. We prove this theorem by constructing a function M that maps each equation $e \in Sat(G) \setminus ID$ to a chain $M(e)$ such that $|O(M(e))| = |e|$ and $E(M(e)) \subseteq G\star$ hold.

Here, we extend the domains of M and E as follows. Let λ be an empty sequence. We define that $M(e) \stackrel{\text{def}}{=} \lambda$ for every $e \in ID$ and that $E(\lambda) \stackrel{\text{def}}{=} \{\}$. For each $e \in ID$, we have $E(M(e)) \subseteq G\star$ and $e \in Sat(G)$. Now, we will construct $M(e)$ for each $e \in Sat(G) \setminus ID$ using mathematical induction on k where k is the smallest integer such that $e \in \mathcal{S}^k(ID \cup G\star)$.

If $k = 0$, we have $e \in G\star$. Let $M(e) = \langle e; \epsilon \rangle$. We have $O(M(e)) = e$ and $E(M(e)) = \{e\} \subseteq G\star$.

Let $k \geq 1$ and assume that the proposition holds for any integer less than k. There exist $p, q, r, s, X, Y, Y', Z, a, b$ such that $e = (Xpa = Zsb)$ and $(Xp = Yq)$, $(Y'r = Zs) \in \mathcal{S}^{k-1}(ID \cup G\star)$ and $|Y|qa \equiv |Y'|rb$, $a \bowtie b$ hold. Let $M(e) = (M(e_1) \cdot a)(M(e_2) \cdot b)$ where $e_1 = (Xp = Yq)$ and $e_2 = (Y'r = Zs)$. By induction hypothesis, we have $E(M(e)) = E(M(e_1) \cdot a) \cup E(M(e_2) \cdot b) = E(M(e_1)) \cup E(M(e_2)) \subseteq G\star \cup G\star = G\star$. We also have that $|O(M(e_i))| = |e_i|$ for $i = 1, 2$ if $e_i \notin ID$.

We have three cases.

If $(Xp = Yq) \notin ID$ and $(Y'r = Zs) \notin ID$, it is the case that

$$\begin{aligned}
|O(M(e))| &= |O((M(e_1) \cdot a)(M(e_2) \cdot b))| \\
&= |\pi_L O(M(e_1) \cdot a) = \pi_R O(M(e_2) \cdot b)| \\
&= (\pi_L |O(M(e_1))| a = \pi_R |O(M(e_2))| b) \\
&= (\pi_L |e_1| a = \pi_R |e_2| b) \\
&= (|X|pa = |Z|sb) \\
&= |Xpa = Zsb| \\
&= |e|.
\end{aligned}$$

If $Y'r \equiv Zs$, we have $|O(M(e))| = |O(M(e_1) \cdot a)| = |O(M(e_1))| \cdot a = |e_1| \cdot a = (|X|pa = |Y|qa) = (|X|pa = |Y'|rb) = (|X|pa = |Z|sb) = |e|$.

If $Xp \equiv Yq$, we have $|O(M(e))| = |O(M(e_2) \cdot b)| = |O(M(e_2))| \cdot b = |e_2| \cdot b = (|Y'|rb = |Z|sb) = (|Y|qa = |Z|sb) = (|X|pa = |Z|sb) = |e|$.

It follows that the proposition holds for k. □

6.3.2 Cycles

Definition 6.15 (Connection Graphs) We assume a program $\boldsymbol{P}$ is given. For any $n \geq 0$, clauses $c_1, \ldots, c_n \in \boldsymbol{P}$, and distinct goal identifiers $w_1, \ldots, w_n$, we call the equation set $\bigcup_{i=1}^{n} \mathcal{G}(w_i, c_i)$ a *connection graph*. □

Every second component of a configuration of the abstract transition system is a connection graph.

Definition 6.16 (Cycles) Let $n \geq 1$ and $u_1, \ldots, u_n$ be node identifiers. A *cycle* is a chain $H = \langle\langle u_{i-1}\rangle q_{i-1} = \langle\overline{u_i}\rangle p_i; r_i\rangle_{i=1}^{n}$ such that (1) $\exists q(\langle u_n\rangle p_n r_n \equiv \langle u_0\rangle q_0 r_1 q)$, and (2) $E(H)$ is a subset of some connection graph. We define $L(H) \stackrel{\text{def}}{=} \{\pi_L(e) \,|\, e \in E(H)\}$ and $R(H) \stackrel{\text{def}}{=} \{\pi_R(e) \,|\, e \in E(H)\}$. □

Note that cycles are defined only after a program $\boldsymbol{P}$ is given. Note also that the expression $\overline{u_i}$ can represent a goal identifier, which happens if and only if the node identifier u_i is a head identifier.

Theorem 6.2 If there is a connection graph with infinite structure, then there exists a cycle.

Proof. Let G be a connection graph that has an infinite structure. By the definition of a connection graph, it holds that $G = G\star$.

By Theorem 6.1, we can choose a shortest chain H such that (a) $O(H)$ is an infinite equation, and (b) $E(H) \subseteq G$. We prove that this H is a cycle.

Let $H = \Sigma_{1,n}$ and $\Sigma_{a,b} = \langle\langle v_{j-1}\rangle q_{j-1} = \langle u_j\rangle p_j; r_j\rangle_{j=a}^{b}$ for any $1 \leq a \leq b \leq n$. By the definition of chains, we have $|\langle u_i\rangle| = |\langle v_i\rangle|$ for each $1 \leq i \leq n-1$. Assume, hypothetically, that $u_i = v_i$. Then, there exist terms t_1, t_2, t_3 such that $\overrightarrow{\overline{q_{i-1}}}(t_1) \equiv \overrightarrow{\overline{p_i}}(t_2) \in \mathit{Var}$ and $\overrightarrow{\overline{q_i}}(t_2) \equiv \overrightarrow{\overline{p_{i+1}}}(t_3) \in \mathit{Var}$ hold. Since $p_i r_i = q_i r_{i+1}$, we have $p_i = q_i$; therefore $\overrightarrow{\overline{q_{i-1}}}(t_1) \equiv \overrightarrow{\overline{p_{i+1}}}(t_3)$. Thus, from the definition of $\mathcal{G}$ and the condition that the head of each clause is linear, we have $e = (\langle v_{i-1}\rangle q_{i-1} = \langle u_{i+1}\rangle p_{i+1}) \in \mathit{ID} \cup G$, and we have $r_i = r_{i+1}$. Let $H' = \Sigma_{1,i-1}\Sigma_{i+2,n}$ if $e \in \mathit{ID}$; otherwise let $H' = \Sigma_{1,i-1}\langle e; r_i\rangle\Sigma_{i+2,n}$. We can confirm H' is a chain that satisfies the conditions (a) and (b), which contradicts the shortestness of H. Hence, we have $u_i = \overline{v_i}$.

What remains to prove is that $u_n = \overline{v_0}$. Since H satisfies (a), there exists $q \neq \epsilon$ such that $|\langle u_n\rangle| p_n r_n \equiv |\langle v_0\rangle| q_0 r_1 q$. Let $H'' = \Sigma_{2,n}\langle\langle v_0\rangle q_0 = \langle u_1\rangle p_1; r_1 q\rangle$. It is easy to show that H'' is a chain that again satisfies (a) and (b) as well as the shortestness condition. Hence the previous paragraph ensures that $u_n = \overline{v_0}$. □

Theorem 6.3 If no cycles exist, the program is NSTO.

Proof. Assume program $\boldsymbol{P}$ is not NSTO. Then there is a connection graph G such that $\langle \{\varepsilon_0 : \mathrm{main}\}, \{\} \rangle \longrightarrow^* \langle Q, G \rangle$ with some Q and that G has an infinite structure. By Theorem 6.2 there is a cycle. □

6.3.3 Initial Graphs

Definition 6.17 (Initial Graphs) For a goal identifier w and a clause $c \in \boldsymbol{P}$, we call $\mathcal{T}(w, c)$ an *initial graph*. □

The following theorem is the motivation of introducing initial graphs. It states that every cycle contains an alias between two body goals of a clause.

Theorem 6.4 For any cycle H there exists an initial graph $\mathcal{T}(w, c)$ such that $E(H) \cap \mathcal{T}(w, c) \neq \{\}$.

Proof. Let $w.m$ be a shortest goal identifier occurring in H where m is an integer. There exist e, p, q, u such that $e = (\langle w.m \rangle p = \langle u \rangle q) \in E(H)$ or $e = (\langle u \rangle q = \langle w.m \rangle p) \in E(H)$. By definition of cycles, there exists a clause c such that $e \in \mathcal{G}(w, c)$. If $u = \overline{w}$ then the goal identifier w occurs in H, next to e, which contradicts the shortestness of $w.m$. Thus, by the definition of $\mathcal{G}(w, c)$ we have $u = w.k$ with some k; therefore $e \in \mathcal{T}(w, c)$. □

Now we define a predicate named fingen, which states that the initial graph of a clause (or its subset) is not a cause of generating infinite structures.

Definition 6.18 Let T be a subset of some initial graph. We write fingen(T) if and only if every cycle H satisfies $E(H) \cap T = \{\}$. □

Theorem 6.5 If $\forall w \forall c(\mathrm{fingen}(\mathcal{T}(w, c)))$, the program is NSTO.

Proof. Assume the program is not NSTO. By Theorem 6.3, we know that there exists a cycle H. By Theorem 6.4, there exist some w, c such that $E(H) \cap \mathcal{T}(w, c) \neq \{\}$. This contradicts the hypothesis that fingen($\mathcal{T}(w, c)$). □

6.4 Using Moding Functions

In this section, we explain how to use mode information in checking that an initial graph T satisfies fingen(T), that is, T does not generate cycles.

6.4.1 Moding functions

Definition 6.19 (Modes) We call the symbols *in* and *out modes*. We call a function from *Path* to $\{in, out\}$ a *moding function*. For every moding function s and every path p, we define the moding function s/p by $(s/p)(q) \stackrel{\text{def}}{=} s(pq)$. We define the moding function $\overline{s}$ so that $\forall p \in Path\,(\{s(p), \overline{s}(p)\} = \{in, out\})$ holds. □

The mode *in* represents that the caller instantiates the value of a path, while the mode *out* represents that the callee instantiates it.

Definition 6.20 (Mode Constraints) We call a logical formula that constrains the values of the distinguished moding function m a *mode constraint*, or constraint[1] in short. For a set E of mode constraints, we denote by $M(E)$ the set of all the moding functions m that satisfy E. A set E of mode constraints is said to be *consistent* if and only if $M(E)$ is not empty. □

Definition 6.21 We consider that any clause $(h \leftarrow b_1, \ldots, b_n)$ imposes the following mode constraints:

(BU) $\quad m/\langle =_k/2, 1\rangle = \overline{m/\langle =_k/2, 2\rangle}$ if $b_i \equiv (s =_k t)$
(BV_+) $\quad \forall X \in Vars(h)\, \forall p(\overrightarrow{p}(h) \equiv X)\ (\mathcal{R}(\{\!|\overline{m/p}|\!\} + \{\!|m/q \mid \overrightarrow{q}(b_i) \equiv X|\!\}))$
(BV_0) $\quad \forall X \in Vars(b_1, \ldots, b_n) \setminus Vars(h)\ (\mathcal{R}(\{\!|m/q \mid \overrightarrow{q}(b_i) \equiv X|\!\}))$

where $\{\!|\ldots|\!\}$ denotes a multiset and $\mathcal{R}(S)$ denotes the *cooperativeness* condition for S defined by:

$$\mathcal{R}(S) \stackrel{\text{def}}{\Leftrightarrow} \forall p \in Path\, \exists s \in S\,(\, s(p) = out \wedge \forall s' \in S - \{\!|s|\!\}\,(s'(p) = in)\,).$$

For a program $\boldsymbol{P}$, we denote by $\mathcal{M}(\boldsymbol{P})$ the set of all mode constraints imposed by the clauses in $\boldsymbol{P}$. We say that $\boldsymbol{P}$ is *cooperatively moded* if $\mathcal{M}(\boldsymbol{P})$ is consistent. □

[1]The term *constraint* is used in this chapter differently from in the other chapters.

(BU) stands for body unification and says that (the two moding functions representing) the two arguments of a unification goal are cooperative—note that $\mathcal{R}(\{\!|\, s, s' \,|\!\})$ is equivalent to $s = \overline{s'}$. (BV_+) and (BV_0) stand for body variables and say that all the body occurrences of a particular variable in a clause, together with the inverse of its head occurrence if any, are cooperative.

The three rules for imposing mode constraints are a subset of those proposed in the paper [34], which defines *well-moded* concurrent logic programs. They additionally require a pointwise mode constraint $m(p) = in$ for each occurrence of a function symbol at the path p in a clause so that m be compatible with the operational semantics of concurrent logic programs. We have discarded the imposition of pointwise constraints because our occurs-check analysis is not based on such compatibility.

Definition 6.22 For moding functions s and s', we define $s \leftrightarrow s' \stackrel{\mathrm{def}}{\Leftrightarrow} \forall r \in Path(s(r) = in \lor s'(r) = in)$. □

The relationship $m/p \leftrightarrow m/p'$ represents that the moding functions m/p and m/p' together do not break the cooperative condition. It is obvious that $m/p \leftrightarrow m/p'$ implies $m/pq \leftrightarrow m/p'q$.

For any node identifier u, we define the operator $I(u)$ on moding functions depending on whether u is a goal identifier or a head identifier:

Definition 6.23 For any goal identifier w and any moding function s, we define $I(w)(s) \stackrel{\mathrm{def}}{=} s$ and $I(\overline{w})(s) \stackrel{\mathrm{def}}{=} \overline{s}$. □

Proposition 6.1 Let $\boldsymbol{P}$ be a program. Every equation $(\langle u \rangle p = \langle v \rangle q) \in \mathcal{G}(w, c)$ satisfies $I(u)(m/p) \leftrightarrow I(v)(m/q)$ for any moding function $m \in M(\mathcal{M}(\boldsymbol{P}))$.

Proof. Let $e = (\langle u \rangle p = \langle v \rangle q) \in \mathcal{G}(w, c)$. As we will see, there exists a multiset S of moding functions such that $\mathcal{R}(\{\!|\, I(u)(m/p), I(v)(m/q) \,|\!\} + S)$; therefore we have $I(u)(m/p) \leftrightarrow I(v)(m/q)$.

We have three sources of e, corresponding to the three parts in the definition of $\mathcal{G}$. When it has been generated in the first part, we have $v = u$ and there exists k such that $\{p, q\} = \{\langle =_k /2, 1\rangle, \langle =_k /2, 2\rangle\}$. Therefore, we have $m/p = \overline{m/q}$, which is equivalent to $\mathcal{R}(\{\!|\, I(u)(m/p), I(v)(m/q) \,|\!\})$. When it has been generated in the second or third part, we have terms s and t in a clause such that $\overrightarrow{p}(s) \equiv \overrightarrow{q}(t) \in Var$. It follows that there exists some S. □

6.4.2 Asymptotic Equality

Next, we will introduce the notion of the asymptotic equality of two moding functions. This notion has been devised by the author in order to formulate a sufficient condition of the NSTO property in our occurs-check analysis.

Definition 6.24 We define two moding functions IN and OUT by $\forall p \in Path(OUT(p) = out)$ and $IN \stackrel{\text{def}}{=} \overline{OUT}$. These two moding functions IN and OUT are said to be *uniform*. For any moding function s and a uniform moding function u, we write $s \to u$ to mean that $\forall q \exists r(\, s/qr = u\,)$ and call this formula a *convergence constraint*. □

Intuitively, $s \to u$ represents that s converges to u. It is easy to confirm that $s \to u$ implies $s/p \to u$.

Definition 6.25 For moding functions s and s', we write $s \simeq s'$ to mean that $\exists u \in \{IN, OUT\}\,(s \to u \wedge s' \to u)$ and say that s and s' are *asymptotically equal* if and only if $s \simeq s'$. □

Note that it is not always the case that $s \simeq s$.

6.4.3 Moding and Infinite Structures

Now, we can express a sufficient condition for guaranteeing that a program cannot generate infinite structures, in terms of moding functions.

Definition 6.26 Let G be an equation set of the form

$$\{\langle u_{2i-1}\rangle p_{2i-1} = \langle u_{2i}\rangle p_{2i} \,|\, i \in 1..n\}\,.$$

We define $Nodes(G) \stackrel{\text{def}}{=} \{u_{2i-1} \,|\, i \in 1..n\} \cup \{u_{2i} \,|\, i \in 1..n\}$ and for every variable X we define $G^X \stackrel{\text{def}}{=} \{p_{2i-1} \,|\, \langle u_{2i-1}\rangle \equiv X\} \cup \{p_{2i} \,|\, \langle u_{2i}\rangle \equiv X\}$. □

Nodes and $-^X$ extract nodes and paths from equation sets, respectively.

The following proposition intuitively states that every goal appearing in a cycle appears twice in that cycle.

Proposition 6.2 For every cycle $\langle\langle u_{i-1}\rangle q_{i-1} = \langle\overline{u_i}\rangle p_i; r_i\rangle_{i=1}^n$ it is the case that $\{u_0, \ldots, u_{n-1}\} = \{\overline{u_1}, \ldots, \overline{u_n}\}$.

Proof. Let $p_n r_n = q_0 r_1 q$. Since $\langle \langle u_{i-1} \rangle q_{i-1} = \langle \overline{u_i} \rangle p_i ; r_i \rangle_{i=2}^{n} \langle \langle u_n \rangle q_0 = \langle \overline{u_1} \rangle p_1 ; r_1 q \rangle$ is also a cycle and $u_n = u_0$, we only have to prove that (a) if $\overline{u_1}$ is a goal identifier then there exists some k such that $u_k = \overline{u_1}$, and (b) if u_{n-1} is a goal identifier then there exists some ℓ such that $\overline{u_\ell} = u_{n-1}$. For any integer m, We can repeatedly perform this index rotation so that either (a) or (b) can be used to prove the inclusion $\overline{u_m} \in \{u_0, \ldots, u_{n-1}\}$ or $u_m \in \{\overline{u_1}, \ldots, \overline{u_n}\}$. Note that (b) covers the case where $\overline{u_m}$ is a head identifier and vice versa. We will prove (a) first.

For each node identifier u, we define $|u|$ by $\langle |u| \rangle = |\langle u \rangle|$.

Let k be the maximum integer in $1..n$ such that $\forall i \in 1..k \, \exists \delta_i (\, |\overline{u_i}| = \overline{u_1}.\delta_i \,)$. If $k = n$, we have $u_n = \overline{u_1}$ because of $(\langle u_n \rangle q_0 = \langle \overline{u_1} \rangle p_1)$ and $|\overline{u_n}| = \overline{u_1}.\delta_n$. Assume $k < n$. If δ_k is not empty, $|\overline{u_{k+1}}|$ also begins with $\overline{u_1}$, which violates the assumption. Thus $|\overline{u_k}| = \overline{u_1}$. If $u_k = u_1$, we have $\exists \delta (\, \overline{u_{k+1}} = \overline{u_k}.\delta \,)$ because of $(\langle u_k \rangle q_k = \langle \overline{u_{k+1}} \rangle p_{k+1})$, which violates the assumption.

In the above proof of (a), we did not use the defining condition (1) of a cycle other than $u_n = u_0$. Hence, replacing in the previous paragraph the cycle by $\langle \langle \overline{u_i} \rangle p_i = \langle u_{i-1} \rangle q_{i-1} ; r_i \rangle_{i=n}^{1}$, we obtain a proof of (b). □

The following theorem states that two paths of a goal appearing in a cycle cannot be strengthened as asymptotically equal.

Theorem 6.6 (Circulation Theorem) Let $\boldsymbol{P}$ be a program and H be a cycle. For any $\langle u' \rangle p' \in L(H)$, $\langle u'' \rangle p'' \in R(H)$, and $m \in M(\mathcal{M}(\boldsymbol{P}))$, we have $I(u')(m/p') \not\approx I(u'')(m/p'')$.

Proof. Let $H = \langle \langle u_{i-1} \rangle q_{i-1} = \langle \overline{u_i} \rangle p_i ; r_i \rangle_{i=1}^{n}$ and $\langle u_n \rangle p_n r_n \equiv \langle u_0 \rangle q_0 r_1 q$. Let $m \in M(\mathcal{M}(\boldsymbol{P}))$. By Proposition 6.1, we have for $i \in 1..n$

$$I(u_{i-1})(m/q_{i-1}r_i) \leftrightarrow I(\overline{u_i})(m/p_i r_i) \tag{6.1}$$

and for $i \in 2..n$ we have $I(u_{i-1})(m/p_{i-1}r_{i-1}) \leftrightarrow I(\overline{u_i})(m/p_i r_i)$ since $p_{i-1}r_{i-1} = q_{i-1}r_i$. Moreover, since $\langle u_n \rangle p_n r_n \equiv \langle u_0 \rangle q_0 r_1 q$ we have $I(u_n)(m/p_n r_n) \leftrightarrow I(\overline{u_1})(m/p_1 r_1 q)$.

Now, choose arbitrarily $X'p' \in L(H)$ and $X''p'' \in R(H)$. There exist j and k such that $X'p' \equiv \langle u_{j-1} \rangle q_{j-1}$ and $X''p'' \equiv \langle \overline{u_k} \rangle p_k$. Let $m' = I(u_{j-1})(m)$, $m'' = I(\overline{u_k})(m)$, $q' = r_j$, and $q'' = r_k$. For any $i \in 1..n$, we have

$$I(u_i)(m/p_i r_i) \leftrightarrow \overline{m'/p'q'q}. \tag{6.2}$$

Let $r \in Path$.

If $\forall i \in j..n(I(u_i)(m)(p_i r_i r) = in)$ and $\forall i \in 1..k(I(u_i)(m)(p_i r_i q r) = in)$, then we have $in = m'(p'q'r) \neq m''(p''q''qr) = out$. If there exists some $i \in j..n$ such that $I(u_i)(m)(p_i r_i r) = out$, then from (6.2) we have $\overline{m'/p'q'q}(r) = in$, that is, $I(u_{j-1})(m)(q_{j-1} r_j q r) = out$. From (6.1), we have $I(\overline{u_j})(m)(p_j r_j q r) = in$, that is, $I(u_j)(m)(p_j r_j q r) = out$. Thus there exists some $i \in 1..n$ s.t. $I(u_i)(m)(p_i r_i q r) = out$. In such a case, from (6.2), we have $\overline{m'/p'q'q}(qr) = in$, that is, $m'(p'q'qqr) = out$. From $m'(p'q') \leftrightarrow m''(p''q''q)$, we have $m''(p''q''qqqr) = in \neq m'(p'q'qqr)$.

Thus, for every $r \in Path$, we have $m'(p'q'r) \neq m''(p''q''qr)$ or $m'(p'q'qqr) \neq m''(p''q''qqqr)$.

Assume, hypothetically, that $m'/p' \simeq m''/p''$. Then, for the path $p'q'$, there exist $u \in \{IN, OUT\}$ and a path s such that $m'/p'q's = u$. Then, for the path $p''q''qs$, there exists a path s' such that $m''/p''q''qss' = u$. Then, for $p'q'qss'$, there is s'' such that $m'/p'q'qss's'' = u$. Then, for $p''q''qqqss's''$, there is s''' such that $m''/p''q''qqqss's''s''' = u$. Let $r = ss's''s'''$. We have $m'/p'q'r = m''/p''q''qr$ and $m'/p'q'qqr = m''/p''q''qqqr$, which contradicts the previous paragraph. □

The following theorem states that we can reduce the proof of fingen of an initial graph T to that of a subset of T.

Theorem 6.7 (Graph Reduction Theorem) Let $\boldsymbol{P}$ be a program and T be a subset of some initial graph. For every $w \in Nodes(T)$ we have:

$$(\exists m \in M(\mathcal{M}(\boldsymbol{P}))\, \forall p, p' \in T^{\langle w \rangle}(m/p \simeq m/p')\,) \\ \Rightarrow\ (\,\mathrm{fingen}(\exists_{\langle w \rangle}(T)) \Rightarrow \mathrm{fingen}(T)\,).$$

Proof. Let $m \in M(\mathcal{M}(\boldsymbol{P}))$ and assume that $\forall p, p' \in T^{\langle w \rangle}(m/p \simeq m/p')$.

Let H be a cycle and that $E(H) \cap T \neq \{\}$. It is sufficient for us to prove that $\exists_{\langle w \rangle}(T) = T$, for it gives the contrapositive proof of $\mathrm{fingen}(\exists_{\langle w \rangle}(T)) \Rightarrow \mathrm{fingen}(T)$.

Assume, hypothetically, that there exists some $p \in T^{\langle w \rangle}$. We have $\langle w \rangle p \in R(H) \cup L(H)$. By Proposition 6.2, there exist i, j such that $\langle w \rangle p_i \in L(H)$ and $\langle w \rangle p_j \in R(H)$. By Theorem 6.6, we have $m/p_i \not\simeq m/p_j$, which contradicts $\forall p, p' \in T^{\langle w \rangle}(m/p \simeq m/p')$. Hence, we have $T^{\langle w \rangle} = \{\}$, and $\exists_{\langle w \rangle}(T) = T$. □

Finally, we can show the main theorem that states the correctness of our algorithm.

```
Variable: a boolean variable modular;
Input: a finite program P;
 E ← M(P);
 if E is inconsistent return 'Analysis failed';
 for each c ∈ P do
   T ← T(ε0, c);
   repeat
     W ← Nodes(T);
     for each w ∈ W do
       E' ← {m/p ≃ m/p' | p, p' ∈ T^⟨w⟩};
       if E ∪ E' is consistent then
         if (modular) E ← E ∪ E';
         T ← ∃⟨w⟩(T)
       end if
     end for
   until W = Nodes(T);
   if W ≠ {} return 'Analysis failed'
 end for;
 return 'P is NSTO'
```

Figure 6.2: Our occurs-check algorithm

Theorem 6.8 (Main Theorem) Let $\boldsymbol{P}$ be a finite program. $\boldsymbol{P}$ is NSTO if the algorithm in Figure 6.2 returns 'NSTO'.

Proof. Note that every success of the consistency check in the innermost loop ensures that there exists some moding function that satisfies the mode constraints $E \cup E'$ and consequently also one of its subsets $\mathcal{M}(\boldsymbol{P}) \cup E'$. By Theorem 6.7, we have $\mathrm{fingen}(\exists\langle w\rangle(T)) \Rightarrow \mathrm{fingen}(T)$. Thus, if 'NSTO' is returned, for each clause c, we have $\mathrm{fingen}(\mathcal{T}(\varepsilon_0, c))$ since $\mathrm{fingen}(\{\})$. In fact, by the same reason, we have $\mathrm{fingen}(\mathcal{T}(w, c))$ for all w and c. By Theorem 6.5 we know that $\boldsymbol{P}$ is NSTO. □

In the algorithm, *modular* is a boolean variable specifying the modularity of the analysis. When it is set to false, the variable E does not change after initialization. When it is set to true and 'NSTO' is returned, E contains the mode constraints imposed by the program $\boldsymbol{P}$ together with extra, asymptotic

equality constraints artificially introduced in order that the algorithm can finish the analysis of $\boldsymbol{P}$. We shall call such a pair $(\boldsymbol{P}, E)$ as a modular result. Every modular result $(\boldsymbol{P}, E)$ satisfies $\mathcal{M}(\boldsymbol{P}) \subseteq E$. Modular results satisfy the following property:

Proposition 6.3 (Modular Analysis) Let $(\boldsymbol{P}_1, E_1)$ and $(\boldsymbol{P}_2, E_2)$ be modular results. If $E_1 \cup E_2$ is consistent, then $(\boldsymbol{P}_1 \cup \boldsymbol{P}_2, E_1 \cup E_2)$ is also a modular result.

Proof. We can apply the occurs-check algorithm to $\boldsymbol{P}_1 \cup \boldsymbol{P}_2$ in the exactly same order as prescribed by $(\boldsymbol{P}_1, E_1)$ and then by $(\boldsymbol{P}_2, E_2)$. The consistency check of $E \cup E'$ in the innermost loop always succeeds because we have $E \cup E' \subseteq E_1 \cup E_2$. This inclusion is obvious since we have $E = E_1 \cup \mathcal{M}(\boldsymbol{P}_2)$ at the end of visiting $\boldsymbol{P}_1$. □

6.5 Example of the Analysis

In this section, we demonstrate how to perform our occurs-check algorithm toward Guarded Horn Clauses (GHC) programs, taking as an example the GHC program in Figure 6.1. This program contains bidirectional communication between goals. To the knowledge of the author, there has been no algorithm that can analyze this program.

GHC is essentially the same as the concurrent logic programming language defined in Chapter 2. The left-hand side of '|' in a clause specifies the *guard* of that clause, expressing the condition for the clause to be used in a reduction.

Logically, guard goals and body goals have the same meaning. However, we can exploit some characteristics of the guard to enhance the precision of the analysis. Specifically, the variables in a clause known to be ground when the reduction takes place can be statically eliminated from the clause, by means of replacing those variables by ground terms, for those variables have nothing to do with infinite structures. We will see this in the example.

The analysis proceeds as follows:

1. *Preprocess*: Translate the input program into the abstract language defined in Section 6.2.3. We must make the program linear headed.

2. *Building Initial Graphs*: For each clause, build the list of the variables that occur more than once. In addition, we compute for each variable in the list, a list of labeled paths at which the variable occurs.

3. *Mode Inference*: Collect the mode constraints imposed by the program, consulting Definition 6.21.

4. *Graph Reduction*: Apply the occurs-check algorithm in Figure 6.2 to the initial graphs and the mode constraints computed above.

In the following, we will explain each of these steps in a detail.

6.5.1 Preprocessing

We will translate source programs into our abstract language defined in Section 6.2.3. Let us assume that the source program is written in a concurrent logic programming language where the guard of a clause consists of guard goals. A guard goal in a clause is said to be *read only* if it does not bind terms to any variables occurring in the head of that clause.

First of all, we introduce a clause ($\texttt{main} \leftarrow B$) if there is a query B to be executed. Then, each clause is translated into the abstract language as follows:

1. Every read-only guard goal $X = t$ where X does not occur in t is statically executed in the clause. Specifically, the goal is removed and every X in the clause is replaced with t.

2. Every read-only guard goal that confirms some variables have been instantiated to particular *ground* term tuples, such as $\texttt{int}(X)$ and numeric comparison, is expanded in the clause. In general, expanding a guard goal requires the use of metavariables for specifying the set of the tuples of terms that together can replace the variables occurring in the goal. Examples of guard expansion can be found in Figure 6.3.

   While guard expansion may generate unbounded number of clauses, they all have the same alias information and hence the same initial graph. Thus, the algorithm can handle those clauses at once by simply ignoring those variables.

3. Every remaining guard goal is converted into an appropriate body goal.

```
main ← print(Hs0), Hs0 =0 [1|Hs1],
        mul(2,Hs0,L2), mul(3,Hs0,L3), mul(5,Hs0,L5),
        merge(L2,L3,L23), merge(L23,L5,Hs1).        % main-1
merge([a|As1],[b|Bs1],Ms) ←
        Ms =1 [a|Ms1], merge(As1,[b|Bs1],Ms1).
        (for all a, b such that a < b)              % merge-1
merge([a|As1],[b|Bs1],Ms) ←
        Ms =2 [b|Ms1], merge([a|As1],Bs1,Ms1).
        (for all a, b such that a > b)              % merge-2
merge([a|As1],[a|Bs1],Ms) ←
        Ms =3 [a|Ms1], merge(As1,Bs1,Ms1).
        (for all number a)                          % merge-3
mul(N,[A|As1],Ms) ←
        Ms =4 [M|Ms1], M :=5 N * A, mul(N,As1,Ms1).  % mul-1
```

Figure 6.3: Program translated from Figure 6.1

4. Every multiple occurrence of a variable in the head is renamed into a fresh variable, and a body goal that performs passive unification between them is introduced to the clause. In short, we assume that multiple head occurrences of a variable always express passive unification. This assumption makes sense if the program is cooperatively moded.

5. For each body goal calling a built-in predicate, including a unification goal, the top-level function symbol is subscripted by its own integer number, thus introducing mode polymorphism to built-in predicates. This subscripting is essential for gaining the precision of the analysis; for otherwise the left-hand side of all the unification goals must have the same moding function.

Example. The GHC program in Figure 6.1 is translated into our language as in Figure 6.3. In the translation process, we make use of the property of guard numeric comparison goals that they reduce only if both the arguments have been instantiated to numbers.

```
main-1::  Hs0  1:<print,1>, 2:<u0,1>, 3:<mul,2>,
               4:<mul,2>, 5:<mul,2>
          Hs1  2:<u0,2><.,2>, 7:<merge,3>
          L2   3:<mul,3>, 6:<merge,1>
          L3   4:<mul,3>, 6:<merge,2>
          L5   5:<mul,3>, 7:<merge,2>
          L23  6:<merge,3>, 7:<merge,1>
merge-1:: Ms1  1:<u1,2><.,2>, 2:<merge,3>
merge-2:: Ms1  1:<u2,2><.,2>, 2:<merge,3>
merge-3:: Ms1  1:<u3,2><.,2>, 2:<merge,3>
mul-1::   M    1:<u4,2><.,1>, 2:<is5,1>
          Ms1  1:<u4,2><.,2>, 3:<mul,3>
          N    2:<is5,2><*,1>, 3:<mul,1>
```

Figure 6.4: Initial graphs of Figure 6.3

6.5.2 Generating Initial Graphs

We will explain how to represent equation sets in the implementation. A close look at the algorithm reveals that any subset of the initial graph of a clause $c = (h \leftarrow b_1, \ldots, b_n)$ that takes place in the algorithm is always of the form $\exists\langle i_k\rangle \ldots \exists\langle i_1\rangle \mathcal{T}(\varepsilon_0, c)$. For each variable X, let us denote by $Z(X)$ the set of the labeled paths at which X occurs, that is, let $Z(X) = \{i{:}p \mid \overrightarrow{p}(b_i) \equiv X\}$. Then, every equation set of the form $\exists\langle i_k\rangle \ldots \exists\langle i_1\rangle \mathcal{T}(\varepsilon_0, c)$ can be represented by a function N that maps each variable X to a subset of $Z(X)$ as follows: $\{\langle i\rangle p = \langle j\rangle q \mid i{:}p \in N(X),\, j{:}q \in N(X)\} \setminus ID$. Thus, we can represent each equation set that appears in the algorithm by $\{\langle X, N(X)\rangle \mid N(X) \neq \{\}\}$.

We assume that initial graphs for built-in predicates are empty and that the mode constraints imposed by built-in predicates are known prior to the analysis. For instance, we know that the built-in predicate `:=` that computes an integer expression imposes no constraints.

Example. Figure 6.4 shows the initial graphs of the program in Figure 6.3. Observe how the set $\{\langle X, N(X)\rangle \mid N(X) \neq \{\}\}$ is written in the text representation. Each pair $\langle X, N(X)\rangle$ is written as X followed by a list of the elements $i{:}p$ contained in $N(X)$. In the text representation, we have abbreviated f/n to f, $=_k/2$ to `u`k, and $:=_5/2$ to `is5`.

6.5.3 Mode Inference

Mode inference consists of two phases: (1) generate mode constraints, and (2) check their consistency. Generation is straightforward from the definition. In what follows, we may abbreviate m/p to p, and $\overline{s}$ to $\text{-}s$.

The consistency check can be performed as constraint simplification. The reader can see that the generated constraint can be described as a set of equality constraints of the forms of $p = q$, $p = \text{-}q$, $p = OUT$ or $p = IN$, convergence constraints of the forms of $p \rightarrow u$ or $p \rightarrow \text{-}u$, or $\mathcal{R}$-constraints where p and q are paths and u is a variable ranged over uniform moding functions satisfying $u = IN \vee u = OUT$.

Clauses with more than two occurrences of a variable impose non-binary $\mathcal{R}$-constraints. Most $\mathcal{R}$-constraints can be iteratively simplified into equality constraints, using for instance the following properties:

- $\mathcal{R}(\{\!|\, s, s' \,|\!\}) \Leftrightarrow s = \overline{s'}$
- $\mathcal{R}(\{\!|\, OUT \,|\!\} + S) \Leftrightarrow \forall m \in S(m = IN)$
- $\mathcal{R}(\{\!|\, IN \,|\!\} + S) \Leftrightarrow \mathcal{R}(S)$.

However, in general, there is a case where some $\mathcal{R}$-constraints remain and they cannot be simplified any more. In this case, the consistency check of the constraints becomes expensive. The simplest solution for overcoming this is to strengthen the remaining $\mathcal{R}$-constraints so that they are removed. For example, assume we have $\mathcal{R}(\{\!|\, m/p \,|\!\} + S)$. If we by some reason know that $m(p) = out$, we can strengthen with the constraint $m/p = OUT$ so that this $\mathcal{R}$-constraint is removed. This solution is sound because the consistency of the strengthened constraints ensures that of the original constraints. When all the $\mathcal{R}$-constraints are successfully removed, the mode information in the program can be represented in the form of a feature graph. Readers are referred to the paper [34] for more information.

When every constraint is either an equality constraint or a convergence constraint, the consistency check is easy. This is because, in this case, the check amounts to remembering for each path p constrained with convergence constraints a single variable u ranging over uniform moding functions such that $m/p \rightarrow u$.

```
IN = <mul,2>
IN = <is5,2><*,2>
<merge,1> =  <merge,1><.,2>
<merge,1> =  <merge,2>
<merge,1> = -<merge,3>
<merge,1> = -<mul,3>
<merge,1> = -<u0,1><.,2>
<merge,1> = -<u1,1>
<merge,1> =  <u1,2>
<merge,1> = -<u2,1>
<merge,1> =  <u2,2>
<merge,1> = -<u3,1>
<merge,1> =  <u3,2>
<merge,1> = -<u4,1>
<merge,1> =  <u4,2>
<merge,1><.,1> = -<is5,1>
<merge,1><.,1> = -<u5,1>
<merge,1><.,1> =  <u5,2>
<mul,1> = <is5,2><*,1>
<print,1> = -<u0,1>
<print,1> =  <u0,2>
```

Figure 6.5: Mode constraints imposed by Figure 6.3

Example. Figure 6.5 shows the result of simplifying the mode constraints imposed by Figure 6.3. Fortunately, all the $\mathcal{R}$-constraints have been reduced into equality constraints and hence we can easily confirm the consistency. □

6.5.4 Applying the Algorithm

We apply the algorithm in Figure 6.2 to each initial graph of a clause and confirm that every variable $\langle i \rangle$ in the graph, which corresponds to the i-th body goal of that clause, can be removed.

Example. First, we arbitrarily choose the clause `merge-1`. The initial graph $\{\langle 1\rangle\langle =_1/2, 2\rangle\langle ./2, 2\rangle = \langle 2\rangle\langle \texttt{merge}/3, 3\rangle, \langle 2\rangle\langle \texttt{merge}/3, 3\rangle = \langle 1\rangle\langle =_1/2, 2\rangle\langle ./2, 2\rangle\}$ has the text representation `Ms1 1:<u1,2><.,2>, 2:<merge,3>`. Of the two goals in the graph, namely `1:Ms=`$_1$`[`a`|Ms1]` and `2:merge(As1,[`b`|Bs1],Ms1)`, let us arbitrarily choose the goal `2`. The goal happens to have only one vari-

able occurrence in the graph, that is, only `Ms1` occurs at `<merge,3>`. We want to prove that `<merge,3>` $\simeq$ `<merge,3>` is consistent with E. To this end, we introduce a uniform moding function `U1` $\in \{IN, OUT\}$ and test the consistency of E strengthened with `<merge,3>` $\rightarrow$ `U1`. We know that the strengthened E entails `<merge,1>` $\rightarrow$ `-U1` and is still consistent. Hence, the goal `2` is removed. This removal lets us remove the variable `Ms1` from the text representation of the graph since it has now only one occurrence, which means that we have no equations from `Ms1` in the graph. Since the removal of `Ms1` makes the graph empty, the analysis for `merge-1` is finished.

Assuming that the variable *modular* is set to true, E is now strengthened with the constraint `<merge,1>` $\rightarrow$ `-U1`.

Next, we choose the clauses `merge-2` and `merge-3`. As you can see, the current E is strong enough to remove the goal `2` of the graph of each of these clauses, because E already entails that `<merge,3>` $\rightarrow$ `U1`. Therefore, the analysis for these clauses is finished.

Next, let us choose the clause `mul-1`. Using Figure 6.5, you can easily confirm that we already have `<u4,1>` $\rightarrow$ `U1`, which entails `<u4,1><.,1>` $\rightarrow$ `U1` and `<u4,1><.,2>` $\rightarrow$ `U1`. Therefore, the goal `1` is removed from the graph. This removal changes the variables `M` and `Ms1` into singletons and hence we can remove them, leaving solely `N` in the graph. Then, introducing a `U2` $\in \{IN, OUT\}$, we can strengthen E with `<is5,2><*,1>` $\rightarrow$ `U2` without causing inconsistency and the goal `2` is removed. Thus, the analysis for `mul-1` is finished.

Finally, we tackle the clause `main-1`. Choose the goal `1`, introduce a `U3` $\in \{IN, OUT\}$ and try `<print,1>` $\rightarrow$ `U3`. Letting `U3 = -U1`, it is consistent with `<u0,1><.,2>` $\rightarrow$ `U1`, and the goal `1` is removed. Next, choose the goal `3`, introduce a `U4` $\in \{IN, OUT\}$, and try `<mul,2>` $\rightarrow$ `U4` and `<mul,3>` $\rightarrow$ `U4`. Since `<mul,2> = IN`, we have `U4 = IN`. Since `<mul,3>` $\rightarrow$ `U1`, we have `U1 = IN`. Now, we can check the consistency, and the goal `3` is removed. Then, the variable `L2` is removed because it is now a singleton. Since we already have `<mul,3>` $\rightarrow$ `IN`, the goal `4` and `L3`, the goal `5` and `L5`, the goal `6` and `L23`, the goal `7` and `Hs1`, and then the goal `2` can all be removed in this order. Thus, the analysis for `main-1` is finished.

These ensure that the program is NSTO.

At the end of the analysis, the mode constraints `<merge,3>` $\rightarrow$ `IN` and `<mul,1>` $\rightarrow$ `IN` and `<mul,3>` $\rightarrow$ `IN` have been added to E. The strengthened set of mode constraints has sufficient information to perform modular analysis. □

6.6 Detection Power of our Analysis

Input programs that can be handled by our analysis are cooperatively moded concurrent logic programs. These include programs with bidirectional communication between goals, a typical example of which is the one in Figure 6.1. Our algorithm can also detect the NSTO property of some programs with bidirectional communication that use only one program variable between goals.

Our algorithm may fail for some programs. In particular, our analysis has inherent limitations in detecting the NSTO property of programs having a clause such that variables that connect between body goals have aliases.

For example, consider the following program:

```
main(Xs,Xs0) :- Xs =1 [accept(As,As0)|Xs0], echo(As,As0).
echo(Es,Es0) :- Es =2 [hello|Es0].
```

The goal `main(Xs,Xs0)` sends to the difference list `Xs` $\setminus$ `Xs0` an `accept` message. This difference list is sent by the first unification goal. The `accept` message contains another difference list `As` $\setminus$ `As0` to which `echo` sends a message of `hello`. Here, the goal `echo(As,As0)` contains an alias since `As0` is returned as a part of `As`. This is formalized by the constraint $m/\langle \texttt{echo}/2, 2\rangle = \overline{m/\langle \texttt{echo}/2, 1\rangle\langle \texttt{./}2, 2\rangle}$, imposed due to (BU) applied to $=_2$.

Now, our algorithm attempts to reduce the initial graph of the first clause, which is $\{\langle 2\rangle\langle \texttt{echo}/2, k\rangle = \langle 1\rangle\langle =_1/2, 2\rangle\langle \texttt{./}2, 1\rangle\langle \texttt{accept}/2, k\rangle \mid k = 1, 2\}\star$, to the empty graph. To do this, we need to coerce `As` and `As0` into being asymptotically equal, by strengthening either with $m/\langle \texttt{echo}/2, 1\rangle \simeq m/\langle \texttt{echo}/2, 2\rangle$ or with $m/\langle =_1/2, 2\rangle\langle \texttt{./}2, 1\rangle\langle \texttt{accept}/2, 1\rangle \simeq m/\langle =_1/2, 2\rangle\langle \texttt{./}2, 1\rangle\langle \texttt{accept}/2, 2\rangle$. However, such strengthening is not consistent with the mode information of the program and therefore the analysis fails.

6.7 Conclusion

We have proposed a mode-based occurs-check algorithm that can handle programs with bidirectional communication. The algorithm iteratively removes body goals from each initial graph that represents a clause, whose correctness relies on the strengthening of mode constraints by asymptotic equality constraints. The notion of asymptotic equality is devised by the author in order to formulate a sufficient condition of the NSTO property in the analysis.

The restriction on the input programs is that the clauses must have linear heads and that the program is cooperatively moded. Although the transformation into linear-headed programs cannot be automated for general logic programs, we have shown how to automate this for cooperatively moded programs.

The main contribution of this work is that our algorithm can prove the NSTO property of a program with bidirectional communication between goals, which cannot be handled by existing analysis.

Chapter 7

Assigning Types to LMNtal Language

7.1 Introduction

Graph rewriting provides a framework for modeling various computation as concurrent transformation of graph structures [1]. It is known that graph structures and rewrite rules for them have mathematical foundations that facilitate the analysis of the described computation. By putting restrictions on the classes of graph structures and rewrite rules, the analysis of the computation can be formulated elegantly. One of the early examples of such attempts is Interaction Nets [24]. As a framework for modeling computation, the computation model should be as simple as possible. On the other hand, graph rewriting can also serve as a model of programming languages. Since programs should be written succinctly, the languages must devise some convenient syntax. For example, attaching a name and attributes to a graph node is particularly useful for practical programming. Such extensions can also be useful for optimizing the implementation of the language system. However, extending the syntax can complicate the formalization of languages.

Strong type systems provide a bridge between the simplicity of the language and its practical usefulness. There is various work on providing strong type systems for particular practical language systems based on graph rewriting, for example AGG [27]. However, there has been little work on how to offer *relatively simple* graph rewriting as a practical programming language. By relatively simple, we mean that processes and data are treated in a unified

way. In our terminology, most of practical graph rewriting languages with implementation, including AGG, are not relatively simple since they allow attribute values to be attached to a graph node. The setting of relatively simple graph rewriting greatly simplifies the formalization of the language.

In this chapter, we take *LMNtal* (pronounced "*elemental*") [38] as an instance of a relatively simple graph rewriting language, and introduce a strong type system into LMNtal. In LMNtal, graph nodes called atoms are connected together with links. The type system is introduced by firstly classifying atoms into active atoms (seen as procedures) and the others (seen as data) and then by formalizing what kind of atoms can be connected to active atoms. We will explain how our type system can be used for describing several static properties of a program as well as for optimizing implementation.

The rest of this chapter is organized as follows. Section 7.2 defines an LMNtal language, and Section 7.3 shows a program example and objectives of our type system. Section 7.4 and 7.5 introduce the type system, and Section 7.6 shows examples of type inference. Section 7.7 explains some theoretical results including type safety, and Section 7.8 gives concluding remarks, discussing related work.

7.2 The Language

In this section, we define a relatively simple graph rewriting language, based on LMNtal. LMNtal is a hierarchical graph rewriting language where graph nodes called atoms are connected together with one-to-one links. LMNtal provides a feature called membranes that allows us to group atoms in a graph structure hierarchically. The language we will define in this section differs from the original one advocated in the paper [38] in that ours is simplified with respect to 'process contexts' and extended with membrane names.

7.2.1 Syntax

The syntax of our LMNtal language is defined in Figure 7.1. p is an *atom name*, m is a *membrane name*, and X is a *link name* (or a link in short). We assume that these names are taken from disjoint sets. We have two syntactic classes in the language: processes P and process templates T.

A *process* represents a graph structure that rewrites itself. $\mathbf{0}$ is an empty process; $p(X_1, \ldots, X_n)$ is an *atom*; P, P is parallel composition of processes;

$$
\begin{array}{rcl}
P & ::= & \mathbf{0} \mid p(X_1, \ldots, X_n) \mid P, P \mid m\{P\} \mid (T \ \texttt{:-}\ T) \\
T & ::= & \mathbf{0} \mid p(X_1, \ldots, X_n) \mid T, T \mid m\{T\} \mid (T \ \texttt{:-}\ T) \mid \texttt{@}p \mid \texttt{\$}p
\end{array}
$$

Figure 7.1: Syntax of LMNtal

$m\{P\}$ is a process grouped by a *membrane* { }; and $(T$:- $T)$ is a rewrite rule for processes that are located in the same membrane as this rule. Each argument of an atom is called as a *port*. An atom of the form X = Y, called a *connector*, is used for connecting the other occurrences of the links X and Y together.

A *process template* is a component of a rule. Intuitively, it represents the processes to be rewritten. A *rule context* @p matches sequences of rules, while a *process context* \$$p$ matches sequences of atoms and membranes.

In what follows, we will always consider a process as a process template to simplify the formulation.

The part of a process template not included in any rule is called the *non-rule part* of it. The formal operational semantics will be given later.

Syntactic Conditions.

P and T must enjoy the following syntactic conditions:

(L1) (Link condition) No links occur more than twice in the non-rule part of a process template.

(L2) (Link condition for rules) Each link occurring in a rule occurs exactly twice in that rule.

(L3) Every process context occurring in a rule occurs exactly twice in that rule.

(L4) Every process context and rule context occurring in a rule occurs exactly once in the left-hand side of that rule and does not occur in other rules inside the rule.

(L5) All process or rule contexts of the same name always occur at membranes of the same name.

(L1) is one of the characteristic conditions of LMNtal. Link condition enables us to find a linked atom quickly. (L2) and (L3) are to preserve (L1) over rewritings. (L4) ensures that process contexts and rule contexts provide pattern matching in a rewriting.

A link occurring exactly once in the non-rule part of a process template T is called a *free* link of T. We denote by $\mathrm{fv}(T)$ the set of free links of T. A link occurring in a process template T is called a *local* link of T if it is not a free link.

An LMNtal program is written as a process. It is convenient to consider that the process representing the entire program is located in some virtual membrane. Let us denote the name of that membrane by `root`.

In this work, to make the formulation simple, we will assume that the arguments of an atom are all distinct. It can easily be checked that this limitation does not make any essential difference to the language. This is because every local link of an atom can be expressed as two free links together with a connector atom between them. For example, the atom `p(X,X)` can be expressed as the process `p(X,Y), X=Y`. The equivalence between these two processes can be stated in terms of the operational semantics defined later.

Notes on Membrane Names.

Only membrane names and its related syntactic condition (L5) are the extension to the original LMNtal language. We have learned from the experience of writing many LMNtal programs that each membrane that appears in a program has its statically determined role, and found it useful to give names to such roles by introducing membrane names. Membranes with different names can have different implementation with respect to data representation of their contents. Assigning names to membranes does not affect the language power since $m\{P\}$ is essentially nothing more than $\texttt{anonymous}\{\texttt{name}(X), m(X), P\}$ where `name` is assumed not to be used for other purposes.

In some occasions, (L5) requires too many membranes to have the same name. For example, consider a program that uses several membranes in order to describe sequential execution of tasks. Each task is executed in a separate membrane and the result is sent to another membrane. Then, if the result is transferred using process contexts, these membranes must have the same name, which can be a cause of lower precision of analysis. We anticipate that the subtyping of membranes can address this issue, but it is beyond the scope of this work.

(E1)	$\mathbf{0}, P$	$\equiv$	P	
(E2)	P, Q	$\equiv$	Q, P	
(E3)	$P, (Q, R)$	$\equiv$	$(P, Q), R$	
(E4)	P	$\equiv$	$P[Y/X]$	if X is a local link of P
(E5)	$P \equiv P'$	$\Rightarrow$	$P, Q \equiv P', Q$	
(E6)	$P \equiv P'$	$\Rightarrow$	$m\{P\} \equiv m\{P'\}$	
(E7)	$X = Y,\ Y = X$	$\equiv$	$\mathbf{0}$	
(E8)	$X = Y$	$\equiv$	$Y = X$	
(E9)	$X = Y,\ P$	$\equiv$	$P[Y/X]$	if P is an atom and X occurs in P
(E10)	$m\{X = Y, P\}$	$\equiv$	$X = Y,\ m\{P\}$	if exactly one of X and Y is a free link of P

Figure 7.2: Structural congruence on LMNtal processes

7.2.2 Structural Congruence

In order to formulate the operational semantics, we first define the structural congruence ($\equiv$) and then the reduction relation ($\longrightarrow$).

We define the relation $\equiv$ on processes as the minimal equivalence relation satisfying the rules shown in Figure 7.2. Two processes related by $\equiv$ are essentially the same and are convertible to each other in zero steps. Here, $[Y/X]$ is a *link substitution* that replaces X with Y.

(E1)–(E3) are the characterization of processes as multisets. (E4) allows the renaming (α-conversion) of local links. Note that the link Y cannot occur free in P for the link condition on $P[Y/X]$ to hold. (E5)–(E6) are structural rules that make $\equiv$ a congruence. (E7)–(E9) are concerned with `=`. (E7) says that a self-absorbed loop is equivalent to $\mathbf{0}$, while (E8) expresses the symmetry of `=`. (E9) is an absorption law of `=`, which says that a connector can be absorbed by another atom, which can again be a connector. Because of the symmetry of $\equiv$, (E9) says that an atom can emit a connector as well.

7.2.3 Reduction Relation

Computation proceeds by rewriting processes using rules collocated in the same "place" of the nested membrane structure.

We define the reduction relation $\longrightarrow$ on processes as the minimal relation

$$\text{(R1)}\ \frac{P \longrightarrow P'}{P, Q \longrightarrow P', Q} \qquad \text{(R2)}\ \frac{P \longrightarrow P'}{m\{P\} \longrightarrow m\{P'\}}$$

$$\text{(R3)}\ \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

(R4) $m\{X = Y, P\} \longrightarrow X = Y,\ m\{P\}$
if X and Y are distinct and do not occur in P

(R5) $X = Y,\ m\{P\} \longrightarrow m\{X = Y, P\}$
if X and Y occur in the non-rule part of P

(R6) $T\theta, (T :\!\!- U) \longrightarrow U\theta, (T :\!\!- U)$

Figure 7.3: Reduction relation on LMNtal processes

satisfying the rules in Figure 7.3. Note that the right-hand side of $\longrightarrow$ must observe the link condition of processes.

Of the six rules, (R1)–(R3) are structural rules. (R4) and (R5) deal with the interaction between connectors and membranes. (R4) says that, when a connector in a membrane connects two links coming from outside, the membrane can expel the connector. (R5) says that, when a connector connects two links coming from the same membrane, the connector itself can go into that membrane.

(R6) is the key rule of LMNtal. The substitution θ is to represent what has been received by each process context and rule context. The whole resulting process, namely $U, (T :\!\!- U)$ and its surrounding context, should observe the link condition, but this can always be achieved by α-converting $T :\!\!- U$ before use so that the new local links don't cause name clashes with the context. The substitution θ in (R6) is represented as a finite set of *substitution elements* of the form β_i/α_i (meaning that α_i is replaced by β_i), and should map every `@`p in T to a sequence of rules and every `$`p in T to a sequence of atoms and membrane processes.

7.3 A Motivating Example

7.3.1 Concatenating Lists

In LMNtal, the skeleton of a list can be represented, using element processes `c`(ons) and a terminal process `n`(il), as $\texttt{c}(A_1, X_1, X_0)$, ..., $\texttt{c}(A_n, X_n, X_{n-1})$, $\texttt{n}(X_n)$. Here, A_i is the link to the i-th element from the skeleton, and X_0 is the link to the whole list from somebody else. This corresponds to a list formed by the constraints $X_0 = \texttt{c}(A_1, X_1)$, ..., $X_{n-1} = \texttt{c}(A_n, X_n)$, $X_n = \texttt{n}$ in constraint logic programming languages, except that the LMNtal list is a resource rather than a value.

Two lists can be concatenated using the following two rules:

```
append(X0,Y,Z), c(A,X,X0) :- c(A,Z1,Z), append(X,Y,Z1)
append(X0,Y,Z), n(X0)     :- Z=Y
```

These rules can be applied to the following process:

```
append(X0,Y0,Z), res(Z),
c(A1,X1,X0),c(A2,X2,X1),c(A3,X3,X2),n(X3),1(A1),2(A2),3(A3),
c(B1,Y1,Y0),c(B2,Y2,Y1),n(Y2),            4(B1),5(B2)
```

Figure 7.4 shows a graphical representation of the program and its execution. Numbers in small letters are to clarify argument positions. In the final state, the concatenated list is connected to a unary atom `res`.

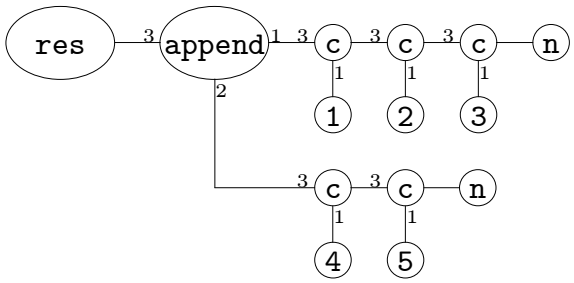
The above rules have clear correspondence with the `append` predicate written in a concurrent logic programming language:

```
append(X0,Y,Z) :- X0=c(A,X) | Z=c(A,Z1), append(X,Y,Z1).
append(X0,Y,Z) :- X0=n      | Z=Y.
```

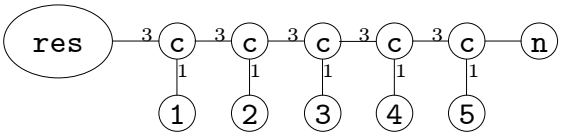
The difference is that LMNtal has eliminated syntactic distinction between processes (i.e., `append`), and data (i.e., list skeleton).

7.3.2 Objectives of Our Type System

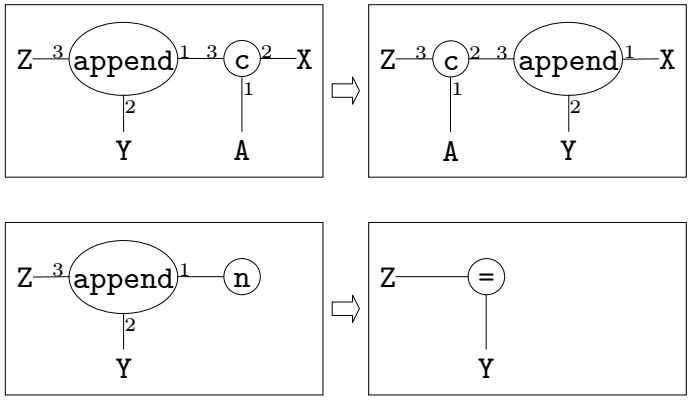
Consider the list concatenation program above. We know the argument of `res` can connect only to a list of integers. This kind of static type analysis has various applications. For instance, it justifies data representation optimization in the runtime system. Moreover, it allows us to describe and



(a) Initial state



(b) Final state



(c) Rewrite rules

Figure 7.4: List concatenation

understand program specification. It is not clear, however, how we can represent and analyze such a property systematically. Previous work could not handle such a property using strong typing. This is because we had no guidelines on how to neglect the `append` atom initially connected to `res` in the analysis.

Probably the most well-known method for typing rewriting graphs is to give a graph homomorphism from process graph structures to a type graph, which is used in the graph rewriting system AGG [27]. This method, however, does not allow the distinction between processes and data within the graph structure, and hence properties like the one mentioned above cannot be handled well.

Interaction Nets [24] provide a partial solution to this, though the language is much restricted. They provide a type system that can type the above example of list concatenation by requiring for each argument of an atom a *fixed* description of its datatype such as lists or integers given prior to the analysis. Hence, they are not suitable for analyzing arbitrarily complex process structures in the program, in the sense as explained in Section 7.6.3. Moreover, what they provide is essentially a type checking algorithm, rather than type inference as we will provide in this work.

In the following two sections, we propose a type system as a solution to the above issues. In our type system, a programmer only needs to classify atoms into active atoms and data atoms in order to obtain by inference a type graph that describes complex process structures in the program.

7.4 Formalization of Types

In this section, we formalize our type system for LMNtal processes that express which atoms can be connected to the ports of active atoms. Recall that a port means an argument of an atom. We first define paths that represent ports. Next, we define a polarized path as a path with an optional polarity inversion symbol. Then, we define types as constraints on the polarized paths.

7.4.1 Principles of our Type System

In this work, we will adopt the following principles to introduce a strong type system to LMNtal:

- Every active atom of the same name (and arity) located in a membrane of the same name has the same type.
- Every data atom of the same name (and arity) connected directly from a port of the same type has the same type.

Here, we say that atoms having the same name (and arity) are typed the same if their corresponding ports have the same types.

The former is justified when we consider an active atom as a procedure call in other languages since most procedures have fixed usage of their arguments. This principle means that active atoms are not treated as polymorphic.

The latter pertains to the treatment of complex type information, saying that the corresponding ports of two data atoms do not always have the same type just because those atoms have the same name. This principle is justified when we consider a data atom as a constructor of data structures in other typed languages since the declarations of most datatypes specify what their components are like. In LMNtal, the relationship of being a component is often described as a link between ports.

There are some circumstances where active atoms should not be regarded as procedures. This happens when active atoms are data structure constructors rewriting by themselves.

7.4.2 Specification of Activeness

We now formalize the activeness of atoms. Our type system requires a mapping, written as *out*, that specifies whether an atom is active and which port of the atom is the output port if it is not active (i.e., data).

Assumption 7.1 Let $\mathbf{N}$ denote the set of all positive integers and define $\mathit{Func} \stackrel{\text{def}}{=} \{p/n \mid p \text{ is an atom name and } n \geq 0 \text{ is an integer}\}$. We assume that a mapping $\mathit{out} : \mathit{Func} \to \mathbf{N} \cup \{\bot\}$ is given and $\mathit{out}(=/2) = \bot$. □

Definition 7.1 (Activeness) Assume a mapping *out* is given. If $u \in \mathbf{N}$, $\mathit{out}(p/n) = u$ means that atoms of name p with n arguments are considered as a *data atom* having the u-th argument as their *output port*. An atom that is neither a data atom nor a connector $=/2$ is said to be *active*. □

Example. We can assume the following setting for analyzing Figure 7.4:

$$\begin{array}{ll} out(\texttt{c}/3) = 3, & out(\texttt{n}/1) = 1, \\ out(\texttt{append}/3) = \bot, & out(\texttt{res}/1) = \bot. \end{array}$$

This specifies lists are taken as data while `append` and `res` are active. □

We assume that *out* is given by the programmer prior to type inference. Since there are many ways of giving the mapping *out*, the corresponding types inferred from a program vary according to the different settings of *out*. We will see this in Section 7.6.3.

There are some general criteria on what kind of atoms should be considered as active. Atoms representing procedures (e.g., `append`) are usually considered as active. On the other hand, atoms representing integer numbers should be data atoms in almost every occasion. This is because not doing so would compel all ports that carry integers to have the same type due to our type principle, which would result in a practically useless outcome. For the same reason, list constructors should usually be data atoms. Atoms reacting by themselves need to be active. Atoms representing messages between processes can be either active atoms or data atoms.

7.4.3 Traceability Condition

We introduce two syntactic conditions for programs: one is called as the active head condition for a rule in the program, and the other is the traceability condition for a process template in the program.

Definition 7.2 (Traceability #1) An atom P is *traceable from an active atom* if and only if:

1. P is an active atom, or
2. P is an atom connected directly to an atom traceable from an active atom. □

Links occurring in the both sides of a rule are not considered as connected.

Assumption 7.2 (Active Head Condition) Any atom occurring in the left-hand side of a rule must be traceable from an active atom. □

Intuitively, the active head condition states that data atoms do not rewrite themselves. For example, the rule `4(X) :- 5(X)` is prohibited when `4`/1 is data. It also excludes the rule `X=Y :- f(X,Y)`. The operational semantics ensures that the active head condition is preserved under reductions. We take the active head condition as mandatory.

Definition 7.3 (Traceability #2) An atom P occurring in the non-rule part of a process template T is *traceable from a free link* of T if and only if:

1. P contains a free link of T, or
2. P is an atom connected directly to an atom traceable from a free link of T. □

Assumption 7.3 (Traceability Condition) Any atom occurring in the non-rule part of a process template T in the program must be traceable from an active atom or a free link of T. □

These two conditions can always be satisfied if we consider more atoms active, though doing so can lower the usefulness of our analysis.

Example. Consider the rule `inc(X,Y) :- s(X,Y)`. Let `inc`/2 be active and `s`/2 data. The process `inc(A,B),inc(B,A),(inc(X,Y) :- s(X,Y))` satisfies both of the active head condition and the traceability condition. On the other hand, the process after reductions `s(A,B),s(B,A),(inc(X,Y) :- s(X,Y))` does not satisfy the traceability condition. □

A group of data atoms having all of their ports connected to themselves are called *garbage*. For example, the process `s(A,B),s(B,A)` is garbage when `s`/2 is a data atom. Garbage does not observe the traceability condition, which means that those data atoms are not captured by our typing scheme and are ignored. However, this does not pose a fundamental problem when every rule satisfies the active head condition, since in this case garbage can only be received by a process context and cannot be compared to something or decomposed into parts.

Note that the traceability condition is not preserved under reductions, as shown in the above example. Hence, we do not take the traceability condition as mandatory. Nonetheless, when the program is well typed in our type system, we can prove that unary data atoms preserve the traceability condition over reductions.

Garbage corresponds to vicious circles in Interaction Nets [24]. Their type system provides the notion of partitions between ports in order to statically ensure that no vicious circles, or garbage, can happen in the execution.

7.4.4 Applications of Our Type System

Several practical applications are known that make use of our type system. Here we outline some of them.

One of the simplest and most important is the space-efficient implementation of unary data atoms, such as integers and other symbolic values like `yes` or `no`. Since unary data atoms are guaranteed to preserve the traceability condition over reductions, every unary data atom is always linked from some atom other than a unary data atom. Hence, we can replace a pointer to a unary data atom by the content it points to, saving the memory space. This optimization of data representation is quite beneficial in implementing a graph rewriting language system in small memory environments such as embedded devices and the active head condition is essential for this purpose.

The other application is the acceleration of finding applicable rules in a membrane. Active atoms can be stored in a queue or a stack so that they can be used to drive the matching, as in some efficient implementations [5, 19] of symbolic concurrent languages. Since LMNtal has no distinction between processes and data and also allows nested membrane structures unlike these languages, more consideration is needed in order to achieve comparable efficiency.

As usual, types can be used for guaranteeing the type safety of the transferred data through foreign language interfaces.

7.4.5 Polarized Paths

Here, we define polarized paths, which are a large portion of *what is typed* in our type system. The rest of what is typed are membranes of whatever names, for which the names of submembranes and active atoms they can contain will be collected.

Definition 7.4 (Polarized Paths) A *polarized path* is an element of the

set *PPath* defined as follows:

$$\begin{aligned} \mathit{Root} &::= \langle r : p/n, i\rangle \mid X \\ \mathit{Path} &::= \mathit{Root} \mid \mathit{Path}\langle f/n, i\rangle \\ \mathit{PPath} &::= \mathit{Path} \mid -\mathit{Path} \end{aligned}$$

where r is a membrane name, X is a link name, p/n is an active atom, f/n is a data atom, and $i = 1, \ldots, n$. □

A polarized path consists of a path and an optional *polarity inversion symbol* $-$. A path corresponds to a port. To begin with, we explain the notion of a root, which is a special case of a path.

A *root* is either a port of an active atom, if it is of the form $\langle r : p/n, i\rangle$, or a link name X. The root $\langle r : p/n, i\rangle$ represents the i-th argument of an active atom p/n located in a membrane of the name r. On the other hand, the root that is a link name X represents the atom argument filled with the link X occurring in a process template where X is a free link of that process template.

A *path* represents a port of an atom satisfying the traceability condition. The construction of a path corresponds to the two traceability definitions and expresses how to reach the represented port from a root. A path that is a root represents the port specified by the root itself. On the other hand, for any path p, the path $p\langle f/n, i\rangle$ represents a port that is the i-th argument of a data atom f/n whose *output port* is connected directly to a port represented by p.

Applying a polarity inversion symbol $-$ to a path or a polarized path corresponds to finding the other side of a link, which is uniquely determined due to the link condition. Hence, let $-(-p)$ mean p and $(-p)\langle f/n, i\rangle$ mean $-(p\langle f/n, i\rangle)$ for any path p. The need for $-$ in the definition of polarized paths will become clear in Section 7.4.6. It is used for expressing the relationship between a sender and a receiver of the same data as an equality between polarized paths.

Example. The polarized path $\langle \texttt{root} : \texttt{append}/3, 1\rangle\langle \texttt{c}/3, 2\rangle\langle \texttt{c}/3, 1\rangle$ represents the port containing a link to the second element of a list where the list is received through the first argument of `append` at a membrane named `root`. The polarized path $-\texttt{X}\langle \texttt{c}/3, 2\rangle\langle \texttt{c}/3, 1\rangle$ represents the port containing a link to the second element of a list where the list is to be received through the free link `X` of another process template, for example `append(X,Y,Z)`. This is not

to be confused with the polarized path $\texttt{X}\langle\texttt{c}/3,2\rangle\langle\texttt{c}/3,1\rangle$, which represents the port containing a link to the second element of a list sent through the free link `X` of a process template, for example `append(A,B,X)`. □

Constraints not only over roots but over paths enables us to formulate the analysis of recursive and cyclic data structures. The way of extending roots by $\langle f/n, i\rangle$'s is borrowed from the research [34] on the static analysis of concurrent logic programs that has also been referred to in Chapter 6.

In the formalization, we have omitted the names of membranes a data atom can be located in. This is not only for simplicity but originated from our experience of writing programs in LMNtal.

7.4.6 Type Constraints

As we have said, we will assign a type to each port in terms of constraints[1] on the polarized paths. We will also assign a type to each membrane name. The result of the type inference for a program is represented by a type constraint.

Definition 7.5 (Type Constraints) We define the syntax of *type constraints* E, or constraints in short, as follows:

$$
\begin{array}{rcll}
E & ::= & \mathit{true} & ;\ \text{no constraints} \\
& | & E \wedge E & ;\ \text{conjunction} \\
& | & \exists X(E) & ;\ \text{localization of a link name} \\
& | & (\mathit{PPath} = \mathit{PPath}) & ;\ \text{polarized paths have the same type} \\
& | & (\mathit{PPath} : f/n) & ;\ \text{polarized path receives data atom } f/n \\
& | & (r : p/n) & ;\ \text{membrane has active atom } p/n \\
& | & (r : m) & ;\ \text{membrane has submembrane } m
\end{array}
$$

where X is a link name, and r and m are membrane names. □

Let p and q be paths. The constraint $(p = q)$ expresses that the two paths are either senders or receivers of the same data, in which case we say that they *have the same type* as each other, while $(p = -q)$ expresses that the two paths are a pair of a sender and a receiver of the same data, in which case we say that they *have the inverse types*.

[1] The term *constraint* is used in this chapter differently from in the other chapters.

For a path p, the constraint $(p : f/n)$ expresses that the path p can receive a data atom f/n, while the constraint $(-p : f/n)$ expresses that the path p can send a data atom f/n.

We have two kinds of constraints for membranes. Let r be a membrane name. The constraint $(r : p/n)$ expresses that the membrane r can contain an active atom p/n. On the other hand, the constraint $(r : m)$ expresses that the membrane r can contain a direct child membrane of the name m.

Intuitively, the constraint $\exists X(E)$ expresses a partial constraint obtained from E by ignoring the information on the link name X. Formally, the meaning of the construct $\exists X$ is defined using the axioms for the equivalence between constraints which are defined in the next paragraph.

Equivalence on Constraints

Finally, we define the equivalence on constraints. We write $D = E$ if and only if two constraints D and E are equivalent in the following sense. First of all, we assume that $\wedge$ is idempotent, commutative and associative and absorbs *true* as its identity. We also assume that $E = E'$ implies $D \wedge E = D \wedge E'$ and $\exists X(E) = \exists X(E')$. The rest of the assumptions on the equivalence are listed in the following axiom. We will write $D \Rightarrow E$ to mean that $D = D \wedge E$.

Axiom 7.1 We assume the following:

(EN1) $(n = n) = \mathit{true}$
(EN2) $(m = n) = (n = m)$
(EN3) $D \wedge (m = n) = D[n/m] \wedge (m = n)$
where D is a constraint not containing $\exists X$ for any X
(EX1) $E \Rightarrow \exists X(E)$
(EX2) $D \Rightarrow E$ implies $\exists X(D) \Rightarrow \exists X(E)$
(EX3) $\exists X \exists Y(D) = \exists Y \exists X(D)$
(EX4) $\exists X(D \wedge \exists X(E)) = \exists X(D) \wedge \exists X(E)$
(EX5) $\exists X(D) \Rightarrow E$ implies $E = \exists X(E)$
(EX6) $\exists X((Xp = n) \wedge E) = \exists X(E[n/Xp])$
(EX7) $\exists X(\bigwedge_i (Xp_i : f_i/k_i) \wedge \bigwedge_j (-Xq_j : g_j/m_j) \wedge E) = E$
where p_i and q_j are distinct for any i and j, and
E is a constraint in which X does not occur syntactically

where $[n/m]$ replaces every m by n. □

In (EX7), the condition that p_i and q_j are distinct is to avoid eliminating the inconsistency.

7.5 Formalization of Type Inference

In this section, we formalize our type inference.

7.5.1 Abbreviations

Here, we introduce abbreviations to make the formalization concise.

Definition 7.6 Let us denote the sequence $X_1, \ldots, X_n$ of already-known $n \geq 0$ distinct links by $\vec{X}$. In particular, we write $\{\vec{X}\}$ to mean $\{X_1, \ldots, X_n\}$.

For any constraint D, we define

$$\nabla\vec{X}(D) \stackrel{\text{def}}{=} D[(-X_1)/X_1] \ldots [(-X_n)/X_n].$$

□

Example.

$$\nabla \mathtt{Y}, \mathtt{Z}((\mathtt{X} = \mathtt{Y}) \wedge (-\mathtt{Y}\langle \mathtt{f}/2, 1\rangle : \mathtt{g}/3)) = ((\mathtt{X} = -\mathtt{Y}) \wedge (\mathtt{Y}\langle \mathtt{f}/2, 1\rangle : \mathtt{g}/3)).$$

□

The operator $\nabla\vec{X}$ is used for coupling two occurrences of a free link.

In addition, we will abbreviate $\exists X_1 \ldots \exists X_n(D)$ to $\exists X_1, \ldots, X_n(D)$.

7.5.2 Constraints Imposed by Process Templates

Finally, we can introduce our type system.

Definition 7.7 Assume a mapping *out* is given. We denote by $C[r : T]$ the constraint imposed by the process template T at membrane r, defined as in Figure 7.5. □

A local link composed on parallel composition requires that its occurrences have the inverse types, while a link occurring in both sides of a rule requires that its occurrences have the same type.

We may abbreviate $C[r : T]$ to $C[T]$ when r is irrelevant.

Definition 7.8 A polarized path n is said to have an *input* polarity in a constraint E if and only if $E \Rightarrow (n : f/k)$ holds with some data atom f/k. A polarized path n is said to have an *output* polarity if and only if $-n$ has input polarity. We say a program P is *well typed* if there are no polarized paths having both of the input polarity and the output polarity in $C[\mathtt{root} : P]$. □

$$
\begin{aligned}
C[r : \mathbf{0}] &= C[r : \texttt{@}p] = C[r : \texttt{\$}p] = \mathit{true} \\
C[r : (m\,\{T\})] &= (r : m) \wedge C[m : T] \\
C[r : (T, U)] &= \exists \vec{X}(C[r : T] \wedge \nabla\vec{X} C[r : U]) \\
&\qquad \text{where } \{\vec{X}\} = \mathrm{fv}(T) \cap \mathrm{fv}(U) \\
C[r : (T \texttt{ :- } U)] &= \exists \vec{X}(C[r : T] \wedge C[r : U]) \\
&\qquad \text{where } \{\vec{X}\} = \mathrm{fv}(T) \cap \mathrm{fv}(U) \\
C[r : p(X_1, \ldots, X_n)] &= (r : p/n) \wedge \textstyle\bigwedge_i(\langle r : p/n, i\rangle = X_i) \\
&\qquad \text{if } p/n \text{ is active} \\
C[r : p(X_1, \ldots, X_n)] &= (-X_u : p/n) \wedge \textstyle\bigwedge_{i \neq u}(-X_u\langle p/n, i\rangle = X_i) \\
&\qquad \text{if } \mathit{out}(p/n) = u \in \mathbf{N} \\
C[r : (X \texttt{ = } Y)] &= (X = -Y)
\end{aligned}
$$

Figure 7.5: Constraints imposed by process templates

We should mention that the link condition of process templates is essential in our typing scheme because it uses a link name to express its free occurrence that is uniquely determined due to the link condition. However, this does not mean that our typing scheme precludes a general graph rewriting language without the link condition. Some other type inference may statically detect those links in the program which always obey the link condition. To integrate such type inference with our typing scheme is beyond the scope of this work.

7.5.3 Type Graphs

Constraints can be visualized in the form of *type graphs*. For instance, type graphs for some basic constraints are shown in Figure 7.6 and Figure 7.7.

A type graph contains three kinds of larger objects: rounded rectangles for membranes, ellipses for active atoms, and circles for data atoms. Formally, a type graph consists of two parts: the upper part and the lower part.

The upper part of a type graph has active atoms and membranes as the vertices. Active atoms and membranes have outgoing arrows, in dashed lines, to the membranes in which they may reside.

The lower part of a type graph has paths as the vertices. Paths that are just link names are written as themselves. The other paths are placed on ellipses and circles with the labels describing their argument positions. Each

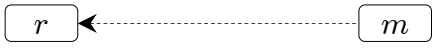
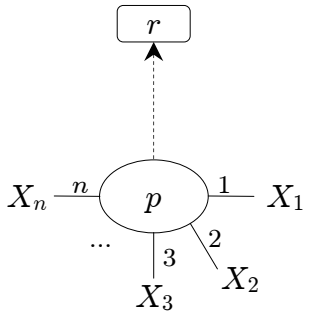
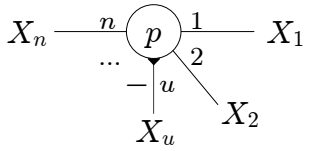


Figure 7.6: Type graph for the constraint $(r : m)$



(a) p/n is active



(b) p/n is a data atom whose u-th argument is output

Figure 7.7: Type graphs for $C[r : p(X_1, \ldots, X_n)]$

path can have a polarity sign (+ for input or − for output) as an additional label attached to it. Labels for link names can be omitted in the graph, for they are usually redundant. The path of the output port of each data atom must have the polarity sign − (a filled triangle can be drawn for emphasizing the output port). If two paths have the same type, they are connected together with lines and have the same polarity sign on them. If two paths have the inverse types, they are connected together and have different polarity signs on them. Polarity signs may be replaced by metavariables, such as q, when the polarity is not known but constrained. In that event, we write $-q$ to denote the inverse polarity sign for q. We sometimes draw arrowheads to emphasize the polarity sign +.

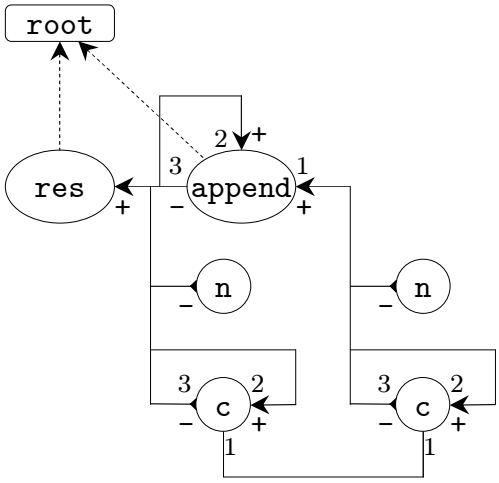


Figure 7.8: Type graph for list concatenation program

7.6 Inference Examples

7.6.1 List Concatenation

Consider the list concatenation program in Section 7.3.1 and Figure 7.4. For the first rule, we can infer the following constraints by structural induction:

$$
\begin{aligned}
&C[\texttt{root} : (\texttt{append(X0,Y,Z)},\texttt{c(A,X,X0)} \texttt{:-}\ \texttt{c(A,Z1,Z)},\texttt{append(X,Y,Z1)})] \\
&\quad = (\texttt{root} : \texttt{append}/3) && \textit{(line 1)} \\
&\quad \wedge (\langle \texttt{root} : \texttt{append}/3, 1\rangle : \texttt{c}/3) && \textit{(line 2)} \\
&\quad \wedge (-\langle \texttt{root} : \texttt{append}/3, 3\rangle : \texttt{c}/3) && \textit{(line 3)} \\
&\quad \wedge (\langle \texttt{root} : \texttt{append}/3, 1\rangle\langle \texttt{c}/3, 1\rangle = -\langle \texttt{root} : \texttt{append}/3, 3\rangle\langle \texttt{c}/3, 1\rangle) && \textit{(line 4)} \\
&\quad \wedge (\langle \texttt{root} : \texttt{append}/3, 1\rangle = \langle \texttt{root} : \texttt{append}/3, 1\rangle\langle \texttt{c}/3, 2\rangle) && \textit{(line 5)} \\
&\quad \wedge (\langle \texttt{root} : \texttt{append}/3, 3\rangle = \langle \texttt{root} : \texttt{append}/3, 3\rangle\langle \texttt{c}/3, 2\rangle) && \textit{(line 6)}
\end{aligned}
$$

This says that `append` at membrane `root` receives from the first argument a *stream*, namely a list of an unbounded length (at lines 2 and 5), sends another stream to the third argument (at lines 3 and 6), and the elements of one of these streams are transferred to the other (at line 4). Similar inference can be done for the rest of the program. Then, we can merge the results, by taking their conjunction, to obtain the whole result. The result of type inference for this program is visualized as the type graph in Figure 7.8.

7.6.2 Stream Merging

In LMNtal, a stream can serve as a message channel between two processes. When the communication is n-to-1, an arbitrary number of streams should be merged into one. The rule for stream merging can be described as follows:

```
merge{i(X0),o(Y0),$p},c(A,X,X0) :- c(A,Y,Y0),merge{i(X),o(Y),$p}
```

Here, a membrane of the name `merge` is assumed to contain n (≥ 1) atoms of the name `i` linked to input streams and one atom of the name `o` linked to an output stream. The process context `$p` is to match the `i` atoms other than the one that happens to be chosen as `i(X0)`. Figure 7.9 shows this rule and a process to which the rule is applied. The result of type inference for this program is visualized in Figure 7.10. We can observe that elements of the two streams have the same type and their polarity is not yet constrained.

7.6.3 Cyclic Data Structures

Graph rewriting languages are good at handing cyclic data structures. For instance, a bidirectional circular buffer with n elements can be represented as:

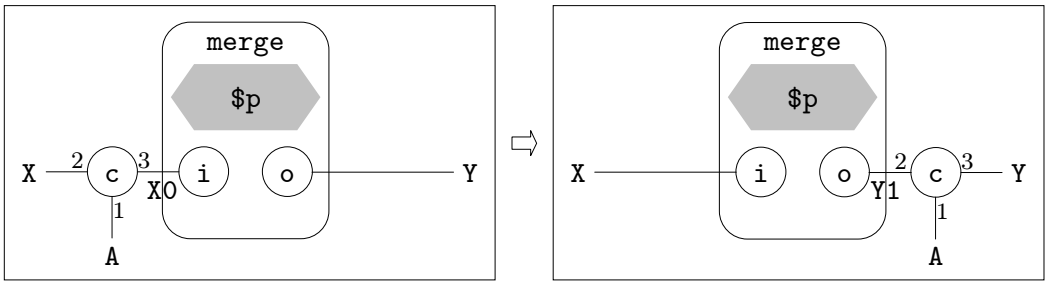
$$\mathtt{b}(S, X_n, X_0), \mathtt{n}(A_1, X_0, X_1), \ldots, \mathtt{n}(A_n, X_{n-1}, X_n)$$

where `b` is a header process, A_i's are links to the elements, and S is the link from the client process to this buffer. Figure 7.11 shows an example of such a circular buffer with five elements.

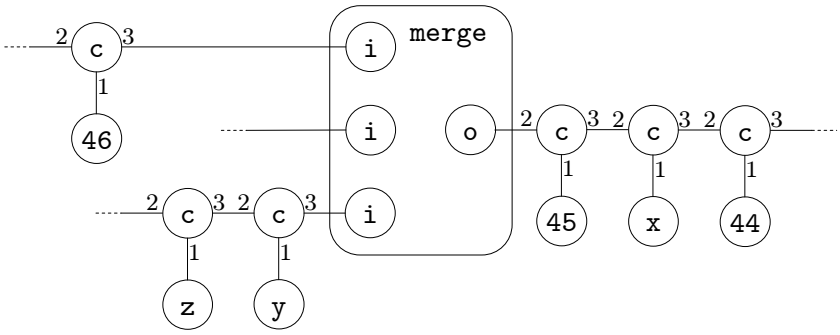
Operations on the buffer are sent through S from the client as messages such as `left`, `right` and `put`. The rewrite rules describing the reaction between messages and the buffer can be written as follows:

```
left(S,S0), n(A,L,C0), b(S0,C0,R) :- b(S,L,C1), n(A,C1,R)
right(S,S0), b(S0,L,C0), n(A,C0,R) :- n(A,L,C1), b(S,C1,R)
put(A,S,S0), b(S0,L,R) :- n(A,L,C1), b(S,C1,R)
```

Figure 7.12 shows two type graphs for these rules according to the choice of which of `b` and messages we take as active. The former corresponds to object oriented programming while the latter corresponds to procedural programming. We have let $out(\mathtt{b}/3) = 1$ when we take `b` as data, and let $out(\mathtt{right}/2) = 2$ and $out(\mathtt{put}/3) = 3$ when we take these messages as data. These specifications of out reflect what we think of data in the left-hand side



(a) Rewrite rule



(b) Target process

Figure 7.9: Stream merging for n-to-1 communication

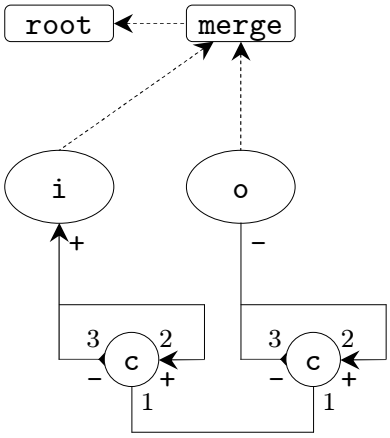


Figure 7.10: Type graph for stream merging

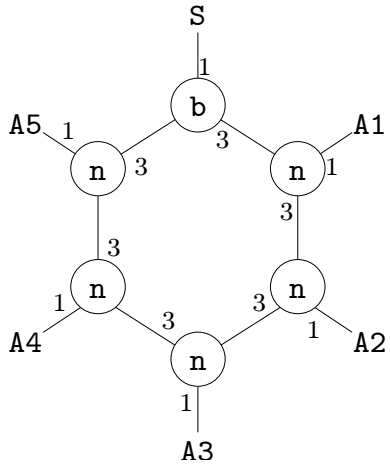
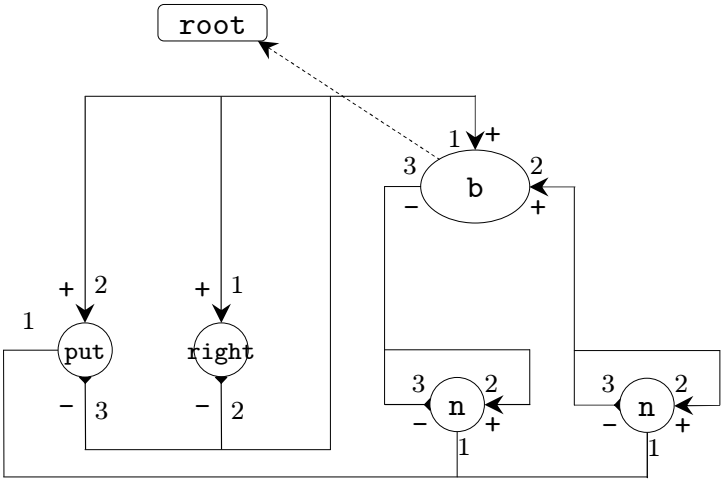
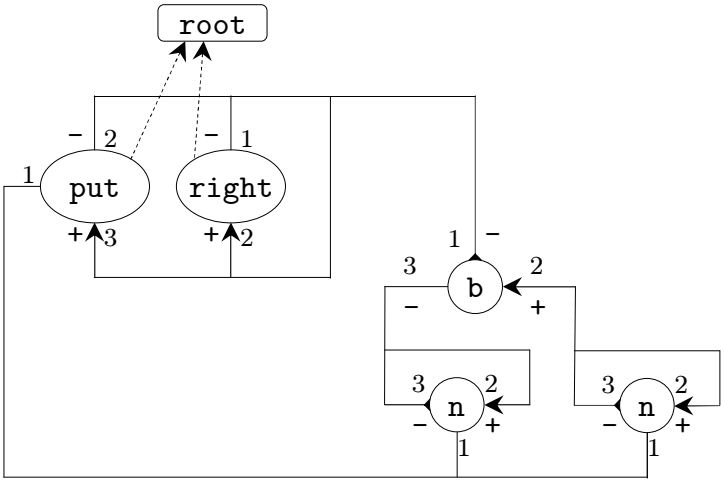


Figure 7.11: A circular buffer



(a) Consider `b` to be active.



(b) Consider messages to be active.

Figure 7.12: Type graphs for circular buffer

of rules, that is, data are sent to the receiver through their output ports. For simplicity, we have ignored the first rule, for `left`, in the analysis.

7.7 Theoretical Results

This section provides some theoretical results including type safety.

7.7.1 Properties on Free Links

We prove some basic properties on free links used in the subsequent proofs in Sections 7.7.2 and 7.7.3.

Definition 7.9 For any constraint D, we define

$$\mathrm{vars}(D) \stackrel{\mathrm{def}}{=} \{X \in \mathit{Links} \mid \exists X(D) \neq D\}\,.$$

where Links denotes the set of all the link names. □

Intuitively, $\mathrm{vars}(D)$ describes the set of links constrained in D. Now, we are going to prove that $\mathrm{fv}(T) = \mathrm{vars}(C[r:T])$ for any membrane name r.

Lemma 7.1 $X \notin \mathrm{fv}(T)$ implies $\exists X C[r:T] = C[r:T]$ for any r.

Proof. By structural induction on T.

- Case $T = \mathbf{0}$.
  We have $\exists X C[\mathbf{0}] = \exists X\, \mathit{true} = \mathit{true} = C[\mathbf{0}]$.
- Case $T = p(X_1, \ldots, X_n)$.
  Since $X \neq X_i$ for every i, we have $\exists X C[T] = C[T]$.
- Case $T = (T_1, T_2)$.
  Let $\{\vec{Z}\} = \mathrm{fv}(T_1) \cap \mathrm{fv}(T_2)$.
  - Case $X \in \{\vec{Z}\}$.
    We have $\exists X C[T_1, T_2] = \exists X \exists \vec{Z}(C[T_1] \wedge \nabla \vec{Z} C[T_2]) = \exists \vec{Z}(C[T_1] \wedge \nabla \vec{Z} C[T_2]) = C[T_1, T_2]$.

– Case $X \notin \{\vec{Z}\}$.

Since for each i we have $X \notin \mathrm{fv}(T_i)$, by induction hypothesis we have $\exists X C[T_i] = C[T_i]$ for each i. Hence, we have

$$\begin{aligned}\exists X C[T_1, T_2] &= \exists X \exists \vec{Z}(C[T_1] \wedge \nabla \vec{Z} C[T_2]) \\ &= \exists \vec{Z} \exists X(\exists X C[T_1] \wedge \nabla \vec{Z} C[T_2]) \\ &= \exists \vec{Z}(\exists X C[T_1] \wedge \exists X \nabla \vec{Z} C[T_2]) \\ &= \exists \vec{Z}(C[T_1] \wedge \nabla \vec{Z} C[T_2]) = C[T_1, T_2].\end{aligned}$$

- Case $T = (T_1 \ \texttt{:-}\ T_2)$.

  Similar to the case of (T_1, T_2) with the difference that $\nabla \vec{Z}$'s are removed.

- Case $T = m\,\{U\}$.

  Since $X \notin \mathrm{fv}(U)$, by induction hypothesis we have $\exists X C[m : U] = C[m : U]$, and therefore $\exists X C[r : m\,\{U\}] = \exists X((r : m) \wedge C[m : U]) = (r : m) \wedge \exists X C[m : U] = (r : m) \wedge C[m : U] = C[r : m\,\{U\}]$. □

Lemma 7.2 If $X \in \mathrm{fv}(T)$, we have $\exists X C[r : T] \neq C[r : T]$ for any r.

Proof. To prove the lemma, it is sufficient to construct for each T a constraint D such that $C[T] = (D \wedge \exists X C[T])$ and $D \neq \exists X(D)$ hold, since this refutes $\exists X C[T] = C[T]$. This is because, if it holds, we have $\exists X C[T] \Rightarrow D$ and hence $D = \exists X(D)$, which contradicts the assumption. We can construct such D by structural induction on T.

- Case $T = \mathbf{0}$.

  $X \in \mathrm{fv}(T)$ is inconsistent with the assumption.

- Case $T = p(X_1, \ldots, X_n)$.

  Let $X = X_i$. We have three cases:

  – Case where p/n is active.

    Let $D = (\langle r : p/n, i\rangle = X_i)$.

  – Case where p/n is a data atom.

    Let $u = \mathit{out}(p/n)$. If $i = u$, let $D = C[T]$. If $i \neq u$, let $D = (-X_u\langle p/n, i\rangle = X_i)$.

– Case where $p/n = \texttt{=}/2$.
Let $D = (X_1 = -X_2)$.

- Case $T = (T_1, T_2)$.

  Let $\{\vec{Z}\} = \mathrm{fv}(T_1) \cap \mathrm{fv}(T_2)$. We first consider the case where $X \in \mathrm{fv}(T_1)$. In this case, we have $X \notin \mathrm{fv}(T_2)$ and $X \notin \{\vec{Z}\}$. By induction hypothesis there exists E such that $C[T_1] = (E \wedge \exists X C[T_1])$ and $E \neq \exists X(E)$. Let ϕ be a substitution that replaces every Z_i occurring in $C[T_1]$ to a polarized path whose root is not in $\{\vec{Z}\}$ and satisfies $\exists X C[T_1] \wedge \nabla \vec{Z} C[T_2] \Rightarrow Z\phi = Z$ for all $Z \in \{\vec{Z}\}$. By Lemma 7.1 we have $\exists X C[T_2] = C[T_2]$. Hence,

$$\begin{aligned} C[T_1, T_2] &= \exists \vec{Z}(C[T_1] \wedge \nabla \vec{Z} C[T_2]) \\ &= \exists \vec{Z}(E \wedge \exists X C[T_1] \wedge \nabla \vec{Z} C[T_2]) \\ &= \exists \vec{Z}(E\phi \wedge \exists X C[T_1] \wedge \nabla \vec{Z} C[T_2]) \\ &= E\phi \wedge \exists \vec{Z}(\exists X C[T_1] \wedge \nabla \vec{Z} C[T_2]) \\ &= E\phi \wedge \exists \vec{Z} \exists X(C[T_1] \wedge \nabla \vec{Z} C[T_2]) \\ &= E\phi \wedge \exists X \exists \vec{Z}(C[T_1] \wedge \nabla \vec{Z} C[T_2]) \\ &= E\phi \wedge \exists X C[T_1, T_2]. \end{aligned}$$

  Thus, we can let $D = E\phi$.

  The case where $X \in \mathrm{fv}(T_2)$ is almost the same.

- Case $T = (T_1 \ \texttt{:-}\ T_2)$.

  Similar to the case of (T_1, T_2) with the difference that $\nabla \vec{Z}$'s are removed.

- Case $T = m\,\{U\}$.

  Since $X \in \mathrm{fv}(U)$, by induction hypothesis there exists D such that $C[m : U] = (D \wedge \exists X C[m : U])$ and $D \neq \exists X(D)$.

  Hence, we have $C[r : m\,\{U\}] = (r : m) \wedge C[m : U] = (r : m) \wedge D \wedge \exists X C[m : U] = D \wedge \exists X((r : m) \wedge C[m : U]) = D \wedge \exists X C[r : m\,\{U\}]$. □

Proposition 7.1 $\mathrm{fv}(T) = \mathrm{vars}(C[r : T])$ for any r.

Proof. From Lemmas 7.1 and 7.2. □

Definition 7.10 A *context* is either a process context $\$p$ or a rule context $@p$. Let T be a process template located in a membrane of the name r. The expression $context(m : \alpha, r : T)$ means that T contains a membrane of the name m such that the context α occurs in the non-rule part of that membrane. □

Given r, T and α, the syntactic condition (L5) guarantees that there is at most one of such m's, since membranes that have the same context must have the same name.

The following two propositions are fundamental in LMNtal.

Proposition 7.2 $P \equiv Q$ implies $\mathrm{fv}(P) = \mathrm{fv}(Q)$.

Proof. By induction on the derivation of the relation $\equiv$. Let $\oplus$ be the operator that computes the symmetric difference.

(E1) $\mathrm{fv}((\mathbf{0}, P)) = \{\} \oplus \mathrm{fv}(P) = \mathrm{fv}(P)$.

(E2) $\mathrm{fv}((P, Q)) = \mathrm{fv}(P) \oplus \mathrm{fv}(Q) = \mathrm{fv}((Q, P))$.

(E3) $\mathrm{fv}((P, (Q, R))) = \mathrm{fv}(P) \oplus \mathrm{fv}((Q, R)) = \mathrm{fv}(P) \oplus (\mathrm{fv}(Q) \oplus \mathrm{fv}(R)) = (\mathrm{fv}(P) \oplus \mathrm{fv}(Q)) \oplus \mathrm{fv}(R) = \mathrm{fv}((P, Q)) \oplus \mathrm{fv}(R) = \mathrm{fv}(((P, Q), R))$.

(E4) If $X \in \mathrm{lv}(P)$, we have $\mathrm{fv}(P) = \mathrm{fv}(P[Y/X])$.

(E5) If $P \equiv P'$, by induction hypothesis $\mathrm{fv}(P) = \mathrm{fv}(P')$, we have $\mathrm{fv}((P, Q)) = \mathrm{fv}(P) \oplus \mathrm{fv}(Q) = \mathrm{fv}(P') \oplus \mathrm{fv}(Q) = \mathrm{fv}((P', Q))$.

(E6) If $P \equiv P'$, by induction hypothesis $\mathrm{fv}(P) = \mathrm{fv}(P')$, we have $\mathrm{fv}(m\{P\}) = \mathrm{fv}(P) = \mathrm{fv}(P') = \mathrm{fv}(m\{P'\})$.

(E7) $\mathrm{fv}(X = Y,\ Y = X) = \{\} = \mathrm{fv}(\mathbf{0})$.

(E8) $\mathrm{fv}(X = Y) = \{X\} \oplus \{Y\} = \mathrm{fv}(Y = X)$.

(E9) The case $(X = Y,\ P) \equiv P[Y/X]$ where $X \in \mathrm{fv}(P)$ and P is an atom.

Since $Y \notin \mathrm{fv}(P)$, we have $\mathrm{fv}(P[Y/X]) = (\mathrm{fv}(P) \setminus \{X\}) \cup \{Y\} = \{X, Y\} \oplus \mathrm{fv}(P) = \mathrm{fv}(X = Y) \oplus \mathrm{fv}(P) = \mathrm{fv}(X = Y,\ P)$.

(E10) $\mathrm{fv}(X = Y,\, m\{P\}) = \mathrm{fv}(X = Y) \oplus \mathrm{fv}(P) = \mathrm{fv}(m\{X = Y, P\})$.

□

Proposition 7.3 $P \longrightarrow Q$ implies $\mathrm{fv}(P) = \mathrm{fv}(Q)$.

Proof. By induction on the derivation of the relation $\longrightarrow$. Let $\oplus$ be the operator that computes the symmetric difference.

(R1) Case $P, Q \longrightarrow P', Q$ where $P \longrightarrow P'$.

By induction hypothesis $\mathrm{fv}(P) = \mathrm{fv}(P')$, we have $\mathrm{fv}(P, Q) = \mathrm{fv}(P) \oplus \mathrm{fv}(Q) = \mathrm{fv}(P') \oplus \mathrm{fv}(Q) = \mathrm{fv}(P', Q)$.

(R2) Case $m\,\{P\} \longrightarrow m\,\{P'\}$ where $P \longrightarrow P'$.

By induction hypothesis $\mathrm{fv}(P) = \mathrm{fv}(P')$, we have $\mathrm{fv}(m\,\{P\}) = \mathrm{fv}(P) = \mathrm{fv}(P') = \mathrm{fv}(m\,\{P'\})$.

(R3) Case $Q \longrightarrow Q'$ where $Q \equiv P \longrightarrow P' \equiv Q'$.

By induction hypothesis $\mathrm{fv}(P) = \mathrm{fv}(P')$ and by Proposition 7.2, we have $\mathrm{fv}(Q) = \mathrm{fv}(P) = \mathrm{fv}(P') = \mathrm{fv}(Q')$.

(R4) Case $m\,\{X = Y,\, P\} \longrightarrow X = Y,\, m\,\{P\}$.

We have $\mathrm{fv}(m\,\{X = Y,\, P\}) = \mathrm{fv}(X = Y) \oplus \mathrm{fv}(P) = \mathrm{fv}(X = Y,\, m\,\{P\})$.

(R5) Case $X = Y,\, m\,\{P\} \longrightarrow m\,\{X = Y,\, P\}$.

The proof is the same as (R4).

(R6) Case $T\theta, (T \;\texttt{:-}\; U) \longrightarrow U\theta, (T \;\texttt{:-}\; U)$.

Since $\mathrm{fv}(T \;\texttt{:-}\; U) = \{\}$, we only have to prove that $\mathrm{fv}(T\theta) = \mathrm{fv}(U\theta)$.

The link condition of a rule ensures that $\mathrm{fv}(T) = \mathrm{fv}(U)$. Each $\$p$ occurs once in T and once in U if it occurs in the rule. For any $@p$, we have $\mathrm{fv}(@p\theta) = \{\}$. Putting it all together, we have

$$\begin{array}{rcl} \mathrm{fv}(T\theta) & = & \mathrm{fv}(T) \oplus \bigoplus_{context(m:\alpha,\, r:T)} \mathrm{fv}(\alpha\theta) \\ & = & \mathrm{fv}(T) \oplus \bigoplus_{context(m:\$p,\, r:T)} \mathrm{fv}(\$p\theta) \\ & = & \mathrm{fv}(U) \oplus \bigoplus_{context(m:\$p,\, r:U)} \mathrm{fv}(\$p\theta) \\ & = & \mathrm{fv}(U) \oplus \bigoplus_{context(m:\alpha,\, r:U)} \mathrm{fv}(\alpha\theta) \\ & = & \mathrm{fv}(U\theta). \end{array}$$

Here, the operator $\bigoplus$ in the first line takes as arguments the family of $\mathrm{fv}(\alpha\theta)$'s prescribed by all α's such that $\mathit{context}(m : \alpha, r : T)$ holds with some m. □

7.7.2 Adequacy

Now, we can prove that the constraints for structurally equivalent processes coincide. In order to give a concise proof of it, we introduce the following operator over constraints. This operator enables us to express the coupling of local links between constraints. The resulting constraint obtained with this operator does not contain any information on the coupled links.

Definition 7.11 We define the operator $\#$ on constraints by

$$D \# E \stackrel{\text{def}}{=} \exists \vec{X}(D \wedge \nabla \vec{X} E)$$

where $\{\vec{X}\} = \mathrm{vars}(D) \cap \mathrm{vars}(E)$. □

We have that $C[r : (T, U)] = C[r : T] \# C[r : U]$. Moreover, we have $D \# E = D \wedge E$ if $\mathrm{vars}(D) = \{\}$. In particular, we have $\mathit{true} \# E = E$.

Lemma 7.3 $D \# E = E \# D$.

Proof. Let $\{\vec{X}\} = \mathrm{vars}(D) \cap \mathrm{vars}(E)$. We have $D \# E = \exists \vec{X}(D \wedge \nabla \vec{X} E) = \exists \vec{X} \nabla \vec{X}(E \wedge \nabla \vec{X} D) = \exists \vec{X}(E \wedge \nabla \vec{X} D) = E \# D$. □

Lemma 7.4 If $\mathrm{vars}(D) \cap \mathrm{vars}(E) \cap \mathrm{vars}(F) = \{\}$, we have $D \#(E \# F) = (D \# E) \# F$.

Proof. Let $\{\vec{X}\} = \mathrm{vars}(D) \cap \mathrm{vars}(E)$, $\{\vec{Y}\} = \mathrm{vars}(D) \cap \mathrm{vars}(F)$, and $\{\vec{Z}\} = \mathrm{vars}(E) \cap \mathrm{vars}(F)$. By assumption we have $\exists \vec{Z} D = D$ and $\exists \vec{X} F = F$. Therefore, we have

$$\begin{aligned}
D \#(E \# F) &= \exists \vec{X} \vec{Y}(D \wedge \nabla \vec{X} \vec{Y} \exists \vec{Z}(E \wedge \nabla \vec{Z} F)) \\
&= \exists \vec{X} \vec{Y} \vec{Z}(D \wedge \nabla \vec{X} \vec{Y}(E \wedge \nabla \vec{Z} F)) \\
&= \exists \vec{X} \vec{Y} \vec{Z}(D \wedge \nabla \vec{X} \vec{Y} E \wedge \nabla \vec{X} \vec{Y} \vec{Z} F)) \\
&= \exists \vec{X} \vec{Y} \vec{Z}(D \wedge \nabla \vec{X} E \wedge \exists \vec{X} \nabla \vec{Y} \vec{Z} F) \\
&= \exists \vec{Y} \vec{Z}(\exists \vec{X}(D \wedge \nabla \vec{X} E) \wedge \nabla \vec{Y} \vec{Z} F) \\
&= (D \# E) \# F.
\end{aligned}$$

□

In the practical use of this operator, the link condition often ensures the condition part of Lemma 7.4.

Lemma 7.3 and Lemma 7.4 ensure that we can safely extend # so that it takes an arbitrary number of arguments. Let $D_1, \ldots, D_n$ be a family of constraints such that for any $X \in \bigcup_{i=1}^{n} \mathrm{vars}(D_i)$ there are at most two indices i satisfying $X \in \mathrm{vars}(D_i)$. Then, we can define $\#_{i=1}^{n} D_i$ as follows:

$$\mathop{\#}_{i=1}^{n} D_i \stackrel{\text{def}}{=} \begin{cases} \mathit{true} & \text{if } n = 0 \\ \left(\#_{i=1}^{n-1} D_i\right) \# D_n & \text{if } n \geq 1. \end{cases}$$

Lemma 7.5 If $\mathrm{vars}(D) = \mathrm{vars}(D')$ and $D \Rightarrow D'$ then $D \# E \Rightarrow D' \# E$.

Proof. Since $D \Rightarrow D'$ we have $D \wedge \nabla \vec{X} E \Rightarrow D' \wedge \nabla \vec{X} E$ and therefore $\exists \vec{X}(D \wedge \nabla \vec{X} E) \Rightarrow \exists \vec{X}(D' \wedge \nabla \vec{X} E)$. This means that $D \# E \Rightarrow D' \# E$. □

Before going on to the main theorems, we must observe the following lemma which states that our typing contains full information on the free links of process templates.

Lemma 7.6 Assume that one of the following holds:

- assumption 1: $X \in \mathrm{fv}(T)$ and $Y \notin \mathrm{fv}(T)$.
- assumption 2: $X \notin \mathrm{fv}(T)$.

Then, we have $C[T[Y/X]] = C[T][Y/X]$.

Proof. Let $\delta = [Y/X]$. By structural induction on T.

- Case $T = \mathbf{0}$.
  $C[\mathbf{0}\delta] = C[\mathbf{0}] = C[\mathbf{0}]\delta$.
- Case $T = p(Z_1, \ldots, Z_n)$.
  Since $X \notin \mathrm{lv}(T)$, we have $X \notin \{\vec{Z}\}$ or $X \in \mathrm{fv}(T)$. The rest is straightforward by definition of $C[T]$.

- Case $T = m\,\{U\}$.

  By induction hypothesis on U, we have $C[m : U]\delta = C[m : U\delta]$. Hence, we have

$$\begin{aligned} C[r : m\,\{U\}]\delta &= (r : m) \wedge C[m : U]\delta \\ &= (r : m) \wedge C[m : U\delta] \\ &= [r : m\,\{U\delta\}]. \end{aligned}$$

- Case $T = (T_1, T_2)$.

  Let $\{\vec{Z}\} = \mathrm{fv}(T_1) \cap \mathrm{fv}(T_2)$.

  - Case where assumption 1 holds.

    Let $X \in \mathrm{fv}(T_i)$. We have $X \notin \mathrm{fv}(T_{3-i})$. By induction hypothesis with assumption 1, we have $C[T_i\delta] = C[T_i]\delta$. By induction hypothesis with assumption 2, we have $C[T_{3-i}\delta] = C[T_{3-i}]\delta$.

  - Case where assumption 2 holds.

    * Case $X \in \{\vec{Z}\}$.
      By the link condition of $(T_1, T_2)\delta$, for $i = 1, 2$ we have $Y \notin \mathrm{fv}(T_i)$ and therefore by induction hypothesis with assumption 1, we have $C[T_i]\delta = C[T_i\delta]$.
    * Case $X \notin \{\vec{Z}\}$.
      For $i = 1, 2$ we have $X \notin \mathrm{fv}(T_i)$ and therefore by induction hypothesis with assumption 2, we have $C[T_i]\delta = C[T_i\delta]$.

  Hence, we have

$$\begin{aligned} C[T_1, T_2]\delta &= (\exists\vec{Z}(C[T_1] \wedge \nabla\vec{Z}C[T_2]))\delta \\ &= \exists\vec{Z}\delta(C[T_1]\delta \wedge (\nabla\vec{Z}C[T_2])\delta) \\ &= \exists\vec{Z}\delta(C[T_1]\delta \wedge \nabla\vec{Z}\delta(C[T_2]\delta)) \\ &= \exists\vec{Z}\delta(C[T_1\delta] \wedge \nabla\vec{Z}\delta C[T_2\delta]) \\ &= C[T_1\delta, T_2\delta] \\ &= C[(T_1, T_2)\delta]. \end{aligned}$$

- Case $T = (T_1 \ \texttt{:-}\ T_2)$.

  Almost the same as the above case. □

Now, we can prove the following important property, namely that our typing is preserved under the structural congruence.

Theorem 7.1 (Adequacy) $P \equiv Q$ implies $C[r : P] = C[r : Q]$.

Proof. By induction on the derivation of the relation $\equiv$.

(E1) Case $(\mathbf{0}, P) \equiv P$.

Since $\mathrm{fv}(\mathbf{0}) = \{\}$, we have $C[\mathbf{0}, P] = C[\mathbf{0}] \wedge C[P] = \mathit{true} \wedge C[P] = C[P]$.

(E2) Case $P, Q \equiv Q, P$.

We have $C[P, Q] = C[P] \# C[Q] = C[Q] \# C[P] = C[Q, P]$.

(E3) Case $P, (Q, R) \equiv (P, Q), R$.

By the link condition we have $\mathrm{fv}(P) \cap \mathrm{fv}(Q) \cap \mathrm{fv}(R) = \{\}$. By Lemma 7.1, we have $\mathrm{vars}(C[P]) \cap \mathrm{vars}(C[Q]) \cap \mathrm{vars}(C[R]) = \{\}$. Hence, by Lemma 7.4, we have

$$\begin{aligned} C[P, (Q, R)] &= C[P] \# C[Q, R] \\ &= C[P] \#(C[Q] \# C[R]) \\ &= (C[P] \# C[Q]) \# C[R] \\ &= C[(P, Q), R]. \end{aligned}$$

(E4) Case $P \equiv P[Y/X]$ where $X \in \mathrm{lv}(P)$.

By Lemma 7.1, $C[P] = \exists X C[P] = \exists Y C[P[Y/X]]$. By Lemma 7.1, $\exists Y C[P[Y/X]] = C[P[Y/X]]$. And by Lemma 7.6 with assumption 2, we have $C[P[Y/X]] = C[P][Y/X]$.

(E5) Case $P, Q \equiv P', Q$ where $P \equiv P'$.

By Theorem 7.2 we have $\mathrm{fv}(P) = \mathrm{fv}(P')$. By induction hypothesis $C[P] = C[P']$, we have $C[P, Q] = C[P] \# C[Q] = C[P'] \# C[Q] = C[P', Q]$.

(E6) Case $m\,\{P\} \equiv m\,\{P'\}$ where $P \equiv P'$.

By induction hypothesis $C[m : P] = C[m : P']$, we have $C[m\,\{P\}] = (r : m) \wedge C[m : P] = (r : m) \wedge C[m : P'] = C[m\,\{P'\}]$.

(E7) Case $(X = Y,\ Y = X) \equiv \mathbf{0}$.

We have

$$\begin{aligned} C[X = Y,\ Y = X] &= \exists XY((X = -Y) \wedge \nabla XY(Y = -X)) \\ &= \exists XY((X = -Y) \wedge (-Y = X)) \\ &= \exists XY(X = -Y) = \mathit{true} = C[\mathbf{0}]. \end{aligned}$$

(E8) Case $(X = Y) \equiv (Y = X)$.

We have

$$\begin{aligned} C[X = Y] &= \exists XY(X = -Y) \\ &= \exists XY(Y = -X) = C[Y = X]. \end{aligned}$$

(E9) Case $X = Y,\ P \equiv P[Y/X]$ where $X \in \mathrm{fv}(P)$ and P is an atom.

By the syntactic condition of $P[Y/X]$, the link Y does not occur in P. Hence, by Lemma 7.6 with assumption 1, we have

$$\begin{aligned} C[X = Y,\ P] &= \exists X(X = -Y \wedge \nabla X C[P]) \\ &= \exists X(X = -Y \wedge C[P][-X/X]) \\ &= \exists X(X = -Y \wedge C[P][Y/X]) \\ &= C[P][Y/X] = C[P[Y/X]]. \end{aligned}$$

(E10) Case $X = Y,\ m\,\{P\} \equiv m\,\{X = Y,\ P\}$.

We assume $X \in \mathrm{fv}(P)$ and $Y \notin \mathrm{fv}(P)$. We have

$$\begin{aligned} C[r : (X = Y,\ m\,\{P\})] &= \exists X(\nabla X(X = -Y) \wedge (r : m) \wedge C[m : P]) \\ &= (r : m) \wedge \exists X(\nabla X(X = -Y) \wedge C[m : P]) \\ &= (r : m) \wedge C[m : (X = Y,\ P)] \\ &= C[r : m\,\{X = Y,\ P\}]. \end{aligned}$$

□

7.7.3 Type Safety

The main result of this chapter is that the constraints imposed by processes are monotonically weakened along rewritings. This means that the constraint imposed by the initial configuration of a program does not conflict with any possible future configuration of that program. Thus, our type system can be used not only for describing specification but also as a foundation of optimized implementation whose correctness hinges on the type safety. A typical example of such optimization is data representation optimization.

Theorem 7.2 (Type Safety) $P \longrightarrow P'$ implies $C[r : P] \Rightarrow C[r : P']$.

Proof. By induction on the derivation of the relation $\longrightarrow$.

(R1) Case $P, Q \longrightarrow P', Q$ where $P \longrightarrow P'$.

By theorem 7.3 we have $\mathrm{fv}(P) = \mathrm{fv}(P')$. By induction hypothesis we have $C[P] \Rightarrow C[P']$. Hence, we have $C[P, Q] = C[P] \# C[Q] \Rightarrow C[P'] \# C[Q] = C[P', Q]$.

(R2) Case $m\,\{P\} \longrightarrow m\,\{P'\}$ where $P \longrightarrow P'$.

By induction hypothesis we have $C[m : P] \Rightarrow C[m : P']$. Hence, we have $C[r : m\,\{P\}] = (r : m) \wedge C[m : P] \Rightarrow (r : m) \wedge C[m : P'] = C[r : m\,\{P'\}]$.

(R3) Case $Q \longrightarrow Q'$ where $Q \equiv P \longrightarrow P' \equiv Q'$.

By induction hypothesis we have $C[P] \Rightarrow C[P']$. Hence, by Theorem 7.1, we have $C[Q] = C[P] \Rightarrow C[P'] = C[Q']$.

(R4) Case $m\,\{X = Y,\ P\} \longrightarrow X = Y,\ m\,\{P\}$.

Let $\{\vec{Z}\} = \mathrm{fv}(X = Y) \cap \mathrm{fv}(P)$. We have

$$\begin{aligned} C[r : (X = Y,\ m\,\{P\})] &= \exists \vec{Z}(\nabla \vec{Z}(X = -Y) \wedge (r : m) \wedge C[m : P]) \\ &= (r : m) \wedge \exists \vec{Z}(\nabla \vec{Z}(X = -Y) \wedge C[m : P]) \\ &= (r : m) \wedge C[m : (X = Y,\ P)] \\ &= C[r : m\,\{X = Y,\ P\}]. \end{aligned}$$

(R5) Case $X = Y,\ m\,\{P\} \longrightarrow m\,\{X = Y,\ P\}$.

The proof is the same as (R4).

(R6) Case $T\theta, (T\ \mathrel{:-}\ U) \longrightarrow U\theta, (T\ \mathrel{:-}\ U)$.

Let $\{\vec{X}\} = \mathrm{fv}(T) \cap \mathrm{fv}(U)$. Let $C[T] = C[T] \wedge \bigwedge_i (X_i\phi = X_i)$ where ϕ is a substitution whose domain is $\mathrm{fv}(T)$ and whose codomain is a set of polarized paths that have active atoms as their roots. Note that such ϕ always exists because of the active head condition.

Let

$$B = \#_{context(m:\alpha,\, r:T)}\, C[m : \alpha\theta]$$

and

$$B' = \#_{context(m:\alpha,\, r:U)}\, C[m : \alpha\theta].$$

The operator $\#$ in B takes as arguments the family of $C[m : \alpha\theta]$'s prescribed by all the pairs of m and α such that $\mathit{context}(m : \alpha, r : T)$ holds.

Note that Condition (L5) ensures that $\mathit{context}(m : \alpha, r : T)$ and $\mathit{context}(m' : \alpha, r' : U)$ together imply that $m = m'$. Since rules have no free links, we have $\mathrm{vars}(C[m : (@p)\theta]) = \{\}$ and hence $E \# C[m : (@p)\theta] = E \wedge C[m : (@p)\theta]$ for any E. Moreover, since every rule context occurring in the non-rule part of U occurs also in T, we have $\bigwedge_{\exists r(\mathit{context}(m:@_{p,\,r:T}))} C[m : (@p)\theta] \Rightarrow \bigwedge_{\exists r(\mathit{context}(m:@_{p,\,r:U}))} C[m : (@p)\theta]$. On the other hand, since every process context in T occurs exactly once in the non-rule part of U, we have $\#_{\mathit{context}(m:\$_{p,\,r:T})} C[m : (\$p)\theta] = \#_{\mathit{context}(m:\$_{p,\,r:U})} C[m : (\$p)\theta]$. Putting it all together, $B \Rightarrow B'$ holds.

This implication states that rules may be removed from a membrane upon a reduction, which weakens the constraints imposed by that membrane. In fact, this is the exact reason we only have a monotonicity theorem rather than a preservation theorem.

Since

$$\begin{aligned} C[T \ \texttt{:-}\ U] &= \exists \vec{X}(C[T] \wedge C[U]) \\ &= \exists \vec{X}(C[T] \wedge C[U]\phi) \\ &= \exists \vec{X}(C[T] \wedge C[U]\phi) \wedge \exists \vec{X}(C[U]\phi) \\ &= C[T \ \texttt{:-}\ U] \wedge \exists \vec{X}(C[U]\phi) \\ &= C[T \ \texttt{:-}\ U] \wedge C[U]\phi, \end{aligned}$$

we have

$$\begin{aligned} & C[T\theta, (T \ \texttt{:-}\ U)] \\ = \ & C[T\theta] \wedge C[T \ \texttt{:-}\ U] \\ = \ & C[T\theta] \wedge C[T \ \texttt{:-}\ U] \wedge (C[T] \# B) \\ \Rightarrow \ & C[T\theta] \wedge C[T \ \texttt{:-}\ U] \wedge (C[T] \# B') \\ = \ & C[T\theta] \wedge C[T \ \texttt{:-}\ U] \wedge (C[T] \# B') \wedge (C[T] \# B') \wedge C[U]\phi \\ = \ & C[T\theta] \wedge C[T \ \texttt{:-}\ U] \wedge (C[T] \# B') \wedge ((C[T] \# B') \# C[U]\phi) \\ = \ & C[T\theta] \wedge C[T \ \texttt{:-}\ U] \wedge (C[T] \# B') \wedge ((C[T] \# C[U]\phi) \# B') \\ = \ & C[T\theta] \wedge C[T \ \texttt{:-}\ U] \wedge (C[T] \# B') \wedge ((C[T] \wedge C[U]\phi) \# B') \\ = \ & C[T\theta] \wedge C[T \ \texttt{:-}\ U] \wedge (C[T] \# B') \wedge ((C[T] \wedge C[U]) \# B') \\ \Rightarrow \ & C[T\theta] \wedge C[T \ \texttt{:-}\ U] \wedge (C[T] \# B') \wedge (C[U] \# B') \\ = \ & C[T\theta] \wedge C[T \ \texttt{:-}\ U] \wedge C[U\theta] \\ = \ & (C[T\theta] \wedge C[T \ \texttt{:-}\ U]) \wedge (C[U\theta] \wedge C[T \ \texttt{:-}\ U]) \\ = \ & C[T\theta, (T \ \texttt{:-}\ U)] \wedge C[U\theta, (T \ \texttt{:-}\ U)] \end{aligned}$$

and this means that $C[T\theta, (T \ :\!\!-\ U)] \Rightarrow C[U\theta, (T \ :\!\!-\ U)]$. □

7.8 Concluding Remarks

We have proposed a method for introducing a strong type system in relatively simple graph rewriting where by relatively simple we mean that processes and data are treated in a unified way. Specifically, we have built a type system for the hierarchical graph rewriting language LMNtal that has atoms as graph nodes. In our type system, a programmer only needs to classify atoms into active atoms and data atoms in order to infer a type graph that describes complex process structures in the program. The application of our type system includes data representation optimization for unary data atoms like integers in the runtime system. The author believes that the unified treatment of these two kinds of atoms in the language together with a type inference mechanism as our type system provides can serve as a bridge between theoretical research and running language systems.

Among related work on typing concurrent graph rewriting languages are Shape Types [17]. They provide strong typing to graph rewriting. Since they do not have the notion of activeness, its range of practical application can be limited. Typed graphs are another standard approach to typing graph structures by giving a graph morphism from concrete graphs to a type graph, see for example [27]. However, it also does not have the notion of activeness.

In this chapter, we have proposed a type system based on classifying graph nodes into active atoms and data atoms. Our method can be thought of as generalization and extension of the analysis [34] for concurrent logic programming we have introduced in Chapter 6, when an active atom in LMNtal is regarded as a predicate call in the concurrent logic programming. In their formulation, the type of a port is expressed as a moding function from paths to polarities and two ports have the same type when they have the same moding function. In our formulation, on the other hand, the type of a port is expressed in terms of type constraints themselves, not as a moding function, and two ports have the same type when they are constrained to be the same polarized path. Our formulation is more general than theirs in that it can distinguish the types of irrelevant nodes that happen to have the same moding function, especially a uniform moding function. Anyhow, the constraint-based approach toward reconstructing data types taken by [34] and us has an advantage that it can handle data structures other than trees

such as non-terminating streams and cyclic structures, which are characteristic of concurrent and graph rewriting languages.

Research in the opposite direction of our work includes the paper [7], which explains how to regard logic programs as hypergraph replacement. They make distinction between predicate symbols and function symbols, as in our work, and moreover they consider shared, uninstantiated logic variables in the hypergraph. Although it may be very fruitful to incorporate their logic-program point of view toward graph rewriting to the typing scheme of graph rewriting languages, they are fundamentally devoted to tree-like structures and cyclic data structures and graph hierarchization are not considered. In fact, the type graphs depicted in this chapter are similar to the hypergraphs explained there, but our type graphs are extended by membranes and polarity information and can represent cyclic data structures that rewrites themselves. It may be an interesting research topic to formalize our type graph as a hypergraph.

Chapter 8

Conclusions

We have introduced several semantics-based static analysis techniques for concurrent programs and presented the proofs of their correctness. These analysis techniques offer a theoetical justification for optimizing compilation of the concurrent programs. In the following, we will summarize the result of this dissertation.

Firstly, we have introduced the following analysis techniques for concurrent logic programming languages:

- the safety of moving synchronization points,
- sequentiality analysis, and
- occurs-check analysis under cooperative modings.

Then, for the graph rewriting language LMNtal we have proposed:

- process structure analysis.

In the work of the motion of synchronization points, we have proved that our program transformation preserves the denotational semantics that takes into account not only interaction with the store but the divergence and termination, which is a new theoretical result of this work.

In the work of sequentiality analysis, we have introduced the notion of interfaces that formalize portions of concurrent processes that can be executed sequentially. We have also demonstrated that inferring interfaces in a bottom-up manner can derive code generation and optimizing compilation.

In the work of occurs-check analysis, we gave a new algorithm for guaranteeing the absence of cyclic data structures in a program. The algorithm

uses cooperative modings and the linear head condition in the program. The analysis had an instance program that had not been covered by the existing methods.

In the work of typing LMNtal processes, we described a way of introducing to LMNtal types that can be practically used by programmers and for optimizing compilation. The type system was based on discriminating the graph nodes into active ones and data and the type inference was formalized as constraint simplification. The advantage of the constraint-based approach was that the types in a program could be systematically inferred from the program, which could be helpful in the implementation of the optimizing compilation of a wide range of graph rewriting programs.

These analysis techniques can be applied to other concurrent languages. For instance, the notion of interfaces introduced in the work of sequentiality analysis can be applied to the analysis of other fine-grained concurrent languages so that the runtime overhead on concurrent operations will be reduced. These languages include the pi-calculus, lazy functional languages, functional languages with a *future* primitive, and concurrent object-oriented languages. A justiciation of this claim is that sequentialization is known to be important to the optimizing compilation of fine-grained concurrent languages. Another justification is that the formalism of interfaces offers semantics-based analysis for extracting sequentiality in the concurrent programs, provided that the communication between processes can be expressed as monotonically strengthening constraints as in concurrent logic programming. In the other languages where communication is not based on constraints, the formalism of interfaces may require some modification. For instance, the pi-calculus relies on name-based communication and would require a careful treatment in expressing communication as constraints.

The author believes that concurrent languages will be more and more important as a means to describe concurrent, parallel and distributed systems when they are supported with useful language systems. The usefulness of the systems should include efficiency and correctness as well as just usability. To this end, the semantics-based approach provides a promising way of proving the correctness of the optimizing compilation of concurrent programs.

Bibliography

[1] Andries, M. *et al.*: Graph Transformation for Specification and Programming. *Science of Computer Programming*, Vol. 34, No. 1 (1999), pp. 1–54.

[2] Appel, A. W.: *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Apt, K. R. and Pellegrini, A.: On the Occur-check Free Prolog Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 16, No. 3 (1994), pp. 687–726.

[4] Bagnara, R., Gori, R., Hill, P. M., and Zaffanella, E.: Finite-Tree Analysis for Constraint Logic-Based Languages. *Static Analysis: 8th Int. Symp.*, LNCS 2126, Springer-Verlag, pp. 165–184, 2001.

[5] Chikayama, T., Fujise, T. and Sekita, D.: A Portable and Efficient Implementation of KL1. *Proc. Programming Language Implementation and Logic Programming (PLILP'94)*, LNCS 844, Springer-Verlag, pp. 25–39, 1994.

[6] Codish, M., Falaschi, M., and Marriot, K.: Suspension Analyses for Concurrent Logic Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 16, No. 3 (1994), pp. 649–686.

[7] Corradini, A., Rossi, F., Parisi-Presicce, F.: Logic Programming as Hypergraph Rewriting. *Proc. the international joint conference on theory and practice of software development (TAPSOFT) on Colloquium on trees in algebra and programming (CAAP '91)*, LNCS 493, Springer-Verlag, pp. 275–295, 1991.

[8] Crnogorac, L., Kelly, A. D., and Søndergaard, H.: A Comparison of Three Occur-Check Analysers. *Proc. the 3rd International Symposium*, LNCS 1145, Springer-Verlag, pp. 159–173, 1996.

[9] de Boer, F. S. and Palamidessi, C.: From Concurrent Logic Programming to Concurrent Constraint Programming, *Advances in Logic Programming Theory*, G. Levi (ed.), pp. 55–113, Oxford University Press, 1994.

[10] de Boer, F. S. and Palamidessi, C.: On the semantics of concurrent constraint programming. *Proc. UK Conference on Logic Programming (ALPUK 92)*, Workshops in Computing, pp. 145–173, Springer-Verlag, 1992.

[11] de Boer, F. S., Di Pierro, A., and Palamidessi, C.: Nondeterminism and Infinite Computations in Constraint Programming. *Proc. International Workshop on Theoretical Computer Science*, Chartres, 1993.

[12] Debray, S.: QD-Janus: A Sequential Implementation of Janus in Prolog. *Software – Practice and Experience*, Vol. 23, No. 12 (1993), pp. 1337–1360.

[13] Debray, S., Gudeman, D., and Bigot, P.: Detection and Optimization of Suspension-Free Logic Programs. *Journal of Logic Programming*, Vol. 29, Nos. 1–3 (1996), pp. 171–194.

[14] Edwards, S. A.: Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol. 8, No. 2 (2003), pp. 141–187.

[15] Etalle, S., Gabbrielli, M., and Meo, C. M.: Transformations of CCP Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 23, No. 3 (2001), pp. 304–395.

[16] Falaschi, M., Hicks, P., and Winsborough, W.: Demand Transformation Analysis for Concurrent Constraint Programs. *Proc. Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, The MIT Press, Cambridge, MA, pp. 333–347, 1996.

[17] Fradet, P. and Le Métayer, D.: Shape Types. *Proc. Principles of Programming Languages (POPL'97)*, ACM Press, pp. 27–39, 1997.

[18] Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[19] Holzbaur C., Fruehwirth T.: A PROLOG Constraint Handling Rules Compiler and Runtime System. *Applied Artificial Intelligence*, Vol. 14, No. 4 (2000), pp. 369–388.

[20] Kato, N. and Ueda, K.: On the Safety of Moving Synchronization Points in Concurrent Logic Programming Languages. *IPSJ Transactions on Programming*, Vol. 41, No. SIG 2 (PRO 6) (2000), pp. 13–28 (in Japanese).

[21] Kato, N. and Ueda, K.: Sequentiality Analysis for Concurrent Logic Programs. *Proc. the 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002)*, Vol. 11, pp. 329–336, International Institute of Informatics and Systemics (IIIS), Orland, Florida, 2002.

[22] Kato, N., Ueda, K.: Occurs-Check Analysis of Concurrent Logic Programs through Asymptotic Uniform Augmentation of Mode Constraints. *Proc. The 19th JSSST General Conference* (in Japanese), 2002.

[23] King, A. and Soper, P.: Schedule analysis of concurrent logic programs *Proc. Joint International Conference and Symposium on Logic Programming (JICSLP'92)*, The MIT Press, Cambridge, MA, pp. 478–492, 1992.

[24] Lafont, Y.: Interaction Nets. *Proc. Principles of Programming Languages (POPL'90)*, ACM Press, pp. 95–108, 1990.

[25] Leroy, X.: Unboxed objects and polymorphic typing. *Proc. Principles of Programming Languages (POPL'92)*, ACM Press, pp. 177–188. 1992.

[26] Leroy, X.: The effectiveness of type-based unboxing. In Proc. Workshop on Types in Compilation '97, published as Technical report BCCS-97-03, Boston College, Computer Science Department, 1997.

[27] Lowe, M. and Beyer, M.: AGG - An Implementation of Algebraic Graph Rewriting, *Proc. the 5th International Conference on Rewriting Techniques and Applications*, LNCS 690, Springer-Verlag, pp. 451–456, 1993.

[28] Milner, R.: *A Calculus of Communicating Systems*. LNCS 92, Springer-Verlag, 1980.

[29] Milner, R.: Bigraphical Reactive Systems. *Proc. CONCUR 2001*, LNCS 2154, Springer-Verlag, pp. 16–35, 2001.

[30] Montanari, U., Rossi, F., Bueno, F., de la Banda, M. G., and Hermenegildo, M.: Towards a Concurrent Semantics-based Analysis of CC and CLP. *Proc. Principles and Practice of Constraint Programming*, LNCS 874, Springer-Verlag, pp. 151–161, 1994.

[31] Overton, D.: *Precise and expressive mode systems for typed logic programming languages*, Ph. D. thesis, University of Melbourne, 2003.

[32] Saraswat, V. A., Rinard, M. C., and Panangaden, P.: Semantic Foundations of Concurrent Constraint Programming, *Proc. Principles of Programming Languages (POPL'91)*, ACM Press, pp. 333–352, 1991.

[33] Ueda, K. and Furukawa, K.: Transformation Rules for GHC Programs. *Proc. Int. Conf. on Fifth Generation Computer Systems 1988 (FGCS'88)*, ICOT, Tokyo, pp. 582–591, 1988.

[34] Ueda, K. and Morita, M.: Moded Flat GHC and Its Message Oriented Implementation Technique. *New Generation Computing*, Vol. 11, No. 3–4 (1993), pp. 3–43.

[35] Ueda, K. and Tsuchiyama, R.: Optimizing KLIC Generic Objects by Static Analysis. *Proc. 11th International Conference on Applications of Prolog (INAP'98)*, Prolog Association of Japan, pp. 27–33, 1998.

[36] Ueda, K. Concurrent Logic/Constraint Programming: The Next 10 Years. *The Logic Programming Paradigm: A 25-Year Perspective*, K. R. Apt, V. W. Marek, M. Truszczynski, and D. S. Warren (eds.), Springer-Verlag, 1999, pp. 53–71.

[37] Ueda, K.: Linearity Analysis of Concurrent Logic Programs. *Proc. International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, pp. 253–270, 2000.

[38] Ueda, K. and Kato, N.: LMNtal: a language model with links and membranes. *Proc. Fifth Workshop on Membrane Computing*, LNCS 3365, Springer-Verlag, pp. 110–125, 2005.

[39] Van Roy, P.: *Can Logic Programming Execute as Fast as Imperative Programming?* Ph. D. thesis, University of California, 1990. Available as Report No. UCB/CSD 90/600.

[40] Yoshida, Y. and Hennessy, M.: Assigning types to processes. *Proc. Fifteenth Annual IEEE Symposium on Logic in Computer Science*, pp. 334–348, 2000.