Set-based Failure Diagnosis for Concurrent Constraint Programming

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Technischen Fakultät
der Universität des Saarlandes

von

Martin Ludwig Müller

Saarbrücken
Mai 1998

Martin Müller

Forschungsbereich Programmiersysteme

Universität des Saarlandes, 66041 Saarbrücken, Germany

`mmueller@ps.uni-sb.de`

`http://www.ps.uni-sb.de/~mmueller/`

Vorsitzender:	Prof. Dr. Harald Ganzinger
Erstgutachter:	Prof. Dr. Gert Smolka
Zweitgutachter:	Priv.-Doz. Dr. Andreas Podelski
Tag des Kolloquiums:	22. Juli 1998

Kurze Zusammenfassung

Oz ist eine anwendungsnahe Programmiersprache, deren Grundlage eine Erweiterung des Modells nebenläufiger Constraintprogrammierung um Prozeduren höherer Stufe und Zustand ist. Oz ist eine Sprache mit dynamischer Typüberprüfung wie Prolog, Scheme oder Smalltalk. Wir untersuchen zwei Ansätze, statische Typüberprüfung für Oz zu ermöglichen: Mengenbasierte Fehlerdiagnose und Starke Typisierung. Wir definieren ein neues System von Mengenconstraints über Featurebäumen, das für die Analyse von Recordstrukturen geeignet ist, und wir untersuchen das Erfüllbarkeits-, das Leerheits- und das Subsumtionsproblem für dieses Constraintsystem. Wir präsentieren eine mengenbasierte Diagnose für Constraint-Logikprogrammierung und für nebenläufige Constraintprogrammierung als Teilsprachen von Oz, und wir beweisen, daß diese unvermeidliche Laufzeitfehler erkennt. Wir schlagen auch eine mengenbasierte Analyse für eine grössere Teilsprache von Oz vor. Komplementär dazu definieren wir eine Oz-artige Sprache genannt Plain, die ein expressives starkes Typsystem erlaubt. Wir stellen ein solches Typsystem vor und beweisen seine Korrektheit.

Zusammenfassung

Das Modell nebenläufiger Constraintprogrammierung (Concurrent Constraint Model, CC) stellt eine einfache und doch mächtige Grundlage für problemnahe nebenläufige Programmiersprachen dar. Die Expressivität des CC-Modells wird erheblich erweitert durch das Oz Programmiermodell (OPM), welches der Programmiersprache Oz zugrunde liegt. Oz subumiert etablierte Programmierparadigmen wie das der funktionalen, der objekt-orientierten, oder der constraintbasierten Programmierung. Insbesondere verfügt Oz über Ausdrucksmittel zur Programmierung von constraintbasierten Inferenzverfahren, die über alle aus der Constraint-Logikprogrammierung bekannten hinaus gehen.

Oz ist eine Sprache mit dynamischer Typüberprüfung wie Prolog, Scheme oder Smalltalk. Das heißt zum einen, daß Oz eine typsichere Sprache ist, die die typkorrekte Verwendung von primitive Operationen garantiert; es bedeutet andererseits, daß Oz keine statische Typüberprüfung durchführt. Dynamische Typüberprüfung ist von Vorteil für die Einfachheit und Flexibilität einer Programmiersprache, aber es erschwert die Fehlersuche in Programmen. In dieser Arbeit untersuchen wir zwei Ansätze, statische Typüberprüfung für Oz zu ermöglichen: Mengenbasierte Fehlerdiagnose und Starke Typisierung.

Mengenbasierte Fehlerdiagnose ist ein Programmanalyseverfahren, dessen Ziel es ist, Programmierfehler schon zur Übersetzungszeit zu erkennen. Das Verfahren wird als mengenbasiert bezeichnet, weil es eine Klasse prädikatenlogischer Formeln verwendet, die über Mengen von Bäumen interpretiert werden (sogenannte Mengenconstraints). Der Entwurf einer mengenbasierten Programmanalyse verläuft in drei Schritten: Zunächst definiert man eine Klasse von Mengenconstraints, die für die gegebene Programmiersprache und das Analyseproblem angemessen ist. Dann definiert man eine Abbildung von Programmen in diese Mengenconstraints und beweist, daß die Abbildung bestimmte Laufzeiteigenschaften des Programms erhält. Schließlich entwickelt man Algorithmen, um die verwendeten Constraints zu lösen.

Wir definieren ein neues System von Mengenconstraints über Featurebäumen. Dieses Constraintsystem ist durch die Analyse von Records motiviert, die in Oz eine zentrale Rolle spielen, und die in Oz durch Gleichheitsconstraints über Featurebäumen in Oz integriert sind. Wir untersuchen das Erfüllbarkeits-, das Leerheits- und das Subsumtionsproblem für Mengenconstraints über Featurebäumen und präsentieren eine Reihe von Algorithmen und Komplexitätsergebnissen. Mengenconstraints über Featurebäumen sind von unabhängigem Interesse, über ihre Verwendung in der Programmanalyse hinaus und insbesondere im Vergleich mit bekannten Mengenconstraintsystemen.

Wir geben eine mengenbasierte Diagnose an für Constraint-Logikprogrammierung und nebenläufige Constraintprogramming als Fragmente erster Stufe von Oz. Als Korrektheitsbeweis für unsere Diagnose zeigen wir, daß sie nur Programme zurückweist, die einen unvermeidlichen Laufzeitfehler enthalten. Für eine grössere Teil-

sprache von Oz, die insbesondere Prozeduren höherer Stufe mit einschließt, geben wir eine Analyse an und illustrieren sie anhand von Beispielen. Das interessante Korrektheitsproblem für diese Analyse lassen wir offen. Durch die Prozeduren höherer Stufe wird das Korrektheitsproblem wesentlich schwieriger und die Beweistechniken für den Fall erster Stufe sind nicht mehr anwendbar.

Komplementär zu der mengenbasierten Diagnose untersuchen wir den Entwurf eines streng statischen Typsystems für Teilsprachen von Oz. Wir definieren Plain und wir zeigen, daß ein expressives starkes Typsystem möglich ist für eine Sprache, die wesentliche Elemente von Oz kombiniert: darunter Prozeduren höherer Stufe, Logische Variablen und partiell determinierte Datenstrukturen, Zellen und Records. Andererseits heben wir einige Einschränkungen von Plain gegenüber Oz hervor. Plains Typsystem unterstützt Recordtypen, Untertypen, polymorphe Typen höherer Stufe, Modi und Modus-Polymorphismus. Wir beweisen die Korrektheit unseres Typsystems mit Hilfe eines Typerhaltungssatzes (subject reduction).

Short Abstract

Oz is a recent high-level programming language, based on an extension of the concurrent constraint model by higher-order procedures and state. Oz is a dynamically typed language like Prolog, Scheme, or Smalltalk. We investigate two approaches of making static type analysis available for Oz: Set-based failure diagnosis and strong typing. We define a new system of set constraints over feature trees that is appropriate for the analysis of record structures, and we investigate its satisfiability, emptiness, and entailment problem. We present a set-based diagnosis for constraint logic programming and concurrent constraint programming as first-order fragments of Oz, and we prove that it correctly detects inevitable run-time errors. We also propose an analysis for a larger sublanguage of Oz. Complementarily, we define an Oz-style language called Plain that allows an expressive strong type system. We present such a type system and prove its soundness.

Abstract

Concurrent constraint (CC) programming is a simple and powerful high-level model for concurrent programming. The expressiveness of the CC model has been considerably extended by the Oz Programming Model (OPM) which is realised in the programming language Oz. Oz subsumes well-established programming paradigms such as higher-order functional and object-oriented programming, and it supports problem solving facilities beyond those known from constraint logic programming.

Oz is a dynamically typed language like Prolog, Scheme, or Smalltalk. This means that Oz is a type safe language that guarantees type-correctness of primitive operations, but that it lacks static (compile-time) type checking. This is advantageous for simplicity and flexibility of the language but it complicates the debugging of programs. In this thesis we investigate two approaches of making static type checking available for Oz: Set-based failure diagnosis and strong typing.

Set-based failure diagnosis is a method for program analysis with the goal to detect programming errors at compile-time. The method is called set-based because it employs set constraints, a class of predicate logic formulas interpreted over sets of trees. The design of a set-based program analysis involves the following steps. First, one defines a class of set constraints that is appropriate for the given language and the analysis problem. Second, one defines a mapping from programs to set constraints and proves that this mapping preserves certain run-time properties of the programs. Third, one provides algorithms to solve the constraints.

We define a new system of set constraints over feature trees. This constraint system is motivated by the analysis of records, since Oz incorporates records as a central data structure through equality constraints over feature trees. We study the satisfiability, emptiness, and entailment problems for set constraints over feature trees and provide a number of algorithms and complexity results. Set constraints over feature trees are also interesting independent from their application in program analysis, and in comparison with other systems of set constraints.

We present a diagnosis for constraint logic programming and concurrent constraint programming as first-order fragments of Oz. We prove our diagnosis correct by showing that it rejects only programs that contain an inevitable run-time error. For a larger sublanguage of Oz including higher-order procedures we present a diagnosis and illustrate it with examples. The interesting problem of proving correctness for this analysis is left open. In presence of higher-order procedures, the correctness problem becomes fundamentally harder, and the proof techniques used for the first-order case fail.

Complementary to the set-based failure diagnosis, we consider the design of strong static type systems for sublanguages of Oz. We define Plain, and we show that an expressive strong type system is possible for a language that combines key features of Oz, namely higher-order procedures, logic variables and partially determined data structures, cells, and records, and we highlight the restrictions of Plain with respect

to Oz. Plain's type system supports record types, subtyping, higher-order polymorphic types, modes, and mode polymorphism. We prove its soundness through a type preservation theorem.

Für Bettina

Acknowledgements

First and foremost I want to thank my advisor Gert Smolka for initiating and guiding my research in the exciting and demanding areas of programming languages and program analysis. Gert Smolka made the Programming Systems Lab in Saarbrücken a stimulating research research environment, and I am proud that I could be a member of his group during the last five years. I am grateful to my second advisor Andreas Podelski for many discussions and cooperation on constraint solving and program analysis. Andreas was the first to recommend set constraints to me and his hints and comments helped to clarify key issues in this work. This thesis owes a great deal to my colleague, roommate, and friend Joachim Niehren. Joachim is the single person who knows most about the ups and downs during the development of this work. In countless discussions and the writing of several papers he provided mathematical guidance, interest, and encouragement. Thank you, Joachim.

I would also like to thank several people for discussion, collaboration, and support: David N. Turner and Benjamin Pierce for discussions on the π-calculus and about typed concurrent programming; Didier Rémy for an email conversation about issues on types in programming languages; Phillipe Devienne, Jean-Marc Talbot, Sophie Tison, and Marc Tommasi for their invitation to Lille and for many discussions on constraints and on program analysis; Witold Charatonik, Joachim Niehren, Andreas Podelski, Gert Smolka, and Ralf Treinen for their permission to use part of joint work here.

It is my pleasure to thank my colleagues at the Programming Systems Lab: Denys Duchier, Martin Henz, Leif Kornstaedt, Benjamin Lorenz, Michael Mehl, Tobias Müller, Konstantin Popow, Peter Van Roy, Christian Schulte, Ralf Scheidhauer, Joachim Walser, and Jörg Würtz. I enjoyed working with this team of outstanding people which brought up, debated, and solved many exciting research issues. My colleagues were always willing to join in for a beer and a good chat, and some of them have become good friends of mine. Bärbel Hussung deserves special thanks for making the secretary's office a pleasure to visit.

For detailed feedback on earlier drafts I am grateful to Joachim Niehren, Andreas Podelski, Gert Smolka, and Ralf Treinen. Especially Ralf delighted me by commenting on paragraphs in this thesis which I never seriously expected anyone to read. Denys Duchier, Tobias Müller, and Joachim Walser also provided helpful feedback on the presentation of this work.

I am happy that Bettina Glück has been with me all the time. Much of the energy I needed to complete this thesis originates from her. She endured when I worked long enthusiastically writing papers, and she cheered me up in times of frustration. Thank you very much, Bettina, for all your patience and support.

Finally, I would like to thank Maria Hergt for encouragement in the last months of the writing of this thesis.

Financial Support. My thesis work was supported by the Graduiertenkolleg "Kognitionswissenschaften" at the Universität des Saarlandes in which I held a Ph.D. scholarship from 1993 to 1996, and by the Sonderforschungsbereich "Ressourcenadaptive Kognitive Prozesse" (SFB 378) at the Universität des Saarlandes in which I am employed since 1996. I received additional support by the German Research Center for Artificial Intelligence (DFKI), the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie through the Hydra Project (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), the Esprit Working Groups CCL (EP 6028) and CCL II (EP 22457), and the HC&M project CONSOLE (CHRXCT 940495).

Contents

1. Introduction

1.1. Motivation

1.1.1. High-level Programming Languages

Computer programming is a complex task. This complexity can be reduced by a programming language that provides expressive abstraction mechanisms, which enable a direct and concise modelling of the application domain. Programming languages are called high-level if they satisfy this requirement.

The design of high-level programming languages for concurrent and distributed programming is an important challenge in computer science today. Many applications are naturally modelled in terms of multiple concurrent processes which proceed largely independently but also need to synchronise and communicate with each other; as a typical example, consider distributed multi-agent systems. Software developers face a quickly increasing demand for concurrent and distributed applications, especially since the advent of the world wide web.

Concurrent constraint (CC) programming [120, 180] is a simple and powerful high-level model for concurrent programming. The expressiveness of this model has been considerably extended by the Oz Programming Model (OPM) which is realised in the programming language Oz [174, 195]. OPM subsumes well-established programming paradigms as facets of a general model, for example higher-order functional and object-oriented programming. By extension of OPM, Oz also supports problem-solving facilities beyond those known from constraint logic programming [182, 183].

1.1.2. Static Program Analysis

Programming is prone to error. Human beings make errors, both due to the formal activity of writing programs, and due to the intrinsic complexity of the application domain at hand. Some of these errors can be avoided by appropriate programming abstractions which help make programs shorter and easier to maintain. Remaining errors can be very hard to find by testing or program inspection. Therefore, it is desirable to have automated support for the static (compile-time) detection of programming errors.

Oz is a dynamically typed language. This means that Oz is type safe in that all primitive operations check the type of their arguments at run-time, but that Oz lacks static type checking. Dynamic typing is advantageous for simplicity and flexibility of the language but it complicates the debugging of programs. The motivation of this thesis is to make some static type checking available for Oz. For a language like Oz, this problem has not been considered before.

We consider two methods for static analysis for Oz: Strong typing and set-based failure diagnosis. Both methods are somewhat dual to each other: *Strong typing* aims at proving that all operations in a program are always type correct, and to accept only programs for which this proof succeeds. Dually, *failure diagnosis* aims at proving that some operation in a program is not type correct, and to reject such programs as erroneous. Strong typing yields a safety guarantee at the price of restricting the expressiveness of the programming language. Failure diagnosis puts few or no restrictions on the programming language but, as a trade-off, it does not yield a safety guarantee.

1.2. The Programming Language Oz

1.2.1. The Oz Programming Model

Concurrent constraint programming is a model of computation that views concurrent processes as independent agents that communicate by imposing constraints on shared variables [120, 180]. Constraints are bits of information that are accumulated in the constraint store and that restrict the possible values a variable can take: the more constraints, the smaller the set. In Oz, constraints are defined as first-order formulas over a fixed predicate logic structure: we refer to the constraint language and the fixed structure jointly as a *constraint system*.

Concurrent processes synchronise on the fact that certain constraints on a variable become entailed (logically implied) by the constraint store (“ask”). The constraint store grows monotonically: constraints can be added (“tell”) but are never retracted. This setup makes it easy to express complex and safe synchronisation patterns [189] and makes CC a powerful model of concurrent programming.

The Oz Programming Model (OPM) [195] is an extension of the concurrent constraint model. The programming language and system Oz [174] is based on OPM with which it was developed hand in hand. OPM makes two essential additions to CC: It adds higher-order procedures and thus enables functional programming as known for example from Scheme [48]. Second, it adds a cell primitive as a primitive for computation with state. In combination with higher-order procedures, cells enable flexible object-oriented programming in a concurrent setting [91, 194].

By extension of OPM, Oz also supports features for problem solving with constraints so that it subsumes the expressiveness of modern CLP languages like cc(FD) [207].

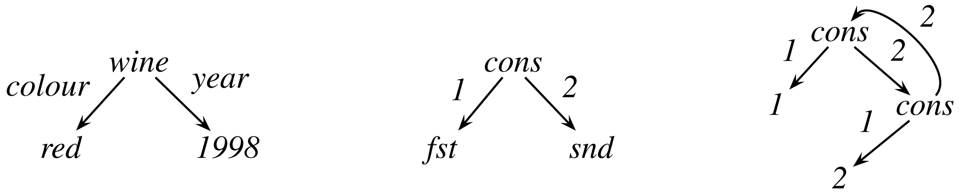


Figure 1.1.: Examples of Feature Trees

These features include primitives for the generation of choice points and for the encapsulation of complete computation states as building blocks for constraint-based inference engines [182–184], and constraint systems over feature trees, finite domain constraints and finite sets of integers [146, 197, 210, 219]. In combination with higher-order procedures and cells, this makes Oz a truly multi-paradigm language , with applications ranging from natural language processing, music composition, time tabling, and the development of graphical user interfaces, to multi-agent systems and distributed programming [93, 95, 174, 209]). In this thesis we focus on OPM without the constraint extensions, and we shall not discuss any distribution issues.

Oz is a dynamically typed language. This is partially due to its heritage from concurrent constraint programming which is based on a traditionally untyped computation model, and also to the initial focus of its developers on the expressiveness of OPM and its new combination of computational primitives. The research presented in this thesis is motivated by the desire to provide some compile-time type checking for Oz, or, more generally, for concurrent constraint programming.

1.2.2. Records and Feature Trees

Records are compound data structures whose components can be accessed by name. This flexibility makes records an important data structure that is supported in many modern programming languages. Record-like structures also have a long tradition in computational linguistics [179, 190] for the analysis of the structure in natural language. *Feature trees* [16, 20, 21, 197] model records: see Figure 1.1 for some typical feature trees. Constraints over feature trees are predicate logic formulas for the description of record structures. This makes them suitable for the incorporation of records into constraint-based languages [197], for example in Oz.

In Oz, constraints over feature trees play a central role, both for the description of records in everyday programming and for constraint programming. Therefore, we take records seriously throughout this thesis. In Oz, records are supported through the constraint system CFT of equality constraints over feature trees [197, 210]. The constraint language of CFT is defined as follows, where the symbols a and f are drawn from sets of *labels* and *features*, respectively.

$$\eta ::= \quad x{=}y \quad | \quad a\langle x\rangle \quad | \quad x\{f_1,\ldots,f_n\} \quad | \quad x[f]y \quad | \quad \eta_1 \wedge \eta_2$$

An equality constraint $x{=}y$ holds if x and y denote the same feature tree; a labelling constraint $a\langle x\rangle$ holds if x denotes a feature tree that is labelled with a at its root; an arity constraint $x\{f_1,\ldots,f_n\}$ holds if the denotation of x has exactly the features (*i. e.*, fields) $f_1,\ldots,f_n$; and a selection constraint $x[f]y$ states that the subtree of x at feature f is y. For example, the leftmost feature tree in Figure 1.1 is uniquely determined by the following constraint (as the solution for x):

$$\mathit{wine}\langle x\rangle \;\wedge\; x[\mathit{colour}]y \;\wedge\; x[\mathit{year}]z \;\wedge x\{\mathit{wine},\mathit{colour}\} \;\wedge$$
$$\mathit{red}\langle y\rangle \;\wedge\; \mathit{1998}\langle z\rangle \;\wedge\; y\{\} \;\wedge z\{\}$$

1.3. Set-based Failure Diagnosis

1.3.1. Set-based Analysis

Set-based failure diagnosis is an instance of *constraint-based* program analysis. This notion refers to a variety of techniques for static program analysis that reduces the reasoning about program properties to the solving of appropriate classes of predicate logic formulas, called *constraints*. *Set-based analysis* is an instance of constraint-based analysis that employs set constraints [85], *i. e.*, predicate logic formulas which are interpreted over sets of trees. Set-based analysis serves to approximate run-time properties of programs statically, for example type information: the set of values a program variable may adopt, or the set of values an expression may evaluate to.

Heintze coined the term “set-based analysis” in his PhD thesis [83]. The history of set-based program analysis dates back to Reynolds in 1969, and Jones and Muchnik in 1979 [110, 176] who applied it to imperative languages, as well as to Mishra in 1984 who analysed logic programming languages [132, 133]. Later, the application of set-based analysis to logic programming was pursued by Heintze and Jaffar [82, 86, 87]. More recently, set-based analysis has been applied to functional languages [12, 38, 65, 84, 121, 216, 218].

The typical setup of constraint-based program analysis consists of the following steps. First, one defines a class of constraints that is appropriate for the given language. Second, one defines a mapping from programs to constraints, for example by traversing the abstract syntax tree of the program and associating constraints to every construct. The conjunction of these constraints is intended as an abstraction of the program properties under consideration. The correctness of this abstraction must be shown in a third step. Finally, the constraint is solved in order to compute a compact representation of the analysis result. Solving a constraint usually means to check *satisfiability* or to compute a *distinguished solution* of the constraint. We apply set-based analysis to concurrent constraint programs [171]. The remainder of this section (Sections 1.3.2 through 1.3.4) and a large part of this thesis (Chapters 2 through 5) is organised according to the three steps mentioned above.

One well-known example for constraint-based program analysis is Wand's formulation of type inference for the simply typed λ-calculus [213]: It derives equality constraints over finite constructor trees from a program, solves them by unification, and accepts a program if the accumulated constraints are all satisfiable: in this case, the constraint associated with a program describes its most general type. The correctness result states that execution of "well-typed programs does not go wrong" [123].

Constraint-based program analysis enjoys a great popularity [5, 88, 157]. This is partly due to its general setup, in which the description of program properties in terms of constraints and the reasoning about them is nicely decoupled: soundness of the abstraction and algorithmic properties of the constraint solving process can be explained and studied separately. More specifically, set constraints are expressive as a formalism, but have a simple and intuitive semantics. In comparison with the general abstract interpretation framework [53] set constraints are often more intuitive (even though abstract interpretation is general enough to express certain set-based analyses [54]).

Theoretical investigations of the various classes of constraints used in program analysis usually focus on the satisfiability problem. More recently, also the *entailment problem* (logic implication) has received some attention [13, 65, 89, 140, 141, 143, 173]. Entailment is interesting in program analysis because it provides explanation for constraint simplification: simplification means to replace a constraint by a smaller one which is either logically equivalent and retains all solutions, or which is entailed and retains the distinguished solution(s). Entailment has also been proposed as a mechanism to explain subtyping on *polymorphic constrained types* [27, 121, 203]. This is relevant to the type checking of module interfaces with polymorphic types.

1.3.2. Set Constraints over Feature Trees

Standard set constraint [85] are interpreted in the domain of sets of finite constructor trees (Herbrand). For the set-based analysis of constraint languages over feature trees (*i. e.*, records) we define a new class of set constraints which are interpreted over sets of feature trees. This system is called $\mathrm{FT}_{\subseteq}(ar, \cup)$ (read "FT-include"). Since Oz allows for infinite (cyclic) records as in Figure 1.1, our constraint system admits sets which contain infinite feature trees. A second important reason for infinite trees is the fact that concurrent programs may be designed for infinite execution and hence are not expected to terminate. For example, the following program scans an infinite stream which it expects to contain feature trees labelled with a or b.[1] Infinite streams can be

[1] In most of the examples to come we use Prolog-style clausal syntax. In examples that rely on higher-order procedures, we switch to a different, roughly Scheme-like notation.

modelled by infinite feature trees.

$$scan(xs) \leftarrow xs[head]x \wedge xs[tail]xr \ \mathsf{then}\ process(x),\ scan(xr)$$

$$process(x_1) \leftarrow a\langle x_1\rangle \ \mathsf{then}\ S_1$$

$$process(x_2) \leftarrow b\langle x_2\rangle \ \mathsf{then}\ S_2$$

The syntax of set constraints over feature trees is defined as follows.

$$\varphi ::= \quad x{\subseteq}x_1{\cup}\ldots{\cup}x_n \quad | \quad x[f]x' \quad | \quad a\langle x\rangle \quad | \quad x\{\overline{f}\} \quad | \quad \varphi_1 \wedge \varphi_2$$

This constraint language is defined like the language of CFT constraints, extended by inclusion constraints of the form $x{\subseteq}x_1{\cup}\ldots{\cup}x_n$. Equations $x{=}y$ can, of course, still be expressed by $x \subseteq y \wedge y \subseteq x$. The semantics is appropriately lifted to the set domain: a labelling constraint $a\langle x\rangle$ holds if x denotes a set of feature trees all of which are labelled with a at the root; an arity constraint $x\{f_1,\ldots,f_n\}$ holds if x denotes a set of feature trees, all of which have exactly the features f_1 through f_n at their root; a selection constraint $x[f]y$ states that x denotes a set of feature trees all of which have a feature f at their root, and the set of all corresponding subtrees equals the denotation of y.[2] Inclusion constraints $x{\subseteq}x_1{\cup}\ldots{\cup}x_n$ are interpreted as usual.

The closest relative of this constraint system is the system of co-definite set constraints of Charatonik and Podelski [44]. Another close relative of set constraints over feature trees is the system $FT_{\leq}$ (read "FT-sub") of ordering constraints over feature trees [143]. For a detailed comparison with related constraint systems see Sections 3.3.1 and 3.3.2.

1.3.3. Solving Set Constraints

We investigate algorithms and complexity issues for various fragments of our set constraints over feature trees. We consider the emptiness problem, *i. e.*, whether or not a variable denotes the empty set in all solutions of a constraint φ, in symbols $\varphi \models x{=}\emptyset$. We show that this problem is DEXPTIME-hard in general and polynomial when union constraints are omitted. We also consider the entailment problems of the form $\varphi \models \varphi'$ and $\varphi \models \exists\overline{x}\varphi'$ for set constraints over feature trees without union constraints. We give an incremental entailment test with polynomial complexity if only the constraints $x[f]y$, $a\langle x\rangle$, and $x{\subseteq}y$ are admitted, we show that entailment becomes coNP-hard when arity constraints $x\{\overline{f}\}$ are added, and that entailment becomes PSPACE-hard when existential quantification is added (even without arity constraints). The entailment problem for the full system remains open.

We also define the system of constraints over non-empty sets of feature trees that is obtained by excluding the empty set from the interpretation domain. We consider this non-standard domain of non-empty sets of trees for two main reasons. On the one hand, our application in program analysis suggests to treat the empty set as an illegal

[2] For discussion on the semantics of the selection constraint see Section 3.3.1.2.

value (the "empty type"). On the other hand, excluding the empty set helps simplify technical arguments: we construct our algorithms for the union-free fragments of set constraints over feature trees by detour through the corresponding systems where the empty set is excluded. We apply techniques that have been developed for systems of tree constraints, in particular for $FT_{\leq}$ [16, 141–143, 145, 197]. We also observe that the first-order theories of *equality constraints* over feature trees and over *non-empty sets* of feature trees coincide.

1.3.4. Set-based Failure Diagnosis for Concurrent Constraint Programming

We apply set constraints over feature trees to the analysis of constraint programs over feature trees. The objective of the analysis is to detect programming errors, in particular such errors which inevitably lead to a run-time error. This choice will become clear below. We consider three Oz-style languages of increasing complexity: First, a language corresponding to constraint logic programming (CLP), then a concurrent constraint (CC) programming language, and finally OPM. All three languages support records through CFT constraints [197].

In the CC model as well as in OPM, an inconsistent constraint store is considered a programming error. This is in contrast to traditional (constraint) logic programming where failure is part of the backtracking mechanism. A CC program has certainly an error if every fair execution leads to failure. Furthermore, our programming experience with Oz indicates that we should also consider a program erroneous that does not fail *only because* some application blocks forever.

1.3.4.1. Constraint Logic Programming

The basic idea of our failure diagnosis is illustrated by the following CLP program.

$$\begin{aligned} & p(x) \leftarrow a\langle x\rangle \\ & q(y) \leftarrow b\langle y\rangle \\ & r(z) \leftarrow p(z),\ q(z) \end{aligned} \tag{P_1}$$

Whenever the procedure (or predicate) r is called, execution will eventually fail since no feature tree can be labelled with both a and b. In CLP terminology, the call $r(z)$ is *finitely failed*. (Since we focus on concurrent programming without backtracking, we favour the view that the procedure r contains an inevitable failure.) We detect this failure as follows. We derive from the program the set constraint

$$\begin{aligned}
&p \subseteq x \ \wedge \ a\langle x\rangle \ \wedge \\
&q \subseteq y \ \wedge \ b\langle y\rangle \ \wedge \\
&r \subseteq z \ \wedge \ z \subseteq p \ \wedge \ z \subseteq q
\end{aligned}$$

and observe that it entails $r=\emptyset$. From this fact we deduce that $r(z)$ is finitely failed for all z and reject the program. A similar but slightly more complex program is this one:

$$\begin{aligned}
&p(x) \leftarrow x[f]x,\ a\langle x\rangle \\
&q(y) \leftarrow y[f]y,\ b\langle y\rangle \\
&r(z) \leftarrow p(z),\ q(z)
\end{aligned} \tag{P_2}$$

The set constraint associated with this program is the following one.

$$\begin{aligned}
&p \subseteq x \ \wedge \ x[f]x' \ \wedge \ a\langle x'\rangle \ \wedge \\
&q \subseteq y \ \wedge \ y[f]y' \ \wedge \ b\langle y'\rangle \ \wedge \\
&r \subseteq z \ \wedge \ z \subseteq p \ \wedge \ z \subseteq q
\end{aligned}$$

Again, this analysis entails $r = \emptyset$. This crucially exploits the fact that the semantics of $\sigma[f]\sigma'$ requires all trees in σ to have the feature f. If $x[f]y$ had only projection semantics, this analysis had a non-empty solution for r (see Example D_{fail3} on Page 94).

1.3.4.2. Concurrent Constraint Programming

The essential difference between CLP programs and CC programs is that the latter may have guarded clauses $p(x) \leftarrow \eta$ then S. Roughly, such a clause tests whether the constraint η or its negation holds for the argument z of an application $p(z)$. If η holds, execution of $p(z)$ can commit to this clause and proceed with S. If $\neg\eta$ holds, then this clause becomes irrelevant for execution of $p(z)$. Otherwise, the clause is said to block.

We define the analysis of CC programs through the analysis of an approximating CLP program. This CLP program is obtained from the CC program by transforming conditional guards into tell statements. For example, the CC program P_3 below is approximated by the program P_2 above. Intuitively, this approximation ignores the synchronisation behaviour of guards.

$$\begin{aligned}
&p(x) \leftarrow x[f]x \ \mathsf{then}\ a\langle x\rangle \\
&q(y) \leftarrow y[f]y \ \mathsf{then}\ b\langle y\rangle \\
&r(z) \leftarrow p(z),\ q(z)
\end{aligned} \tag{P_3}$$

The interpretation of the analysis result needs more care now due to the possibility that a guarded clause blocks. For instance, if $r(z)$ is called on a free variable z, then the clauses of p and q both block forever: the constraint store will not accumulate any information on z and hence will never entail $z[f]z$ or its negation. In general, we cannot statically exclude this possibility, and hence we obtain a weaker correctness result for CC or CLP: from the fact that the analysis entails $r{=}\emptyset$ we deduce that every application of r is finitely failed or blocks forever.

1.3.4.3. Oz: Higher-order Concurrent Constraint Programming

OPM extends CC to a higher-order programming language: procedures are first-class data structures that can be passed as arguments to a procedure and returned as a result. For example, consider the following statement:[3]

$$(\mathsf{proc}\; x\; (y,z)\; (\mathsf{proc}\; z\; (u)\; u[tag]y)) \qquad (P_4)$$

This program binds x to a binary procedure with formal arguments y and z. Application of this procedure returns a unary procedure in the second argument. This second procedure asserts that its unique argument u is a feature tree with feature *tag* leading to the first argument y of the procedure x. We analyse P_4 as follows.[4]

$$\varphi_{P_4} = \quad x{\subseteq}\mathsf{proc}(arg_1{:}y,\ arg_2{:}z)\ \wedge\ z{\subseteq}\mathsf{proc}(arg_1{:}u)\ \wedge\ u[tag]y$$

An application of the procedure x to two variables v and w, followed by an application of w to x will lead to a failure due to the assertion that the *procedure* x has a *feature tag*:

$$P_4 \parallel x(v,w) \parallel w(x) \qquad \longrightarrow \qquad (\mathsf{proc}\; x\; (y,z)\; \ldots) \parallel x[tag]v \parallel \ldots$$

We analyse this statement as follows.

$$\begin{aligned}\varphi_{P_4}\ \wedge\ &\mathsf{proc}\langle x\rangle\ \wedge\ x[arg_1]x_1\ \wedge\ x[arg_2]x_2\ \wedge\ v{\subseteq}x_1\ \wedge\ w{\subseteq}x_2\ \wedge\\ &\mathsf{proc}\langle w\rangle\ \wedge\ w[arg_1]w_1\ \wedge\ x{\subseteq}w_1\end{aligned}$$

This constraint entails that $x = \emptyset$ (because there exists no feature tree that has exactly the features arg_1 and arg_2 but at the same time the feature *tag*). We conclude that the statement $P_4 \parallel x(v,w) \parallel w(x)$ will inevitably fail and reject it.

Our analysis is weak with respect to the analysis of higher-order procedures as the following example illustrates. The procedure

$$(\mathsf{proc}\; x\; (y,z)\; (y\;\, z)) \qquad (P_5)$$

[3] We discuss our analysis for CLP and CC based on a Prolog-style clausal syntax of programs. For OPM we switch to a Scheme-style syntax that is more convenient to deal with higher-order programming. Embedding CC programs into OPM is straightforward.

[4] The term notation is used for conciseness here, as an abbreviation for a set constraint over feature trees. For example, $x{\subseteq}\mathsf{proc}(arg_1{:}y,\ arg_2{:}z)$ abbreviates the constraint $x \subseteq x' \wedge \mathsf{proc}\langle x'\rangle \wedge x'[arg1]y \wedge x'[arg2]z \wedge x'\{arg1, arg2\}$ for a fresh variable x'. For the formal definition of this notation see Chapter 2.

that applies its first argument y to its second argument z is analysed by the constraint

$$x \subseteq \mathsf{proc}(arg_1{:}y,\ arg_2{:}z) \wedge y[arg_1]y' \ \wedge\ \mathsf{proc}\langle y\rangle \ \wedge\ y\{arg_1\} \ \wedge\ z \subseteq y'$$

which reveals that y must be a unary procedure (expressed by the constraint $y\{arg_1\}$). Now assuming a binary procedure *inc* on integers whose analysis is

$$\varphi_{inc} = \quad inc \subseteq \mathsf{proc}(arg_1{:}\mathsf{int},\ arg_2{:}\mathsf{int})\,,$$

and consider the following statement:

$$P_5 \parallel (x\ inc\ v) \tag{P_6}$$

Our analysis of this program in conjunction with φ_{inc} will not entail $v \subseteq \mathsf{int}$, and so not find out that v must be an integer: the relationship between the formal arguments y and z, and hence between the actual ones *inc* and v, is lost here. The reason is that we only propagate information *from* the formal arguments *to* the actual arguments of procedures, and not vice versa. In principle, this order could be inverted for input arguments. However, the mode (*i. e.*, input or output) of procedural arguments is *not syntactically apparent* in constraint programs, much in contrast to functional languages: unification and constraint solving allow for data flow in both directions, and all procedural arguments can, in principle, be input, output, or both. A static *mode analysis*, however, seems to requires a full-fledged control-flow analysis for Oz [191], which is out of scope of this thesis.

On the other hand, the accuracy of the analysis can be easily improved by annotating procedures with type information. Such annotations can be fit nicely in the constraint framework, when modelled as prescriptive constraints that a program must satisfy in addition to the constraints derived by descriptive means. For example, the constraint φ_{inc} above can originate from the analysis of another statement as well as from an explicit type annotation on *inc*.

Our analysis for OPM is a reasonable extension of our analysis for CC. The correctness problem for this analysis, however, is harder than in the first-order case due to the lack of a denotational semantics for OPM. We leave the correctness problem open; instead, we illustrate our method with examples and provide style conventions that summarise the intuitions underlying it. The analysis for Oz has been implemented in an experimental prototype with an incomplete constraint solver. The feasibility of the analysis in a development system remains to be explored.

1.4. Strong Static Typing for Oz

Complementary to set-based failure diagnosis, we investigate the possibility of designing an OPM-style language that has a static type system similar to functional languages like ML or Haskell [130, 162].

1.4.1. Static versus Dynamic Typing

A *type* is a set of data objects with a common structure that allow the same set of operations. Typical types are the set of all integers, and the set of all records that have a field named *address*. The attempt to apply an operation to a data structure of inappropriate type (such as the attempt to multiply a record by 2 or to select the field *address* from the integer 42) is called a *type error*. A programming language is called *type safe* if it is checked whether or not the primitive operations are applied to arguments of proper types, and if their behaviour is well-defined even if the types are not the expected ones: Thus type errors are detected at least at run-time. Many modern programming languages are type safe in this sense. In unsafe languages, such as assembly languages or C [112], programs may behave randomly after a type error.

A programming language is commonly called *statically* or *dynamically typed* depending on whether (most of) the *type checking* is done at compile time or at run time. A language is called *strongly typed* if *all* type checking is done at compile-time so that the run-time system can safely ignore types.[5] Typical examples for statically typed and type safe languages are the functional languages SML and Haskell, the concurrent language Pict, the imperative language Modula-3, and the object-oriented language Java [35, 78, 130, 162, 169]. Amongst the dynamically typed languages there are the logic programming language Prolog, the functional language Scheme, the object-oriented language Smalltalk, and the concurrent languages Erlang and Oz [17, 48, 77, 195, 200].

Strong typing is usually formalised in two steps. First, one determines the run-time situations that one wants to exclude as *type errors*. Second, one defines a *type system* consisting of a language of *type expressions* and a set of *typing rules*. Type expressions describe run-time *invariants* of a program such as "the identifier *x* always refers to an integer". At compile-time, every relevant program phrase (identifiers, terms, expressions, statements, *etc.*) is assigned a type, either automatically or according to explicit program annotations. A *type checker* tries to verify the corresponding invariants using the typing rules, and a successful proof guarantees the impossibility of type errors.

A disadvantage of strong typing is the additional level of complexity that a type system adds to a language. One source of complexity is simply the formal language of type expressions which must be mastered by programmers to provide type declarations and to understand error messages. Automated type inference as in SML [55, 123] alleviates this problem because it relieves the programmer from many type declarations. Of course, the programmer must still understand the type language.

To date, type systems for expressive object-oriented programming languages are fairly complicated and require many type declarations (see, *e. g.*, [1] for recent references). Another problem is the fact that it is impossible to define a type checker that terminates

[5] Since all strongly typed languages in this sense are statically typed, and many statically typed languages are strongly typed, static and strong typing are often used synonymously.

on all programs and accepts exactly those programs that will never exhibit a run-time type error (the absence of run-time type errors is an undecidable problem). As a trade-off, a strong type system enforces a decidable discipline that must reject many type correct programs and hence restricts the expressiveness of programming language.

The case is dual for dynamically typed languages. Their main strength is the great flexibility with which they support the encoding of high-level programming abstractions, based on only a small set of simple primitives. This makes dynamically typed languages ideal as platforms for rapid prototyping and teaching of programming concepts (see, for instance, the text books based on Scheme [2, 68]). On the other hand, tracing down a programming error can be a lengthy undertaking and can make the absence of compile-time type checking painfully apparent.

1.4.2. Types for Oz

As a dynamically typed language, Oz freely supports features that would complicate strong typing. We focus on the combination of features that sets Oz apart from its relatives, namely logic variables introduced by explicit declaration, first-class procedures, constraints, and parallel composition. In contrast to constraint logic programming, Oz has higher-order procedures and explicitly declared logic variables; in contrast to functional programming, Oz has logic variables and constraints; and in contrast to the π-calculus [129], communication and synchronisation in Oz is through shared logic variables instead of message exchange over channels (see also Section 5.4.1).

It is straightforward to devise for Oz a *monomorphic* type system, which assigns exactly one type to every identifier. It is also possible to adapt an ML-style *polymorphic* type system [55, 123], which assigns type schemes to certain procedures. This requires some more care but works if one follows Wright [217] for the interaction of polymorphism and logic variables (see Section 6.3.2). ML-polymorphism is too weak, however, to type check a number of programs that we found important in the programming practice of Oz. For instance, polymorphic procedures (or objects with polymorphic methods) cannot be placed in a cell, assigned to an object's state variable, or sent along a channel. This considerably restricts the flexibility of object-oriented programming and the communication patterns in a concurrent or distributed language. Secondly, ML-polymorphism cannot type check many convenient higher-order or object-oriented programming abstractions (see, *e. g.*, [101, 166]).[6]

Therefore, we consider a type system with universal *higher-order polymorphic* types [76, 177]. We also assume a *subtyping* order on types: type systems with subtyping allow operations defined on a type T to be applied to all objects whose type refines T

[6] This is not to say that no object-oriented programming at all is possible with ML-polymorphism. O'Caml [175] is a language that supports object-oriented programming with ML-style polymorphic types, but it requires all methods to have a monomorphic types. Also notice that Haskell's type classes [80] can express some form of inheritance.

("subtypes" of T). Subtyping is, *e. g.*, used to type check the assignment of an integer value to a variable of type number (given that "integer" is a subtype of "number"). The combination of higher-order polymorphic types with subtyping [32, 36] is especially convenient in object-oriented programming. For instance, a procedure that receives an object and sends it the message m should have a type that admits as argument *every* object implementing the method m. Subtyping is a flexible way to achieve this.

As these examples indicate, one must statically know the data flow to make use of polymorphism and subtyping ("*assign* subtypes to supertypes", "*use* functions at subtype", "specialise *input* arguments before application"). In (pure) functional languages the data flow is given by the syntactic structure of the program. In constraint (logic) programming and in Oz this is not the case. Unification and constraint solving have a *bidirectional* nature, and procedural arguments can be input, output, or both. The need to statically know the data flow in programs lead us to the definition of Plain, a language with higher-order procedures, cells, records, and pattern matching (see Chapter 6). The key change in which Plain differs from Oz is that the equality constraint on variables is replaced by a *(single) assignment* statement

$$x := y\,.$$

Execution of $x := y$ does not *unify* the current bindings of x and y but blocks until y is bound to some data structure and then *binds* x to the same data structure, too. This is a considerable restriction of Oz as a constraint programming language, in particular with respect to feature tree constraints. But Plain still admits computation with partial information; for example, through records with embedded logic variables.

$$(\mathsf{local}\ (y,\ z)\ x := \{hd{:}y,\ tl{:}z\} \parallel \ldots)$$

Like in Oz, Plain's procedures do not statically distinguish between input and output arguments. So the type system must enforce a strict static *mode discipline* in presence of higher-order polymorphism and subtyping. To this end, we adapt Pierce and Sangiorgi's mode system for channels [165] to a language with logic variables. We do not consider the type inference problem for Plain, which is very likely to be undecidable [214].

1.4.3. Failure Diagnosis versus Strong Typing

The two methods for program analysis, strong typing and set-based failure diagnosis, are roughly dual to each other. *Strong typing* aims at proving that all operations in a program are always type correct, and it accepts only programs for which this proof succeeds. It is desirable to accept as many type correct programs as possible, but it is absolutely necessary to not accept a single type incorrect program. Dually, *failure diagnosis* aims at proving that some operation in a program is not type correct, and it rejects such programs as erroneous. It is desirable to detect as many type errors as

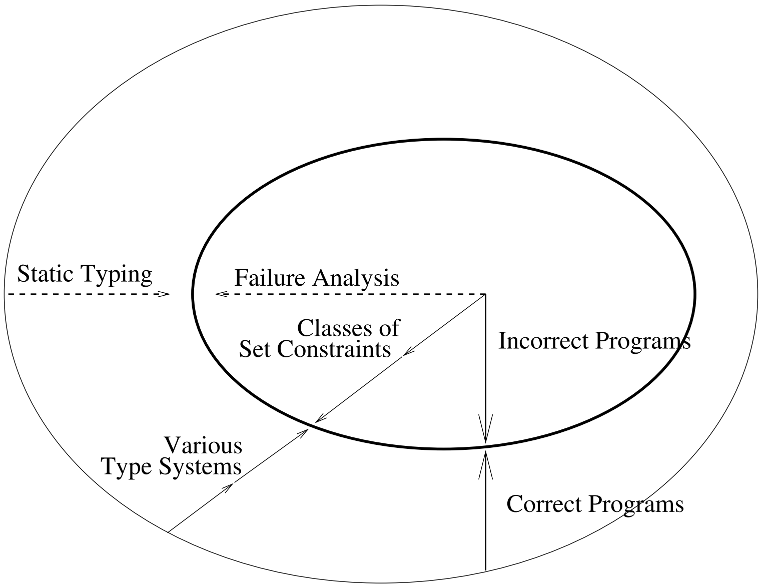


Figure 1.2.: Failure Diagnosis versus Strong Typing

possible, but ideally no type correct program is rejected. The extreme cases are given by a strong type system that accepts no program at all, and a failure diagnosis system that accepts all programs.

Figure 1.2 illustrates this point of view, and it also gives a pictorial summary of the material presented in this thesis. The complete oval represents all programs in a given programming language. Assuming a fixed notion of type errors, the fat line separates the sets of programs that have a type error from those that do not. As mentioned above, the set of type correct programs is undecidable so that some approximation is fundamentally needed: strong typing approximates the set of type correct programs, while failure diagnosis approximates the set of type incorrect programs.[7]

A strong type system accepts the more type correct programs the more expressive it is: this dimension is indicated by the arrow superscripted "Various Type Systems". Higher expressiveness usually comes with more complicated type expressions and typing rules and with more expensive type checking problem. *Monomorphic* type systems (as known, *e. g.*, from Pascal) are fairly inexpressive and often not satisfactory in practice. More expressive type systems are obtained by adding different forms of *polymorphism*, for instance *parametric polymorphism* as in ML [55, 123], *subtype polymorphism* as studied in object-oriented programming languages [31], or mixed

[7]This duality does not, in general, withstand formal scrutiny. This is mainly due to the fact that strong typing and failure diagnosis do not talk about the same class of type errors. First, the precise dual of being provably free of type errors is *possibly* containing an error, whereas our failure diagnosis for CC checks whether a program *inevitably* contains an error.

forms [32, 36]. A major goal of research in this area is to find expressive type systems with a decidable, hopefully efficiently decidable, type checking problem whose type expressions remain intelligible to programmers.

Analogously, failure diagnosis detects the more type errors the more accurately it is can describe the run time behaviour of programs. In set-based program analysis this is the case the finer-grained and the more expressive the set description language is chosen. The expressiveness of such a language depends on the choice of set operators provided (*e. g.*, union, intersection, projection, complementation). Again, highly expressive set description languages can become very complex and expensive to deal with. For example, the satisfiability problem for standard set constraints including all mentioned set operators is NEXPTIME-complete [10, 19, 40]. One major goal of research in set-based analysis is to find constraint systems which are expressive but can be efficiently solved. Preferably, some application-relevant problems like entailment should be efficiently decidable, too.

1.5. Contributions

1.5.1. Summary

The underlying motivation of the research reported in this thesis is the question:

> *How can we provide some static type checking for the dynamically typed language Oz or, more generally, for a concurrent constraint language with higher-order procedures?*

We tackle this question from the complementary points of view given by strong typing and failure diagnosis. Our contributions to these two areas correspond roughly to the two programming paradigms that are most closely related to Oz, namely the paradigms of functional and logic programming. Notice that Oz actually subsumes both of them.

Strong typing for languages with higher-order procedures has been studied extensively in the context of functional programming languages and is applied very successfully there. So a natural rephrasing of the search for a strong type system for Oz is:

> *How can we adapt strong type systems developed for functional language to a language based on logic variables?*

The language Plain that we design in Chapter 6 answers this question for an expressive type system with higher-order polymorphism and subtyping.

A main focus of the research in set-based analysis was on logic programming languages. Since the logic programming tradition has had a major impact on the development of Oz and since (constraint) logic programming is an important sublanguage of Oz, it is reasonable to ask:

> *How can we adapt set-based analysis techniques from logic programming languages for a failure diagnosis of Oz?*

There are several aspects to this question: (i) Which constraint system is appropriate for an analysis of Oz? (ii) How does a (set-based) failure diagnosis look for first-order fragments of Oz, and how can one generalise the diagnosis to a language with higher-order procedures? (iii) What kind of correctness result is obtained and how can one prove it?

As an answer to (i), we propose a new constraint system over sets of possibly infinite feature trees and analyse it in detail. This investigation constitutes a large part of this thesis. Answering (ii), we define an analysis for first-order concurrent constraint programming, and we extend it to OPM. (iii) As a correctness result, we show that the analysis for CLP detects finite failure, and that the analysis for CC detects finite failure unless an application blocks forever.

1.5.2. Technical Summary

In this section we summarise the technical contributions of the thesis.

1.5.2.1. Set Constraints over Feature Trees

We introduce a set constraint system $\mathrm{FT}_{\subseteq}(ar, \cup)$ (read "FT-include") with three distinguishing properties that distinguishes it from the set constraints that are usually considered in the literature.

- Constraints are interpreted over sets of feature trees, instead of constructor trees as usual. This makes our constraint system suitable for the analysis of records in programming languages, and for the analysis of feature tree constraints in constraint programming. The analysis of tuples as a special case of records remains possible.
- Constraints are interpreted over sets of infinite trees, instead of finite ones as usual. This is necessary for the analysis of infinite data structures as they are common in constraint logic programming. It is also needed to establish a relation between the denotational and the operational semantics of constraint logic programs with possibly non-terminating computations.
- Every constraint is satisfiable and has a greatest solution: This makes the constraint system appropriate for the analysis of concurrent programs that specify infinite computations. Our constraint system shares this property with the co-definite set constraints [44], which can be embedded into set constraints over feature trees such that emptiness in the greatest solution is preserved.

We also consider a non-standard system, called $\mathrm{FT}^{ne}_{\subseteq}(ar, \cup)$ and read "FT-include-nonempty", of constraints over *non-empty sets of feature trees*. Our motivation is threefold:

- The investigation of constraints over non-empty sets is a conceptual contribution which credits the central role that emptiness plays in the solving of set constraints and in set-based program analysis.
- Constraints over non-empty sets can help to simplify technical arguments. We consider both constraint solving and entailment first for fragments of $\mathrm{FT}^{ne}_{\subseteq}(ar, \cup)$ and derive the related results for $\mathrm{FT}_{\subseteq}(ar, \cup)$ from them. Charatonik and Podelski have proven decidability for set constraints with intersection by detour through set constraints over non-empty sets [42].
- By the exclusion of the empty set, we establish a close relationship between constraints over trees and constraints over non-empty sets of trees. In particular, the first-order theories of equality constraints over both domains coincide.

1.5.2.2. Solving Set Constraints

We consider fragments of the system $\mathrm{FT}_{\subseteq}(ar,\cup)$ with restricted constraint languages, denoted by $\mathrm{FT}_{\subseteq}(ar)$, $\mathrm{FT}_{\subseteq}(\cup)$, and $\mathrm{FT}_{\subseteq}$, and we provide algorithms and complexity results for these as well as for the corresponding fragments of $\mathrm{FT}^{ne}_{\subseteq}(ar,\cup)$.

- We investigate incremental constraint solving which is important for modular program analysis.
- We show that the satisfiability problem for $\mathrm{FT}^{ne}_{\subseteq}(ar)$ (no union constraints) can be solved in incremental cubic time and provide an appropriate algorithm. By extension of the satisfiability test for $\mathrm{FT}^{ne}_{\subseteq}$ we obtain an incremental algorithm to compute the greatest solution of an $\mathrm{FT}_{\subseteq}(ar)$ constraint and to decide emptiness. The algorithm is shown to have polynomial complexity of degree 4. We observe that the satisfiability problem for $\mathrm{FT}^{ne}_{\subseteq}(ar,\cup)$ and the emptiness problem for $\mathrm{FT}_{\subseteq}(ar,\cup)$ are DEXPTIME-hard, and we conjecture that a DEXPTIME algorithm can be derived from the literature.
- We give an incremental algorithm that solves the satisfiability problem for positive and negative $\mathrm{FT}^{ne}_{\subseteq}$ constraints (neither union nor arity constraints) in cubic time; this implies that also the entailment problem is solvable in cubic time. The proof relies on the independence property of this constraint system which we show. We apply the result for $\mathrm{FT}^{ne}_{\subseteq}$ to prove that the entailment problem for $\mathrm{FT}_{\subseteq}$ can be solved by an incremental algorithm in time $O(n^4)$.
- We show that the entailment problem of set constraints over feature trees becomes coNP-hard when arity constraints are added, and that it becomes even PSPACE-hard when existential quantifiers are added. Both hardness results carry over to $\mathrm{FT}_{\subseteq}(ar)$: the entailment problem for $\mathrm{FT}_{\subseteq}(ar)$ is coNP-hard, and the entailment problem for $\mathrm{FT}_{\subseteq}$ with existential quantifiers is PSPACE-hard.

All results hold independent of whether the constraints are interpreted over finite or infinite trees.

1.5.2.3. Set-based Failure Diagnosis for Oz

We present a method for automated set-based failure diagnosis for concurrent constraint programs over feature trees (*i. e.*, records) in terms of set constraints over feature trees.

- To date, set-based analysis for constraint (logic) programs has focussed on the least-model semantics of terminating programs. Since we are interested in possibly non-terminating computations, we consider the greatest model semantics.

We show that our analysis safely approximates the greatest model of constraint logic and concurrent constraint programs.

- We relate the greatest model semantics of constraint logic programs over infinite trees to finite failure. We conclude that our analysis safely approximates the inevitability of failure for constraint logic programs, and that it approximates the inevitability of failure for concurrent constraint programs unless an application blocks forever.
- We also discuss generalisation of our correctness result to larger fragments of Oz. For a large part of Oz we present a set-based analysis in terms of set constraints over feature trees and we give examples to illustrate its appropriateness.

1.5.2.4. Strong Typing for Logic Variables

We define a sublanguage of Oz called Plain to which standard strong type systems known from functional programming can be applied.

- We give a strong type system with record subtyping, universal higher-order polymorphism, and mode polymorphism, and we prove a type preservation and a type safety result.
- Plain pinpoints some aspects in the definition of OPM which complicate strong typing, and it marks a starting point from which strongly typed OPM-style languages can be developed.
- Plain's expressiveness is comparable to that of Pict, a recent concurrent language based on the π-calculus. Thereby, Plain contributes to relating two prominent concurrent programming models: concurrent constraints and process calculi.

1.6. Publication Remarks

Several of the results presented in this thesis have been obtained in collaboration with my colleagues Joachim Niehren, Witold Charatonik, Andreas Podelski, and Gert Smolka, or are influenced by this collaboration. Some of them have partly been published before. This section lists the relevant papers and reports. I would like to thank my co-authors for the permission to use part of the material therein.

Some of our results for set constraints over feature trees have been developed for ordering constraints over feature trees. We adapt them to a set constraint system that is more flexibly applied to program analysis.

1. MÜLLER, MARTIN, JOACHIM NIEHREN, & ANDREAS PODELSKI (1997). Ordering constraints over feature trees. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming* (CP'97), edited by G. Smolka, vol. 1330 of *Lecture Notes in Computer Science*, pp. 297–311, Schloß Hagenberg, Linz, Austria. Springer-Verlag, Berlin.

This paper defines and investigates the constraint system $\mathrm{FT}_{\leq}$ (read "FT-sub") of ordering constraints over feature trees and presents algorithms for deciding satisfiability and entailment in cubic time. The corresponding results for $\mathrm{FT}^{ne}_{\subseteq}(ar)$ and $\mathrm{FT}^{ne}_{\subseteq}$ (Theorems 3 and 12 on Pages 34 and 54) have been adapted from this paper.

2. MÜLLER, MARTIN, JOACHIM NIEHREN, & ANDREAS PODELSKI (1997). Inclusion constraints over non-empty sets of trees. In *Theory and Practice of Software Development* (TAPSOFT'97), edited by M. Bidoit & M. Dauchet, vol. 1214 of *Lecture Notes in Computer Science*, pp. 345–356, Lille, France. Springer-Verlag, Berlin.

This paper defines and investigates the constraint system Ines of inclusion constraints over non-empty sets of constructor trees. It was the first to investigate set constraints over non-empty sets. It also contains the observation that the first-order theories of equality constraints over infinite constructor trees and over non-empty sets of infinite constructor trees coincide (corresponding to Theorem 23 on Page 87). The satisfiability test for $\mathrm{FT}_{\leq}$ was inspired by the one for Ines. In particular, the detailed complexity analysis of the satisfiability test for $\mathrm{FT}^{ne}_{\subseteq}(ar)$ is adapted from there (see Section 2.2.3 on Page 39).

3. MÜLLER, MARTIN & JOACHIM NIEHREN (1997). Entailment for set constraints is not feasible. Tech. rep., Programming Systems Lab, Universität des Saarlandes.

This report proves that entailment for Ines constraints (as well as for atomic set constraints [85]) is coNP-hard. The coNP-hardness result on entailment for $\mathrm{FT}^{ne}_{\subseteq}(ar)$ (Theorem 16 on Page 62) has its origins here.

4. MÜLLER, MARTIN & JOACHIM NIEHREN (1998). Ordering constraints over feature trees expressed in second-order monadic logic. In *International Conference on Rewriting Techniques and Applications* (RTA'98), edited by T. Nipkow, vol. 1379 of *Lecture Notes in Computer Science*, pp. 196–210, Tsukuba, Japan. Springer-Verlag, Berlin.

5. MÜLLER, MARTIN, JOACHIM NIEHREN, & RALF TREINEN (1998). The first-order theory of ordering constraints over feature trees. In *Proceedings of the 13th IEEE Symposium on Logic in Computer Science* (LICS'98) . IEEE Computer Society Press. To appear.

Both papers investigate, amongst other issues, the entailment problem with existential quantifiers for the constraint system $\mathrm{FT}_{\leq}$. Paper (5) show this problem to be coNP-hard in the case of finite trees and PSPACE-hard in case of infinite trees. The paper (4) shows that the hardness proof of paper (5) can be transformed such that it applies to both the case of finite and infinite trees. The PSPACE-hardness results for $\mathrm{FT}^{ne}_{\subseteq}$ and $\mathrm{FT}_{\subseteq}$ with existential quantifiers (Theorem 20 on Page 74) have been adapted from there.

6. PODELSKI, ANDREAS, WITOLD CHARATONIK, & MARTIN MÜLLER (1998). Set-based error diagnosis of concurrent constraint programs. Tech. rep., Programming Systems Lab, Universität des Saarlandes.

This paper gives an analysis for concurrent constraint programs over infinite constructor trees and shows that the analysis safely approximates the greatest model of the program. It observes that finite failure can be characterised by the greatest model semantics and so proves that the analysis correctly approximates a run-time error in concurrent constraint programs. To a large extent, Chapter 4 is based on this paper.

7. MÜLLER, MARTIN (1996). Polymorphic types for concurrent constraints. Tech. rep., Programming Systems Lab, Universität des Saarlandes.

8. MÜLLER, MARTIN, JOACHIM NIEHREN, & GERT SMOLKA (1998). Typed concurrent programming with logic variables. Tech. rep., Programming Systems Lab, Universität des Saarlandes.

The second report defines the language Plain along with a type system with higher-order polymorphic types, modes and subtypes, and is the basis for Chapter 6. A main insight underlying Plain is the fact that a type system with higher-order polymorphism and subtyping only works in presence of static data flow information; this insight was formulated earlier in the first report.

1.7. Overview

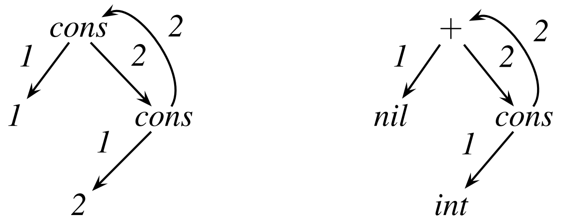
In Chapter 2 we introduce feature tree constraints to model records in constraint programming, and we define set constraints over feature trees. We also investigate the satisfiability and the emptiness problem for fragments of this system. In Chapter 3 we consider the entailment problem in addition. Chapter 4 defines a set-based failure diagnosis for constraint logic programming and concurrent constraint programming in terms of set constraints over feature trees. Chapter 5 generalises the failure diagnosis to higher-order procedures, and hence to OPM. Chapter 6 complements the work on set-based failure diagnosis by designing an Oz-style language with higher-order procedures and logic variables with a static type system. Chapter 7 assesses the achievements of this thesis and outlines some directions of future research. Appendix A introduces some basic mathematical concept and notation.

We imagine three paths through this thesis. The reader interested in set constraints and their formal properties should read Chapters 2 and 3. The reader interested in set-based program analysis should read the definitional parts of Chapter 2, and then proceed to Chapters 4 and 5. The reader interested in typed concurrency with logic variables can read Chapter 6 independently. To understand the comparison of Plain with OPM, the reader may want to read the introduction to OPM in Chapter 5. All paths may end in Chapter 7 on future work.

2. Set Constraints over Feature Trees

Feature trees [16, 20, 21, 197] are a kind of trees which is appropriate for the description of record-like structures. The picture below shows two typical feature trees.

A feature tree is a possibly infinite tree with unordered marked edges and with marked nodes. Edge labels (called *features*) are functional in that the edges departing from the same node must be pairwise distinct. The node marks are called *labels*. The feature trees above mention the features *colour*, *year*, *1*, and *2*, and the labels *wine*, *red*, *1998*, *cons*, *fst*, and *snd*. Feature trees are more general than constructor trees: by using as features consecutive positive natural numbers starting from *1*, constructor trees can be modelled as feature trees (see the second tree above). Feature trees may be *infinite*. Some of them, the *rational feature trees* can be represented by finite cyclic directed graphs The graph on the left hand side below describes a feature tree that models the infinite list $[1,2,1,2,\ldots]$, while the graph on the right hand side represents the type of integer lists, $\mathsf{list}(\mathit{int}) = \mathit{nil} + \mathit{cons}(\mathit{int}, \mathsf{list}(\mathit{int}))$:



In the concurrent constraint language Oz, records are modelled as feature trees and incorporated through the system CFT of *feature constraints* [197, 210]. Feature constraints are a predicate logic formalism for the description of objects by the values of their attributes. Feature description languages and feature logics of various kinds have a long tradition in natural language processing [109, 179, 190, 193]. Their use for the integration of records in constraint languages is more recent [14, 15, 197].

In this chapter we present a new set constraint system that is appropriate for the analysis of records and, more specifically, for the analysis of constraint programs over feature trees; hence *set constraints over feature trees*. The constraint system consists of labelling, selection, arity, union, and inclusion constraints.

Traditionally, set constraints have been considered over the domain of finite constructor trees.

Our main motivation to consider *feature trees* is the analysis of feature tree constraints in Oz. In CFT, the selection constraint $x[f]y$ plays a central role. It asserts that x denotes some feature tree with a feature f leading to the denotation of y, without mentioning the label at the root of x; to assert the label at the root of x, there is a labelling constraint $a\langle x\rangle$. Our system of set constraints over feature trees has a similar selection constraint $x[f]y$ stating that x denotes a set of feature trees with feature f and that y denotes the projection of x at f. In standard set constraint [85], the most closely related projection constraint $x \subseteq a^{-1}_{(f)}(y)$ denotes projection both at the feature f *and* the tree constructor a. Therefore, the projection constraint is not appropriate for the analysis of feature selection constraints.

Our interest in *infinite trees* is also motivated by the program analysis for Oz: since Oz provides for infinite data structures, we must be able to handle sets of infinite feature trees. Secondly, infinite trees are needed to give meaning to non-terminating computations.

Finally, the separation of constraints on labels and features adds flexibility to set-based analysis. For instance, one can integrate analyses along different dimensions by placing different bits of information under different associated features. The flexibility of selection constraints was also found convenient by Flanagan and Felleisen for an analysis of Scheme [64, 65].

Every set constraint over feature trees is satisfiable and has a greatest solution. The latter property is crucial for the analysis of non-terminating programs as we shall see in Chapter 4. Our system shares this property with its closest relative amongst the set constraint systems, the system of co-definite set constraints [44]. Set constraints over feature trees can be viewed as a refinement of co-definite set constraints, in analogy to the fact that CFT refines the constraint system RT [50] of equations over rational constructor trees [197]. We show that co-definite set constraints can be embedded into set constraints over feature trees such that the greatest solution is preserved, and we conjecture that the embedding actually preserves validity for arbitrary first-order formulas.

This and the following chapter investigate in detail the system of set constraints over feature trees. We also consider a system of set constraints over the domain of *non-*

empty sets of feature trees, which has the same constraint language but comes with the global restriction that every variable denotes a non-empty set. The technical motivation for this is that it is sometimes simpler to solve constraints in two steps, where the first step considers non-empty sets only and the second step adds the reasoning about emptiness. We obtain some of our results for constraints over non-empty sets of feature trees first, and then derive the corresponding result for the set constraint system which admits the empty set. Also Charatonik and Podelski [42] have proven decidability for set constraints with intersection by detour through set constraints over non-empty sets of trees.

We give an $O(n^3)$ algorithm to decide satisfiability for union-free constraints over non-empty sets of feature trees. For the corresponding system which admits the empty set, we derive an $O(n^4)$ algorithm to compute the greatest solution and to decide the emptiness problem, (that is, whether or not a variable denotes the empty set in the greatest solution of a constraint). We consider union constraints only briefly and show that the emptiness problem for the complete system of set constraints over feature trees is DEXPTIME-hard.

Additional motivation to pay special attention to the empty set includes the following. First, our application in program analysis suggests to treat the empty set as an undesirable value: a variable that cannot adopt any value at run-time has the "empty type" and is bogus. Second, the empty set can be the reason for efficiency problems and the motivation for ad-hoc optimisations. For example, the implication $a(x,y)\subseteq a(x',y') \rightarrow x \subseteq x' \wedge y \subseteq y' \vee x\subseteq\emptyset \vee y\subseteq\emptyset$ that is valid over sets of constructor trees is sometimes replaced by $a(x,y)\subseteq a(x',y') \rightarrow x\subseteq x' \wedge y\subseteq y'$ for efficiency reasons (*e. g.*, in [12], an analogous optimisation is used in a solver for a kind of set constraints). This simplification is unsound because it does not preserve satisfiability. It is sound, however, when variables are interpreted over non-empty sets.

Finally, the domain of non-empty sets is related to the domain of trees. We exploit this by adapting several techniques directly from the constraint system $\mathrm{FT}_{\leq}$ of ordering constraints over feature trees [141, 143, 145]. We also show that the first-order theories of equality constraints is the same when interpreted over trees or non-empty sets of trees.

The discussion of related work is postponed to Section 3.3 in the following chapter.

2.1. Constraint Systems over Feature Trees

2.1.1. Feature Trees

We assume a set $\mathcal{V}$ of *variables* ranged over by x, y, z, and a signature that defines a set $\mathcal{L}$ of *labels* ranged over by a, b, c, and an infinite set $\mathcal{F}$ of *features* ranged over by f, g, h. We base our definition of feature trees on the notion of paths. A *path* p is a finite sequence of labels. The *empty path* is denoted by ε and the concatenation of paths p and p' as pp'; we have $\varepsilon p = p\varepsilon = p$. Given paths p and q, p' is called a *prefix of* p if $p = p'p''$ for some path p''. Note that every non-empty tree domain contains the empty path ε. A set P of paths is *prefix-closed* if, for all paths p and p', $pp' \in P$ implies $p \in P$. A *tree domain* is a non-empty and prefix-closed set of paths.

A *feature tree* τ is a pair (D, S) consisting of a tree domain D and function $S : D \to \mathcal{L}$ from D into the labels $\mathcal{L}$, called a *labelling*. Given a feature tree τ, we write D_τ for its domain and S_τ for its labelling; hence $\tau = (D_\tau, S_\tau)$. We identify the function S_τ with the set of pairs (p, a) such that $S_\tau(p) = a$. The set of features defined at the root of a feature tree τ is called the *arity* of τ: $\mathsf{ar}(\tau) = D_\tau \cap \mathcal{F}$. For every feature tree τ with $p \in D_\tau$ we denote with $\tau.p$ the *subtree of* τ *at path* p. Formally:

$$\tau.p \quad =_{def} \quad (\{p' \mid pp' \in D_\tau\}, \{(p', a) \mid (pp', a) \in S_\tau\}) \qquad \text{if } p \in D_\tau \qquad (2.1)$$

We call a feature tree τ *finite* if D_τ is finite, and *infinite* otherwise. A feature tree τ is called *rational* if it has only finitely many subtrees and is finitely branching, *i. e.*, if the set $\{f \mid \text{exists } f : pf \in D_\tau\}$ is finite for all p. The set of all feature trees is denoted by $\mathcal{FT}$ where $\mathcal{F}$ and $\mathcal{L}$ remain implicit.

The set $\mathcal{T}$ of *constructor trees* can be defined along the same lines. We do not elaborate on this definition but only remark that constructor trees are isomorphic to feature trees whose features are the natural numbers $\mathbb{N}$ and which conform to an arity function $\mathsf{a} : \mathcal{L} \to \mathbb{N}$; a feature tree τ is said to *conform* to $\mathsf{a} : \mathcal{L} \to \mathbb{N}$ if $\{n \mid pn \in D_\tau\} = \{1, \ldots, n\}$ whenever $(p, a) \in S_\tau$ and $\mathsf{a}(a) = n$. The corresponding embedding of constructor trees into feature trees is denoted by $[\![\cdot]\!]$.

2.1.2. Equality Constraints over Feature Trees (CFT)

We recall the definition of the feature constraint system CFT [197]. The abstract syntax of CFT constraints is defined as follows:

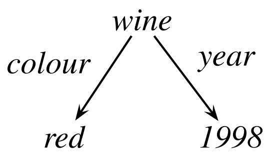
$$\eta ::= \quad x{=}y \quad | \quad a\langle x\rangle \quad | \quad x\{\overline{f}\} \quad | \quad x[f]y \quad | \quad \eta_1 \wedge \eta_2$$

CFT constraints η are conjunctions of so-called *primitive* constraints. We call $x{=}y$ an *equality*, $a\langle x\rangle$ a *labelling*, $x[f]y$ a *selection*, and $x\{\overline{f}\}$ an *arity constraint*. The constraint system CFT is defined by the constraints above and their interpretation in the following structure. Its domain is $\mathcal{FT}$, the equality symbol $=$ is interpreted as equality on $\mathcal{FT}$, every label a [*resp.*, every arity $\{\overline{f}\}$] is interpreted as a unary predicate a [*resp.*, $\{\overline{f}\}$], and every feature is interpreted as a binary predicate $[f]$ such that the following holds.

$$\begin{array}{lll} \tau[f]\tau' & \text{iff} & \tau.f = \tau' \\ a\langle\tau\rangle & \text{iff} & (\varepsilon, a) \in S_\tau \\ \tau\{\overline{f}\} & \text{iff} & \mathsf{ar}(\tau) = \{\overline{f}\} \end{array}$$

We identify this structure with its domain and write $\mathcal{FT}$ for both. The constraint system FT is the subsystem of CFT that one obtains by dropping the arity constraint [16, 21]. The concept of an $\mathcal{FT}$-valuation satisfying a constraint η (or, equivalently, being a solution of η), written $\alpha \models_{\mathcal{FT}} \eta$, is defined as usual. For instance, every solution of the constraint below maps x to the feature tree on the right hand side.

$$\exists y \exists z \begin{pmatrix} \mathit{wine}\langle x\rangle \;\wedge\; x[\mathit{colour}]y \;\wedge\; x[\mathit{year}]z \;\wedge \\ x\{\mathit{wine}, \mathit{colour}\} \;\wedge \\ \mathit{red}\langle y\rangle \;\wedge\; \mathit{1998}\langle z\rangle \;\wedge\; y\{\} \;\wedge z\{\} \end{pmatrix}$$



We also use *feature terms* t as a generalisation of first-order terms [197]. Their syntax is defined as follows, where we always assume that the features in a sequence $\overline{f}$ are pairwise distinct.

$$t ::= \quad x \quad | \quad a(\overline{f}{:}\bar{t})$$

Occasionally, we write $\top$ as an abbreviation of $x \subseteq x$ for an arbitrary variable x. We also use equational constraints of the form $x{=}t$ whose meaning is defined by an existential formula $[\![x{=}t]\!]$ as follows.

$$[\![x{=}a(f_1{:}t_1,\ldots,f_n{:}t_n)]\!] \;=\; a\langle x\rangle \wedge x\{f_1,\ldots,f_n\} \;\wedge\; \bigwedge_{i=1}^{n} \exists y_i x[f_i]y_i \qquad (2.2)$$

$$[\![x{=}a(f_1{:}t_1,\ldots,f_n{:}t_n\ldots)]\!] \;=\; a\langle x\rangle \wedge \bigwedge_{i=1}^{n} \exists y_i x[f_i]y_i \qquad (2.3)$$

For a typical example consider

$$[\![x{=}\mathit{cons}(1{:}y, 2{:}\mathit{nil})]\!] = \quad \exists y' \,(\mathit{cons}\langle x\rangle \wedge x\{1,2\} \wedge x[\mathit{1}]y \wedge x[2]y' \wedge \mathit{nil}\langle y'\rangle \wedge y'\{\})$$

This constraint determines separately the label a of x, the arity $\{1,2\}$, and the associated subtrees y and z. This separation of labelling, selection, and arity constraints in CFT enable a more fine-grained description of trees than that possible with equational constraints over infinite constructor trees [50]. When we use $x{=}a(\overline{f}{:}\bar{t})$ in the sequel, we mean the corresponding CFT formula $[\![x{=}a(\overline{f}{:}\bar{t})]\!]$ unless otherwise stated.

2.1.3. Set Constraints over Feature Trees (FT$_{\subseteq}(ar, \cup)$)

We write $\mathcal{P}(\mathcal{FT})$ for the powerset of the domain $\mathcal{FT}$ of feature trees. Elements of $\mathcal{P}(\mathcal{FT})$ are denoted by σ. For every set σ of feature trees such that $p \in D_\tau$ for all $\tau \in \sigma$ we define $\sigma.p$ as the *set of subtrees* $\tau.p$ *of trees* τ *in* σ. Formally:

$$\sigma.p \;=_{\mathit{def}}\; \{\tau.p \mid \tau \in \sigma\} \qquad \text{if} \quad \forall \tau \in \sigma : p \in D_\tau\,. \qquad (2.4)$$

The class of *constraints over sets of feature trees* is defined by this abstract syntax.

$$\varphi ::= \quad x{\subseteq}x_1{\cup}\ldots{\cup}x_n \quad | \quad x[f]x' \quad | \quad a\langle x\rangle \quad | \quad x\{\overline{f}\} \quad | \quad \varphi_1 \wedge \varphi_2$$

These constraints are defined like CFT constraints extended by inclusion constraints of the form $x{\subseteq}x_1{\cup}\ldots{\cup}x_n$. (Equations $x{=}y$ can, of course, still be expressed by $x \subseteq y \wedge y \subseteq x$.)x As in CFT, we call the primitive constraints $x[f]x'$, $a\langle x\rangle$, and $x\{\overline{f}\}$ *selection*, *labelling*, and *arity constraints*. The primitive constraint $x{\subseteq}x_1{\cup}\ldots{\cup}x_n$ is called an *inclusion*. Given a constraint φ, we write $\mathcal{V}(\varphi)$, $\mathcal{L}(\varphi)$, and $\mathcal{F}(\varphi)$ for the variables, labels, and features occurring in φ. The *size* of a constraint φ is defined as the number of symbols, *i. e.*, variables, labels, and features occurring in φ.

The constraint system $\mathrm{FT}_{\subseteq}(ar,\cup)$ is defined by the constraint language above and their interpretation in the first-order structure which lifts the interpretation of labels, arities, and features from $\mathcal{FT}$ to $\mathcal{P}(\mathcal{FT})$. Its domain is $\mathcal{P}(\mathcal{FT})$, the inclusion and union symbols $\subseteq$ and $\cup$ are interpreted by set inclusion and set union, every label a [every arity $\{\overline{f}\}$] is interpreted as a unary predicate a [resp. $\{\overline{f}\}$], and every feature is interpreted as a binary predicate $[f]$ such that

$$\begin{array}{lll}
\sigma[f]\sigma' & \text{iff} & \sigma.f \text{ defined and } \sigma' = \{\tau' \mid \exists\tau\in\sigma : \tau[f]\tau'\} \\
a\langle\sigma\rangle & \text{iff} & \forall\tau\in\sigma : a\langle\tau\rangle \\
\sigma\{\overline{f}\} & \text{iff} & \forall\tau\in\sigma : \tau\{\overline{f}\}
\end{array}$$

Again, we identify the structure of $\mathrm{FT}_{\subseteq}(ar,\cup)$ with its domain $\mathcal{P}(\mathcal{FT})$.[8]

The name $\mathrm{FT}_{\subseteq}(ar,\cup)$ reflects the collection of set operators in addition to labelling and selection constraints. By restricting the constraint language, we obtain less expressive subsystems. For example, $\mathrm{FT}_{\subseteq}(ar)$ is the restriction of $\mathrm{FT}_{\subseteq}(ar,\cup)$ that does not contain union constraints, and $\mathrm{FT}_{\subseteq}$ contains neither union nor arity constraints. So $\mathrm{FT}_{\subseteq}(ar)$ corresponds to CFT just as $\mathrm{FT}_{\subseteq}$ corresponds to FT. The definition of a $\mathcal{P}(\mathcal{FT})$-valuation α being a solution of φ, written $\alpha \models_{\mathcal{P}(\mathcal{FT})} \varphi$, is the usual one.

We use the *equality constraint* $x{=}y$ as an abbreviation for $x{\subseteq}y \wedge y{\subseteq}x$. Sometimes we admit $\perp$ as a primitive constraint for falsity, since we cannot express it (Proposition 2.1). We also use inclusion constraints like $x{\subseteq}t$ which mention a feature term t and whose meaning is defined by the following existential formulas over set constraints.

$$[\![x{\subseteq}a(\overline{f}{:}\overline{t})]\!] \quad = \quad x\{\overline{f}\} \;\wedge\; a\langle x\rangle \;\wedge\; \bigwedge_{i=1}^{n} \exists y_i\,(x[f_i]y_i \;\wedge\; [\![y_i{\subseteq}t_i]\!]) \tag{2.5}$$

$$[\![x{\subseteq}a(\overline{f}{:}\overline{t}\ldots)]\!] \quad = \quad a\langle x\rangle \;\wedge\; \bigwedge_{i=1}^{n} \exists y_i\,(x[f_i]y_i \;\wedge\; [\![y_i{\subseteq}t_i]\!]) \tag{2.6}$$

[8] Most specific for our system is the interpretation of feature selection, in particular in contrast to the standard *projection* constraint $x \subseteq a^{-1}_{(k)}(y)$ considered in the literature. For motivation and illustration of the semantics of $\sigma[f]\sigma'$ compare Proposition 2.5 below, and Section 3.3.1.2.

2.1.4. Constraints over Non-empty Sets of Feature Trees ($\mathbf{FT}^{ne}_{\subseteq}(ar,\cup)$)

We call $\mathrm{FT}^{ne}_{\subseteq}(ar,\cup)$ the constraint system that we obtain when we restrict the interpretation domain of $\mathrm{FT}_{\subseteq}(ar,\cup)$ constraints to $\mathcal{P}^{+}(\mathcal{FT})$, the domain of non-empty sets of feature trees. We read $\mathrm{FT}^{ne}_{\subseteq}(ar,\cup)$ as "FT-include-nonempty". Again we identify the structure of $\mathrm{FT}^{ne}_{\subseteq}(ar,\cup)$ with the system $\mathrm{FT}^{ne}_{\subseteq}(ar,\cup)$ itself. In analogy to the notation used above we write $\mathrm{FT}^{ne}_{\subseteq}(ar)$ and $\mathrm{FT}^{ne}_{\subseteq}$ for the subsystems of $\mathrm{FT}^{ne}_{\subseteq}(ar,\cup)$ without union constraints, and without union and arity constraints.[9]

2.1.5. Basic Properties

We mention a number of basic properties of the constraint systems $\mathrm{FT}_{\subseteq}(ar,\cup)$ and $\mathrm{FT}^{ne}_{\subseteq}(ar)$. Constraint solving will be considered in the following chapter.

Proposition 2.1 (Least Solution)
Every $FT_{\subseteq}(ar,\cup)$ constraint is satisfiable. The valuation which maps all variables to the empty set is a solution of every $FT_{\subseteq}(ar,\cup)$ constraint.

Proof. A simple check of all primitive constraints. □

Notice that in contrast to $\mathrm{FT}_{\subseteq}(ar,\cup)$ constraints a satisfiable $\mathrm{FT}^{ne}_{\subseteq}(ar)$ constraint need not have a *least* solution: all singleton sets are minimal but incomparable elements with respect to set inclusion.

Proposition 2.2 (Solutions are Closed under Unions)
The set of solutions of any $FT_{\subseteq}(ar,\cup)$ or $FT^{ne}_{\subseteq}(ar,\cup)$ constraint is closed under pointwise union (possibly infinite).

Proof. Given an $\mathrm{FT}_{\subseteq}(ar,\cup)$ or $\mathrm{FT}^{ne}_{\subseteq}(ar,\cup)$ constraint φ and a set S of solutions of φ, one easily checks that the pointwise union of the elements of S satisfies all primitive constraints in φ. □

In contrast, neither $\mathrm{FT}_{\subseteq}(ar,\cup)$ nor $\mathrm{FT}^{ne}_{\subseteq}(ar,\cup)$ constraints are closed under intersection. For instance, the constraint $x \subseteq y \cup z$ has the solutions α and α' with $\alpha(x) = \alpha(y) = \{a\}, \alpha(z) = \{b\}$, and $\alpha'(y) = \{b\}, \alpha'(x) = \alpha'(z) = \{a\}$, whose intersection is not a solution since it assigns $\emptyset$ to y and z but $\{a\}$ to x.

The next property is crucial for our set-based analysis as described in Chapters 4 and 5.

[9] In [142] we have anticipated the system $\mathrm{FT}^{ne}_{\subseteq}$ and suggested that it should be called Ines($\mathcal{FT}$); we do not follow this suggestion here, to point out the different constraint languages of $\mathrm{FT}^{ne}_{\subseteq}$ and Ines, and to stress the relationship to $\mathrm{FT}_{\subseteq}(ar,\cup)$. Notice, however, that we only consider the union-free fragments of $\mathrm{FT}^{ne}_{\subseteq}(ar,\cup)$ in this thesis.

Proposition 2.3 (Greatest Solution)
Every $FT_{\subseteq}(ar,\cup)$ or $FT^{ne}_{\subseteq}(ar,\cup)$ constraint has a greatest solution.

Proof. Since, by Proposition 2.1, every $\mathrm{FT}_{\subseteq}(ar,\cup)$ or $\mathrm{FT}^{ne}_{\subseteq}(ar)$ constraint φ is satisfiable, the set $\mathrm{Sol}(\varphi)$ of solutions of φ is non-empty. So the pointwise union of all solutions in $\mathrm{Sol}(\varphi)$ is a well-defined set-valuation and, by Proposition 2.2, a solution of φ. □

There is a close relationship between emptiness in the greatest solution and satisfiability over non-empty sets of feature trees. For any φ, denote with $gsol(\varphi)$ the *greatest solution* of φ over $\mathcal{P}(\mathcal{FT})$.

Proposition 2.4 (Greatest Solution and Empty Sets)
For all $FT_{\subseteq}(ar,\cup)$ constraints φ: The greatest solution of φ maps some variable to the empty set ($\exists x \in \mathcal{V} : gsol(\varphi)(x) = \emptyset$) if and only if φ is non-satisfiable over $\mathcal{P}^+(\mathcal{FT})$.

Proof. If, for some x, $gsol(\varphi)(x) = \emptyset$, then x denotes the empty set in all $\mathcal{P}(\mathcal{FT})$-solutions of φ. Hence φ is non-satisfiable over $\mathcal{P}^+(\varphi)$. *Vice versa*, if $gsol(\varphi)(x) \neq \emptyset$ for all x, then $gsol(\varphi)(x) \neq \emptyset$ is also a $\mathcal{P}^+(\mathcal{FT})$-solution of φ. □

Another interesting relationship between CFT constraints and $\mathrm{FT}^{ne}_{\subseteq}(ar)$ constraints is the following one. It states that, with respect to a collection of CFT constraints, the satisfiability of a sequence of equality constraints can be characterised by the satisfiability of a sequence of inclusion constraints. (In this statement, we identify every CFT constraint with a set constraint over feature trees by replacing $x = y$ with $x \subseteq y \wedge y \subseteq x$.)

Proposition 2.5
Let $\eta_1, \ldots, \eta_n$ be satisfiable and variable-disjoint CFT constraints not containing the variable y. If the constraint $\bigwedge_{i=1}^{n}(\eta_i \wedge y \subseteq x_i)$ is satisfiable over $\mathcal{P}^+(\mathcal{FT})$, then the constraint $\bigwedge_{i=1}^{n} \eta_i \wedge \bigwedge_{i=1}^{n-1} x_i = x_{i+1}$ is satisfiable over $\mathcal{FT}$.

Proof. One uses the greatest solution of the set constraint $\bigwedge_{i=1}^{n}(\eta_i \wedge y \subseteq x_i)$ (Definition 3 on Page 36) to construct a solution of $\bigwedge_{i=1}^{n} \eta_i \wedge \bigwedge_{i=1}^{n-1} x_i = x_{i+1}$. □

This proposition would not hold if the semantics of $\sigma[f]\sigma'$ would not require $\sigma.f$ to be defined. For a counterexample see the analysis of Example D_{fail3} on Page 94.

2.2. Solving Constraints over Non-empty Sets of Feature Trees

In this section, we devise an incremental algorithm to decide satisfiability of $\mathrm{FT}^{ne}_{\subseteq}(ar)$ constraints $x \subseteq y$, $x[f]y$, $a\langle x\rangle$, and $x\{\overline{f}\}$ in cubic time. In Section 2.3 we derive an emptiness test for $\mathrm{FT}_{\subseteq}(ar)$.

Intuitively, a satisfiability test is *incremental* if the input constraint can be input piecewise without changing the complexity measure [105].[10] Let a satisfiability test with complexity $O(f(n))$ be given, assume a sequence of constraints $\varphi_0, \varphi_1, \ldots, \varphi_k$ with respective sizes $n_0, \ldots, n_k$, and ask for every $m \in \{0, \ldots, k\}$ whether the conjunction $\bigwedge_{i=1}^{m} \varphi_i$ is satisfiable. The naive solution to this problem is to run the satisfiability test on all these conjunctions. This approach has worst-case complexity $O(\Sigma_{m=0}^{k} f(\Sigma_{i=0}^{m} n_i))$, or simpler $O(k \cdot f(n))$ if $n = \Sigma_{i=1}^{k} n_i$ is size of the complete conjunction and k is the length of the sequence. In contrast, an incremental algorithm can build upon its work to check of $\bigwedge_{i=1}^{m} \varphi_i$ to decide $\bigwedge_{i=1}^{m+1} \varphi_i$ more efficiently, such that the total complexity of the problem does not exceed $O(f(n))$.

2.2.1. Satisfiability

The main algorithmic problem to be solved is to guarantee termination in presence of infinite trees.

Example 1 (Termination Problem)

Pick two distinct labels $a \neq b$ and consider the constraint

$$x[f]x' \wedge a\langle x' \rangle \wedge y[f]y' \wedge b\langle y' \rangle \wedge z \subseteq x \wedge z \subseteq y \tag{2.7}$$

This constraint is non-satisfiable over non-empty sets: the denotation of z must be a subset of the denotations of x and y and hence of their intersection. Since no feature tree can be labelled with both a and b, the denotations of x and y must be disjoint and hence z must denote the empty set. To detect the inconsistency in (2.7) we derive the following constraints step by step:

$z[f]z'$	(from $z \subseteq x$ and $x[f]x'$ where z' fresh)
$z' \subseteq x' \wedge z' \subseteq y'$	(from $z[f]z', x[f]x'$ and $y[f]y'$ and $z \subseteq x$ and $z \subseteq y$)
$a\langle z' \rangle \wedge b\langle z' \rangle$	(from $z' \subseteq x' \wedge a\langle x' \rangle$ and $z' \subseteq y' \wedge b\langle y' \rangle$)

When we apply a similar argument to the constraint

$$y \subseteq x \wedge x[f]x \tag{2.8}$$

we run into a loop, as the reader can easily verify. The critical step of reasoning here is the first one above which introduces the fresh variable z'.

[10] Incremental algorithms are synonymously called *on-line*, in contrast to *off-line* algorithms that receive their input at once.

$$
\begin{array}{llcll}
(\text{Refl}) & x \subseteq x & & & \\
(\text{Trans}) & x \subseteq y \wedge y \subseteq z & \rightarrow & x \subseteq z & \\
(\text{Incl-Nondis}) & x \subseteq y & \rightarrow & x \nparallel y & \\
(\text{Symm-Nondis}) & x \nparallel y & \rightarrow & y \nparallel x & \\
(\text{Quasi-Trans}) & x \subseteq y \wedge x \nparallel z & \rightarrow & y \nparallel z & \\
(\text{Desc-Incl}) & x[f]x' \wedge x \subseteq y \wedge y[f]y' & \rightarrow & x' \subseteq y' & \\
(\text{Desc-Nondis}) & x[f]x' \wedge x \nparallel y \wedge y[f]y' & \rightarrow & x' \nparallel y' & \\
(\text{Clash-Sort}) & a\langle x\rangle \wedge x \nparallel y \wedge b\langle y\rangle & \rightarrow & \bot & \text{if } a \neq b \\
(\text{Clash-Arity-I}) & x[f]x' \wedge x \nparallel y \wedge y\{\bar{g}\} & \rightarrow & \bot & \text{if } f \notin \{\bar{g}\} \\
(\text{Clash-Arity-II}) & x\{\bar{f}\} \wedge x \nparallel y \wedge y\{\bar{g}\} & \rightarrow & \bot & \text{if } \{\bar{f}\} \neq \{\bar{g}\}
\end{array}
$$

Figure 2.1.: Satisfiability of $\mathrm{FT}^{ne}_{\subseteq}(ar)$ Constraints

The reason for the inconsistency of (2.7) is the disjointness of two sets that are required to have a non-empty intersection. In order to reason about this phenomenon in a terminating manner, we introduce an additional primitive *non-disjointness* constraint $x \nparallel y$. We also consider $\bot$ as a primitive constraint in this section.

$$\varphi ::= \quad \bot \quad | \quad x \nparallel y \quad | \quad x \subseteq x' \quad | \quad x[f]x' \quad | \quad a\langle x\rangle \quad | \quad x\{\bar{f}\} \quad | \quad \varphi_1 \wedge \varphi_2$$

The semantics of $x \nparallel y$ is defined as follows.

$$\alpha \models_{\mathcal{P}(\mathcal{FT})} x \nparallel y \quad \text{iff} \quad \alpha(x) \cap \alpha(y) \neq \emptyset$$

Notice that non-disjointness is not transitive. Both $\bot$ and the non-disjointness constraint $x \nparallel y$ are expressible in $\mathrm{FT}^{ne}_{\subseteq}$, since

$$
\begin{array}{lll}
\bot & \leftrightarrow & \exists x(a\langle x\rangle \wedge b\langle x\rangle) \quad \text{if } a \neq b \\
x \nparallel y & \leftrightarrow & \exists z(z \subseteq x \wedge z \subseteq y)
\end{array}
$$

are valid $\mathrm{FT}^{ne}_{\subseteq}$-equivalences. In $\mathrm{FT}_{\subseteq}(ar)$, neither of them holds; in particular, notice that the formula $\exists z(z \subseteq x \wedge z \subseteq y)$ holds vacuously in $\mathrm{FT}_{\subseteq}(ar)$ (pick the empty set as the denotation of z) while its left does not hold in general.

Figure 2.1 contains axiom schemes which define an infinite set of axioms. Every axiom is either a primitive constraint, or the implication between a constraint and a primitive constraint of the form $x \subseteq y$ or $x \nparallel y$. These axioms describe the satisfiability problem of $\mathrm{FT}^{ne}_{\subseteq}(ar)$. In a subsequent step we shall interpret these axioms as rewriting rules to yield a satisfiability check of $\mathrm{FT}^{ne}_{\subseteq}(ar)$ constraints.

Proposition 2.6 (Soundness)
The structure $\mathcal{P}^{+}(\mathcal{FT})$ is a model of the axioms in Figure 2.1.

Proof. By a routine check. For illustration, we prove the statement for rule (Quasi-Trans). The implications below follow with $x \not\parallel y \leftrightarrow \exists z(z \subseteq x \wedge z \subseteq y)$ and transitivity of set inclusion.

$$x \subseteq y \wedge x \not\parallel z \quad \leftrightarrow \quad x \subseteq y \wedge \exists v(v \subseteq x \wedge v \subseteq z) \quad \rightarrow \quad \exists v(v \subseteq y \wedge v \subseteq z) \quad \leftrightarrow \quad x \not\parallel z$$

We consider the remaining axioms informally. Axioms (Refl) and (Trans) hold since set inclusion $\subseteq$ is a partial order. Axiom (Incl-Nondis) states that inclusion implies non-disjointness. This does not hold over the domain of arbitrary sets of feature trees since the empty set is included in every set but also disjoint with every set. Non-disjointness is symmetric (Symm-Nondis) since set intersection is commutative. Axioms (Desc-Incl) and (Desc-Nondis) state that set projection at a feature f is a homomorphism with respect to set inclusion and non-disjointness. The axiom (Clash-Sort) states that labelling is a partial function. Axioms (Clash-Arity-I) and (Clash-Arity-II) state that two sets with incompatible arity restrictions must be disjoint. □

Figure 2.1 induces a naive fixed point algorithm on sets of primitive constraints that computes the closure of an input constraint φ under the given axioms. (Here, we identify a constraint with the set of its primitive constraints.) In order for this fixed-point to be finite we restrict applicability of the reflexivity axiom $x \subseteq x$ to those variables which actually occur in a given constraint; thus, no fresh variables are introduced. Call this algorithm S. We call a *step* of this algorithm the addition of a new primitive constraint to some given constraint according to one of the axioms. A constraint is called S-*closed* if the algorithm S cannot proceed. The fixed-point of a constraint under algorithm S is called its S-*closure*. If some axiom (A) does not apply to a set of primitive constraints, it is called *A-closed*, or *closed under (A)*.
For illustration on how the algorithm works we consider two examples.

Example 2 (Satisfiability Test)
Assume $a \neq b$. Reconsider the constraint $x[f]x' \wedge a\langle x' \rangle \wedge y[f]y' \wedge b\langle y' \rangle \wedge z \subseteq x \wedge z \subseteq y$ (where $a \neq b$) from above (2.7). From this constraint, algorithm S derives $z \not\parallel x$ by (Incl-Nondis), $y \not\parallel x$ with (Quasi-Trans), and eventually $\perp$ via (Clash-Sort). Now consider another non-satisfiable constraint:

$$a\langle x \rangle \wedge x[f]x \wedge z \subseteq x \wedge z \subseteq y \wedge y[f]y' \wedge b\langle y' \rangle \tag{2.9}$$

In several steps as above, algorithm S derives from (2.9) the non-disjointness constraint $x \not\parallel y$. Then it derives $x \not\parallel y'$ via (Desc-Nondis) and $\perp$ via (Clash-Sort).

Proposition 2.7 (Termination)

If φ is an $FT^{ne}_{\subseteq}(ar)$ constraint of size n, then algorithm S *terminates after at most* $2 \cdot n^2 + 1$ *steps.*

Proof. If φ has size n, then it contains at most n variables. Since algorithm S does not introduce fresh ones, it may add at most n^2 constraints of each of the forms $x \not\parallel y$ or $x \subseteq y$ and possibly $\perp$. $\square$

Proposition 2.8 (Completeness)

Every S*-closed* $FT^{ne}_{\subseteq}(ar)$ *constraint which does not contain* $\perp$ *is satisfiable over* $\mathcal{P}^+(\mathcal{FT})$.

Proof. The proof is postponed to Section 2.2.2 beginning on Page 35. Its structure is as follows. First, we define a syntactic notion of path consistency on constraints (Definition 2) and show that every S-closed constraint not containing $\perp$ is indeed path consistent (Lemma 2.12). Second, we show that every path consistent constraint is satisfiable (Lemma 2.11). $\square$

Theorem 3 (Decidability and Complexity for Satisfiability of $\mathbf{FT}^{ne}_{\subseteq}(ar)$)

The satisfiability problem of $FT^{ne}_{\subseteq}(ar)$ constraints of size n is decidable and has incremental time complexity $O(n^3)$ and space complexity $O(n^2)$.

Proof. By Proposition 2.6, φ is equivalent to its S-closure. Hence φ is inconsistent if the S-closure of φ contains $\perp$. Otherwise φ is satisfiable by Proposition 2.8. Since S terminates for all input constraints by Proposition 2.7, S is an effective decision procedure. The complexity statement is detailed in Section 2.2.3 beginning on Page 39. There, we use a table of quadratic size to show that we can implement every step of algorithm S such that it takes time $O(n)$. This yields an overall time complexity of $O(n^3)$. $\square$

In the incremental case, this complexity statement relies on the assumption that applicability of (Clash-Arity-II) can be checked in linear time. This is the case under one of the following conditions. (i) No arity constraint ever occurs, (ii) the size of the arity constraints is bounded, or (iii) the arity constraints list the features according to a fixed order. Under any of these preconditions, the equality test for two arities $\{\overline{f}\}$ and $\{\overline{g}\}$ in the side condition of rule (Clash-Arity-II) can be checked in at most linear time $O(n)$. If none of these assumptions hold, the equality of $\{\overline{f}\}$ and $\{\overline{g}\}$ requires time $n \cdot \log n$ such that the time complexity of the satisfiability test rises to $O(n^3 \cdot \log n)$.

Finite Trees

The satisfiability test for $\mathrm{FT}^{ne}_{\subseteq}(ar)$ can be adapted to the case of finite trees by extending the algorithm with the following occurs check axiom scheme:

$$(\text{Occurs}) \qquad x \subseteq x_i \wedge \bigwedge_{i=1}^{n}(x_i \subseteq y_i \wedge y_i[f_i]x_{i+1}) \wedge x_{n+1} \subseteq x \quad \rightarrow \quad \perp \qquad n \geq 1$$

We can implement this occurs check such that we stay in incremental time $O(n^3)$ and space $O(n^2)$. This can be done by means of reachability constraints of the form $x \rightsquigarrow^{+} y$ which state that there exists $n \geq 1$, variable $x_1, \ldots, x_{n+1}$ and $y_1, \ldots, y_n$, and features $f_1, \ldots, f_n$ such that

$$x \subseteq x_i \wedge \bigwedge_{i=1}^{n}(x_i \subseteq y_i \wedge y_i[f_i]x_{i+1}) \wedge x_{n+1} \subseteq y$$

holds. There are at most $O(n^2)$ such constraints; so Theorem 3 carries over to the finite tree case.

2.2.2. Completeness of the Satisfiability Test

In this section we complete the proof of Proposition 2.8. It relies on two syntactic properties of $\mathrm{FT}^{ne}_{\subseteq}(ar)$ constraints, called path reachability and path consistency that we define first. For every S-closed constraint φ not containing $\perp$ we then define a mapping from variables into non-empty sets of feature trees (see Definition 3) and show in Proposition 2.10 that this is the greatest solution of φ.

Throughout this section we use the notion "constraint" to mean "$\mathrm{FT}^{ne}_{\subseteq}(ar)$ constraint".

Definition 1 (Path Reachability)

For all paths p and constraints φ, we define a binary relation $\overset{\varphi}{\rightsquigarrow}_p$ between variables, where $x \overset{\varphi}{\rightsquigarrow}_p y$ reads as "y is reachable from x over path p in φ":

$$\begin{array}{lll} x \overset{\varphi}{\rightsquigarrow}_{\varepsilon} y & \textit{if} & x \subseteq y \in \varphi \\ x \overset{\varphi}{\rightsquigarrow}_{f} y & \textit{if} & x[f]y \in \varphi \\ x \overset{\varphi}{\rightsquigarrow}_{pq} y & \textit{if} & \textit{exists } z \textit{ such that } x \overset{\varphi}{\rightsquigarrow}_p z \textit{ and } z \overset{\varphi}{\rightsquigarrow}_q y. \end{array}$$

For all paths p and constraints φ, we define a binary relation $\overset{\varphi}{\rightsquigarrow}_p$ between variables and labels a [finite sets of features $\{\overline{f}\}$] where $x \overset{\varphi}{\rightsquigarrow}_p a$ [$x \overset{\varphi}{\rightsquigarrow}_p \{\overline{f}\}$] reads as "$a$ [$\{\overline{f}\}$] can be reached from x over path p in φ":

$$\begin{array}{lll} x \overset{\varphi}{\rightsquigarrow}_p a & \textit{if} & x \overset{\varphi}{\rightsquigarrow}_p y \textit{ and } a\langle y\rangle \in \varphi, \\ x \overset{\varphi}{\rightsquigarrow}_p \{\overline{f}\} & \textit{if} & x \overset{\varphi}{\rightsquigarrow}_p y \textit{ and } y\{\overline{f}\} \in \varphi \end{array}$$

Finally we derive, for all paths p, a unary relation $x \overset{\varphi}{\leadsto}_p$ *which says that a variable "x has path p in* φ*":*

$$\begin{array}{lll} x \overset{\varphi}{\leadsto}_p & \textit{if} & \textit{exists } y \textit{ such that } x \overset{\varphi}{\leadsto}_p y \\ x \overset{\varphi}{\leadsto}_{pf} & \textit{if} & \textit{exist } y, \overline{g} \textit{ such that } x \overset{\varphi}{\leadsto}_p \{\overline{g}\} \textit{ and } f \in \{\overline{g}\} \end{array}$$

Example 4 (Path Reachability)

Let φ be the constraint

$$x \subseteq y \wedge x\{f,g,h\} \wedge a\langle y\rangle \wedge x[f]u \wedge x[g]z \wedge z[f]x \wedge b\langle z\rangle$$

and observe that, among others, the following reachability relations hold for x:

$$\begin{array}{l} x \overset{\varphi}{\leadsto}_\varepsilon y, \quad x \overset{\varphi}{\leadsto}_g z, \quad x \overset{\varphi}{\leadsto}_{gf} x, \quad x \overset{\varphi}{\leadsto}_{gf} y, \\ x \overset{\varphi}{\leadsto}_h, \quad x \overset{\varphi}{\leadsto}_{gf} \{f,g,h\} \\ x \overset{\varphi}{\leadsto}_\varepsilon a, \quad x \overset{\varphi}{\leadsto}_g b, \quad x \overset{\varphi}{\leadsto}_{gf} a \end{array}$$

In the proof of Lemmas 2.11 and 2.12 we make implicit use of the following simple property of path reachability.

Fact 1 *If* $x \overset{\varphi}{\leadsto}_{fp} y$ *then there exists* z *such that* $x \overset{\varphi}{\leadsto}_f z \wedge z \overset{\varphi}{\leadsto}_p y$.

Definition 2 (Path Consistency)

We call a constraint φ path consistent *if the following two conditions hold for all* $x, y \in \mathcal{V}$, $a, b \in \mathcal{L}$, $g \in \mathcal{F}$ *and* $p \in \mathcal{F}^*$.

1. *If* $x \overset{\varphi}{\leadsto}_p a$, $x \not\parallel y \in \varphi$, *and* $y \overset{\varphi}{\leadsto}_p b$ *then* $a = b$.

2. *If* $x \overset{\varphi}{\leadsto}_{pf}$, $x \not\parallel y \in \varphi$, *and* $y \overset{\varphi}{\leadsto}_p \{\overline{g}\}$, *then* $f \in \{\overline{g}\}$.

Definition 3 (Greatest Solution)

Assume a path consistent constraint φ *closed under (Refl) and (Incl-Nondis), and define for all* $x \in \mathcal{V}$

$$D_x^\varphi =_{def} \quad \{p \mid x \overset{\varphi}{\leadsto}_p\} \qquad\qquad S_x^\varphi =_{def} \quad \{(p,a) \mid x \overset{\varphi}{\leadsto}_p a\}$$

Furthermore, for all x define a set $gsol(\varphi)(x)$ *of feature trees as follows:*

$$gsol(\varphi)(x) =_{def} \quad \left\{ \tau \;\middle|\; \begin{array}{ll} 1. & D_x^\varphi \subseteq D_\tau \quad \textit{and} \quad S_x^\varphi \subseteq L_\tau \\ 2. & \forall p : \textit{if} \quad x \overset{\varphi}{\leadsto}_p \{\overline{g}\} \quad \textit{then} \quad \mathsf{ar}(\tau.p) = \{\overline{g}\} \end{array} \right\}$$

Lemma 2.9 (*$gsol(\varphi)$* **is well-defined)**

If φ is a path consistent constraint φ which is closed under (Refl) and (Incl-Nondis), then for all $x \in \mathcal{V}$, $p \in \mathcal{F}^*$, $a, b \in \mathcal{L}$, $f \in \mathcal{F}$ and $\overline{g} \in \mathcal{P}(\mathcal{F})$ it holds that

1. *$(p,a) \in S^{\varphi}_x$ and $(p,b) \in S^{\varphi}_x$ imply $a = b$.*
2. *$pf \in D^{\varphi}_x$ and $x \overset{\varphi}{\rightsquigarrow}_p \{\overline{g}\}$ imply $f \in \{\overline{g}\}$.*

Proof. Path reachability statements only hold for variables which actually occur in φ. Hence, in both cases $x \subseteq x \in \varphi$ and $x \not\parallel x \in \varphi$ due to the asserted closure conditions (Refl) and (Incl-Nondis). The claims now follow immediately from the definition of path consistency. □

This lemma shows that, for all $x \in \mathcal{V}$, S^{φ}_x is a partial labelling on D^{φ}_x. Hence there exists at least one feature tree extending it so that $gsol(\varphi)(x)$ is a non-empty set of feature trees.

Proposition 2.10 (Greatest Solution)

The valuation $gsol(\varphi)$ is the greatest solution of every S*-closed $FT^{ne}_{\subseteq}$ constraint φ not containing $\perp$.*

Proof. Follows from the combination of the Lemmas 2.11 and 2.12. □

Lemma 2.11 (Closedness and Path-Consistency Imply Satisfiability)

For every (Refl), (Trans), (Incl-Nondis), (Desc-Incl)-closed and path consistent constraint φ not containing $\perp$, $gsol(\varphi)$ is the greatest solution.

Proof. Let φ be (Refl), (Trans), (Incl-Nondis), (Desc-Incl)-closed and path consistent. By (Refl) and (Incl-Nondis)-closedness and path consistency, $gsol(\varphi)$ is a variable assignment into non-empty sets of feature trees, as Lemma 2.9 shows. We verify that $gsol(\varphi)$ satisfies all primitive constraints in φ. Maximality is obvious.

Case $x \subseteq y \in \varphi$: For all y', if $y \overset{\varphi}{\rightsquigarrow}_p y'$ then $x \overset{\varphi}{\rightsquigarrow}_p y'$ by the definition of path reachability. Thus, $D_{gsol(\varphi)(y)} \subseteq D_{gsol(\varphi)(x)}$.[11] Similarly, for all a $[\{\overline{g}\}]$, if $y \overset{\varphi}{\rightsquigarrow}_p a$ $[y \overset{\varphi}{\rightsquigarrow}_p \{\overline{g}\}]$ then $x \overset{\varphi}{\rightsquigarrow}_p a$ $[x \overset{\varphi}{\rightsquigarrow}_p \{\overline{g}\}]$ by the definition of path reachability. Thus, $L_{gsol(\varphi)(y)} \subseteq L_{gsol(\varphi)(x)}$. In combination, we obtain that $gsol(\varphi)(x) \subseteq gsol(\varphi)(y)$.

Case $x[f]y \in \varphi$: We prove the following two equivalences for all p, z, and b:

1. $x \overset{\varphi}{\rightsquigarrow}_{fp} z$ iff $y \overset{\varphi}{\rightsquigarrow}_p z$

[11] For sake of clarity, we drop some of the superscripted φ's in the proof.

2. $x \overset{\varphi}{\leadsto}_{fp} a$ iff $y \overset{\varphi}{\leadsto}_{p} a$
3. $x \overset{\varphi}{\leadsto}_{fp} \{\overline{g}\}$ iff $y \overset{\varphi}{\leadsto}_{p} \{\overline{g}\}$

The first property implies $D_{gsol(\varphi)(y)} = \{p \mid fp \in D_{gsol(\varphi)(x)}\}$ and the second one is equivalent to $L_{gsol(\varphi)(y)} = \{(p,b) \mid (fp,b) \in L_{gsol(\varphi)(x)}\}$.

1. If $y \overset{\varphi}{\leadsto}_{p} z$ then $x \overset{\varphi}{\leadsto}_{fp} z$ since $x[f]y \in \varphi$. Suppose $x \overset{\varphi}{\leadsto}_{fp} z$. By definition of path reachability there exist x' and y' such that

   $$x \overset{\varphi}{\leadsto}_{\varepsilon} x', \qquad x'[f]y', \qquad \text{and} \qquad y' \overset{\varphi}{\leadsto}_{p} z.$$

   Reflexivity and transitivity, that is, (Refl)- and (Trans)-closedness of φ, and $x \overset{\varphi}{\leadsto}_{\varepsilon} x'$ imply that $x \subseteq x' \in \varphi$. (Desc-Incl)-closedness ensures $y \subseteq y' \in \varphi$ such that $y \overset{\varphi}{\leadsto}_{p} z$ holds.
2. If $x \overset{\varphi}{\leadsto}_{fp} a$ then there exists z such that $x \overset{\varphi}{\leadsto}_{fp} z$ and $a\langle z\rangle$. The first equivalence implies $y \overset{\varphi}{\leadsto}_{p} z$ and thus $y \overset{\varphi}{\leadsto}_{p} a$. The converse is simple.
3. Similar to the previous case.

Case $a\langle x\rangle \in \varphi$**:** Reflexivity, *i. e.*, (Refl)-closedness of φ, implies that $x \subseteq x \in \varphi$. Thus $x \overset{\varphi}{\leadsto}_{\varepsilon} a$ such that $(\varepsilon, a) \in L_x$.

Case $x\{\overline{f}\} \in \varphi$**:** If $g \in \{\overline{f}\}$, then $x \overset{\varphi}{\leadsto}_{\varepsilon} g$ by definition of path reachability, and hence $g \in D_\tau$ for all $\tau \in gsol(\varphi)(x)$. Conversely, if $x\{\overline{f}\} \in \varphi$ then $x \overset{\varphi}{\leadsto}_{\varepsilon} \{\overline{f}\}$, such that $x \overset{\varphi}{\leadsto}_{\varepsilon} g$ implies $g \in \{\overline{f}\}$ by path consistency.

Case $x \not\parallel y \in \varphi$**:** We have to show that the set $L_x \cup L_y$ is partial function and that, for all p, $x \overset{\varphi}{\leadsto}_{p} \{\overline{f}\}$ and $y \overset{\varphi}{\leadsto}_{p} g$ imply $g \in \{\overline{f}\}$ (and vice versa with x and y swapped). For both, path consistency suffices.

Thus $gsol(\varphi)$ is a solution of φ. □

Lemma 2.12 (Closedness Implies Path-Consistency)
A constraint is path consistent whenever it does not contain $\perp$ and is closed under (Incl-Nondis), (Symm-Nondis), (Quasi-Trans), (Desc-Nondis), and the three clash rules (Clash-Sort), (Clash-Arity-I), and (Clash-Arity-II)

Proof. Let φ be a constraint not containing $\perp$, and assume that φ is closed under the rules (Incl-Nondis), (Symm-Nondis), (Quasi-Trans), (Desc-Nondis), (Clash-Sort), (Clash-Arity-I), and (Clash-Arity-II). The proof is by induction over paths p. Let x, y, a, b, $\overline{f}$, and g be arbitrary.

Case $x \overset{\varphi}{\rightsquigarrow}_p a$, $x \not\Vert y \in \varphi$, and $x \overset{\varphi}{\rightsquigarrow}_p b$:

If $p = \varepsilon$ then there exist $n, m \geq 0$, $x_1, \ldots, x_n$, $y_1, \ldots, y_m$ such that

$$x \subseteq x_1 \wedge \ldots \wedge x_{n-1} \subseteq x_n \wedge a\langle x_n \rangle \in \varphi \quad \text{and}$$
$$y \subseteq y_1 \wedge \ldots \wedge y_{m-1} \subseteq y_m \wedge b\langle y_m \rangle \in \varphi$$

Rules (Incl-Nondis), (Symm-Nondis), and (Quasi-Trans)-closedness imply that $x_n \not\Vert y_m \in \varphi$; this can be shown by a simple induction over n and m. Hence $a = b$ since φ is closed under (Clash-Sort) but does not contain $\perp$.

If $p = fq$ then there exists there exist x', y', x'', and y'' such that:

$$x \overset{\varphi}{\rightsquigarrow}_\varepsilon x',\ x'[f]x'' \in \varphi,\ x'' \overset{\varphi}{\rightsquigarrow}_p a, \quad \text{and}$$
$$y \overset{\varphi}{\rightsquigarrow}_\varepsilon y',\ y'[f]y'' \in \varphi,\ y'' \overset{\varphi}{\rightsquigarrow}_p b.$$

Since $x \not\Vert y \in \varphi$ we have $x' \not\Vert y' \in \varphi$ by definition of path reachability and (Incl-Nondis), (Symm-Nondis), and (Quasi-Trans)-closedness (see case $p{=}\varepsilon$). (Desc-Nondis)-closedness thus implies $x'' \not\Vert y'' \in \varphi$ such that $a{=}b$ follows by induction hypothesis.

Case $x \overset{\varphi}{\rightsquigarrow}_p \{\overline{f}\}$, $x \not\Vert y \in \varphi$, and $x \overset{\varphi}{\rightsquigarrow}_p g$: The proof is similar to the previous case, of course using axioms (Clash-Arity-I/II) instead of (Clash-Sort). □

2.2.3. Incrementality and Complexity of the Satisfiability Test

We complete the proof of Theorem 3 by proving the following Proposition. Throughout this section we use the notion "constraint" to mean "$\mathrm{FT}^{ne}_{\subseteq}(ar)$ constraint".

Proposition 2.13 (Complexity for Satisfiability of $\mathbf{FT}^{ne}_{\subseteq}$)
Algorithm S *can be implemented in space $O(n^2)$ and incremental time complexity $O(n^3)$ where n is the size of the input constraint, provided one of the following holds:*

- *No arity constraint ever occurs.*
- *The size of arity constraints is bounded.*
- *The arity constraints list the features according to a fixed order.*

Otherwise, every step of algorithm S *can be implemented such that it has incremental time complexity $O(n^3 \cdot \log n)$.*

We organise algorithm S as a rewriting on pairs of the form (P,S) where P and S are called *pool* and *store*. Pool P and store S are data structures for primitive constraints as described below. In what follows below, we confuse pools and stores with the constraints they represent.

To start out, we fix an input constraint φ_0 and consider the non-incremental case. The incremental case is dealt with then.

2.2.3.1. The Non-incremental Case

Initially, the pool contains all primitive constraints contained in φ_0 and the store is empty. In order to decide satisfiability of φ, we start with the pair $(\varphi, \top)$. A reduction step $(P,S) \leadsto (P',S')$ consists in picking a primitive constraint μ from P, and then applying all rules of S to S and μ. Reduction terminates with an empty pool or the detection of an inconsistency; that is with one of $(\top,S)$ or (P,S) where $\bot \in S$. We denote the reflexive transitive closure of $\leadsto$ by $\leadsto^*$.

Call a pair (P,S) *locally closed* if S is closed under one-step consequences with respect to algorithm S. *Reduction* is smallest binary relation on pairs (P,S) closed under the following rule:

$$(P,S) \quad \leadsto \quad (P \backslash \{\mu\} \wedge S', S \wedge \mu)$$

if S' contains all one-step consequences of $S \wedge \mu$ under S
which S does not already contain.

Lemma 2.14 (Invariants of Reduction)

Reduction performs equivalence transformations and preserves local closure. That is,

1. *If $(P,S) \leadsto (P',S')$ and (P,S) is locally closed, then (P',S') is locally closed, too.*

2. *If $(P,S) \leadsto (P',S')$, then $P \wedge S$ and $P' \wedge S'$ are equivalent.*

Proof. Straightforward using correctness of S. □

Corollary 5 (Correctness)

1. *If $(P,\top) \leadsto^* (\top,S)$, then S is S-closed and equivalent to P.*

2. *If $(P,\top) \leadsto^* (P',S)$ where $\bot \in S$, then P is non-satisfiable.*

Proof. Immediate from Lemma 2.14. □

Data Structures

Now let us consider the data structures for pool and store more closely. Let n_v, n_f, and n_l be the cardinalities of $\mathcal{V}(\varphi_0)$, $\mathcal{F}(\varphi_0)$, and $\mathcal{L}(\varphi_0)$, respectively, and let n be the size of φ_0. The *pool* can be implemented such that selection and deletion, as well as addition of arbitrary primitive constraints can be performed in constant time $O(1)$. We assume the *store* to consist of the following components.

1. An array of size n_v containing arity constraints $x\{\overline{f}\}$ (at most one per variable);
2. an array of size n_v containing label constraints $a\langle x\rangle$ (at most one per variable);
3. a table of size $n_v \cdot n_f$ containing selection constraints $x[f]y$ (at most one per variable and feature);
4. a table of size n_v^2 containing ordering constraints $x \subseteq y$;
5. a table of size ${n_v}^2$ containing non-disjointness constraints $x \not\parallel y$.

Access Operations

The store can be realised with tables and arrays of boolean values such as to provide for the following operations.

1. Add a primitive constraint in time $O(1)$,
2. given a variable x, test the presence of a label constraint $a\langle x\rangle$ and retrieve it in time $O(1)$;
3. given a variable x, test the presence of an arity constraint $x\{\overline{f}\}$ and retrieve it in time $O(1)$;
4. given a variable x and a feature f, test the presence of a selection constraint $x[f]y$ and retrieve it in $O(1)$;
5. test the presence of an inclusion $x \subseteq y$ or a non-disjointness constraint $x \not\parallel y$ in time $O(1)$;
6. given a variable x, retrieve the set of all y such that $x \subseteq y$ or $y \subseteq x$ is in the store in time $O(n_v)$;
7. given a variable x, retrieve the set of all y such that $x \not\parallel y$ is in the store in time $O(n_v)$.

Furthermore, for arbitrary symbols (features, labels, and variables) we can arrange for on $O(1)$-test of whether it occurs in φ at all.

Every primitive constraint and store can have at most $O(n)$ one-step consequences with respect to S. For instance, consider axiom (Desc-Incl): if $x[f]x'$ is fixed, then (Desc-Incl) has at most $O(n_v)$ one-step consequences, and if $x \subseteq y$ is fixed, then (Desc-Incl) has at most $O(n_f)$ one-step consequences. In both cases there are $O(n)$ since $n_v, n_f \leq n$. As the reader may want to check, this implies that for all rules but (Clash-Arity-II) the one-step consequences of some primitive constraint and a store can be computed in time $O(n)$. Now consider rule (Clash-Arity-II). We can check applicability of this rule in time $O(n)$ if one of the following holds.

- If no arity constraint every occurs, *i. e.*, if we consider satisfiability of $\mathrm{FT}_{\subseteq}^{ne}$ constraints only, rule (Clash-Arity-II) never applies.
- If the size of arity constraints is bounded, then the test $\{\overline{f}\} \neq \{\overline{g}\}$ takes constant time (in the size of the bound).
- If the arity constraints list the features according to a fixed order, then the test $\{\overline{f}\} \neq \{\overline{g}\}$ takes time $O(\mathcal{F}(\varphi)) = O(n)$.

If the size of the arity constraints is unbounded, and the features in the arity constraints are not statically ordered, we must define an order dynamically and order the features in $\{\overline{f}\}$ and $\{\overline{g}\}$ before we compare them. In this case, the test $\{\overline{f}\} \neq \{\overline{g}\}$ takes time $O(n_f \cdot \log n_f + n_f) = O(n \cdot \log n)$.

There are at most $O(n_f \cdot n_v^2 + 2 \cdot {n_v}^2 + n_l \cdot n_v)$ distinct primitive constraints. But since algorithm S derives only primitive constraints of the form $x \subseteq y$ and $x \not\parallel y$, there are at most $O({n_v}^2)$ proper addition operations on the store. The pool is extended only when some new primitive constraint is added to the store, hence $O({n_v}^2)$ times. In each case, there are at most $O(n)$ new consequences. Hence, the pool may grow up to $O(n^3)$ primitive constraints in the worst-case. However, for most of these (namely all but $O(n^2)$) one only needs to do the $O(1)$ test to notice that they are already contained in the store. Hence, the overall complexity is

$$O(1 \cdot n^3 + n \cdot n^2) \quad = \quad O(n^3)\,.$$

2.2.3.2. The Incremental Case

Now let us check that our analysis remains true for the incremental version of our algorithm. In an incremental algorithm, the input constraint can be added piece-wise to the pool. Note that our algorithm is already insensitive to the order in which primitive constraints are picked from the pool. The additional complication is that the number of symbols n_v, n_f, and n_l in φ is not known statically. However, by replacing the static tables and arrays by dynamically extensible hash tables we can still guarantee the complexity estimations on the access operations [60]. In the previous section, we have assumed the input constraint φ_0, and hence n_v, n_f, and n_l fixed. The argument, however, is valid for *arbitrarily large* n_v, n_f, and n_l. Hence, running the described algorithm on the conjunction of a sequence of constraints

$$\varphi_0,\ \varphi_1,\ \varphi_2,\ \ldots,\varphi_m$$

can never cost more than $O(n^3)$ where n is the overall size of $\bigwedge_{i=0}^{m} \varphi_i$. So, the algorithm has an *incremental* time complexity of $O(n^3)$.

2.3. Dropping the Non-emptiness Restriction

We extend our results from set constraints over non-empty sets of feature trees to the full domain of possibly empty sets of feature trees. Essentially all results carry over, with the notable exception that the satisfiability test now becomes an emptiness test.

The detour through the non-standard domain of non-empty sets is worthwhile. In this section we shall see that the emptiness test for $\mathrm{FT}_{\subseteq}(ar)$ is more complicated than the satisfiability test for $\mathrm{FT}^{ne}_{\subseteq}$ since it requires an explicit propagation of non-emptiness information.[12]

2.3.1. Emptiness Test

Since every $\mathrm{FT}_{\subseteq}(ar)$ constraint is satisfiable, a satisfiability test for $\mathrm{FT}_{\subseteq}(ar)$ does not make sense. Instead, we transform the satisfiability test for $\mathrm{FT}^{ne}_{\subseteq}(ar)$ into an *emptiness test* for the $\mathrm{FT}_{\subseteq}(ar)$. This is done in Figure 2.2 which uses a new ternary constraint of the form $x \not\parallel_z y$ with the following semantics:

$$\alpha \models_{\mathcal{P}(\mathcal{FT})} x \not\parallel_z y \quad \text{iff} \quad \alpha(x) \cap \alpha(y) \neq \emptyset \vee \alpha(z) = \emptyset$$

So, $x \not\parallel_z y$ is equivalent to the formula $x \cap y{=}\emptyset \rightarrow z{=}\emptyset$. Furthermore, $x{=}\emptyset$ is used as an abbreviation of $a\langle x\rangle \wedge b\langle x\rangle$ for arbitrarily fixed distinct a, b.

Notice in passing that the constraint $x \not\parallel y$ is not equivalent to the formula $\exists z(z{\subseteq}x \wedge z{\subseteq}y)$ over $\mathcal{P}(\mathcal{FT})$ since $\exists z(z{\subseteq}x \wedge z{\subseteq}y)$ is trivially true over possibly empty sets. More strongly, $x \not\parallel y$ is not expressible in $\mathrm{FT}_{\subseteq}(ar)$. To see this, notice that a set is non-disjoint with itself exactly if it is non-empty. Hence $x \not\parallel x$ is equivalent to $x \neq \emptyset$, whereas $\mathrm{FT}_{\subseteq}(ar)$ constraint cannot express emptiness (otherwise, $\mathrm{FT}_{\subseteq}(ar)$ could also express the inconsistent constraint $x \neq \emptyset \wedge a\langle x\rangle \wedge b\langle x\rangle$, in contradiction to Proposition 2.1).

[12] A similar observation holds for the polynomial result on entailment for $\mathrm{FT}_{\subseteq}$ in Section 3.1 that we obtain by detour through $\mathrm{FT}^{ne}_{\subseteq}$.

(Refl)	$x \subseteq x$			
(Trans)	$x \subseteq y \wedge y \subseteq z$	$\rightarrow$	$x \subseteq z$	
(Symm-Nondis)	$x \nparallel_z y$	$\rightarrow$	$y \nparallel_z x$	
(Incl-Nondis)	$x \subseteq y$	$\rightarrow$	$x \nparallel_x y$	
(Quasi-Trans)	$x \subseteq x' \wedge x \nparallel_z y$	$\rightarrow$	$x' \nparallel_z y$	
(Desc-Incl)	$x[f]x' \wedge x \subseteq y \wedge y[f]y'$	$\rightarrow$	$x' \subseteq y'$	
(Desc-Nondis)	$x[f]x' \wedge x \nparallel_z y \wedge y[f]y'$	$\rightarrow$	$x' \nparallel_z y'$	
(Empty-Sort)	$a\langle x\rangle \wedge x \nparallel_z y \wedge b\langle y\rangle$	$\rightarrow$	$z = \emptyset$	if $a \neq b$
(Empty-Arity-I)	$x[f]x' \wedge x \nparallel_z y \wedge y\{\overline{g}\}$	$\rightarrow$	$z = \emptyset$	if $f \notin \{\overline{g}\}$
(Empty-Arity-II)	$x\{\overline{f}\} \wedge x \nparallel_z y \wedge y\{\overline{g}\}$	$\rightarrow$	$z = \emptyset$	if $\{\overline{f}\} \neq \{\overline{g}\}$
(Empty-Prop-I)	$x = \emptyset \wedge x[f]y$	$\rightarrow$	$y = \emptyset$	
(Empty-Prop-II)	$x = \emptyset \wedge y[f]x$	$\rightarrow$	$y = \emptyset$	
(Empty-Prop-III)	$x = \emptyset \wedge y \subseteq x$	$\rightarrow$	$y = \emptyset$	
(Empty-Prop-IV)	$x = \emptyset \wedge x \nparallel_z y$	$\rightarrow$	$z = \emptyset$	

Figure 2.2.: Emptiness Test for $\mathrm{FT}_{\subseteq}(ar)$

Now consider Figure 2.2 more closely. Axioms (Refl) through (Empty-Arity-II) are obtained by straightforward adaptation from Figure 2.1. The most interesting axiom is (Ines-Nondis) which now derives from an inclusion $x \subseteq y$ the constraint $x \not\parallel_x y$ stating that x is empty if x and y have an empty intersection. The clash axioms are modified to infer emptiness now, and axioms (Empty-Prop-I) through (Empty-Prop-IV) propagate emptiness along all primitive constraints. Call E the fixed point algorithm induced by the axioms in Figure 2.2 with the additional control that the axiom (Refl) is only applied to occurring variables.

Theorem 6 (Emptiness Test for $\mathbf{FT}_{\subseteq}(ar)$)

Algorithm E is sound and complete, and it can be implemented such that it decides the emptiness problem of $FT_{\subseteq}(ar)$ constraints in incremental time $O(n^4)$ and space $O(n^3)$. In more detail:

(Soundness) The structure $\mathcal{P}(\mathcal{FT})$ is a model of the axioms in Figure 2.2.

(Complexity) If φ has size n, then algorithm E terminates after at most $O(n^3)$ steps, each of which can be implemented to take at most linear time.

(Completeness) If φ is an E-closed constraint with $\varphi \models_{\mathcal{P}(\mathcal{FT})} x{=}\emptyset$, then $x{=}\emptyset \in \varphi$.

Proof. Soundness follows easily by inspection of the rules. The complexity statement can be proven just as Theorem 3 in Section 2.2.3. The higher degrees of the polynomials are due to the fact that there is a cubic number of constraints of the form $x \not\parallel_z y$, where there were only quadratically many of the form $x \not\parallel y$ before. Completeness is shown as Proposition 2.17 below. □

Theorem 6 holds under the same preconditions as Theorem 3 (see the remark on Page 34): if the size of arity constraints is unbounded and a static order on the features cannot be assumed, then the incremental time complexity is $O(n^4 \cdot \log n)$ rather than $O(n^4)$.

In order to complete the proof, we need some additional machinery. For every $\mathrm{FT}_{\subseteq}$ constraint φ let $Empty(\varphi) = \{x \mid x{=}\emptyset \in \varphi\}$, and obtain $\varphi_{\neq\emptyset}$ from φ by first dropping all constraints that mention a variable $x \in Empty(\varphi)$, and then replacing all remaining constraints $x \not\parallel_z y$ by $x \not\parallel y$.

Proposition 2.15 (Eliminating Empty Variables)

Let φ be an E-closed $FT_{\subseteq}(ar)$ constraint. Then $\varphi_{\neq\emptyset}$ is satisfiable over $\mathcal{P}^+(\mathcal{FT})$.

Proof. One shows that $\varphi_{\neq\emptyset}$ is S-closed (and does not contain $\perp$) so that $\varphi_{\neq\emptyset}$ must be satisfiable over $\mathcal{P}^+(\mathcal{FT})$ by Proposition 2.8. □

Lemma 2.16 (Extending $\mathcal{P}^+(\mathcal{FT})$ solutions)

Let φ be an E*-closed $FT_\subseteq(ar)$ constraint and α be a $\mathcal{P}^+(\mathcal{FT})$-solution of $\varphi_{\neq\emptyset}$. Extend α to α' by mapping all $x \in Empty(\varphi)$ to the empty set and all remaining variables $x \in \mathcal{V}(\varphi)$ to an arbitrary non-empty set. Then $\alpha' \models_{\mathcal{P}(\mathcal{FT})} \varphi$.*

Proof. If α is a $\mathcal{P}^+(\mathcal{FT})$-solution of $\varphi_{\neq\emptyset}$ then α' is a $\mathcal{P}(\mathcal{FT})$-solution of $\varphi_{\neq\emptyset}$. One checks for all primitive constraints that mention a variable in *Empty*(φ) that α' is a solution for it. Hence $\alpha' \models_{\mathcal{P}(\mathcal{FT})} \varphi$. □

Proposition 2.17 (Completeness of the Emptiness Test)

If φ is an E*-closed constraint with $\varphi \models_{\mathcal{P}(\mathcal{FT})} x{=}\emptyset$ then $x{=}\emptyset \in \varphi$.*

Proof. Let φ be E-closed and assume that $x{=}\emptyset \notin \varphi$. This means that $x \notin Empty(\varphi)$. By Proposition 2.15, there exists a $\mathcal{P}^+(\mathcal{FT})$-solution α of $\varphi_{\neq\emptyset}$. By Lemma 2.16, then there exists a $\mathcal{P}(\mathcal{FT})$-solution α' of φ that extends α and satisfies $\alpha'(x) \neq \emptyset$. Hence $\varphi \not\models_{\mathcal{P}(\mathcal{FT})} x{=}\emptyset$. □

Finite Trees

In analogy to the case of non-empty sets (see Page 35) we can adapt the emptiness test in Figure 2.2 for $\mathrm{FT}_\subseteq(ar)$ to the case of finite trees by an occurs check axiom.

$$\text{(Empty-Occurs)} \quad x \subseteq x_i \wedge \bigwedge_{i=1}^{n}(x_i \subseteq y_i \wedge y_i[f_i]x_{i+1}) \wedge x_{n+1} \subseteq x \;\rightarrow\; x{=}\emptyset \qquad n \geq 1$$

and we can implement it such that we stay in incremental time $O(n^4)$ and space $O(n^3)$. So Theorem 6 carries over to the finite tree case.

2.3.2. Solving Union Constraints

Theorem 7 (Hardness of Satisfiability for $\mathbf{FT}_\subseteq(ar,\cup)$)

The satisfiability problem of $FT_\subseteq(ar,\cup)$ constraints is DEXPTIME-hard.

Proof. By reduction of the well-known DEXPTIME-complete emptiness problem of the intersection of two deterministic top-down tree automata [69, 185]. □

We do not elaborate on the details of this proof, because similar reductions have been given to prove DEXPTIME-hardness for co-definite set constraints and for set constraints with intersection [42, 44, 58]. Given these completeness results, it is also a good guess to assume DEXPTIME-completeness.

Conjecture 8 (Complexity of Satisfiability for FT$_\subseteq(ar,\cup)$)
The satisfiability problem of FT$_\subseteq(ar,\cup)$ is decidable in DEXPTIME.

The source of this high complexity is the union constraint; so it is natural to consider weaker approximations of union. The most prominent one is Mishra's interpretation of set constraints over over the non-standard domain of *path-closed sets* [132]. In this interpretation, all set expressions denote the smallest path-closed supersets of their standard set interpretation.[13] For instance, the term $f(a,a) \cup f(b,b)$ is interpreted by the set $\{f(a,a), f(a,b), f(b,a), f(b,b)\}$. Unfortunately, the satisfiability problem of co-definite set constraints interpreted over path-closed sets remains DEXPTIME-complete [44].

It is tempting to consider an even weaker approximation of union which would, for example, interpret the term $f(a,a) \cup g(b,b)$ by the set $\{f(a,a), f(a,b), f(b,a), f(b,b), g(a,a), g(a,b), g(b,a), g(b,b)\}$. This approximation may be interesting if its complexity is strictly smaller than DEXPTIME. We conjecture this to be the case, since, as it seems, this constraint system cannot encode the emptiness problem of the intersection of two deterministic top-down tree automata.

[13] This approximation is also called Cartesian closure or tuple-distributive approximation [132, 220].

3. Entailment for Set Constraints

In this chapter, we investigate the entailment problem for union-free fragments of our system of set constraints over feature trees. We give an algorithm to decide entailment $\varphi \models \varphi'$ for $\mathrm{FT}^{ne}_{\subseteq}$ (no union or arity constraints) in time $O(n^3)$ and we derive an entailment test for $\mathrm{FT}_{\subseteq}$ that takes time $O(n^4)$. For both $\mathrm{FT}^{ne}_{\subseteq}(ar)$ and $\mathrm{FT}_{\subseteq}(ar)$ (with arity constraints but no union constraints) we show that entailment $\varphi \models \varphi'$ is coNP-hard, and for both $\mathrm{FT}^{ne}_{\subseteq}$ and $\mathrm{FT}_{\subseteq}$ we show that entailment with existential quantification $\varphi \models \exists\overline{x}\varphi'$ is PSPACE-hard. All results hold over both the domains of sets of finite trees and sets of infinite trees.

Entailment is interesting in program analysis because it provides explanation for constraint simplification [13, 65, 89, 140, 141, 143, 173]. Simplification means to replace every constraint φ by a smaller one which is either entailed by φ and retains the distinguished solution(s), or which is logically equivalent to φ and retains all solutions. Retaining *all* solutions is crucial for a modular program analysis where the analysis of a complete program should be equivalent to the combination of separate analysis results for program components.

Consider some typical simplification steps. If a constraint entails the equality between two variables, then one of them can be replaced by the other one and then be eliminated. This strictly reduces the number of occurring variables and has an immediate impact on all further constraint processing. One also needs to get rid of variables in a constraint whose denotation is irrelevant for the analysis, provided the constraint is satisfiable so that there exists an appropriate denotation at all. Since such variables are often existentially quantified, this simplification implies minimising the number of existential quantifiers.

If the constraint system allows only "flat" terms like $f(x,y)$ that have only variables as immediate subterms, then terms like $f(g(a,b))$ are "flattened out" with auxiliary, existentially quantified, variables; for example, a constraint like $x{=}f(g(a))$ is replaced by the formula $\exists y \exists z (x{=}f(y) \wedge y{=}g(z) \wedge z{=}a)$. In this case, constraint simplification may involve entailment with existential quantification of the form $\varphi \models \exists \overline{x} \varphi'$; for example, to show that $f(a,x)$ is an instance of $f(y,x)$ one may have to check that $z{=}f(a,x)$ entails $\exists y (z{=}f(y,x))$.

Entailment has also been proposed as a mechanism to explain subtyping on so-called polymorphic constrained types [27, 121, 203]. There, entailment with existential quantification is used to model subtyping on polymorphic types with constrained quantification; for example the type $\forall \overline{x} \backslash \varphi . t$ (read: "type t for all $\overline{x}$ that satisfy φ") is a subtype of $\forall \overline{y} \backslash \varphi' . t'$ if $x{=}t \wedge \varphi$ entails $\exists \overline{y} (x{=}t' \wedge \varphi')$ for a fresh variable x.

These applications of entailment have motivated the research for complete entailment tests for various constraint systems and the related complexity questions. This issue is a fundamental one, but it also has some practical impact: complete entailment tests correspond to optimal constraint simplification algorithms that could always transform a constraint to an equivalent one with minimal size.

The design of complete entailment tests was more difficult than many researchers expected. Henglein and Rehof showed that the entailment problem of so-called structural subtyping constraints over finite trees is coNP-complete [89]. We have adapted their proof technique to show coNP-hardness of the entailment problem for two systems of set constraints (Ines [142] and atomic set constraints [85]) in [140]; furthermore, we showed for a system of ordering constraints over feature trees that the entailment problem with existential quantifiers even becomes PSPACE-complete [141, 145]. We present both hardness results in the context of our system of set constraints over feature trees. Very recently, Henglein and Rehof showed that entailment for structural subtyping constraints over infinite trees to be PSPACE-complete [90].

Luckily, constraint simplification needs not be optimal if it is "good enough" and can be implemented efficiently. From this point of view, the mentioned intractability results encourage the investigation of *sound approximations* of entailment for constraint simplification. For the application in subtyping constrained types these results seem to be more serious, since there complete entailment plays a crucial role to model well-typedness.

As an aside notice also that the entailment problem is needed for a constraint system to be integrated into concurrent constraint programming, because entailment explains the semantics of CC-conditionals ("ask").

3.1. Entailment with Polynomial Complexity

We show that the entailment problem $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \varphi'$ for $\mathrm{FT}^{ne}_{\subseteq}$ has cubic complexity. We also prove that $\mathrm{FT}^{ne}_{\subseteq}$-constraints have the independence property of negated constraints [50, 115, 116]: We conclude that even the satisfiability problem for positive and negative $\mathrm{FT}^{ne}_{\subseteq}$ constraints remains within the same complexity.

$$
\begin{array}{rll}
\varphi \vdash a\langle x\rangle & \text{iff} & \text{exists } x' \text{ such that } x \subseteq x' \wedge a\langle x'\rangle \in \varphi \\
\varphi \vdash x \subseteq y & \text{iff} & x \subseteq y \in \varphi \ \text{ or } \ x = y \\
\varphi \vdash x \nparallel y & \text{iff} & x \nparallel y \in \varphi \ \text{ or } \ x = y \\
\varphi \vdash x[f]y & \text{iff} & \varphi \vdash y \subseteq x[f] \text{ and } \varphi \vdash x?[f] \subseteq y \\
\text{where } \varphi \vdash x \subseteq y[f] & \text{iff} & \text{exist } x', y' \text{ such that } x \subseteq x' \wedge y'[f]x' \wedge y' \subseteq y \in \varphi \\
\varphi \vdash x?[f] \subseteq y & \text{iff} & \text{exist } x', y' \text{ such that } x \subseteq x' \wedge x'[f]y' \wedge y' \subseteq y \in \varphi
\end{array}
$$

Figure 3.1.: Syntactic Containment for $FT^{ne}_{\subseteq}$

In order to decide entailment $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \varphi'$ between $FT^{ne}_{\subseteq}$ constraints we must first decide satisfiability of φ, since entailment is trivial if φ is non-satisfiable. As we shall prove, the entailment problem for $FT^{ne}_{\subseteq}$ is not harder than its satisfiability problem.

3.1.1. Syntactic Containment

Let us write μ for the primitive $FT^{ne}_{\subseteq}$ constraints.

$$\mu \quad ::= \quad x \subseteq y \quad | \quad x \nparallel y \quad | \quad x[f]x' \quad | \quad a\langle x\rangle$$

An $FT^{ne}_{\subseteq}$ constraint φ entails another one φ' if and only if φ entails all primitive constraints in φ'. As it turns out, the constraint system $FT^{ne}_{\subseteq}$ is so weak that φ only entails primitive constraints that are already *syntactically* contained in φ (Proposition 3.2). Since primitive entailment is linear (Lemma 3.3), this yields an incremental entailment test that takes quadratic time in the size of φ if φ is S-closed, and cubic time in general.

Figure 3.1 defines the notion that a constraint φ *syntactically contains* μ, written $\varphi \vdash \mu$.

Example 9 (Entailment of Selection Constraints)

As an illustration for the most complicated case of syntactic containment, namely the one dealing with selection constraints, define the following constraint (depicted to the right) and observe that it entails $x[f]y$.

$$
\begin{tikzcd}
 & & u \arrow[d, dash, "f"'] & \subseteq & x & \subseteq & v \arrow[d, dash, "f"] & & \\
y & \subseteq & u' & & & & v' & \subseteq & y
\end{tikzcd}
$$

$$y \subseteq u' \wedge u[f]u' \wedge u \subseteq x \wedge x \subseteq v \wedge v[f]v' \wedge v' \subseteq y \tag{3.1}$$

In order to show that syntactic containment coincides with entailment, we must show in particular that syntactic containment is complete with respect to entailment. Before we show this, notice the impact of arity constraints:

Example 10 (Syntactic Containment with Arity Constraints)

Syntactic containment is no longer complete if arity constraints are added. In particular, constraints may entail many non-trivial inclusions then. For instance, consider the following judgement:

$$a\langle x\rangle \wedge x\{\} \wedge a\langle y\rangle \wedge y\{\} \quad \models_{\mathcal{P}^+(\mathcal{FT})} \quad x\subseteq y \wedge y\subseteq x \tag{3.2}$$

We must show that no primitive constraint μ is entailed by an $\mathrm{FT}_{\subseteq}^{ne}$ constraint φ that is not already contained in φ. To show this it suffices to find a solution of φ that contradicts μ. More strongly, we show that there is a single solution that contradicts all such μ *at the same time*. We show this by means of a satisfiable formula that strengthens φ and entails the negation of all relevant μ. Such a formula is called saturated. Its existence will also give us the independence property for $\mathrm{FT}_{\subseteq}^{ne}$.

Lemma 3.1 (Existence of a Saturated Formula)

For every satisfiable $FT_{\subseteq}^{ne}$ constraint φ, there exists a formula $Sat(\varphi)$, called a saturation *of φ, with the following properties.*

1. *$Sat(\varphi)$ is satisfiable.*
2. *$Sat(\varphi) \models_{\mathcal{P}^+(\mathcal{FT})} \varphi$.*
3. *$\forall\mu$: If $\mathcal{V}(\mu) \subseteq \mathcal{V}(\varphi)$, then $\varphi \nvdash \mu$ implies $Sat(\varphi) \models_{\mathcal{P}^+(\mathcal{FT})} \neg\mu$.*

Proof. The constructive existence proof of $\mathrm{Sat}(\varphi)$ is technically involved and postponed to Section 3.1.2 which begins on Page 55. There, Definition 5 defines a formula $\mathrm{Sat}(\varphi)$ in such a way that $\mathrm{Sat}(\varphi)$ entails φ by construction. Lemmas 3.4 and 3.5 prove that $\mathrm{Sat}(\varphi)$ is satisfiable. The third claim follows from Lemma 3.7. $\square$

Proposition 3.2 (Entailment = Syntactic Containment)

Entailment and syntactic containment coincide for primitive $FT_{\subseteq}^{ne}$ constraints $x \subseteq y$, $x \not\parallel y$, $a\langle x\rangle$, and $x[f]y$: if φ is an S-closed constraint not containing $\perp$ and μ is a primitive constraint, then $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \mu$ if and only if $\varphi \vdash \mu$.

Proof. It is easy to see that syntactic containment is *semantically correct* (*i. e.*, $\varphi \vdash \mu$ implies $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \mu$). It remains to show that syntactic containment is *semantically complete* (*i. e.*, $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \mu$ implies $\varphi \vdash \mu$). So, assume $\varphi \models \mu$. If $\mathcal{V}(\mu) \not\subseteq \mathcal{V}(\varphi)$ then μ is of the form $x\subseteq x$ or $x \not\parallel x$ such that $\varphi \vdash \mu$ is trivial. Otherwise, assume $\mathcal{V}(\mu) \subseteq \mathcal{V}(\varphi)$. Now let $\mathrm{Sat}(\varphi)$ be the saturation formula postulated by Lemma 3.1. By Property 2, $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \mu$ implies $\mathrm{Sat}(\varphi) \models \mu$. With Property 1, this yields $\mathrm{Sat}(\varphi) \not\models_{\mathcal{P}^+(\mathcal{FT})} \neg\mu$, and Property 3 implies $\varphi \vdash \mu$. $\square$

Lemma 3.3 (Primitive Entailment is Linear)
Given an S*-closed constraint* φ *of size n, we can compute a representation of* φ *in time* $O(n)$ *that allows to test syntactic containment* $\varphi \vdash \mu$ *for selection constraints in time* $O(n)$*, and for all other* μ *in time* $O(1)$.

Proof. In a linear operation we enter all primitive constraints in φ into the data structures described in Section 2.2.3. The complexity statement for labelling, arity, and inclusion constraints immediately follows directly from the properties of the data structures. To check containment of a selection constraint $x[f]y$, we proceed as follows.

1. Check whether there exists $z \in \mathcal{V}(\varphi)$ such that $z \subseteq x \in \varphi$, and then whether there exists z' such that $z[f]z' \in \varphi$ and $y \subseteq z' \in \varphi$, and
2. check whether there exists $z \in \mathcal{V}(\varphi)$ such that $x \subseteq z \in \varphi$, and then whether there exists z' such that $z[f]z' \in \varphi$ and $z' \subseteq y \in \varphi$

Clearly, $x[f]y$ is syntactically contained in φ if and only if both checks succeed. Since there are two tests for every variable $z \in \mathcal{V}(\varphi)$, this is a linear-time operation. □

Theorem 11 (Independence for $\mathbf{FT}^{ne}_{\subseteq}$)
The constraint system $FT^{ne}_{\subseteq}$ *has the following independence property: for every* $k \geq 1$ *and constraints* $\varphi, \varphi_1, \ldots, \varphi_k$*, it holds that*

$$\textit{if} \quad \varphi \models_{\mathcal{P}^+(\mathcal{FT})} \bigvee_{i=1}^{k} \varphi_i \quad \textit{then} \quad \exists i, 1 \leq i \leq k : \varphi \models_{\mathcal{P}^+(\mathcal{FT})} \varphi_i$$

Proof. Assume $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \bigvee_{i=1}^{k} \varphi_i$. If φ is unsatisfiable we are done. Also, if $\varphi \wedge \varphi_j$ is nonsatisfiable for some j, then

$$\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \bigvee_{i=1}^{k} \varphi_i \quad \text{iff} \quad \varphi \models_{\mathcal{P}^+(\mathcal{FT})} \bigvee_{i=1, i \neq j}^{k} \varphi_i$$

Hence we can assume, without loss of generality, that φ and $\varphi \wedge \varphi_i$ are satisfiable for all i, and that φ is S-closed and does not contain $\perp$. If there exists an i such that $\varphi \vdash \mu$ for all μ with $\mu \in \varphi_i$, then $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \varphi_i$ and we are done by Proposition 3.2. Otherwise, there exists $\mu_i \in \varphi_i$ for every i such that $\varphi \not\vdash \mu_i$. Let $\mathrm{Sat}(\varphi)$ be the formula postulated by Lemma 3.1. Without loss of generality, we can assume that $\mathcal{V}(\varphi_i) \subseteq \mathcal{V}(\varphi)$ for all i. Hence $\mathcal{V}(\mu_i) \subseteq \mathcal{V}(\varphi_i)$ implies $\mathrm{Sat}(\varphi) \models \neg\mu_i$ by Property 3. Therefore:

$$\mathrm{Sat}(\varphi) \models_{\mathcal{P}^+(\mathcal{FT})} \bigwedge_{i=1}^{k} \neg\varphi_i \,.$$

Since $\mathrm{Sat}(\varphi)$ is satisfiable and entails φ (Properties 1 and 2), this contradicts our assumption that $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \bigvee_{i=1}^{k} \varphi_i$. $\square$

The independence property of negated constraints is a fundamental property of constraint systems, which intuitively says that the constraint system cannot express disjunction [105]; this can drastically simplify reasoning with disjunctive formulas. We do not investigate the independence property in its own right (but see the remark at the end of this section). For more details and further references on independence see [42, 105].

Theorem 12 (Entailment and Negation for $\mathrm{FT}_{\subseteq}^{ne}$)
If $\varphi, \varphi_1, \ldots, \varphi_k$, $k \geq 1$ are $FT_{\subseteq}^{ne}$ constraints with sizes $n, n_1, \ldots, n_k$, then satisfiability of $\varphi \wedge \neg\varphi_1 \wedge \ldots \wedge \neg\varphi_k$ is decidable in time $O(n^3 + n \cdot \Sigma_{i=1}^{k} n_i)$ and space $O(n^2 + \Sigma_{i=1}^{k} n_i)$.

Proof. If φ is non-satisfiable then $\varphi \wedge \neg\varphi_1 \wedge \ldots \wedge \neg\varphi_k$ is trivially non-satisfiable. By Theorem 3, satisfiability of φ can be decided in time $O(n^3)$ and space $O(n^2)$. Now assume φ to be satisfiable and S-closed. By the Independence Theorem 11, $\varphi \wedge \neg\varphi_1 \wedge \ldots \wedge \neg\varphi_k$ is non-satisfiable if and only if $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \varphi_i$ for some i, and this is equivalent to $\varphi \not\models_{\mathcal{P}^+(\mathcal{FT})} \mu$ for some i and all primitive constraints $\mu \in \varphi_i$. By Proposition 3.2, $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \mu$ iff $\varphi \vdash \mu$, hence it suffices to decide syntactic containment for every μ contained in some φ_i. For each i, there are $O(n_i)$ many such μ to be tested for syntactic containment, each of which takes time $O(n)$ by Lemma 3.3. Hence non-satisfiability of $\varphi \wedge \neg\varphi_1 \wedge \ldots \wedge \neg\varphi_k$ can be tested in an additional time $O(n \cdot \sum_{i=1}^{k} n_i)$. The overall time complexity adds up to $O(n^3 + n \cdot \sum_{i=1}^{k} n_i)$, and the total space complexity is $O(n^2 + n \cdot \sum_{i=1}^{k} n_i)$. $\square$

Corollary 13 (Satisfiability of Positive and Negative $\mathrm{FT}_{\subseteq}^{ne}$ Constraints)
If φ and φ' are $FT_{\subseteq}^{ne}$ constraints with sizes n and n', then entailment $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \varphi'$ is decidable in time $O(n^3 + n \cdot n')$ and space $O(n^2 + n')$.

Finite Trees

Theorem 12 carries over to the domain of finite trees. In order to check $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \varphi'$ we just need to bring φ into closed form with respect to algorithm S plus the occurs check (Occurs) on Page 35. The second step remains unchanged: Simply check whether all primitive constraints in φ' are syntactically contained in the closure of φ.

Conjectures on Independence

We conjecture that the independence property holds for $\mathrm{FT}_{\subseteq}^{ne}(ar)$ if we are given an infinite set of labels, and even remains to hold when existential quantifiers are admitted (that is, $\varphi \models \bigvee_i \exists\overline{x}_i\varphi_i$ implies $\bigvee_i \varphi \models \exists\overline{x}_i\varphi_i$). For $\mathrm{FT}_{\subseteq}(ar)$, independence fails because $a\langle x\rangle \wedge a\langle y\rangle \models x\subseteq y \vee y \subseteq x$ holds. For $\mathrm{FT}^{ne}_{\subseteq}(ar)$ with a finite set $\mathcal{L} = \{a_1,\ldots,a_n\}$ of labels, independence also fails because $x\{\} \wedge a_1\langle y_1\rangle \wedge \ldots \wedge a_n\langle y_n\rangle \models_{\mathcal{P}^+(\mathcal{F}\mathcal{T})} y_1\subseteq x \vee \ldots \vee y_n\subseteq x$. Given an infinite set of labels, however, independence for $\mathrm{FT}^{ne}_{\subseteq}(ar)$ may well hold.

We have two reasons for these conjectures on $\mathrm{FT}^{ne}_{\subseteq}(ar)$. First, Charatonik and Podelski have shown that set constraints with intersection have the independence property when interpreted over non-empty sets of trees [42] and given an infinite signature; over this domain, set constraints with intersection subsume the constraint system Ines of inclusion constraints over constructor trees [142], and Ines is closely related to $\mathrm{FT}^{ne}_{\subseteq}(ar)$. For the extensibility with existential quantifiers we draw intuition from the related constraint system $\mathrm{FT}_{\leq}$ [143] (see Section 3.3.2). The constraints of $\mathrm{FT}_{\leq}$ coincide with $\mathrm{FT}^{ne}_{\subseteq}$ constraints but their interpretation is over the domain $\mathcal{F}\mathcal{T}$ of feature trees. We have shown in [143] that $\mathrm{FT}_{\leq}$ has the independence property without existential quantifiers, and that independence fails in presence of existential quantifiers; but a basic counter example for independence with existential quantification in $\mathrm{FT}_{\leq}$ does not work for $\mathrm{FT}^{ne}_{\subseteq}$.[14]

3.1.2. Saturation

We complete the proof of Proposition 3.2 by constructing a saturated formula as postulated by Lemma 3.1. To this end, we employ operators Γ_1 and Γ_2 on constraints. The operator Γ_2 is defined such that $\Gamma_2(\varphi)$ disentails all μ except selection constraints (that is, those of the form $x \not\parallel y$, $x\subseteq y$, and $a\langle x\rangle$), which are not syntactically contained in φ (Lemma 3.6). The operator Γ_1 is necessary to also disentail selection constraints. Given a constraint φ, $\Gamma_1(\varphi)$ extends it such that $\Gamma_2(\Gamma_1(\varphi)$ disentails all relevant μ. In a sense, Γ_1 serves as a "preprocessor" for Γ_2.

Definition 4 (Γ_1 and Γ_2)

Let φ be a constraint. For all $x \in \mathcal{V}(\varphi)$ and $f \in \mathcal{F}(\varphi)$ let v_{xf} be a fresh variable. Depending on this choice we define $\Gamma_1(\varphi)$ as follows, where cl denotes the S*-closure of a constraint:*

$$\Gamma_1(\varphi) =_{def} \quad cl(\varphi \wedge \bigwedge\{x[f]v_{xf} \mid x \in \mathcal{V}(\varphi),\ f \in \mathcal{F}(\varphi)\})$$

Furthermore, let v_1 and v_2 be distinct fresh variables, a_1 and a_2 be distinct labels, and for every pair of variables $x,y \in \mathcal{V}(\varphi)$, let f_x and f_{xy} be fresh features. We define

[14] The corresponding $\mathrm{FT}^{ne}_{\subseteq}$ formula $a\langle y\rangle \wedge y\subseteq x \rightarrow \exists b\,(b\langle z\rangle \wedge z\subseteq x) \vee a\langle x\rangle$ (where $a \neq b$) is *not* valid: if $a \neq c$ and $b \neq c$, then the set $\{a,c\}$ satisfies the left hand side of the implication but none of the disjuncts on the right hand side.

$\Gamma_2(\varphi)$ *depending on* $v_1, v_2, a_1, a_2, f_x, f_{xy}$ *as follows:*

$$\begin{array}{rlll} \Gamma_2(\varphi) =_{def} & \varphi & \wedge \ \bigwedge\{x[f_x]v_x \ \wedge \ \neg\exists y' (y[f_x]y') \mid \varphi \not\vdash y{\subseteq}x, \ x,y \in \mathcal{V}(\varphi)\} & (1) \\ & & \wedge \ \bigwedge\{x[f_{xy}]v_1 \ \wedge \ y[f_{xy}]v_2 \mid \varphi \not\vdash x \not\parallel y, \ x,y \in \mathcal{V}(\varphi)\} & (2) \\ & & \wedge \ \bigwedge\{x \not\parallel v_1 \ \wedge \ x \not\parallel v_2 \mid \forall a \in \mathcal{L}: \ \varphi \not\vdash a\langle x\rangle, \ x \in \mathcal{V}(\varphi)\} & (3) \\ & & \wedge \ a_1\langle v_1\rangle \wedge a_2\langle v_2\rangle & (4) \end{array}$$

Example 14 (Contradicting Feature Selection Constraints)
For illustration of Γ_1 and Γ_2 consider the constraint

$$\varphi_{contra} \ =_{def} \ x[f]x \wedge y{\subseteq}x \tag{3.3}$$

which is S-closed up to trivial and non-disjointness constraints and which does not entail $x[f]y$. In order to disentail $x[f]y$ we first compute $\Gamma_1(\varphi)$ by adding $x[f]v_{xf}$ and $y[f]v_{yf}$ to φ_{contra} and then computing the S-closure. Now, $\Gamma_1(\varphi_{contra})$ is (up to trivial and non-disjointness constraints)

$$\begin{array}{rcl} \Gamma_1(\varphi_{contra}) & = & x[f]x \wedge y{\subseteq}x \wedge x[f]v_{xf} \wedge y[f]v_{yf} \wedge \\ & & v_{yf}{\subseteq}v_{xf} \wedge v_{xf}{\subseteq}x \wedge x{\subseteq}v_{xf} \wedge y{\subseteq}v_{xf} \end{array} \tag{3.4}$$

Observe that $\Gamma_1(\varphi_{contra})$ does not contain $v_{xf}{\subseteq}y$; that is, $\Gamma_1(\varphi_{contra}) \not\vdash v_{xf}{\subseteq}y$. Now clause (1) of $\Gamma_2(\Gamma_1(\varphi_{contra}))$ disentails $v_{xf}{\subseteq}y$ by asserting that y allows selection at feature f_y while v_{xf} does not. Hence, $\Gamma_2(\Gamma_1(\varphi_{contra}))$ also disentails $x[f]y$.

Lemma 3.4 (Properties of Γ_1)
Let φ be an S-closed constraint not containing $\bot$. Then $\Gamma_1(\varphi)$ is satisfiable and satisfies the following two properties for all primitive constraints μ:

1. *If $\varphi \not\vdash \mu$ and $\mathcal{V}(\mu) \subseteq \mathcal{V}(\varphi)$, then $\Gamma_1(\varphi) \not\vdash \mu$.*
2. *If $\varphi \not\vdash x[f]y$, then $\Gamma_1(\varphi) \not\vdash y{\subseteq}v_{xf}$ or $\Gamma_1(\varphi) \not\vdash v_{xf}{\subseteq}y$.*

Proof. Let n be the cardinality of the set $V = \{v_{xf} \mid x \in \mathcal{V}(\varphi) \text{ and } f \in \mathcal{F}(\varphi)\}$ and fix an enumeration *var* from $\{1,\ldots,n\}$ into V. Then consider the following sequence of constraints

$$\begin{array}{rcll} \varphi_0 & = & \varphi & \\ \varphi_n & = & \mathrm{cl}(\varphi_{n-1} \wedge x[f]v_{xf}) & \text{if } n > 0 \text{ and } var(n) = v_{xf} \end{array}$$

Apparently, $\Gamma_1(\varphi) = \varphi_n$. In order to show that $\Gamma_1(\varphi)$ is satisfiable, we give an inductive construction of the form of the φ_i, for all i, and show that each of them is satisfiable.

φ_0 is S-closed and hence satisfiable by assumption. For the induction step, assume that φ_{i-1} is S-closed for an i, $0 < i \leq n$, with $var(i) = v_{xf}$. We show that $\varphi_i = \mathrm{cl}(\varphi_{i-1} \wedge x[f]v_{xf}) = \widetilde{\varphi_{i-1}}$ where

$$
\begin{aligned}
\widetilde{\varphi_{i-1}} = \varphi_{i-1} \wedge\ & x[f]v_{xf} \wedge v_{xf}\subseteq v_{xf} \wedge v_{xf} \not\parallel v_{xf} && (4.1)\\
\wedge\ & \bigwedge\{z\subseteq v_{xf} \mid \varphi_{i-1} \vdash z \subseteq x[f]\} && (4.2)\\
\wedge\ & \bigwedge\{v_{xf}\subseteq z \mid \varphi_{i-1} \vdash x?[f] \subseteq z\} && (4.3)\\
\wedge\ & \bigwedge\{v_{xf} \not\parallel z \wedge z \not\parallel v_{xf} \mid \text{ex. } y : \varphi_{i-1} \vdash y?[f] \subseteq z \text{ and } x \not\parallel y \in \varphi_{i-1}\} && (4.4)\\
\wedge\ & \bigwedge\{v_{xf} \not\parallel z \wedge z \not\parallel v_{xf} \mid \text{ex. } y : \varphi_{i-1} \vdash y \subseteq x[f] \text{ and } z \not\parallel y \in \varphi_{i-1}\} && (4.5)
\end{aligned}
$$

It is clear that $\widetilde{\varphi_{i-1}}$ is contained in φ_i, hence it suffices to show that $\widetilde{\varphi_{i-1}}$ is S-closed. The S-closedness of φ_i is proved by a case distinction. (Refl) follows from clause (4.1), and (Symm-Nondis) follows from clauses (4.4) and (4.5). The descend axioms (Desc-Incl) and (Desc-Nondis) do not apply to $\widetilde{\varphi_{i-1}}$ since no selection constraint on v_{xf} is added, and the clash axioms does not apply to $\widetilde{\varphi_{i-1}}$ because no labeling or arity constraints on v_{xf} are added. We check the remaining cases (Trans), (Incl-Nondis) and (Quasi-Trans).

(Trans) Assume $u\subseteq v \wedge v\subseteq w \in \widetilde{\varphi_{i-1}}$. We must show that $u\subseteq w \in \widetilde{\varphi_{i-1}}$. We make a case distinction depending on which of the variables x, y, z equal v_{xf}.

If $u, v, w \neq v_{xf}$, then $u\subseteq v \wedge v\subseteq w \in \varphi_{i-1}$. Hence, due to S-closedness of φ_i, $u\subseteq w \in \varphi_{i-1}$, and therefore $u\subseteq w \in \widetilde{\varphi_{i-1}}$.

If $u = v = v_{xf}$, then $u\subseteq w \equiv v_{xf}\subseteq v_{xf} \in \widetilde{\varphi_{i-1}}$ follows from clause (4.1).

If $u = v = v_{xf}$ and $w \neq v_{xf}$, then $u\subseteq w \equiv v_{xf}\subseteq w \in \widetilde{\varphi_{i-1}}$ follows from the assumption that $v\subseteq w \equiv v_{xf}\subseteq w \in \widetilde{\varphi_{i-1}}$. The case $u \neq v_{xf}$ and $y = z = v_{xf}$ is symmetric.

If $u = v_{xf}$ and $v, w \neq v_{xf}$, then $u\subseteq v \equiv v_{xf}\subseteq v \in \widetilde{\varphi_{i-1}}$ implies, by clause (4.3), that $\varphi_{i-1} \vdash x?[f] \subseteq v$. By S-closedness of φ_{i-1} (Trans) it follows that $\varphi_{i-1} \vdash x?[f] \subseteq w$ and hence, by clause (4.3) again, $u\subseteq w \equiv v_{xf}\subseteq w \in \widetilde{\varphi_{i-1}}$.

The case $w = v_{xf}$ and $u, v \neq v_{xf}$ is symmetric, using clause (4.2) instead of clause (4.3).

If $u, w \neq v_{xf}$ and $v = v_{xf}$, then, by clauses (4.2) and (4.3), $\varphi_{i-1} \vdash u \subseteq x[f]$ and $\varphi_{i-1} \vdash x?[f] \subseteq w$. By S-closedness of φ_{i-1}, (Trans) and (Desc-Incl), it follows that $u\subseteq w \in \varphi_{i-1}$ and hence $u\subseteq w \in \widetilde{\varphi_{i-1}}$.

(Incl-Nondis) Assume $u\subseteq v \in \widetilde{\varphi_{i-1}}$. We must show that $u \not\parallel v \in \widetilde{\varphi_{i-1}}$.

If $u, v \neq v_{xf}$, then $u \not\parallel v \in \widetilde{\varphi_{i-1}}$ follows from S-closedness of φ_{i-1}.

If $u = v = v_{xf}$, then $u \not\parallel v \in \widetilde{\varphi_{i-1}}$ follows from clause (4.1).

If $u = v_{xf}$ and $v \neq v_{xf}$, then, by clause (4.3) $\varphi_{i-1} \vdash x?[f] \subseteq v$. By S-closedness of φ_{i-1}, (Refl) and (Incl-Nondis), $x \not\parallel x \in \varphi_{i-1}$ and hence, by clause (4.4), $u \not\parallel v \equiv v_{xf} \not\parallel v \in \widetilde{\varphi_{i-1}}$.

The case $v = v_{xf}$ and $u \neq v_{xf}$ is symmetric, using clause (4.5) instead of clause (4.4) and $u \not\parallel u \in \varphi_{i-1}$.

(Quasi-Trans) Assume $u \not\parallel v \wedge v \subseteq w \in \widetilde{\varphi_{i-1}}$. We must show that $u \not\parallel w \in \widetilde{\varphi_{i-1}}$.

If $u, v, w \neq v_{xf}$, then $u \not\parallel w \in \widetilde{\varphi_{i-1}}$ follows from S-closedness of φ_{i-1}.

If $u = v = v_{xf}$ and $w \neq v_{xf}$, then, by clause (4.3), $\varphi_{i-1} \vdash x?[f] \subseteq w$. By S-closedness of φ_{i-1} we know that $x \not\parallel x \in \varphi_{i-1}$ and hence, by clause (4.4), $u \not\parallel w \equiv v_{xf} \not\parallel w \in \widetilde{\varphi_{i-1}}$.

If $v = w = v_{xf}$ and $u \neq v_{xf}$, then $u \not\parallel w \equiv u \not\parallel v_{xf} \in \widetilde{\varphi_{i-1}}$ follows from $u \not\parallel v \equiv u \not\parallel v_{xf} \in \widetilde{\varphi_{i-1}}$.

If $u = w = v_{xf}$ and $v \neq v_{xf}$, then $u \not\parallel w \equiv v_{xf} \not\parallel v_{xf} \in \widetilde{\varphi_{i-1}}$ follows from clause (4.1).

If $w = v_{xf}$ and $u, v \neq v_{xf}$, then by clause (4.2), $\varphi_{i-1} \vdash v \subseteq x[f]$, and hence, by clause (4.5), $u \not\parallel w \equiv u \not\parallel v_{xf} \in \widetilde{\varphi_{i-1}}$.

If $u = v_{xf}$ and $v, w \neq v_{xf}$, then $u \not\parallel v \equiv v_{xf} \not\parallel v$ could have been added by clause (4.4) or clause (4.5).

(4.4) Then, by clause (4.4), there exists v' such that $\varphi_{i-1} \vdash v'?[f] \subseteq v$ and $x \not\parallel v' \in \varphi_{i-1}$. By S-closedness of φ_{i-1} (Trans), $\varphi_{i-1} \vdash v'?[f] \subseteq w$, and hence, by clause (4.4) again, $u \not\parallel w \equiv v_{xf} \not\parallel w \in \widetilde{\varphi_{i-1}}$.

(4.5) Then, by clause (4.5), there exists v' such that $\varphi_{i-1} \vdash v' \subseteq x[f]$ and $v \not\parallel v' \in \varphi_{i-1}$. By S-closedness of φ_{i-1}, (Quasi-Trans) and (Symm-Nondis), $w \not\parallel v' \in \varphi_{i-1}$, so that $u \not\parallel w \equiv v_{xf} \not\parallel w \in \widetilde{\varphi_{i-1}}$ by clause (4.5) again.

If $v = v_{xf}$ and $u, w \neq v_{xf}$, then $u \not\parallel v \equiv u \not\parallel v_{xf}$ could have been added by clause (4.4) or clause (4.5). The argument is similar to the previous one.

Now we check properties (1) and (2) of $\Gamma_1(\varphi)$. In both cases, we prove the contraposed claim.

1. Assume that $\Gamma_1(\varphi) \vdash \mu$ and $\mathcal{V}(\mu) \subseteq \mathcal{V}(\varphi)$. We show that $\varphi \vdash \mu$ by case distinction over μ.

$\mu = x{\subseteq}y$ or $\mu = x \nparallel y$: If $\Gamma_1(\varphi) \vdash \mu$ then $\mu \in \Gamma_1(\varphi)$ or $x = y$. If $x = y$, then trivially $\varphi \vdash \mu$. Otherwise, if $x \neq y$, note that all basic constraints which are contained in $\Gamma_1(\varphi)$ but not in φ contain at least one fresh variable. Hence from $x, y \in \mathcal{V}(\mu) \subseteq \mathcal{V}(\varphi)$ we obtain $\mu \in \varphi$, and therefore $\varphi \vdash \mu$.

$\mu = a\langle x\rangle$: If $\Gamma_1(\varphi) \vdash a\langle x\rangle$ then there exists a variable x' such that $x{\subseteq}x' \wedge a\langle x'\rangle \in \Gamma_1(\varphi)$. By inspection of the form of $\Gamma_1(\varphi) = \varphi_n$ one obtains that $x' \in \mathcal{V}(\varphi)$ and hence $a\langle x'\rangle \in \varphi$. In combination with the assumption that $\mathcal{V}(\mu) \subseteq \mathcal{V}(\varphi)$ which gives $x \in \mathcal{V}(\varphi)$ we conclude that $\varphi \vdash a\langle x\rangle$.

$\mu = x[f]y$: If $\Gamma_1(\varphi) \vdash x[f]y$ then there exist variables u, u' and v, v' such that

$$\Gamma_1(\varphi) \vdash u{\subseteq}x \wedge x{\subseteq}v\,,$$
$$\Gamma_1(\varphi) \vdash y{\subseteq}u' \wedge v'{\subseteq}y\,, \text{ and}$$
$$u[f]u' \wedge v[f]v' \in \Gamma_1(\varphi)$$

By assumption, $x, y \in \mathcal{V}(\mu) \subseteq \mathcal{V}(\varphi)$. Also $u, v \in \mathcal{V}(\varphi)$ holds since $\Gamma_1(\varphi) = \varphi_n$ contains no selection constraints on fresh variables.

We can without loss of generality assume that $u', v' \in \mathcal{V}(\varphi)$. In this case $\varphi \vdash x[f]y$ follows easily.

To see why we can assume $u', v' \in \mathcal{V}(\varphi)$, suppose $u' \notin \mathcal{V}(\varphi)$. Then $u' = v_{uf}$ by construction of $\Gamma_1(\varphi) = \varphi_n$: Let $var(v_{uf}) = i$. Then by Clause (4.2) $\varphi_{i-1} \vdash y \subseteq u[f]$ which means that there must exist variables $w, w' \in \mathcal{V}(\varphi_{i-1})$ such that $y{\subseteq}w' \wedge w[f]w' \wedge w{\subseteq}x \in \varphi_{i-1}$. Hence, we can replace w, w' for u, u' above and obtain the same situation up to renaming. By induction over $var(v_{uf})$ we find replacement for u', v' in $\mathcal{V}(\varphi)$. The argument for v' is dual.

2. Assume that $\Gamma_1(\varphi) \vdash z{\subseteq}v_{xf}$ and $\Gamma_1(\varphi) \vdash v_{xf}{\subseteq}z$. Then by clauses (4.2) and (4.3) there must exist variables $y, y', u, u' \in \mathcal{V}(\Gamma_1(\varphi))$ such that $\Gamma_1(\varphi) \vdash z \subseteq x[f]$ and $\Gamma_1(\varphi) \vdash x?[f] \subseteq z$. By definition of syntactic containment these assumptions imply $\Gamma_1(\varphi) \vdash x[f]z$ and hence, by case (1) above, $\varphi \vdash x[f]z$. □

□

Lemma 3.5 (Γ_2 Preserves Satisfiability)
If φ is S-closed and does not contain $\perp$, then $\Gamma_2(\varphi)$ is satisfiable.

Proof. Let φ_Γ be the constraint part of $\Gamma_2(\varphi)$, *i. e.*, with existential quantifiers and negated constraints dropped. It is not difficult to see that φ_Γ does not contain $\perp$ and that φ_Γ is S-closed up to trivial constraints ($x{\subseteq}x$ and $x \nparallel x$) and symmetric compatibility constraints. Note in particular, that the fresh features f_x occur only once in $\Gamma_2(\varphi)$ (and hence neither (Desc-Incl) nor (Desc-Nondis) applies), and that the fresh features f_{xy} occur exactly twice in $\Gamma_2(\varphi)$, namely in selections at x and y, for which neither $x \nparallel y$ nor, by (Incl-Nondis)-closedness of φ, $x{\subseteq}y$ or $y{\subseteq}x$ occur in φ.

Hence, by Proposition 2.10, $gsol(\varphi_\Gamma)$ as defined in Definition 3 is a solution of φ_Γ. It suffices to check that $gsol(\varphi_\Gamma)$ also satisfies the negated selection constraints added in clause (1) of $\Gamma_2(\varphi)$.

Assume $\neg\exists y'\,(y[f_x]y') \in \Gamma_2(\varphi)$, hence also $x[f_x]v_x \in \Gamma_s(\varphi)$ and $\varphi \nvdash y{\subseteq}x$. S-closedness of φ and $\varphi \nvdash y{\subseteq}x$ imply that $y \overset{\varphi}{\not\rightsquigarrow}_{\varepsilon} x$ and hence $y \overset{\varphi_\Gamma}{\not\rightsquigarrow}_{\varepsilon} x$ holds. Since f_x has a unique occurrence in $\Gamma_2(\varphi)$, this implies that $y \overset{\varphi_\Gamma}{\not\rightsquigarrow}_{\varepsilon} f_x$, and hence $f_x \notin D_{gsol(\varphi_\Gamma)(y)}$. □

Lemma 3.6 (Γ_2 Contradicts Non-selection Constraints)
Let φ be an S*-closed constraint which does not contain $\bot$, and let μ be a primitive constraint of the form $x \nparallel y$, $x{\subseteq}y$, or $a\langle x\rangle$. Then $\Gamma_2(\varphi) \models_{\mathcal{P}^+(\mathcal{FT})} \neg\mu$ if and only if $\varphi \nvdash \mu$.*

Proof. If $\Gamma_2(\varphi) \models_{\mathcal{P}^+(\mathcal{FT})} \neg\mu$ then $\varphi \nvdash \mu$ by Lemma 3.5 and correctness of syntactic containment. For the inverse direction we inspect the definition of $\Gamma_2(\varphi)$.

Clause (1) If $\varphi \nvdash x{\subseteq}y$, then $\Gamma_2(\varphi)$ disentails $x{\subseteq}y$ by forcing x to have a feature f_x which y must not have.

Clause (2) If $\varphi \nvdash x \nparallel y$, then $\Gamma_2(\varphi)$ disentails $x \nparallel y$ by forcing x and y to have a common feature f_{xy} such that the subtrees of x and y at f_{xy} are incompatible.

Clauses (3) and (4) If $\varphi \nvdash a\langle x\rangle$, then $\Gamma_2(\varphi)$ disentails $a(x)$ for every label a by forcing x to contain at least two trees with distinct label. □

Definition 5 (Saturation)
Let φ be an S*-closed constraint not containing $\bot$. The* saturation $\mathrm{Sat}(\varphi)$ of φ *is defined by*

$$Sat(\varphi) \quad =_{def} \quad \Gamma_2(\Gamma_1(\varphi))\,.$$

Lemma 3.7 (Saturation Characterises Syntactic Entailment)
Let φ be an S*-closed constraint not containing $\bot$, and let μ be such that $\mathcal{V}(\mu) \subseteq \mathcal{V}(\varphi)$. Then $\varphi \nvdash \mu$ implies $Sat(\varphi) \models_{\mathcal{P}^+(\mathcal{FT})} \neg\mu$.*

Proof. Let $\mathrm{Sat}(\varphi) = \Gamma_2(\Gamma_1(\varphi))$. If $\varphi \nvdash \mu$ then $\Gamma_1(\varphi) \nvdash \mu$ holds by case (1) of Lemma 3.4. If μ is not a selection constraint, then $\Gamma_2(\Gamma_1(\varphi)) \models_{\mathcal{P}^+(\mathcal{FT})} \neg\mu$ holds by Lemma 3.6. Otherwise, let $\mu = x[a]y$. Hence, one of $\Gamma_1(\varphi) \nvdash v_{xa}{\subseteq}y$ or $\Gamma_1(\varphi) \nvdash y{\subseteq}v_{xa}$ holds by case (2) of Lemma 3.4. By Lemma 3.6, either $\Gamma_2(\Gamma_1(\varphi)) \models_{\mathcal{P}^+(\mathcal{FT})} \neg v_{xa}{\subseteq}y$ or $\Gamma_2(\Gamma_1(\varphi)) \models_{\mathcal{P}^+(\mathcal{FT})} \neg y{\subseteq}v_{xa}$ holds, and hence again $\Gamma_2(\Gamma_1(\varphi)) \models_{\mathcal{P}^+(\mathcal{FT})} \neg\mu$. □

$$
\begin{array}{llll}
\varphi \vdash_\emptyset a\langle x\rangle & \text{iff} & x{=}\emptyset \in \varphi, & \text{or } \varphi \vdash a\langle x\rangle \\
\varphi \vdash_\emptyset x{\subseteq}y & \text{iff} & x{=}\emptyset \in \varphi, & \text{or } \varphi \vdash x{\subseteq}y \\
\varphi \vdash_\emptyset x[f]y & \text{iff} & x{=}\emptyset \in \varphi \text{ and } y{=}\emptyset \in \varphi, & \text{or } \varphi \vdash x[f]y
\end{array}
$$

Figure 3.2.: Syntactic Containment up to Emptiness for $\mathrm{FT}_{\subseteq}(ar)$

3.1.3. Dropping the Non-emptiness Restriction

We show that entailment for $\mathrm{FT}_{\subseteq}$ can also be decided in polynomial time, more precisely, in time $O(n^4)$. We obtain this result by extending the corresponding result for $\mathrm{FT}^{ne}_{\subseteq}$ and it seems that a direct proof would be substantially more involved.

The key to the polynomial complexity result is an extension of our notion of syntactic containment. Figure 3.2 extends the definition of syntactic containment in Figure 3.1 and defines a relation $\varphi \vdash_\emptyset \mu$ between $\mathrm{FT}_{\subseteq}$ constraints and primitive constraints $x[f]y$, $a\langle x\rangle$ or $x{\subseteq}y$. If $\varphi \vdash_\emptyset \mu$ holds we say that φ *contains* μ *up to emptiness*. Syntactic containment up to emptiness suffices to characterise entailment for $\mathrm{FT}_{\subseteq}$.

Proposition 3.8 (Entailment = Syntactic Containment up to Emptiness)
The notions of entailment and syntactic containment up to emptiness coincide for primitive constraints: If φ *is an* E*-closed* $\mathrm{FT}_{\subseteq}$ *constraint and* μ *is a primitive constraint, then* $\varphi \models_{\mathcal{P}(\mathcal{FT})} \mu$ *if and only if* $\varphi \vdash_\emptyset \mu$.

Proof. The direction from right to left (soundness) is clear. For the direction from left to right (completeness) assume that there exists $\mu \in \varphi'$ such that $\varphi \not\vdash_\emptyset \mu$. From Proposition 2.15 we know that $\varphi_{\neq\emptyset}$ is satisfiable over $\mathcal{P}^+(\mathcal{FT})$. By Proposition 3.2 we know that $\varphi \not\models_{\mathcal{P}^+(\mathcal{FT})} \mu$. Hence there exists a $\mathcal{P}^+(\mathcal{FT})$-solution α of $\varphi_{\neq\emptyset}$ such that $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \neg\mu$ and hence also $\alpha \models_{\mathcal{P}(\mathcal{FT})} \neg\mu$. By Lemma 2.16, the extension α' of α that maps all variables in $\mathit{Empty}(\varphi)$ to the empty set and all other variables to non-empty sets is a $\mathcal{P}(\mathcal{FT})$-solution of φ. We show by case distinction over the possible forms of μ that $\alpha' \models_{\mathcal{P}(\mathcal{FT})} \neg\mu$. This means that $\varphi \not\models_{\mathcal{P}(\mathcal{FT})} \mu$ and hence $\varphi \not\models_{\mathcal{P}(\mathcal{FT})} \varphi'$.

$\mu = a\langle x\rangle$**:** Since $\varphi \not\vdash_\emptyset a\langle x\rangle$, we know that $x \notin \mathit{Empty}(\varphi)$. Therefore $\alpha \models_{\mathcal{P}(\mathcal{FT})} \neg a\langle x\rangle$ implies $\alpha' \models_{\mathcal{P}(\mathcal{FT})} \neg a\langle x\rangle$.

$\mu = x[f]y$**:** Since $\varphi \not\vdash_\emptyset x[f]y$, we know that $x{=}\emptyset \notin \varphi$ or $y{=}\emptyset \notin \varphi$. We consider two cases (the remaining one is symmetric).

If $x{=}\emptyset \notin \varphi$ and $y{=}\emptyset \notin \varphi$: Then $x, y \notin \mathit{Empty}(\varphi)$, and therefore $\alpha \models_{\mathcal{P}(\mathcal{FT})} \neg x[f]y$ implies $\alpha' \models_{\mathcal{P}(\mathcal{FT})} \neg x[f]y$.

If $x{=}\emptyset \in \varphi$ and $y{=}\emptyset \notin \varphi$: Then $x \in \mathit{Empty}(\varphi)$ and $y \notin \mathit{Empty}(\varphi)$. Hence α' maps x to the empty set and y to a non-empty set. Therefore, $\alpha' \models_{\mathcal{P}(\mathcal{FT})} \neg x[f]y$.

$\mu = x{\subseteq}y$**:** Since $\varphi \not\vdash_{\emptyset} x{\subseteq}y$, we know that $x{=}\emptyset \notin \varphi$. By a case distinction on whether or not $y{=}\emptyset \in \varphi$ as in the previous case we obtain that $\alpha \models_{\mathcal{P}(\mathcal{FT})} \neg x{\subseteq}y$ implies $\alpha' \models_{\mathcal{P}(\mathcal{FT})} \neg x{\subseteq}y$. □

Theorem 15 (Entailment for $\mathbf{FT}_{\subseteq}$ is Polynomial)
Let φ and φ' be $FT_{\subseteq}$ constraints whose sizes are n and n'. Then entailment $\varphi \models_{\mathcal{P}(\mathcal{FT})} \varphi'$ is decidable in time $O(n^4 + n \cdot n')$ and space $O(n^3 + n')$.

Proof. By Theorem 6, we can compute the E-closure in φ in time $O(n^4)$ and space $O(n^3)$. By Proposition 3.8 it suffices to test syntactic containment up to emptiness for all primitive constraints in φ', of which there are at most n'. In analogy to Lemma 3.3, we can assume the E-closure of φ to be represented such that every such test takes at most linear time. Hence the overall procedure takes time $O(n^4 + n \cdot n')$ and space $O(n^3 + n')$. □

Finite Trees

Theorem 15 carries over to the case of finite trees: we must only adapt the first step of checking $\varphi \models_{\mathcal{P}(\mathcal{FT})} \varphi'$ so that it computes the closure of φ with respect to E and the occurs check axiom (Empty-Occurs) on Page 46. The second step remains unchanged.

3.2. Hardness Results on Entailment

The complexity of entailment between set constraints becomes coNP-hard when arity constraints are added. This is proven in Section 3.2.1 for $\mathrm{FT}_{\subseteq}(ar)$ and in Section 3.2.2 for $\mathrm{FT}^{ne}_{\subseteq}(ar)$. Using the same proof technique, the corresponding results can be be obtained for inclusion constraints over sets of constructor trees [140] and for entailment for $\mathrm{FT}^{ne}_{\subseteq}$ with existential quantifiers (see Section 3.2.3). Section 3.2.4 strengthens this result by proving PSPACE-hardness for the entailment problem with existential quantification; this result holds even without arity constraints.

3.2.1. Entailment with Arity Constraints is coNP-hard

We prove the following result.

Theorem 16 (Entailment for $\mathbf{FT}^{ne}_{\subseteq}(ar)$ is coNP-hard)
The entailment problem $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \varphi'$ for $FT^{ne}_{\subseteq}(ar)$ is coNP-hard.

Proof. Follows from Proposition 3.9 on Page 64. □

Corollary 17 (Satisfiability of Positive and Negative $\mathrm{FT}^{ne}_{\subseteq}(ar)$ Constraints)
The satisfiability problem of positive and negative $FT^{ne}_{\subseteq}(ar)$ constraints is coNP-hard.

For the proof, we reduce the complement of the propositional satisfiability problem SAT to an entailment problem between $\mathrm{FT}^{ne}_{\subseteq}(ar)$ constraints. Crucially, the reduction uses arity constraints. This implies Theorem 16 because SAT is NP-complete, actually the very first problem for which NP-completeness was proven [52]. The reduction is based on an idea of Henglein and Rehof [89]. They have considered entailment between ordering constraints over finite constructor trees with the so-called structural subtyping order.

3.2.1.1. A Complication of Entailment

Before we give the proof of Theorem 16, notice that it is in contrast to the paper [41] which claims polynomial complexity for entailment (over the domain of non-empty sets of *finite constructor* trees). The algorithm given there is incomplete. This incompleteness is not easily fixed. The next example illustrates a complication of entailment.

The following is a valid entailment over $\mathrm{FT}^{ne}_{\subseteq}(ar)$-constraints; notably one that depends on the implicit non-emptiness restriction for the denotation of x.

$$x{\subseteq}a(f{:}y) \wedge x{\subseteq}a(f{:}z) \wedge a\langle y\rangle \wedge y\{\} \quad \models_{\mathcal{P}^+(\mathcal{FT})} \quad y{\subseteq}z \tag{3.5}$$

A possible argument is as follows: since the denotation of x is non-empty, the intersection of the denotations of y and z must be non-empty. The constraint $a\langle y\rangle \wedge y\{\}$ implies that y denotes the singleton set $\{a\}$. By non-disjointness of y and z, the denotation of z must at least contain a. Thus $y{\subseteq}z$ is entailed. By a similar argument, the following entailment proposition can be shown valid for $\mathrm{FT}^{ne}_{\subseteq}(ar)$.

$$\begin{array}{l} x{\subseteq}a(f{:}y',\ g{:}y'') \wedge y'{\subseteq}b(f{:}z',\ g{:}z'') \wedge z'{\subseteq}a \wedge z'\{\} \wedge \\ x{\subseteq}a(f{:}u',\ g{:}u'') \wedge u'{\subseteq}b(f{:}v',\ g{:}v'') \end{array} \quad \models_{\mathcal{P}^+(\mathcal{FT})} \quad z'{\subseteq}v' \tag{3.6}$$

The variables z' and v' are related to each other through x which does not denote a singleton set itself. Rather, *for some path* (here *ff*) does selection from the denotation of x yield a singleton. Notice that two distinct features $f \neq g$ are necessary to describe this situation.

This example also illustrates the problem of the algorithm in [41], transposed to the feature tree notation. Roughly, the algorithm in [41] derives singleton information, for example by reasoning as follows:

$$a\langle x\rangle \wedge x\{\} \quad \rightarrow \quad \mathsf{singleton}(x)$$

$$a\langle x\rangle \wedge x\{f\} \wedge x[f]y \wedge \mathsf{singleton}(y) \quad \rightarrow \quad \mathsf{singleton}(x)$$

$$\ldots$$

$$\mathsf{singleton}(y) \wedge y \not\parallel z \quad \rightarrow \quad y\subseteq z$$

But as we have seen, the derivation of constraints $\mathsf{singleton}(x)$ does not suffice for a complete entailment algorithm. Rather, one needs a path-based argument like

$$\mathsf{singleton}(z') \wedge x \rightsquigarrow_{ff} z' \wedge x \rightsquigarrow_{ff} v' \quad \rightarrow \quad z'\subseteq v' \,.$$

This is what the algorithm in [41] fails to do. Hence, the entailment in Example (3.5) is correctly detected, while the entailment in Example (3.6) is not.

3.2.1.2. The Reduction

We assume an infinite set of boolean variables ranged over by u. A *clause* C is a finite disjunction of *literals* u or $\neg u$. We write *false* for the empty clause. A *solution* of a finite conjunction of clauses is a boolean variable assignment under which each of the clauses evaluates to *true*. The *clause satisfiability problem* SAT is whether a given conjunction has a solution. Without loss of generality we assume that no clause contains both a literal and its negation.

Proposition 3.9 (Reducing SAT to Entailment for $\mathbf{FT}^{ne}_{\subseteq}(ar)$)
For all $x \in \mathcal{V}$ there exists a function Φ_x from clauses C and integers k to existential $FT^{ne}_{\subseteq}(ar)$ formulas such that for all C:

1. *The size of $\Phi_x(C,k)$ is proportional to k.*
2. *For all SAT problems $\bigwedge_{i=1}^{n} C_i$ over k variables the following holds if $x \neq y$:*

$$\bigwedge_{i=1}^{n} \Phi_x(C_i,k) \wedge \Phi_y(\mathit{false}) \models_{\mathcal{P}^+(\mathcal{FT})} x\subseteq y \quad \textit{iff} \quad \bigwedge_{i=1}^{n} C_i \textit{ is non-satisfiable.}$$

Theorem 16 is an immediate corollary of this Proposition. To see this, notice that the size of the entailment problem $\Phi_x(C_i,k) \wedge \Phi_y(\mathit{false}) \models_{\mathcal{P}^+(\mathcal{FT})} x\subseteq y$ is $O(k \cdot n)$ and hence polynomial in the size of the given SAT problem.

Before we prove the Proposition, we illustrate the basic idea by an example. Consider the following clauses over three boolean variables u_1, u_2, and u_3, and observe that $C_1 \wedge C_2$ is satisfiable while $C_1 \wedge C_2 \wedge C_3$ is not.

$$C_1 \quad =_{def} \quad \neg u_1 \vee u_3,$$

$$C_2 \quad =_{def} \quad \neg u_1 \vee \neg u_3, \text{ and}$$

$$C_3 \quad =_{def} \quad u_1$$

$$
\begin{array}{l}
\exists y \\ \exists y_0 \\ \exists y_1 \\ \exists z \\ \exists z'
\end{array}
\left(
\begin{array}{ll}
x[\mathit{1}]y & \wedge\, a\langle x\rangle \,\wedge\, x\{0,1\} \\
 & \wedge\, a\langle y\rangle \wedge y\{\} \\
y[\mathit{0}]y_0 \wedge z\subseteq y_0 & \wedge\, a\langle y_0\rangle \,\wedge\, y_0\{0,1\} \\
y[\mathit{1}]y_1 \wedge z\subseteq y_1 & \wedge\, a\langle y_0\rangle \,\wedge\, y_1\{0,1\} \\
z[\mathit{0}]z' & \wedge\, a\langle z\rangle \,\wedge\, z\{0,1\} \\
 & \wedge\, a\langle z'\rangle \,\wedge\, z'\{\}
\end{array}
\right)
$$

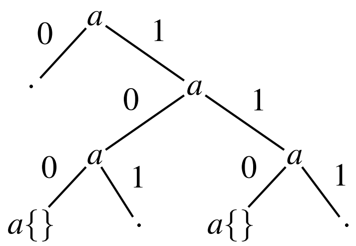


Figure 3.3.: An Example for the Reduction of SAT to Entailment for $\mathrm{FT}^{ne}_{\subseteq}(ar)$

Now fix distinct variables x and y. Proposition 3.9 claims the existence of formulas $\Phi_x(C_1,3)$ through $\Phi_x(C_3,3)$ and Φ'_y such that

$$\Phi_x(C_1,3)\wedge\Phi_x(C_2,3)\wedge\Phi_y(\mathit{false},3) \quad \not\models_{\mathcal{P}^+(\mathcal{F}\mathcal{T})} \quad x\subseteq y \tag{3.7}$$

$$\Phi_x(C_1,3)\wedge\Phi_x(C_2,3)\wedge\Phi_x(C_3,3)\wedge\Phi_y(\mathit{false},3) \quad \models_{\mathcal{P}^+(\mathcal{F}\mathcal{T})} \quad x\subseteq y \tag{3.8}$$

The formula $\Phi_x(C_1,3)$ (to be defined) and the form of its greatest solution are depicted in Figure 3.3: the formula on the left asserts all feature trees in the denotation of x to have at least the paths and labels of the tree on the right; at the mentioned paths they may have at most features 0 and 1, and no feature at all at the mentioned paths of length 3.

The maximal paths correspond to the boolean valuations of u_1 through u_3 under which C_1 evaluates to *false* [89], where the features 0 and 1 correspond to the truth values *false* and *true*. Similarly, as the empty clause evaluates to *false* under *all* valuations, the formula $\Phi_y(\mathit{false},3)$ constrains y to the set of trees that have exactly the paths in $\{0,1\}^3$ and are completely labelled with a. As there is only one such tree, call it τ^3, $\Phi_y(\mathit{false},3)$ entails that $y=\{\tau^3\}$. (This only holds because the empty set is excluded from $\mathcal{P}^+(\mathcal{F}\mathcal{T})$; over $\mathcal{P}(\mathcal{F}\mathcal{T})$ only the inclusion $y\subseteq\{\tau^3\}$ is entailed. See also Section 3.2.2.)

Likewise, the formula $\Phi_x(C_1,3)\wedge\Phi_x(C_2,3)\wedge\Phi_x(C_3,3)$ will constrain x to $\{\tau^3\}$, and hence (3.8) will be valid. In contrast, $\Phi_x(C_1,3)$ (and also $\Phi_x(C_1,3)\wedge\Phi_x(C_2,3)$) are less restrictive with respect to x than $\Phi_y(\mathit{false},3)$ is with respect to y, and hence $\Phi_x(C_1,3)\wedge\Phi_y(\mathit{false},3)\not\models_{\mathcal{P}^+(\mathcal{F}\mathcal{T})} x\subseteq y$ as well as (3.7) hold.

3.2.1.3. Proof of Proposition 3.9

This proof covers this whole section. Let us first formalise the intuition given in the example above. We fix a label a and three distinct features $0, 1$ and 2, and we confuse the truth values *true* and *false* with the features 1 and 0, respectively. (The feature 2 will be used only farther below.) We represent every boolean valuation β on $\{u_1, \ldots, u_k\}$ as the following path p^k_β.

$$p^k_\beta =_{def} \quad \beta(u_k) \ldots \beta(u_1)$$

We say that a feature tree τ *contains a valuation* β, written $\beta \in \tau$, if the following conditions hold.

1. $\mathsf{ar}(\tau.p^k_\beta) = \emptyset$
2. $\forall p, p$ a prefix of p^k_β : $\quad (p, a) \in S_\tau \quad$ and $\quad \mathsf{ar}(\tau.p) \subseteq \{0, 1\}$

By generalisation, we say that a tree τ *contains a set B of valuations*, written $B \subseteq \tau$, if $\forall \beta \in B : \beta \in \tau$. The injective function T establishes a correspondence between the sets of boolean valuations B and the sets $T(B)$ of feature trees containing B.

$$T(B) =_{def} \quad \{\tau \mid \text{ if } B \subseteq \tau\}$$

For instance, $T(\{0, 1\}^k)$ is the singleton set containing just the complete binary feature tree of depth k over the features 0 and 1 which is completely labelled with a.

Now assume that the function Φ_x has the following properties.

$$\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \bigwedge_{i=1}^{n} \Phi_x(C_i, k) \quad \text{iff} \quad \alpha(x) \subseteq T(\mathrm{Sol}(\neg \bigwedge_{i=1}^{n} C_i)) \tag{3.9}$$

$$\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Phi_y(\mathit{false}, k) \quad \text{iff} \quad \alpha(y) = T(\{0, 1\}^k) \tag{3.10}$$

Lemma 3.10

If, for all $x \in \mathcal{V}$, there exists a function Φ_x with properties (3.9) and (3.10) then clause (2) *in Proposition 3.9 holds.*

$$
\begin{array}{rcl}
\Phi_x(C,k) & = & \hat{\Phi}_x(p^k_C) \\
\hat{\Phi}_x(\varepsilon) & = & x\{\} \wedge a\langle x\rangle \\
\hat{\Phi}_x(1p) & = & x\{0,1\} \wedge a\langle x\rangle \wedge \exists x' \, (x[\mathit{1}]x' \wedge \hat{\Phi}_{x'}(p)) \\
\hat{\Phi}_x(0p) & = & x\{0,1\} \wedge a\langle x\rangle \wedge \exists x' \, (x[\mathit{0}]x' \wedge \hat{\Phi}_{x'}(p)) \\
\hat{\Phi}_x(2p) & = & x\{0,1\} \wedge a\langle x\rangle \wedge \exists x_1 x_2 x_3 \, (x[\mathit{1}]x_1 \wedge x[\mathit{0}]x_2 \wedge x_1 \subseteq x_3 \wedge x_2 \subseteq x_3 \wedge \hat{\Phi}_{x_3}(p))
\end{array}
$$

Figure 3.4.: Reducing SAT to Entailment for $\mathrm{FT}^{ne}_{\subseteq}(ar)$

Proof.

$$
\begin{array}{ll}
& \bigwedge_{i=1}^{n} \Phi_x(C_i,k) \wedge \Phi_y(\mathit{false},k) \models_{\mathcal{P}^+(\mathcal{FT})} x \subseteq y \\
\text{iff} & \forall \alpha : \text{ if } \alpha \models_{\mathcal{P}^+(\mathcal{FT})} \bigwedge_{i=1}^{n} \Phi_x(C_i,k) \wedge \Phi_y(\mathit{false},k) \text{ then } \alpha(x) \subseteq \alpha(y) \\
\text{iff} & \forall \alpha : \text{ if } \alpha \models_{\mathcal{P}^+(\mathcal{FT})} \bigwedge_{i=1}^{n} \Phi_x(C_i,k) \wedge \Phi_y(\mathit{false},k) \text{ then } \alpha(x) \subseteq T(\{0,1\}^k) \text{ by (3.10)} \\
\text{iff} & T(\mathrm{Sol}(\neg \bigwedge_{i=1}^{n} C_i)) \subseteq T(\{0,1\}^k) \qquad \text{by (3.9)} \\
\text{iff} & \bigwedge_{i=1}^{n} C_i \text{ is non-satisfiable}
\end{array}
$$

For the downward implication of equivalence marked (3.9) note that, by Property (3.9), every valuation α with $\alpha(x) = T(\mathrm{Sol}(\neg \bigwedge_{i=1}^{n} C_i))$ is a solution of $\bigwedge_{i=1}^{n} \Phi_x(C_i,k)$. The upward implication follows directly from (3.9). For the upward implication of the last equivalence note that $\mathrm{Sol}(\neg \bigwedge_{i=1}^{n} C_i) = T(\{0,1\}^k)$ if $\bigwedge_{i=1}^{n} C_i$ is non-satisfiable. For the downward implication first note that

$$\forall B \subseteq \{0,1\}^k : T(B) \neq \emptyset$$

which implies that $\emptyset \neq T(\mathrm{Sol}(\neg \bigwedge_{i=1}^{n} C_i))$. Hence $T(\mathrm{Sol}(\neg \bigwedge_{i=1}^{n} C_i)) = T(\{0,1\}^k)$ since $T(\{0,1\}^k)$ is a singleton set, and $\mathrm{Sol}(\neg \bigwedge_{i=1}^{n} C_i) = \{0,1\}^k$ since T is injective. Thus $\bigwedge_{i=1}^{n} C_i$ is non-satisfiable. □

It remains to show that there are indeed formulas $\Phi_x(C,k)$ with Property (3.9) whose size is proportional to k. For every $i, 1 \leq i \leq k$ and every clause C over $\{u_1, \ldots, u_k\}$ let

$$\begin{array}{ll} \delta_i(C) = 1 & \text{if } \neg u_i \text{ in } C \\ \delta_i(C) = 0 & \text{if } u_i \text{ in } C \\ \delta_i(C) = 2 & \text{otherwise.} \end{array}$$

This is well-defined because no clause C contains both u_i and $\neg u_i$ for a boolean variable u_i. Every clause C corresponds to the path p^k_C given by

$$p^k_C =_{def} \quad \delta_k(C) \ldots \delta_1(C) .$$

The definition of $\Phi_x(C,k)$ by recursion over p^k_C is given in Figure 3.4. Since every step of this definition introduces at most three new variables, these formulas have size proportional to k.

It is easy to verify directly that every solution α of $\Phi_y(\mathit{false},k)$ satisfies $\alpha(y) = T(\{0,1\}^k)$, so that Property (3.10) holds. Note also, that Property (3.10) is a consequence of Property (3.9) since $T(\{0,1\}^k)$ is a singleton such that $\alpha(y) \subseteq T(\{0,1\}^k)$ implies $\alpha(y) = T(\{0,1\}^k)$ over non-empty sets.

Observe that $\bigwedge_{i=1}^n \Phi_x(C_i,k) \wedge \Phi_y(\mathit{false},k)$ is always satisfiable, for example by every valuation mapping both x and y to $T(\{0,1\}^k)$. Hence we know that a greatest solution exists by Proposition 2.10.

Lemma 3.11

For all clauses C over k variables and all x: $T(Sol(\neg C)) = gsol(\Phi_x(C,k))(x)$.

Proof. We show by induction over k that, for all x, τ, and all clauses C over k variables

$$\mathrm{Sol}(\neg C) \subseteq \tau \quad \text{if and only if} \quad \tau \in gsol(\Phi_x(C,k))(x) .$$

Case $k = 0$: We have

$$\Phi_x(\mathit{false},k) \quad = \quad \hat{\Phi}_x(\varepsilon) \quad = \quad x\{\} \wedge a\langle x\rangle$$

By definition of $\mathrm{Sol}(\neg C) \subseteq \tau$, the fact that the only valuation over 0 variables is the empty one, and the definition of $gsol(\Phi_x(\mathit{false},k))$, we reason as follows.

$$\begin{array}{rll} \mathrm{Sol}(\neg \mathit{false}) \subseteq \tau & \text{iff} & \mathrm{Sol}(\mathit{true}) \subseteq \tau \\ & \text{iff} & \varepsilon \in D_\tau,\ (\varepsilon, a) \in S_\tau, \text{ and } \mathsf{ar}(\tau) = \emptyset \\ & \text{iff} & gsol(\Phi_x(\mathit{false},k))(x) \end{array}$$

Case $k > 0$: The clause C can have one of three forms, $u_k \vee C'$, $\neg u_k \vee C'$, or C' for some clause C' over variables $u_{k-1}, \ldots, u_1$. We only consider the case $C = u_k \vee C'$. The other cases are similar. For ease of reading, we introduce the following abbreviations for all $f \in F$:

$$\begin{array}{rclcrcl} D_\tau.f & = & \{p \mid fp \in D_\tau\} & \qquad & fD_\tau & = & \{fp \mid p \in D_\tau\} \\ S_\tau.f & = & \{(p,a) \mid (fp,a) \in D_\tau\} & & fS_\tau & = & \{(fp,a) \mid (p,a) \in S_\tau\} \end{array}$$

If $C = u_k \vee C'$, we have

$$\Phi_x(C,k) \quad = \quad x\{0,1\} \wedge a\langle x\rangle \wedge \exists x' (x[0]x' \wedge \Phi_{x'}(C',k-1)) \tag{3.11}$$

Fix a fresh x'. By definition of $\mathrm{Sol}(\neg C) \subseteq \tau$ and by induction assumption we have that

$$\begin{array}{ll} & \mathrm{Sol}(\neg C) \subseteq \tau \\ \text{iff} & \mathrm{Sol}(\neg(u_k \vee C')) \subseteq \tau \\ \text{iff} & \exists \tau' \colon \mathrm{Sol}(\neg C') \subseteq \tau' \qquad \text{and} \left\{ \begin{array}{ll} 1. & D_{\tau'} = D_\tau.0,\ S_{\tau'} = S_\tau.0 \\ 2. & \mathsf{ar}(\tau) = \{0,1\},\ (\varepsilon,a) \in S_\tau \end{array} \right. \\ \text{iff} & \exists \tau' \colon \tau' \in gsol(\Phi_{x'}(C',k-1))(x') \quad \text{and} \left\{ \begin{array}{ll} 1. & D_{\tau'} = D_\tau.0,\ S_{\tau'} = S_\tau.0 \\ 2. & \mathsf{ar}(\tau) = \{0,1\},\ (\varepsilon,a) \in S_\tau \end{array} \right. \end{array}$$

It remains to show that this is equivalent to $\tau \in gsol(\Phi_x(C,k))(x)$.[15]

($\Rightarrow$) By definition of the greatest solution, $\tau' \in gsol(\Phi_{x'}(C',k-1))(x')$ holds if and only if:

3. $D_{\Phi_{x'}(C',k-1)(x')} \subseteq D_{\tau'}$, $S_{\Phi_{x'}(C')(x',k-1)} \subseteq S_{\tau'}$, and
4. for all p and $\overline{f}$: if $\Phi_{x'}(C',k-1) \vdash x' \rightsquigarrow_p \{\overline{f}\}$ then $\mathsf{ar}(\tau'.p) = \{\overline{f}\}$.

Given equation (3.11), we conclude from (1), (3) and the definition of path reachability that

$$\begin{array}{rcccccccc} D_{\Phi_x(C,k)}(x) & = & 0D_{\Phi_{x'}(C',k-1)}(x') & \subseteq & 0D_{\tau'} & \subseteq & 0(D_\tau.0) & \subseteq & D_\tau \\ S_{\Phi_x(C,k)}(x) & = & 0S_{\Phi_{x'}(C',k-1)}(x') & \subseteq & 0S_{\tau'} & \subseteq & 0(S_\tau.0) & \subseteq & S_\tau \end{array}$$

Further, if $\Phi_x(C,k) \vdash x \rightsquigarrow_p \{\overline{f}\}$, then there two possibilities:

[15] Here, we allow path reachability with respect to existential formulas Φ instead of just constraints, if the mentioned variables are free in Φ. For instance, we write $\exists y \exists z (x \subseteq y \wedge y[f]z \wedge z \subseteq x') \vdash x \rightsquigarrow_f x'$.

1. If $p = \varepsilon$ and $\{\overline{f}\} = \{0,1\}$, then $\mathsf{ar}(\tau.p) = \{0,1\} = \{\overline{f}\}$ follows from (2).
2. If $\Phi_{x'}(C', k-1) \vdash x' \rightsquigarrow_{p'} \{\overline{f}\}$ for some p' with $p = 0p'$, then $\mathsf{ar}(\tau'.p') = \{\overline{f}\}$ follows from (4). Furthermore, $D_\tau.0 = D_{\tau'}$ in (1) implies that $\mathsf{ar}(\tau.0p') = \mathsf{ar}(\tau'.p')$, hence again $\mathsf{ar}(\tau.p) = \{\overline{f}\}$.

In combination, $\Phi_x(C,k) \vdash x \rightsquigarrow_p \{\overline{f}\}$ implies $\mathsf{ar}(\tau.p) = \{\overline{f}\}$ for all p and $\overline{f}$, and thus $\tau \in gsol(\Phi_x(C,k))(x)$.

($\Leftarrow$) For the converse, we assume $\tau \in gsol(\Phi_x(C,k))(x)$ and set $\tau' = \tau.0$. Then we check that τ satisfies (2), that τ' satisfies (1), and that $\tau' \in gsol(\Phi_{x'}(C,k-1))(x')$. □

Lemma 3.12
Let $\bigwedge_{i=1}^n C_i$ be a SAT problem over k variables. Then:

$$\forall\alpha : \alpha \models_{\mathcal{P}^+(\mathcal{FT})} \bigwedge_{i=1}^n \Phi_x(C_i,k) \quad \textit{if and only if} \quad \alpha(x) \subseteq T(Sol(\neg \bigwedge_{i=1}^n C_i))$$

Proof. We show that the greatest solution of $\bigwedge_{i=1}^n \Phi_x(C_i)$ equals $T(\mathrm{Sol}(\neg \bigwedge_{i=1}^n C_i))$.

$$\begin{array}{rcll}
gsol(\bigwedge_{i=1}^n \Phi_x(C_i,k)) & = & \bigcap_{i=1}^n gsol(\Phi_x(C_i,k)) & \text{since } \bigwedge_{i=1}^n \Phi_x(C_i,k) \text{ sat.} \\
& & & \text{by Proposition 2.1} \\
& = & \bigcap_{i=1}^n \{\tau \mid \mathrm{Sol}(\neg C_i) \subseteq \tau\} & \text{by Lemma 3.11} \\
& = & \{\tau \mid \bigcup_{i=1}^n \mathrm{Sol}(\neg C_i) \subseteq \tau\} & \\
& = & \{\tau \mid \mathrm{Sol}(\neg \bigwedge_{i=1}^n C_i) \subseteq \tau\} & \\
& = & T(\mathrm{Sol}(\neg \bigwedge_{i=1}^n C_i)) &
\end{array}$$
□

3.2.2. Dropping the Non-emptiness Restriction

We show that the same reduction idea of SAT to entailment applies, with slight modifications, also to the case of possibly empty sets of trees.

Theorem 18 (Entailment for $\mathrm{FT}_\subseteq(ar)$ is coNP-hard)
The entailment problem $\varphi \models_{\mathcal{P}(\mathcal{FT})} \varphi'$ for $FT_\subseteq(ar)$ is coNP-hard.

$$
\begin{array}{rcl}
\Phi^z_x(C,k) & = & \hat{\Phi}^z_x(p^k_C) \\
\hat{\Phi}^z_x(\varepsilon) & = & \boxed{x{\subseteq}z} \\
\hat{\Phi}^z_x(1p) & = & x\{0,1\} \wedge a\langle x\rangle \wedge \exists x'\,(x[\mathit{1}]x' \wedge \hat{\Phi}^z_{x'}(p)) \\
\hat{\Phi}^z_x(0p) & = & x\{0,1\} \wedge a\langle x\rangle \wedge \exists x'\,(x[\mathit{0}]x' \wedge \hat{\Phi}^z_{x'}(p)) \\
\hat{\Phi}^z_x(2p) & = & x\{0,1\} \wedge a\langle x\rangle \wedge \exists x_1x_2x_3\,(x[\mathit{1}]x_1 \wedge x[\mathit{0}]x_2 \wedge x_1{\subseteq}x_3 \wedge x_2{\subseteq}x_3 \wedge \hat{\Phi}^z_{x_3}(p)) \\
\Phi'^z_y(\mathit{false}) & = & \hat{\Phi}'^z_y(p^k_{\mathit{false}}) \\
\hat{\Phi}'^z_y(2p) & = & N(y) \wedge \exists y'\,(y[\mathit{1}]y' \wedge y[\mathit{0}]y' \wedge \hat{\Phi}'^z_{y'}(p)) \\
\hat{\Phi}'^z_y(\varepsilon) & = & \boxed{y{=}z}
\end{array}
$$

Figure 3.5.: Reducing SAT to Entailment for $\mathrm{FT}_{\subseteq}(ar)$

Corollary 19 (Satisfiability of Positive and Negative $\mathrm{FT}_{\subseteq}(ar)$ Constraints)
The satisfiability problem of positive and negative constraints $FT_{\subseteq}(ar)$ is coNP-hard.

The proof is by adaptation of the proof of Theorem 16 in the previous section. There, we have exploited that we can express singleton sets with $\mathrm{FT}^{ne}_{\subseteq}(ar)$ constraints. This is no longer the case for $\mathrm{FT}_{\subseteq}(ar)$ constraints. For illustration, observe that the following implication holds over non-empty sets of feature trees:

$$x\{\} \wedge a\langle x\rangle \wedge y\{\} \wedge a\langle y\rangle \quad \models_{\mathcal{P}^+(\mathcal{FT})} \quad x{\subseteq}y \tag{3.12}$$

Over possibly empty sets it does not. Only a weaker implication holds:

$$x\{\} \wedge a\langle x\rangle \wedge y\{\} \wedge a\langle y\rangle \quad \models_{\mathcal{P}^+(\mathcal{FT})} \quad x{\subseteq}y \vee y{\subseteq}x \tag{3.13}$$

In analogy, Property (3.10) on page 66 does not hold over $\mathcal{P}(\mathcal{FT})$ because the empty set is always a solution for $\Phi_y(\mathit{false},k)$. Only the following weaker equivalence holds.

$$\alpha \models_{\mathcal{P}(\mathcal{FT})} \Phi_y(\mathit{false},k) \quad \text{iff} \quad \alpha(y) \subseteq T(\{0,1\}^k) \tag{3.14}$$

If the constraint system can express non-emptiness, we can correct this easily by requiring x to denote a non-empty set in the ε-clause of Figure 3.4:

$$\hat{\Phi}_x(\varepsilon) \quad = \quad x \neq \emptyset \wedge x\{\} \wedge a\langle x\rangle \tag{3.15}$$

Unfortunately, in $\mathrm{FT}_{\subseteq}(ar)$ we cannot express non-emptiness.[16] We adapt Proposition 3.9 as follows in order to prove Theorem 18.

[16] In contrast, set constraints over constructor trees can if the signature contains constants. Therefore, the adaptation (3.15) indeed works for standard set constraints as we show in [140]. There, of course $\hat{\Phi}_x(\varepsilon)$ is defined as $x{=}a$.

Proposition 3.13 (Reducing SAT to Entailment for $\mathrm{FT}_{\subseteq}(ar)$)

For all $x,y,z \in \mathcal{V}$ there exists a function Φ_x^z from clauses C and integers k to existential $FT_{\subseteq}(ar)$ formulas, and an existential $FT_{\subseteq}(ar)$ formula $\Phi'^z_y(false)$ such that for all C:

1. *The sizes of $\Phi_x^z(C,k)$ and $\Phi'^z_y(false)$ are proportional to k.*
2. *For all SAT problems $\bigwedge_{i=1}^n C_i$ over k variables the following holds if $x \neq y$:*

$$\bigwedge_{i=1}^{n} \Phi_x^z(C_i,k) \wedge \Phi'^z_y(\mathit{false}) \models_{\mathcal{P}^+(\mathcal{FT})} x \subseteq y \quad \mathit{iff} \quad \bigwedge_{i=1}^{n} C_i \text{ is non-satisfiable.}$$

The reduction is given in Figure 3.5. It adapts the reduction of Figure 3.4 in the ε-clause, and gives an special definition for the formula associated with the clause *false*. Instead of forcing all maximal paths in $\Phi_x^z(C,k)$ to the singleton set $\{a\}$, it asserts all maximal paths in $\Phi_x^z(C,k)$ to be included in the fixed z, and all maximal paths in $\Phi'^z_y(false)$ to be equal to z. The proof of Proposition 3.13 is completely analogous to the proof of Proposition 3.9, except that all notions related to valuations (such as: τ contains β, $B \subseteq \tau$, $T(B)$, etc.) must be made relative to some set σ that denotes the valuation of z.

3.2.3. Entailment with Existential Quantifiers is coNP-hard

We apply the idea of the two previous sections to the entailment problems $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \exists\overline{x}\varphi'$ and $\varphi \models_{\mathcal{P}(\mathcal{FT})} \exists\overline{x}\varphi'$ *with existential quantification but without arity constraints*, and show them to be coNP-hard, too. The idea still rests on Henglein's and Rehof's idea from [89], but the details are original. The reduction works for both sets of infinite trees and sets of finite trees. In the following section, we improve this result by showing entailment with existential quantification to be even PSPACE-hard.

With existential quantification, the reduction of SAT to an entailment problem becomes simpler. In the previous sections we have encoded an inconsistent SAT problem by a set such that all trees in the set have *exactly* all paths in $\{0,1\}^k$ and are either completely labelled with a or such that selection at all the paths in $\{0,1\}^k$ yields the same set. With existential quantification it suffices to encode an inconsistent SAT problem by a set of trees which contain *at least* a given set of paths.

Proposition 3.14 (Reducing SAT to Entailment for $\mathrm{FT}_{\subseteq}^{ne}$ with Existentials)

For all $x \in \mathcal{V}$ there exists a function Ψ_x from clauses C and integers k to existential $FT_{\subseteq}^{ne}$ formulas such that for all C:

1. *The size of $\Psi_x(C,k)$ is proportional to k.*

$$
\begin{array}{lcl}
\Psi_x(C,k) & = & \bigwedge_{i=1}^{n} \hat{\Psi}_x(p_{C_i}^k) \\
\hat{\Psi}_x(\varepsilon) & = & \mathit{true} \\
\hat{\Psi}_x(tp,x) & = & \exists x' \, (x[\mathit{1}]x' \wedge \hat{\Psi}_{x'}(p)) \\
\hat{\Psi}_x(fp,x) & = & \exists x' \, (x[\mathit{0}]x' \wedge \hat{\Psi}_{x'}(p)) \\
\hat{\Psi}_x(?p,x) & = & \exists x_1 \exists x_2 \exists x_3 \, (x[\mathit{1}]x_1 \wedge x[\mathit{0}]x_2 \wedge x_1 \subseteq x_3 \wedge x_2 \subseteq x_3 \wedge \hat{\Psi}_{x_3}(p,)
\end{array}
$$

Figure 3.6.: Reducing SAT to Entailment for $\mathrm{FT}_{\subseteq}^{ne}$ with Existential Quantifiers

$$
\begin{array}{l}
\exists y \\ \exists y_0 \\ \exists y_1 \\ \exists z \\ \exists z'
\end{array}
\left(
\begin{array}{l}
x[\mathit{1}]y \\
y[\mathit{0}]y_0 \ \wedge \ z \subseteq y_0 \\
y[\mathit{1}]y_1 \ \wedge \ z \subseteq y_1 \\
z[\mathit{0}]z'
\end{array}
\right)
$$

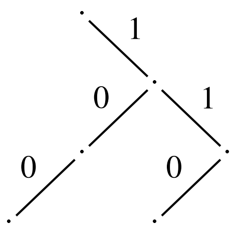


Figure 3.7.: An Example for the Reduction of SAT to Entailment for $\mathrm{FT}_{\subseteq}^{ne}$ with Existential Quantifiers

2. *For all SAT problems $\bigwedge_{i=1}^{n} C_i$ over k variables the following holds:*

$$
\bigwedge_{i=1}^{n} \Psi_x(C_i,k) \models_{\mathcal{P}^+(\mathcal{FT})} \Psi_x(\mathit{false},k) \quad \textit{iff} \quad \bigwedge_{i=1}^{n} C_i \textit{ is non-satisfiable}
$$

Note the distinction of this Proposition to Proposition 3.9: while the latter requires the conjunction of $\bigwedge_{i=1}^{n} \Psi_x(C_i,k)$ and $\Psi_y(\mathit{false},k)$ to entail the (quantifier-free) constraint $x \subseteq y$, Proposition 3.14 requires $\bigwedge_{i=1}^{n} \Psi_x(C_i,k)$ to entail $\Psi_x(\mathit{false},k)$ which does contain existential quantifiers.

The function Ψ_x is defined in Figure 3.6. Again, the size of the formulas $\Psi_x(\mathit{false},k)$ and $\Psi_x(C,k)$ is $O(k \cdot n)$, *i. e.*, polynomial in the size of the given instance of SAT. Their construction is similar to the one of Figure 3.4 but, as promised above, strictly simpler since it does not mention arity constraints or label constraints at all. The clause $C_1 = \neg u_1 \vee u_3$ over variables u_1, u_2, and u_3 which we considered above will now be mapped to the formula $\Psi_x(C_1,k)$ in Figure 3.7. In every solution of $\Psi_x(C_1,k)$, all feature trees in the denotation of x must have at least the paths in the tree depicted on the right; they may have further paths and arbitrary labellings, though. In order to justify this reduction, we must adapt Property (3.9) which does no longer apply.

We say that a feature tree τ *weakly contains a valuation* β, written $\beta \in_w \tau$, if

$$\forall p, p \text{ a prefix of } p^k_\beta : p \in D_\tau .$$

By generalisation, we define $B \subseteq_w \tau$ if $\forall \beta \in B : \beta \in_w \tau$, and similarly we define

$$T_w(B) =_{def} \quad \{\tau \mid B \subseteq_w \tau\}$$

Note that $T_w(\{0,1\}^k)$ is not a singleton anymore. Then the following properties hold.

$$\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \bigwedge_{i=1}^{n} \Psi_x(C_i, k) \quad \text{iff} \quad \alpha(x) \subseteq T_w(\mathrm{Sol}(\neg \bigwedge_{i=1}^{n} C_i)) \tag{3.16}$$

Lemma 3.15

If, for all $x \in \mathcal{V}$, there exists a function Ψ_x with properties (3.16) then clause (2) *in Proposition 3.14 holds.*

Proof.

$$\begin{array}{ll}
\multicolumn{2}{l}{\bigwedge_{i=1}^{n} \Psi_x(C_i,k) \models_{\mathcal{P}^+(\mathcal{FT})} \Psi_x(\mathit{false},k)} \\
\text{iff} & \forall \alpha : \alpha \models_{\mathcal{P}^+(\mathcal{FT})} \bigwedge_{i=1}^{n} \Psi_x(C_i,k) \text{ implies } \alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Psi_x(\mathit{false},k) \\
\text{iff} & \forall \alpha : \alpha \models_{\mathcal{P}^+(\mathcal{FT})} \bigwedge_{i=1}^{n} \Psi_x(C_i,k) \text{ implies } \alpha(x) \subseteq T_w(\{0,1\}^k) \qquad \text{by (3.16)} \\
\text{iff} & T_w(\mathrm{Sol}(\neg \bigwedge_{i=1}^{n} C_i)) \subseteq T_w(\{0,1\}^k) \\
\text{iff} & \bigwedge_{i=1}^{n} C_i \text{ is non-satisfiable} \qquad \text{by (3.16)}
\end{array}$$

□

The remainder of the proof is closely following the lines of the one in Section 3.2.1. We do not elaborate on it further since the following section contains a stronger result.

3.2.4. Entailment with Existential Quantifiers is PSPACE-hard

We prove the following results.

Theorem 20 (Entailment with Existential Quantifiers is PSPACE-hard)

The entailment problem $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \exists \overline{x} \varphi_1$ for both $FT_\subseteq$ and $FT^{ne}_\subseteq$ is PSPACE-hard.

Proof. Follows from Proposition 3.16 on Page 75. □

Corollary 21 (Negation and Existential Quantification for $\mathrm{FT}_{\subseteq}$ and $\mathrm{FT}^{ne}_{\subseteq}$)
Satisfiability of positive and negative existential $FT^{ne}_{\subseteq}$ formulas, and emptiness of positive and negative existential $FT_{\subseteq}$ formulas is PSPACE-hard ($\varphi \wedge \neg\exists\overline{x}_1\,\varphi_1 \wedge \ldots \neg\exists\overline{x}_n\,\varphi_n$).

We reduce in linear time the inclusion problem between regular languages REG over finite words to the entailment problem $\varphi \models_{\mathcal{P}^+(\mathcal{FT})} \exists\overline{x}\varphi'$ over $\mathrm{FT}^{ne}_{\subseteq}$. Since the problem REG is well-known to be PSPACE-complete [70, 103], this proves PSPACE-hardness of entailment with existential quantifiers.[17]

Interestingly, the reduction works both for the case of sets of infinite trees and of finite trees: Notably, it is possible to encode the Kleene star without referring to infinite trees. This is in contrast to our earlier result for $\mathrm{FT}_{\leq}$ [141] that suggested the need for infinite trees (or sets of infinite trees) for the encoding of the Kleene star. We could drop this restriction in [145].

We consider regular expressions over a finite subset $\mathcal{F}_0 \subseteq \mathcal{F}$ defined as follows:

$$R \quad ::= \quad \varepsilon \quad | \quad f \quad | \quad R^* \quad | \quad R_1 \cup R_2 \quad | \quad R_1 R_2 \qquad \text{where } f \in \mathcal{F}_0$$

Note that $\mathcal{F}_0 \subseteq \mathcal{F}$ allows arbitrary large alphabets since $\mathcal{F}$ is assumed to be infinite. Every regular expression R defines a non-empty set $\mathcal{L}(R)$ of finite words over $\mathcal{F}_0$.

Proposition 3.16 (Reducing REG to Entailment with Existential Quantifiers)
Let x and y be arbitrary variables. For every pair of regular expressions R_1 and R_2 there are existential $FT^{ne}_{\subseteq}$ formulas $\Theta(x, R_1, y)$ and $\Theta(x, R_2, y)$ whose sizes are linear in the sizes of R_1 and R_2, such that

$$\Theta(x, R_1, y) \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x, R_2, y) \quad \textit{if and only if} \quad \mathcal{L}(R_2) \subseteq \mathcal{L}(R_1).$$

3.2.4.1. First Solution: Infinite Feature Trees

An immediate idea of the proof is to encode every regular set of words (over features) as a set σ of feature trees all of which share the regular structure: namely such that all trees in σ contain all paths in $\mathcal{L}(R)$ and are labelled with a at all paths in $\mathcal{L}(R)$. For instance, one may encode the finite set $\{1, 111\}$ as the set σ of all feature trees τ with

$$\{(1, a), (111, a)\} \quad \subseteq \quad S_\tau\,,$$

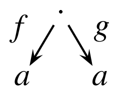
[17] Essentially the same proof also applies to entailment for $\mathrm{FT}_{\subseteq}$. The details will be discussed at the end of this section.

and the infinite set $\{\varepsilon, 1, 11, \ldots\}$ as the set σ of all feature trees τ with

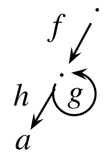
$$\{(\varepsilon, a), (1, a), (11, a), \ldots\} \quad \subseteq \quad S_\tau .$$

This information can conveniently be represented by a single feature tree [141]. The set described thus is given by all feature trees that contain at least the information in this tree. Here are some typical regular expressions and the associated trees:

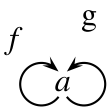
$f \cup g$



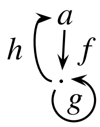
fg^*h



$(f \cup g)^*$



$(fg^*h)^*$



A consequence of this encoding is, however, that infinite regular languages are necessarily encoded by sets of infinite trees. Hence, for sets of finite trees this encoding only works for *star-free* regular expressions that induce finite languages; since the inclusion problem for languages defined by star-free regular expressions is coNP-complete, we obtain only coNP-hardness for the entailment problem over sets of finite trees. But we can do better.

3.2.4.2. Better Solution: Finite and Infinite Feature Trees

Instead of encoding a regular language $\mathcal{L}(R)$ by a set of feature trees all of which have all paths in $\mathcal{L}(R)$ and are labelled with a there, we encode it by a set σ of feature trees that contains one tree that has all the paths $p \in \mathcal{L}(R)$ and is labelled with a there. Intuitively, this is dual to the encoding above. The regular language is not encoded by an upper bound on the sets (which affects the shape of all contained trees) but a lower bound (which asserts the existence of one).

The proof covers the remainder of this section. In Figure 3.8 we define an existential $\mathrm{FT}^{ne}_{\subseteq}$ formula $\Theta(x, R, y)$ for every regular expression R and variables x, y. The formula $\Theta(x, R, y)$ clearly has size linear in the size of R. Define the *projection* $p^{-1}(\sigma)$ of a set σ to some path p by $p^{-1}(\sigma) = \{\tau' \mid \text{exists } \tau \in \sigma : \tau.p = \tau'\}$.

Lemma 3.17

Let α *be a variable assignment and* R *a regular expression. Then* $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x, R, y)$ *if and only if* $\forall p \in \mathcal{L}(R) : p^{-1}(\alpha(x)) \supseteq \alpha(y)$.

Proof. By structural induction over R. Let α be a variable assignment.

$$
\begin{array}{lcl}
\Theta(x,\varepsilon,y) & = & x \supseteq y \\
\Theta(x,f,y) & = & \exists z (x \supseteq z \wedge z[f]y) \\
\Theta(x,R_1 \cup R_2,y) & = & \Theta(x,R_1,y) \wedge \Theta(x,R_2,y) \\
\Theta(x,R^*,y) & = & \exists z (z \supseteq y \wedge \Theta(z,R,z) \wedge x \supseteq z) \\
\Theta(x,R_1R_2,y) & = & \exists z (\Theta(x,R_1,z) \wedge \Theta(z,R_2,y))
\end{array}
$$

Figure 3.8.: Reducing Inclusion of Regular Languages of Finite Words to Entailment for $\mathrm{FT}^{ne}_{\subseteq}$ with Existential Quantifiers

ε: $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x,\varepsilon,y)$ iff $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} x \supseteq y$

iff $\varepsilon^{-1}(\alpha(x)) = \alpha(x) \supseteq \alpha(y)$

f: $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x,f,y)$ iff $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \exists z (x \supseteq z \wedge z[f]y)$

iff $\exists \sigma : \alpha, z \mapsto \sigma \models_{\mathcal{P}^+(\mathcal{FT})} x \supseteq z \wedge z[f]y$

iff $\exists \sigma : f^{-1}(\alpha(x)) \supseteq \alpha(y)$

For the downward implication notice that $\alpha, z \mapsto \sigma \models_{\mathcal{P}^+(\mathcal{FT})} x \supseteq z \wedge z[f]y$ implies that $\sigma.f$ is always defined, and that $f^{-1}(\alpha(x)) \supseteq f^{-1}(\sigma)$; hence $f^{-1}(\alpha(x)) \supseteq f^{-1}(\sigma) \supseteq \alpha(y)$. For the upward implication simply set $\sigma = f^{-1}(\alpha(x))$.

R^*: $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x,R^*,y)$

iff $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \exists z (z \supseteq y \wedge \Theta(z,R,z) \wedge x \supseteq z)$

iff $\exists \sigma : \alpha, z \mapsto \sigma \models_{\mathcal{P}^+(\mathcal{FT})} z \supseteq y \wedge \Theta(z,R,z) \wedge x \supseteq z$

iff $\exists \sigma : \alpha, z \mapsto \sigma \models_{\mathcal{P}^+(\mathcal{FT})} z \supseteq y \wedge x \supseteq z$ and $\forall p \in \mathcal{L}(R) : p^{-1}(\sigma) \supseteq \sigma$ (IA)

iff $\exists \sigma : \alpha(x) \supseteq \sigma \wedge \sigma \supseteq \alpha(y)$ and $\forall p \in \mathcal{L}(R^*) : p^{-1}(\sigma) \supseteq \sigma$ (**)

(The upward implication of the last equivalence holds since $\mathcal{L}(R) \subseteq \mathcal{L}(R^*)$. The downward implication holds since $\forall p \in \mathcal{L}(R) : p^{-1}(\sigma) \supseteq \sigma$ implies $\forall p,q \in \mathcal{L}(R) : q^{-1}(p^{-1}(\sigma)) \supseteq q^{-1}(\sigma) \supseteq \sigma$, and so on.) The last formula (**) is equivalent to

$$\forall p \in \mathcal{L}(R^*) : \quad p^{-1}(\alpha(x)) \supseteq \alpha(y) \tag{3.17}$$

The downward implication is simple. For the inverse direction assume (3.17) and define

$$\sigma \quad =_{def} \quad \bigcap\{p^{-1}(\alpha(x)) \mid p \in \mathcal{L}(R^*)\}$$

Now, $\alpha(x) = \varepsilon^{-1}(\alpha(x)) \supseteq \sigma$ holds by definition since $\varepsilon \in \mathcal{L}(R^*)$, and $\sigma \supseteq \alpha(y)$ holds since $\forall p \in \mathcal{L}(R^*) : p^{-1}(\alpha(x)) \supseteq \alpha(y)$. Furthermore, for all $p \in \mathcal{L}(R^*)$, and

$$\begin{array}{rcl} p^{-1}(\sigma) & = & \{\tau' \mid \text{exists } \tau \in \sigma : \tau.p = \tau'\} \\ & = & \{\tau' \mid \forall q \in \mathcal{L}(R^*) : \tau' \in p^{-1}(q^{-1}(\alpha(x)))\} \\ & = & \bigcap\{qp^{-1}(\alpha(x)) \mid q \in \mathcal{L}(R^*)\} \quad \supseteq \quad \sigma \end{array}$$

$R_1 \cup R_2$**:** $\quad \alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x, R_1 \cup R_2, y)$

$$\begin{array}{ll} \text{iff} & \alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x, R_1, y) \wedge \Theta(x, R_2, y) \\ \text{iff} & \forall p \in \mathcal{L}(R_1) : p^{-1}(\alpha(x)) \supseteq \alpha(y) \text{ and } \forall q \in \mathcal{L}(R_2) : q^{-1}(\alpha(x)) \supseteq \alpha(y) \\ \text{iff} & \forall p \in \mathcal{L}(R_1 \cup R_2) : p^{-1}(\alpha(x)) \supseteq \alpha(y) \end{array}$$

R_1R_2**:** Note that we have assumed $\mathcal{L}(R_1) \neq \emptyset$ and $\mathcal{L}(R_2) \neq \emptyset$.

$$\begin{array}{ll} \multicolumn{2}{l}{\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x, R_1R_2, y)} \\ \text{iff} & \alpha \models_{\mathcal{P}^+(\mathcal{FT})} \exists z(\Theta(x, R_1, z) \wedge \Theta(z, R_2, y)) \\ \text{iff} & \exists \sigma : \alpha, z \mapsto \sigma \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x, R_1, z) \wedge \Theta(z, R_2, y) \\ \text{iff} & \exists s,\ \forall p{\in}\mathcal{L}(R_1)\forall q{\in}\mathcal{L}(R_2) : p^{-1}(\alpha(x)) \supseteq \sigma \text{ and } q^{-1}(\sigma) \supseteq \alpha(y) \quad \text{by (IA)} \end{array}$$

Due to our assumption that $\mathcal{L}(R_1)$ and $\mathcal{L}(R_2)$ are non-empty, the last clearly formula is clearly equivalent to $\forall p \in \mathcal{L}(R_1R_2) : p^{-1}(\alpha(x)) \supseteq \alpha(y)$. □

3.2.4.3. Proof of Proposition 3.16

Let a and b two distinct labels.

($\Rightarrow$) Assume that $\mathcal{L}(R_2) \not\subseteq \mathcal{L}(R_1)$, so that there exists $p_0 \in \mathcal{L}(R_2)$ such that $p_0 \notin \mathcal{L}(R_1)$. Define a valuation α by

$$\begin{array}{rcl} \alpha(y) & = & \{a\} \\ \alpha(x) & = & \left\{ \tau \;\middle|\; \begin{array}{l} D_\tau = \text{prefix-closure}(\mathcal{L}(R_1 \cup R_2)) \\ \forall p \in \mathcal{L}(R_1) : \tau.p = a,\ \forall q \in \mathcal{L}(R_2 \backslash R_1) : \tau.q = b \end{array} \right\} \end{array}$$

where prefix-closure(S) is the smallest prefix-closed superset of S. Clearly, α defines a valuation into sets of feature trees. From Lemma 3.17 we obtain that $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x,R_1,y)$, and $\alpha \not\models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x,R_2,y)$, because $p_0^{-1}(\alpha(x)) = \{b\}$ and $\{b\} \not\supseteq \{a\}$. Hence $\Theta(x,R_1,y) \not\models \Theta(x,R_2,y)$.

($\Leftarrow$) Assume $\mathcal{L}(R_2) \subseteq \mathcal{L}(R_1)$. Then apparently for all α

$$(\forall p \in \mathcal{L}(R_1) : p^{-1}(\alpha(x)) \supseteq \alpha(y)) \quad \text{implies} \quad (\forall p \in \mathcal{L}(R_2) : p^{-1}(\alpha(x)) \supseteq \alpha(y)).$$

By Lemma 3.17, this is equivalent to saying that for all α the following holds: if $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x,R_1,y)$ then $\alpha \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x,R_2,y)$; that is $\Theta(x,R_1,y) \models_{\mathcal{P}^+(\mathcal{FT})} \Theta(x,R_2,y)$. □

3.2.4.4. Dropping the Non-emptiness Restriction

We check that Proposition 3.16 also holds for the domain of possibly empty sets of feature trees. We check Lemma 3.17 again: the interesting direction is the one from left to right. To show this, pick R and an $\mathcal{P}(\mathcal{FT})$-solution α of $\Theta(x,R,y)$. We make a case distinction on emptiness of $\alpha(x)$ in order to prove

$$\forall p \in \mathcal{L}(R) : p^{-1}(\alpha(x)) \supseteq \alpha(y) \tag{3.18}$$

$\alpha(x) = \emptyset$: By induction over R one shows that this implies $\alpha(y) = \emptyset$ under the assumption that $\alpha \models_{\mathcal{P}(\mathcal{FT})} \Theta(x,R,y)$. Hence (3.18) holds trivially.

$\alpha(x) \neq \emptyset$: If $\alpha(y) = \emptyset$ then again (3.18) holds trivially. Otherwise, one checks that α is a $\mathcal{P}^+(\mathcal{FT})$-solution of $\Theta(x,R,y)$. Then (3.18) follows from Lemma 3.17.

3.3. Discussion and Related Work

3.3.1. Set Constraint Systems

In this section we compare set constraints over feature trees with standard set constraints. We briefly survey classes of standard set constraints (for more exhaustive overviews see [5, 88, 157]), and we consider two standard set constraints more closely, namely projections $x \subseteq a^{-1}_{(k)}(y)$ and term inclusions $a(x,y) \subseteq z$.

3.3.1.1. Standard Set Constraints

A *general set expression* e is built from first-order terms x or $a(\bar{e})$, union $e_1 \cup e_2$, intersection $e_1 \cap e_2$, complement e^c, and projection $a^{-1}_{(k)}(e)$ [85]. All set constraints mentioned below are interpreted in the domain $\mathcal{P}(\mathcal{T})$ of sets of constructor trees. The denotation of the projection term $a^{-1}_{(k)}(\sigma)$ is defined by

$$a^{-1}_{(k)}(\sigma) \quad =_{def} \quad \{\tau \mid \exists \tau_1, \ldots, \tau_n : a(\tau_1, \ldots, \tau_{k-1}, \tau, \tau_{k+1}, \ldots, \tau_n) \in \sigma\} \qquad (3.19)$$

where $1 \leq k \leq n = \mathsf{ar}(a)$, and $y \subseteq a^{-1}_{(k)}(x)$ holds under a $\mathcal{P}(\mathcal{T})$-valuation α if $\alpha(y) \subseteq a^{-1}_{(k)}(\alpha(x))$. A *general set constraint* is a conjunction of inclusions of the form $e \subseteq e'$. A *positive set constraint* is built from *positive set expressions* that do not contain the complement operator. A *definite set constraint* [85] is a conjunction of inclusions $e_l \subseteq e_r$ between positive set expressions, where the set expressions e_r on the right hand side of an inclusion are furthermore restricted to contain only variables, constants and function symbols and the intersection operator (that is, no projection or union). Heintze and Jaffar have called this class definite because every satisfiable constraint of this class has a least solution.

Charatonik and Podelski [42] define the class of *set constraints with intersections* (inclusions between set expressions built from variables, constructors, and intersection only) and show them to be equivalent to definite set constraints. They also define the class of *co-definite set constraints* [44] whose (flattened) syntax is as follows:

$$\psi ::= \quad a \subseteq x \mid x \subseteq y_1 \cup \ldots \cup y_n \mid x \subseteq a(\bar{x}) \mid x \subseteq a^{-1}_{(k)}(y) \mid \psi \wedge \psi'$$

An essential property of co-definite set constraints is that they have a greatest solution if satisfiable (over finite as well as infinite trees). This property is dual to the least model property of definite set constraints; hence the name "co-definite". Otherwise, both systems are not dual to each other. Devienne, Talbot, and Tison [58, 59] have extended both definite and co-definite set constraints by so-called membership expressions. Membership expressions are set comprehensions whose body is an existentially quantified conjunction of inclusions $t \in e$ between first-order terms and set expressions.

At the lower end of the scale of expressiveness, *atomic set constraints* are inclusions between first-order terms and no further set operators [85]. *Inclusion constraints over non-empty sets* Ines [142] are inclusion between first-order terms (their syntax coincides with the syntax of atomic set constraints) interpreted over non-empty sets of trees.

3.3.1.2. Projections

The selection constraint $x[f]y$ in our set constraint system $\mathrm{FT}_{\subseteq}(ar,\cup)$ corresponds to the projection constraint $x\subseteq a^{-1}_{(k)}(y)$ in standard set constraints. There are two differences, though.

First, recall that the constraint $x[f]y$ requires all trees in the denotation of x to have the feature f. In addition, it constrains y to the projection of x at f: so $x[f]y$ is a constraint on both x and y. In contrast, the projection constraint $y\subseteq a^{-1}_{(k)}(x)$ does not restrict the possible values of x: for every value of x there is a solution of $y\subseteq a^{-1}_{(k)}(x)$. An alternative set-up of our constraint system would have used two constraints to represent the meaning of $x[f]y$, namely $y = f^{-1}(x)$ to express projection at f and $x[f]\downarrow$ to require definedness of f. The latter one and the labelling constraint $a\langle x\rangle$ are used to define non-trivial sets. In standard set constraints, this is expressed by *set terms* such as $a(\overline{y})$, where the denotation of $a(\sigma_1,\ldots,\sigma_n)$ is defined as follows.

$$a(\sigma_1,\ldots,\sigma_n) \quad =_{def} \quad \{a(\tau_1,\ldots,\tau_n) \mid \tau_1 \in \sigma_1,\ldots,\tau_n \in \sigma_n\} \tag{3.20}$$

A third alternative would have been a system based on feature terms like $a(\overline{f}{:}\overline{x})$, $a(\overline{f}{:}\overline{x}\ldots)$, $(\overline{f}{:}\overline{x})$, and $(\overline{f}{:}\overline{x}\ldots)$ whose denotation is defined similar to (3.20) with the additional flexibility that label, feature, and arity information can be freely combined or piece-wise omitted. Amongst these alternatives, the system $\mathrm{FT}_{\subseteq}(ar,\cup)$ is intriguing by its simplicity and its similarity to the feature constraint system CFT over trees which it analyses. Furthermore, the semantics of selection constraints in $\mathrm{FT}_{\subseteq}(ar,\cup)$ seems most appropriate for the analysis of selection constraints of CFT; see example D_{fail3} on Page 94.

An interesting property of the selection constraint is the validity of the following implication over $\mathcal{P}(\mathcal{FT})$.

$$x[f]y \quad\rightarrow\quad (x{=}\emptyset \leftrightarrow y{=}\emptyset) \tag{3.21}$$

Given two distinct features f and g, we can even express that emptiness of one variable is equivalent to emptiness of another one (we exploit this in Chapter 4):

$$\exists z(z[f]x \wedge z[g]y) \quad\leftrightarrow\quad (x{=}\emptyset{\leftrightarrow}y{=}\emptyset) \tag{3.22}$$

In contrast, the projection constraint $x \subseteq a^{-1}_{(1)}(y)$ does not entail $x{=}\emptyset \leftrightarrow y{=}\emptyset$, since it has a solution that maps y to the set $\{b\}$ and x to the empty set. However, in combination with a set constructor $x \subseteq a(\overline{y})$ a similar formula (albeit constructor dependent) holds:[18]

$$x \subseteq a(y_1, \overline{y}) \wedge y_1 \subseteq a^{-1}_{(1)}(x) \quad \rightarrow \quad (x{=}\emptyset \leftrightarrow y_1{=}\emptyset) \tag{3.23}$$

In $\mathrm{FT}_{\subseteq}(ar)$, we can express the projection constraint $x \subseteq a^{-1}_{(k)}(y)$ as follows:

$$[\![x \subseteq a^{-1}_{(k)}(y)]\!] \quad =_{def} \quad \exists x' \, (x'[k]x \wedge a\langle x' \rangle \wedge x' \subseteq y) \tag{3.24}$$

Theorem 22 below makes precise what "expressing" means. Intuitively, labelling and selection constraints in encoding (3.24) separate the two services of the projection operator $a^{-1}_{(k)}$: applied to a set σ, first determines a subset σ' of σ of trees that are labelled with a and have a k^{th} subtree, and then collect the k^{th} subtrees of all the trees in σ'.[19]

Constructor trees can be embedded into feature trees; denote with $[\![\cdot]\!] : \mathcal{T} \mapsto \mathcal{FT}$ the canonical embedding which we have mentioned on Page 26. On constructor trees, the constraint system CFT is a refinement of the constraint system RT of infinite constructor trees [50] that is used in Prolog II [51]. To show this, it was proven in [197] that the embedding $[\![\cdot]\!]$ (extended to first-order connectives) preserves validity of arbitrary first-order formulas over RT.

Analogously, $\mathrm{FT}_{\subseteq}(ar, \cup)$ refine co-definite set constraints in a sense made precise by the following theorem. If α is a $\mathcal{P}(\mathcal{T})$-valuation then let $[\![\alpha]\!]$ the $\mathcal{P}(\mathcal{FT})$-valuation that maps all x to $\{[\![\tau]\!] \mid \tau \in \alpha(x)\}$. Consider the embedding $[\![\cdot]\!]$ of co-definite set constraints ψ into $\mathrm{FT}_{\subseteq}(ar, \cup)$ constraints φ as defined by the clauses (2.5) and (2.6) on Page 28 and clause (3.24) above.

Theorem 22 (Embedding Co-definite Set Constraints wrt. Greatest Solutions)
For all co-definite set constraints ψ without constraints of the form $a \subseteq x$, the greatest $\mathcal{P}(\mathcal{T})$-solution of ψ and the greatest $\mathcal{P}(\mathcal{FT})$-solution coincide up to the canonical embedding $[\![\cdot]\!]$ of constructor trees into feature trees: α is the greatest solution of ψ if and only if $[\![\alpha]\!]$ is the greatest solution of $[\![\psi]\!]$.

Proof. Straightforward. □

Notice that this theorem would fail to hold if the semantics of $\sigma[f]\sigma'$ did not require all trees in σ' to have the feature f. As an experiment, assuming the slightly weaker

[18] The inverse implication, of course, does not hold because the right hand side does not mention the label a.

[19] Notice in passing, that the inverse inclusion $a^{-1}_{(k)}(y) \subseteq x$ cannot be expressed in $\mathrm{FT}_{\subseteq}(ar)$. Doing so, *e. g.*, in order to embed definite set constraints [85], would probably require a term-based constraint syntax as mentioned on the previous page.

definition: $\sigma[f]\sigma'$ if and only if $\sigma' = \{\tau' \mid \exists\tau \in \sigma : \tau[f]\tau'\}$; then the greatest solution of

$$x \subseteq a(y) \wedge y = b \wedge x \subseteq a(z) \wedge z = c$$

would map x to the empty set (if $b \neq c$), while the greatest solution of

$$x \subseteq y' \wedge a\langle y'\rangle \wedge y'\{1\} \wedge y'[1]y \wedge b\langle y\rangle \wedge y\{\}$$
$$x \subseteq z' \wedge a\langle z'\rangle \wedge z'\{1\} \wedge z'[1]z \wedge c\langle z\rangle \wedge y\{\}$$

would map x to the set of all feature trees that are labelled with a but that do not have the feature 1.

Co-definite set constraints can express inconsistency, for example by

$$a \subseteq x \wedge x \subseteq b \leftrightarrow \bot \qquad \text{if } a \neq b\,.$$

whereas this is impossible in $FT_{\subseteq}(ar, \cup)$. In particular, $FT_{\subseteq}(ar, \cup)$ constraints cannot express the co-definite set constraint $a \subseteq x$. However, it can be expressed in the first-order theory of $FT_{\subseteq}(ar, \cup)$ as follows:

$$\exists y(a\langle y\rangle \wedge y\{\} \wedge \neg b\langle y\rangle \wedge y \subseteq x) \qquad \text{where } a \neq b\,.$$

We conjecture that, using this trick, we can embed the full first-order theory of co-definite set constraints into the first-order theory of $FT_{\subseteq}(ar, \cup)$ such that validity is preserved.

3.3.1.3. Finite versus Infinite Trees

Consider the following two constraints:

$$\eta_1 \;=_{def}\; x[f]x$$
$$\eta_2 \;=_{def}\; x \subseteq a^{-1}_{(1)}(x)$$

Over sets of finite trees, the $FT_{\subseteq}(ar, \cup)$ constraint η_1 implies $x = \emptyset$, because for every solution α the selection $\alpha(x).f$ must be defined and $\alpha(x).f \subseteq \alpha(x)$ holds; hence every tree in $\alpha(x)$ must have the infinite path $fff\ldots$ and therefore be infinite. In contrast, the projection constraint η_2 has the solution $\alpha(x) = \{b, a(b), a(a(b)), \ldots\}$ which is a set of finite trees only.

3.3.1.4. Term Inclusion and Greatest Solutions

The set constraint $a(x,y) \subseteq \emptyset$ (expressible as $a(x,y) \subseteq z \wedge z \subseteq a \wedge z \subseteq b$ if $a \neq b$) has two maximal but incomparable solutions over sets of constructor trees, namely the ones that map one variable to the empty set $\emptyset$ and the other one to the full domain $\mathcal{P}(\mathcal{T})$. Maximality holds for both because the following equivalence is valid.

$$a(x,y) \subseteq \emptyset \quad \leftrightarrow \quad (x = \emptyset \vee y = \emptyset) \tag{3.25}$$

In other words, the constraint $a(x,y) \subseteq \emptyset$ has no greatest solution because emptiness of x and y is not independent from each other. Similarly, the constraint $f(x,y) \subseteq f(a,a) \cup f(b,b)$ has two maximal solutions but no greatest one.

Co-definite set constraints (over sets of constructor trees) and inclusion constraints over sets of feature trees are two options to avoid this dependency. A third option is to exclude the empty set [142].

Co-definite set constraints. If only constants or monadic terms are allowed on the left hand side of an inclusion, the critical dependency cannot arise.

Constraints over sets of feature trees. The co-definite set constraint $a(x,y) \subseteq \emptyset$ corresponds to the $\mathrm{FT}_{\subseteq}(ar)$ constraint $z \subseteq \emptyset \wedge z\{1,2\} \wedge a\langle z \rangle \wedge z[\mathit{1}]x \wedge z[\mathit{2}]y$ which entails $x = \emptyset \wedge y = \emptyset$ (that is, that *both* x and y denote the empty set) due to formula (3.21).

Constraints over non-empty sets of trees. If the empty set is excluded from the interpretation domain, there are greatest solutions even if terms are admitted on the left of an inclusion. For example, when Mishra's set constraints (see Section 2.3.2) are interpreted over non-empty path-closed sets, the critical dependency cannot arise. To see this, observe that $a(x,y) \subseteq z \leftrightarrow x \subseteq a^{-1}_{(1)}(z) \wedge y \subseteq a^{-1}_{(2)}(z)$ is a valid equivalence over path-closed sets (for the simple proof, see [43]).

The independence of emptiness between neighbouring projections can also simplify the satisfiability test for systems of set constraints. Notice that the following implication between co-definite set constraints is not valid.

$$a(x_1,\ x_2) \subseteq a(y_1,\ y_2) \quad \rightarrow \quad (x_1 \subseteq y_1 \wedge x_2 \subseteq y_2) \tag{3.26}$$

In contrast, the analogous $\mathrm{FT}_{\subseteq}$ implication

$$x[\mathit{1}]x' \wedge x \subseteq y \wedge y[\mathit{1}]y' \quad \rightarrow \quad x' \subseteq y' \tag{3.27}$$

is valid, and is part of the satisfiability check for $\mathrm{FT}_{\subseteq}(ar)$ presented in Section 2.2.1. Notice that implication (3.26) does hold over sets of constructor trees under the additional assumption that both x_1 and x_2 denote a non-empty set. So it is a valid implication in the constraint system Ines [142] that excludes the empty set globally from the interpretation domain of all variables.

3.3.1.5. Decidability and Complexity of Set Constraints

Various decidability and complexity results have been obtained for different classes of set constraints. For many the complexity of the satisfiability problem for the full class of standard set constraints is very high.

Heintze and Jaffar [85] show the satisfiability problem for definite set constraints to be decidable, thereby giving the first decidability result for a class of set constraints. Aiken and Wimmers show the class of *positive set constraints* to be decidable in NEXPTIME [10]. Gilleron, Tison, and Tommasi prove decidability for the satisfiability problem of positive set constraints using so-called tree set automata [74]. Bachmair, Ganzinger, and Waldmann [19] have noticed the equivalence of positive set constraints to a certain first-order theory called the *monadic class*, and could thus show the satisfiability problem of positive set constraints to be NEXPTIME-complete. Later, the decidability result has been extended to include negated inclusion constraints $e \not\subseteq e'$ by various researchers [8, 39, 75, 199], and to projection by Charatonik and Pacholski [40]. None of these extensions changes the worst-case complexity of the satisfiability problem. Aiken, Kozen, Vardi and Wimmers, have studied complexity of satisfiability for various subclasses of positive set constraints [7], defined by restrictions on the arities of the function symbols in the given signature.

Charatonik and Podelski show that the satisfiability problem of both set constraints with intersection and co-definite set constraints is DEXPTIME-complete [42, 44].The result on set constraints with intersection has also settled the complexity of the satisfiability problem of definite set constraints. Devienne, Talbot, and Tison [58, 59] have applied tree automata techniques to solve set constraints with membership expressions (with respect to both the greatest and the least model semantics) and could show that membership expressions do not change the DEXPTIME-completeness of the satisfiability problem for either definite or co-definite set constraints, nor affect the greatest-solution property of co-definite set constraints.

The two set constraint systems without any set operators apart from term construction, atomic set constraints and inclusion constraints over non-empty sets Ines [85, 142] have a cubic satisfiability problem [142]. For atomic set constraints, this result was implicit in the existing literature [84, 85]. In contrast to the polynomial satisfiability, the entailment problem for both classes is infeasible: we show it to be coNP-hard in [140].

3.3.2. Tree Constraint Systems

In this section, we compare systems of set constraints with systems of tree constraints, both with respect to equality constraints and ordering constraints. Figure 3.9 summarises the relationship between some of the mentioned constraint systems over trees and sets of trees. The constraint systems occupy the nodes of the cube. They are

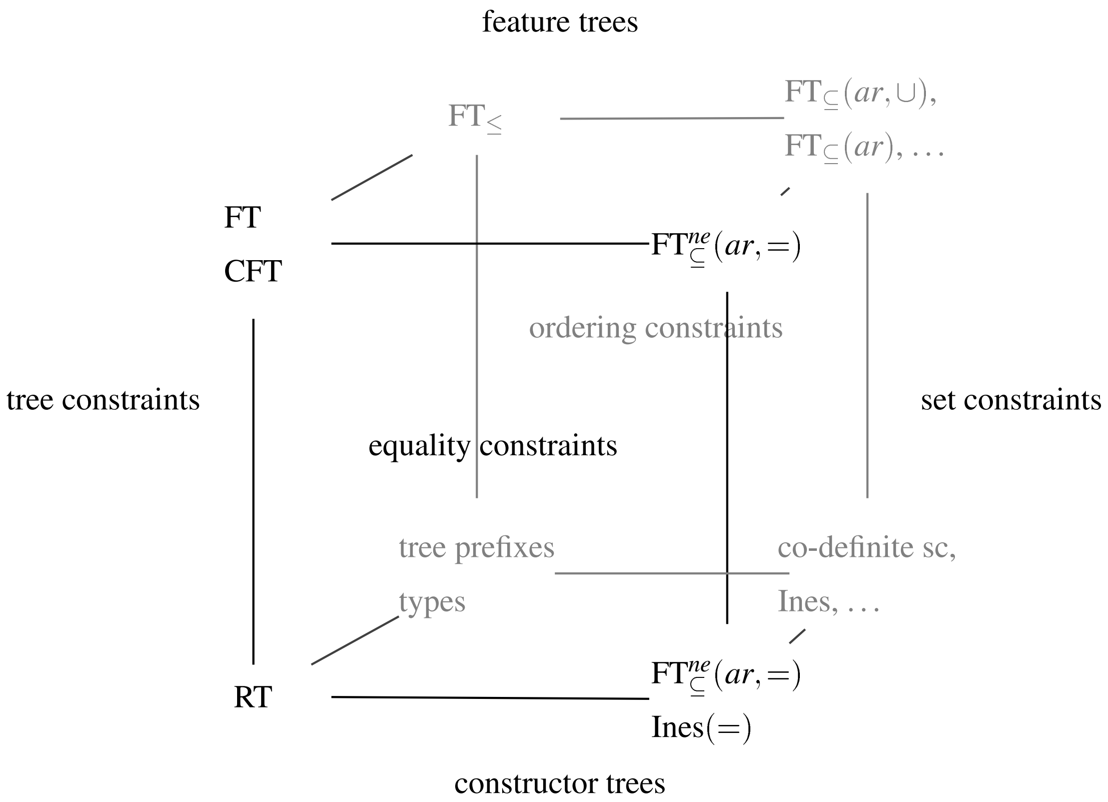


Figure 3.9.: Related Tree and Set Constraint Systems

arranged along three dimensions that tell whether a constraint system talks about constructor trees or feature trees (top – bottom), about equality or an ordering relation (front – back), and about trees or sets of trees (left – right). The edges of the cube do not imply any further formal relationship between the constraint systems at the nodes.

3.3.2.1. Equality Constraints over Sets and Trees

Set constraints over non-empty sets of trees are closely related to constraint system over trees. This relation is tightest on the fragment of equality constraints. It was noticed in [142] that the first-order theories of CFT and Ines coincide when their constraint languages are restricted to equality constraints. The analogous result holds for $\mathrm{FT}^{ne}_{\subseteq}(ar)$ and CFT. Intuitively, the following Theorem 23 says that we can solve equality constraints over $\mathrm{FT}^{ne}_{\subseteq}(ar)$ by unification. (Of course, we do not obtain the quasi-linear complexity of unification [197] by simply applying our algorithm to an equality constraint with $x{=}y$ replaced by $x{\subseteq}y \wedge y{\subseteq}x$.)

Theorem 23 (First-order Theory of Equality Constraints)
The first-order theories of equality constraints (including arity constraints) over feature trees and over non-empty sets of feature trees coincide.

Proof. This follows from the fact that all axioms of the complete axiomatisation of CFT [20, 197] are valid for non-empty sets of feature trees.

Amongst the five axioms of the theory CFT in [197], four are immediately seen to hold over $\mathcal{P}^+(\mathcal{FT})$: Functionality of features, $\forall x \forall y \forall z (x[f]y \wedge x[f]z \rightarrow y{=}z)$, clash between different labels, $\forall x (a\langle x\rangle \wedge b\langle x\rangle \rightarrow \perp)$ if $a \neq b$, clash of feature selection at a tree with inappropriate arity, $\forall x \forall y (x[f]y \wedge x\{\overline{g}\})$ if $f \notin \{\overline{g}\}$, and the axiom $\forall x \forall y (x\{f\overline{g}\} \rightarrow \exists y x[f]y)$. The first three are actually part of our satisfiability test for $\mathrm{FT}_{\subseteq}(ar)$ in Section 2.2.1. The fifth one is based on the notion of *determinants*:A determinant is a conjunction of the form $x_1{=}a_1(\overline{f}_1{:}\overline{y}_1) \wedge \ldots \wedge x_n{=}a_n(\overline{f}_n{:}\overline{y}_n)$ for pairwise distinct variables $x_1, \ldots, x_n$, and looks as follows (where $\tilde{\forall}\varphi$ denotes the universal closure of φ):

$$\tilde{\forall} \exists! x_1, \ldots, x_n (x_1{=}a_1(\overline{f}_1{:}\overline{y}_1) \wedge \ldots \wedge x_n{=}a_n(\overline{f}_n{:}\overline{y}_n)) \tag{3.28}$$

Its validity in $\mathcal{P}^+(\mathcal{FT})$ is again easily seen. □

Notice that axiom (3.28) does not hold (and so Theorem 23 fails) when the empty set is admitted. For example, consider the following instance of (3.28),

$$\forall x \forall y \exists! z (z{=}a(f{:}x, g{:}y)) \quad \equiv \quad \forall x \forall y \exists! z (a\langle z\rangle \wedge z\{f, g\} \wedge z[f]x \wedge z[g]y)\,,$$

and notice that it does not hold over $\mathcal{P}(\mathcal{FT})$, because

$$\alpha \models_{\mathcal{P}(\mathcal{FT})} \neg \exists z (z{=}a(f{:}x, g{:}y)) \quad \text{if } \alpha(x) = \emptyset \text{ and } \alpha(y) \neq \emptyset\,.$$

3.3.2.2. Set Inclusion Constraints versus Tree Ordering Constraints

The constraint system $\mathrm{FT}_{\leq}$ of ordering constraints over feature trees (not sets) is the second closest relative of $\mathrm{FT}_{\subseteq}(ar, \cup)$ [141, 143, 145]. In particular the fragment $\mathrm{FT}_{\subseteq}$ is, roughly, $\mathrm{FT}_{\leq}$ transformed into a set constraint system.

The constraints in $\mathrm{FT}_{\leq}$ coincide with those in $\mathrm{FT}^{ne}_{\subseteq}$ (no arity and no union constraints) where inclusion constraints $x \subseteq y$ are replaced by the inverse tree ordering constraints $y \leq x$. $\mathrm{FT}_{\leq}$ constraints are interpreted over feature trees τ whose labelling function L_τ may be partial on the tree domain D_τ. The ordering constraint is interpreted by

$$\tau \leq \tau' \quad \text{iff} \quad D_\tau \subseteq D_{\tau'} \text{ and } L_\tau \subseteq L_{\tau'} \tag{3.29}$$

In the following chapter, we exploit the close relationship between tree constraints and set constraints over non-empty sets by applying techniques to the solving of set constraints that have been originally developed for tree constraint systems.

- The satisfiability test for $\mathrm{FT}^{ne}_{\subseteq}(ar)$ in Section 2.2.1 is essentially the same as the one we have given for $\mathrm{FT}_{\leq}$ [143].

- Surprisingly, also the cubic entailment test for $\mathrm{FT}_{\leq}$ can be transferred almost unchanged to $\mathrm{FT}^{ne}_{\subseteq}$ [143]; see Section 3.1.

- The same holds for the coNP- and PSPACE-hardness results for entailment with existential quantification [141]; see Section 3.2.

- An extension of $\mathrm{FT}_{\leq}$, called $\mathrm{FT}_{\leq}(sort)$, that allows the labels of feature trees to be partially ordered is discussed in [137]. It was shown that the satisfiability test for $\mathrm{FT}_{\leq}(sort)$ remains polynomial under certain assumptions on the partial order of labels. It seems straightforward to also extend the satisfiability test for $\mathrm{FT}^{ne}_{\subseteq}(ar)$ to $\mathrm{FT}^{ne}_{\subseteq}(ar, sort)$ along the lines of $\mathrm{FT}_{\leq}(sort)$.

Notice in passing that higher fragments of the first-order theory of $\mathrm{FT}_{\leq}$ and $\mathrm{FT}^{ne}_{\subseteq}$ do not coincide (see the Footnote 14 on Page 55.) For $\mathrm{FT}_{\leq}$, we have shown that entailment with existential quantification is PSPACE-complete, both for the cases of finite and infinite trees [141, 145]; the full first-order theory of $\mathrm{FT}_{\leq}$ is undecidable, in both the finite tree and the infinite tree case [145]. For $\mathrm{FT}^{ne}_{\subseteq}(ar)$ and $\mathrm{FT}_{\subseteq}(ar)$, the decidability question and a precise complexity characterisation of entailment with existential quantifiers is open. Undecidability for the first-order theory of $\mathrm{FT}_{\subseteq}(ar, \cup)$ is likely [186].

3.3.2.3. Ordering Constraints over Feature and Constructor Trees

There are different options to extend the domain $\mathcal{T}$ of constructor trees by an ordering.

- One option is to enlarge $\mathcal{T}$ to also contain *tree prefixes* [142], that is constructor trees that may have unlabelled maximal paths. Tree prefixes can be ordered naturally according to equivalence (3.29). On $\mathcal{T}$, this ordering collapses to equality.

- A second, equivalent possibility is to distinguish a special constant symbol $\bullet$ and define an order by requiring

$$\bullet \leq f(\bar{t}) \qquad \text{and} \qquad \bar{t} \leq \bar{t}' \quad \text{iff} \quad f(\bar{t}) \leq f(\bar{t}')) \tag{3.30}$$

  for all f and tree sequences $\bar{t},\bar{t}'$ of appropriate length. Paths leading to $\bullet$ correspond to maximally unlabelled paths in tree prefixes.

- A third option is to fix an order $\leq$ on labels and to consider the ordering

$$\tau \leq \tau' \quad \text{iff} \quad D_\tau = D_{\tau'} \text{ and } L_\tau \leq L_{\tau'} \tag{3.31}$$

  where $L_\tau \leq L_{\tau'}$ extends $\leq$ path-wise to trees.

Orders on various classes of constructor trees have been considered in the context of type systems for programming languages [34, 134]. Mostly, types are modelled by finite constructor trees over a signature that contains a binary function symbol $\rightarrow$ (arrow), where the ordering on trees with the arrow $\rightarrow$ as their top-level constructor is monotonic (covariant) in the second position but antimonotonic (contravariant) in the second one (see also Page 158 for the subtyping rule on functions). In this context, the orderings (3.29) and (3.31) roughly correspond to what is called *non-structural* and *structural subtyping*. Henglein and Rehof show entailment of ordering constraints with respect to *structural* subtyping to be coNP-complete [89] for finite types, and, more recently, for infinite types [90]. None of the corresponding hardness results relies on the arrow constructor. The entailment problem for *non-structural* subtyping constraints is PSPACE-hard [90] but the exact complexity is open.

4. Set-based Failure Diagnosis for CLP and CC

We consider a concurrent constraint programming language over records, that we model by means of the constraint system CFT [197] over possibly infinite feature trees. As common in concurrent programming, we consider non-terminating computations meaningful. Typical examples for applications that are intended to run forever include operating systems or web servers. We define a set-based analysis for this language in terms of set constraints over feature trees, and we prove that it detects the inevitability of a class of run-time errors. We proceed in two steps: First, we consider the sublanguage that contains only unguarded clauses and thus corresponds to *constraint logic programming* [104]. This allows us to use results from the theory of (constraint) logic programming [105, 118] to prove correctness of our analysis for this fragment. Second, we adapt our result to *concurrent constraint programs* by considering guarded clauses.

Constraint Logic Programming. The standard semantics for terminating constraint logic programs is given by the *least model* of their *completion* [46, 118]. This choice is natural for programs that always terminate because the least model is given by all procedure applications (*i. e.*, *goals*) that terminate successfully in finite time. Traditionally, constraint programs have been interpreted over *finite trees* (even though modern Prolog dialects have followed Prolog II [50, 51] in providing constraints over infinite constructor trees).

In contrast, the standard semantics for possibly non-terminating constraint logic programs[20] is given by the *greatest model* of the completion. Moreover, the natural inter-

[20] In logic programming the term *perpetual processes* has been used synonymously [118].

pretation of non-terminating logic programs is over *infinite trees* [118]. The greatest model is needed to give semantics to infinite computations that build data structures of arbitrary size. For example, an infinite data structure is needed to explain an infinite stream of messages.

So far, set-based analysis for (constraint) logic programs has focussed on terminating computations and hence striven to approximate the least model semantics [69, 86–88, 132]. Heintze and Jaffar state explicitly that they do not see any use for greatest models [87]. Furthermore, set-based analysis has usually considered (constraint) logic programs over finite trees.[21] We base our set-based failure diagnosis on greatest models and constraints over infinite trees.

With every CLP program D we associate a set constraint over feature trees φ_D such that the greatest solution $gsol(\varphi_D)$ of φ_D is an upper approximation of the greatest model of the program's completion.

$$gm(D) \subseteq gsol(\varphi_D)$$

In order to prove this, we apply a technique that is well-known from abstract interpretation [53]. We associate to every CLP program D over feature trees an abstract program $D^\#$ over sets of feature trees and prove that the semantics of the latter is an upper approximation the semantics of the former. More precisely, if $gm(D)$ and $gm(D^\#)$ are the greatest models of D and $D^\#$, respectively, then for every predicate symbol p the maximal element in $gm(D^\#)(p)$ includes $gm(D)(p)$. Second, we prove that the greatest model of the abstract program $D^\#$ and the greatest solution of φ_D coincide (again, up to the projection to maximal elements).

We can relate this approximation result of the *denotational* semantics to the *operational* semantics by characterising finite failure over *infinite trees* through the greatest model.[22] This allows us to infer finite failure of the CLP program from emptiness of some variable in $gsol(\varphi)$.

Our analysis of CLP programs is more flexible than the one that we give in the paper [171]. There, we require the constraints in program clauses to be solved before they can be analysed; here, we show how to analyse CLP program which freely use constraints in unsolved form.

Concurrent Constraint Programming. The analysis of CC programs is given by the analysis of the CLP program that we obtain by transforming all conditional guards

[21] An analysis for Prolog II that approximates the *least* model over *infinite* trees seems to be much more difficult than it occurs at first glance. This is due to the fact that infinite trees are defined by a greatest fixed point construction, so that a corresponding analysis would have to approximate the least fixed point of an operator that refers to the greatest fixed point of another one; this "alternation" seems to make the analysis considerably harder.

[22] While results of Jaffar and Stuckey on infinite tree logic programming are closely related [107], the exact result that we show seems not to be explicit in the literature. See Section 4.1.4.2 for details.

into tell statements; this means that our analysis ignores the synchronisation behaviour of conditional guards. This is one reason for our interest in inevitable failure: for most CC programs with non-trivial recursions (over lists) say, the corresponding CLP program that approximates it has more failed computation branches. Hence, diagnosing possible failure in the CLP program would not imply possible failure in the original CC program (see also Section 4.2.3).

While we can carry over the approximation result of the greatest model, the operational interpretation of $gsol(\varphi)$ needs some more care: in CC programs, there may be statements whose reduction blocks forever because a synchronisation condition is never satisfied. For such CC programs in which every application can eventually reduce, our characterisation of finite failure through the greatest model still holds. Since we cannot guarantee statically that application will not block, we must weaken our result for CC programs: We show that emptiness in $gsol(\varphi)$ implies finite failure in every fair execution of a CC program *D unless* there is an application that blocks forever.

In concurrent constraint programming, failure is considered a run-time error. This is in contrast to CLP where failure is part of the backtracking mechanism. A CC program has certainly an error if every fair execution leads to failure. In our programming experience with Oz, programs are also erroneous if they do not fail *only because* some application blocks forever. In other words, emptiness in $gsol(\varphi)$ correctly approximates a run-time property in CLP and CC programs that is useful for debugging.

4.1. Set-based Failure Diagnosis for CLP over Infinite Trees

We define our set-based failure diagnosis in Figure 4.4 on Page 101. Before we discuss it in detail, we consider some examples where we suppress irrelevant technical detail. The reader unfamiliar with CC and CLP may want to consult the definition of the language in Section 4.1.2 before reading on.

4.1.1. Examples

4.1.1.1. Basic Examples

We call a procedure p finitely failed if every fair execution of an application $p(x)$ inevitably leads to a failure. The procedure p in D_{fail1} is obviously finitely failed in this sense (if $a \neq b$):

$$p(x) \leftarrow a\langle x\rangle \wedge b\langle x\rangle \qquad (D_{fail1})$$

With D_{fail1} we associate the following $\mathrm{FT}_{\subseteq}(ar, \cup)$ constraint (the *analysis* of D_{fail1}):

$$p \subseteq x \ \wedge \ a\langle x\rangle \ \wedge \ b\langle x\rangle$$

from which we infer finite failure of p by noticing that it entails $p{=}\emptyset$. Slightly less obvious but similar is the finite failure of procedure r in D_{fail2}:

$$\begin{aligned} & p(x) \leftarrow a\langle x\rangle && (D_{fail2})\\ & q(y) \leftarrow b\langle y\rangle \\ & r(z) \leftarrow p(z),\ q(z) \end{aligned}$$

The constraint associated with D_{fail2} states that every actual argument of the procedure p [q] must allow labelling with a [b], and that z must be a valid argument for both procedures p and q.

$$\begin{aligned} & p{\subseteq}x \ \wedge\ a\langle x\rangle \ \wedge \\ & q{\subseteq}y \ \wedge\ b\langle y\rangle \ \wedge \\ & r{\subseteq}z \ \wedge\ z{\subseteq}p \ \wedge\ z{\subseteq}q \end{aligned}$$

Since this constraint entails $r{=}\emptyset$, we conclude that r is finitely failed. The procedure r is also finitely failed in the next example which is still a little more complicated:

$$\begin{aligned} & p(x) \leftarrow x[f]x' \wedge a\langle x'\rangle && (D_{fail3})\\ & q(y) \leftarrow y[f]y' \wedge b\langle y'\rangle \\ & r(z) \leftarrow p(z),\ q(z) \end{aligned}$$

The analysis of D_{fail3} is this one:

$$\begin{aligned} & p{\subseteq}x \ \wedge\ x[f]x' \ \wedge\ a\langle x'\rangle \ \wedge \\ & q{\subseteq}y \ \wedge\ y[f]y' \ \wedge\ b\langle y'\rangle \ \wedge \\ & r{\subseteq}z \ \wedge\ z{\subseteq}p \ \wedge\ z{\subseteq}q \end{aligned}$$

We reject the program because its analysis entails $r = \emptyset$. Notice that this were not the case if the semantics of $\sigma[f]\sigma'$ did not require all trees in σ to have the feature f. Assuming the weaker semantics, $\sigma[f]\sigma'$ if and only if $\sigma' = \{\tau' \mid \exists\tau \in \sigma : \tau[f]\tau'\}$, the analysis of D_{fail3} would have a solution α with $\alpha(p) = \alpha(x) = \{a, b(f{:}a)\}$, $\alpha(q) = \alpha(y) = \{a, b(f{:}b)\}$, and $\alpha(r) = \alpha(z) = \{a\}$.

As the program D_{fail1} above indicates, our analysis can deal with clauses containing non-satisfiable constraints. More generally, we do not require the constraints in clause bodies to be solved before they can be analysed: our analysis is invariant under equivalence transformations of CFT constraints. For instance, the analysis of the two programs below is the same.

$$\begin{aligned} & p(x) \leftarrow x[f]y,\ x[f]z && (D_{solve1})\\ & p(x) \leftarrow x[f]y,\ y{=}z && (D_{solve2}) \end{aligned}$$

4.1.1.2. Guarded Clauses

Now consider a program with guarded clauses.

$$p(x_1) \leftarrow \eta_1 \ \mathsf{then}\ a\langle x_1 \rangle \qquad (D_{susp1})$$
$$p(x_2) \leftarrow \eta_2 \ \mathsf{then}\ b\langle x_2 \rangle$$
$$r(z) \leftarrow c\langle z \rangle,\ p(z)$$

In D_{susp1}, the procedure r is considered erroneous. Its body requires z to be labelled with c, but at the same time z should allow labelling with either a or b according to the clauses of p. The analysis

$$p \subseteq x_1 \cup x_2 \ \wedge\ a\langle x_1 \rangle \ \wedge\ b\langle x_2 \rangle \ \wedge$$
$$r \subseteq z \ \wedge\ c\langle z \rangle \ \wedge\ z \subseteq p \ \wedge \ldots$$

detects this because it entails $r{=}\emptyset$. On application $r(u)$ it is not clear, however, whether any of the clauses of p will ever be executed. If both guards η_1 or η_2 are never entailed, the label inconsistency with respect to u will not be exhibited. So the fact that the analysis of D_{susp1} entails emptiness of z implies finite failure of the procedure r *unless* the application of to procedure p blocks forever.

The next program shows a similar phenomenon. There is no way to execute applications of p and q on the same argument z without failure.

$$p(x_1) \leftarrow \eta_1 \ \mathsf{then}\ a\langle x_1 \rangle \qquad q(y_1) \leftarrow \eta_3 \ \mathsf{then}\ c\langle y_1 \rangle \qquad (D_{susp2})$$
$$p(x_2) \leftarrow \eta_2 \ \mathsf{then}\ b\langle x_2 \rangle \qquad q(y_2) \leftarrow \eta_4 \ \mathsf{then}\ d\langle y_2 \rangle$$
$$r(z) \leftarrow p(z),\ q(z)$$

The analysis detects this the constraint associated with D_{susp2} entails $r{=}\emptyset$:

$$p \subseteq x_1 \cup x_2 \ \wedge\ a\langle x_1 \rangle \ \wedge\ b\langle x_2 \rangle \ \wedge$$
$$q \subseteq y_1 \cup y_2 \ \wedge\ c\langle y_1 \rangle \ \wedge\ d\langle y_2 \rangle \ \wedge$$
$$r \subseteq z \ \wedge\ z \subseteq p \ \wedge\ z \subseteq q \ \wedge\ \ldots$$

4.1.1.3. Infinite Trees

A program that explicitly talks about infinite data structures is the following one.

$$p(x) \leftarrow x[f]y,\ p(y) \qquad (D_{inf1})$$

Reduction of the application $p(z)$ will enter an infinite recursion, which will constrain z to a feature tree with an arbitrarily long but finite path of f's. The program is deterministic and will never fail. This program is accepted since its analysis does not entail x or y to be empty:

$$p \subseteq x \ \wedge\ x[f]y \ \wedge\ y \subseteq x$$

In comparison, consider the following program and its analysis:

$$p(x) \leftarrow x[f]x \qquad (D_{\mathit{inf2}})$$

$$p \subseteq x \ \wedge \ x[f]x$$

Execution of $p(x)$ will instantaneously terminate and constrain x to an infinite tree containing the path $fff \ldots f$. The greatest solutions of the analyses of D_{inf1} and D_{inf2} coincide, but the need for (sets of) infinite trees has different reasons: in D_{inf1} it is due to an infinite computation approximating an infinite tree with arbitrary accuracy, and in D_{inf2} due to a cyclic constraint.[23]

For a more realistic example consider the following procedure that reads an infinite stream of variables (constructed with the features *head* and *tail*, and the label *nil*) and then executes either S_1 or S_2, depending on whether the variable is labelled with a or b:

$$\mathit{scan}(xs) \leftarrow xs[\mathit{head}]x \wedge xs[\mathit{tail}]xr \ \texttt{then} \ \mathit{process}(x),\ \mathit{scan}(xr) \qquad (D_{\mathit{scan}})$$

$$\mathit{process}(x_1) \leftarrow a\langle x_1 \rangle \ \texttt{then} \ S_1$$

$$\mathit{process}(x_2) \leftarrow b\langle x_2 \rangle \ \texttt{then} \ S_2$$

The associated constraint is this one:

$$\mathit{scan} \subseteq xs \ \wedge \ xs[\mathit{head}]x \ \wedge xs[\mathit{tail}]xr \ \wedge \ x \subseteq \mathit{process} \ \wedge \ xr \subseteq \mathit{scan} \ \wedge$$

$$\mathit{process} \subseteq x_1 \cup x_2 \ \wedge \ a\langle x_1 \rangle \ \wedge \ b\langle x_2 \rangle$$

In the context of D_{scan} both of the following clauses are erroneous.

$$p(u) \leftarrow u[\mathit{tail}]w \wedge w[\mathit{head}]w' \wedge c\langle w' \rangle,\ \mathit{scan}(u) \qquad (D_{\mathit{scan1}})$$

$$q(v) \leftarrow v[\mathit{tail}]w \wedge \mathit{nil}\langle w' \rangle,\ \mathit{scan}(v) \qquad (D_{\mathit{scan2}})$$

An application $p(y)$ (wrt. D_{scan1}) will fail since in procedure p a list is constructed and passed to *scan* which contains an element that is not labelled with a or b. An application $q(y)$ (wrt. D_{scan2}) will fail since the argument passed to *scan* in procedure q cannot be an infinite list.

The analysis of D_{scan1} contains the following constraint and, in conjunction with the analysis of D_{scan}, entails $p = \emptyset$.

$$p \subseteq u \ \wedge \ u[\mathit{tail}]w \ \wedge \ v[\mathit{head}]w' \ \wedge \ c\langle w' \rangle \ \wedge \ u \subseteq \mathit{scan} \ \wedge \ldots$$

Similarly, the analysis of D_{scan2} contains

$$q \subseteq v \ \wedge \ v[\mathit{tail}]w'' \ \wedge \ \mathit{nil}\langle w'' \rangle \ \wedge \ v \subseteq \mathit{scan} \ \wedge \ldots$$

and entails $q = \emptyset$ in conjunction with the analysis of D_{scan}.

[23] As another remark to Footnote 21 on Page 92, notice that an analysis of Prolog II approximating the least model over infinite trees would probably have to distinguish the programs D_{inf1} and D_{inf2}.

Programs	D	$\subseteq$	set of all clauses $p(x) \leftarrow S$	
Statements	S	$::=$	η	(Tell Statement)
		$\mid$	$p(x)$	(Application)
		$\mid$	$S_1,\ S_2$	(Parallel Composition)
		$\mid$	skip	(Null Statement)
Constraints	η	$::=$	$x{=}y \quad \mid \quad a\langle x\rangle \quad \mid \quad x[f]y \quad \mid \quad x\{\overline{f}\} \quad \mid \quad \eta_1 \wedge \eta_2$	
Configurations	C	$::=$	$V D \eta\, [\,]\ S$	
Variables	V	$\subseteq$	$\mathcal{V}$ (V a finite set)	

Figure 4.1.: Syntax of CLP(CFT): Constraint Logic Programming over Feature Trees

4.1.1.4. Procedure Clause and Program Points

The procedure p in the program D_{choice} below is not finitely failed. Accordingly, the analysis of D_{choice} does not entail $q{=}\emptyset$ so that we accept the program.

$$
\begin{aligned}
&p(x) \leftarrow \mathsf{skip} \\
&p(x) \leftarrow a\langle x\rangle,\ b\langle x\rangle
\end{aligned}
\qquad (D_{choice})
$$

Yet, whenever the second clause of p will be executed, failure will inevitably occur. In order to detect this, we can introduce a new procedure for each clause of p, yielding

$$
\begin{aligned}
&p(x) \leftarrow p'(x) \qquad && p'(x) \leftarrow \mathsf{skip} \\
&p(x) \leftarrow p''(x) \qquad && p''(x) \leftarrow a\langle x\rangle,\ b\langle x\rangle
\end{aligned}
\qquad (D'_{choice})
$$

In this program, p'' is finitely failed, corresponding to the fact that *the second clause* of p in D_{choice} is finitely failed. Accordingly, the analysis of D'_{choice} entails $p''{=}\emptyset$.

4.1.2. Constraint Logic Programming over Feature Trees

We assume a set $\mathcal{V}$ of *variables* ranged over by x, y, z, and an alphabet $\mathcal{P}$ of constants, ranged over by p, q, which we call *procedure names*.

$$S_1,\, S_2 \;\equiv\; S_2,\, S_1 \qquad (S_1,\, S_2),\, S_3 \;\equiv\; S_2,\, (S_2,\, S_3) \qquad \mathsf{skip},\, S \equiv S$$

$$\text{consistent renaming of bound variables } (\alpha) \qquad \frac{V = V' \quad \eta \models\!\!\!\mid \eta' \quad S \equiv S'}{V D \eta\, [\,]\; S \;\equiv\; V' D \eta'\, [\,]\; S'}$$

Figure 4.2.: Structural Congruence of CLP(CFT)

4.1.2.1. Syntax

We define a simple concurrent constraint language over feature trees with unguarded clauses, which we consider as a concurrent constraint language without guards.[24] Its operational semantics is defined by a transition system that corresponds to the standard one for a constraint logic programming language. Therefore, we call this language CLP(CFT). We shall also borrow the logic semantics of CLP for this language, and make use of standard concepts from the CLP literature [105, 118].

The abstract syntax of CLP(CFT) is given in Figure 4.1. A program D consists of a set of *clauses* $p(x) \leftarrow S$ where x is called the *formal argument*, and S the clause *body*.[25] Every clause body consists of a sets of *constraints* η and *procedure applications* $p(x)$; in $p(x)$, the variable x is called the *actual argument* of p. As constraint system we fix CFT. We consider only unary procedures for ease of notation. This does not restrict the expressiveness of the language if the constraints can express pairing: for CFT this is the case if we assume at least two distinct features.

The formal argument x of a clause $p(x) \leftarrow S$ is bound with scope S. All other variables occurring in S are implicitly bound within S by an existential quantifier. The variables *free in a statement* S or a constraint are denoted by $\mathit{fv}(S)$ and $\mathit{fv}(\eta)$. A clause $p(x) \leftarrow S$ does not contain any free variable.

Given a program of the form $D = p_1(x_1) \leftarrow S_1, \ldots, p_n(x_n) \leftarrow S_n$, we denote with $\mathcal{P}_D$ the set $\{p_1, \ldots p_n\}$ of procedure names defined in D. We define the *definition of p in D*, written $\mathit{Def}(p,D)$ as the set of all clauses $p(x) \leftarrow S$ in D. A *configuration* $V D \eta\, [\,]\; S$ consists of a statement S, a *constraint store* η, a collection of procedure definitions D, and a set V of variables such that $\mathit{fv}(S) \cup \mathit{fv}(\eta) \subseteq V$ holds. Configurations describe computation states. A configuration $V D \eta\, [\,]\; S$ is *well-formed* if for every application $p(x)$ in D or S there is a corresponding procedure $p \in \mathcal{P}_D$. Throughout this chapter, we use the term "configuration" to mean "well-formed configuration".

We identify statements S, definitions D, and configurations C up to consistent renaming of bound variables and we assume once and for all that bound variables in any S, D,

[24] We add guards in Section 4.2 and obtain a language that is essentially the committed choice language ALPS considered by Maher in [120].

[25] The symbol D should allude to "definition".

$$V D\eta \,[]\, \eta' \;\longrightarrow\; V D\, \eta \wedge \eta' \,[]\, \mathsf{skip} \qquad (\textsc{Tell})$$

$$V D\eta \,[]\, p(x) \;\longrightarrow\; V \cup fv(S) \setminus \{y\}\, D\eta \,[]\, S[x/y] \qquad (\textsc{Apply})$$
$$\text{if } p(y) \leftarrow S \in D \text{ and } V \cap fv(S) \subseteq \{y\}$$

$$\frac{V D\eta \,[]\, S_1 \;\longrightarrow\; V' D' \eta' \,[]\, S_2}{V D\eta \,[]\, S_1,\, S \;\longrightarrow\; V' D' \eta' \,[]\, S_2,\, S} \qquad (\textsc{Closure})$$

Figure 4.3.: Operational Semantics of CLP(CFT)

or C are pairwise distinct and distinct from the free variables. Furthermore, we identify S, D, and C up to the smallest *structural congruence* $\equiv$ that satisfies the equations given in Figure 4.2; structural congruence makes parallel composition of statements commutative and associative with neutral element skip; two definitions are congruent if they are identical up to consistent renaming of variables (α-renaming), and two configurations are congruent if all their components are.

4.1.2.2. Operational Semantics

The operational semantics is given in terms of a one-step reduction relation on configurations C. *Reduction* $\longrightarrow$ is defined in Figure 4.3 as the smallest binary relation on configurations that satisfies the axioms (Tell) and (Apply), and that is closed under the inference rule (Closure).

Tell. A tell statement η reduces without synchronisation by conjoining η to the current store. For technical reasons, we allow the constraint store to become non-satisfiable.

Apply. If the procedure p is defined in D, then reduction of an application $p(x)$ picks one of the clauses $p(y) \leftarrow S$ in D nondeterministically, and replaces $p(x)$ by $S[y/x]$, that is by the clause body S with the actual argument y replacing the formal one x. The second side condition requires that the local variables in S are fresh for the current configuration before they are added to the set of used variables. Hence the bound variables may need renaming before reduction can take place.

Closure. In combination with the structural congruence, the closure rule states that the next reduction step for $V D\eta \,[]\, S$ can indeterministically deal with any application or tell statement in S.

For illustration of rules (TELL) and (APPLY), let $D = p(x) \leftarrow x[f]y \wedge a\langle y\rangle$ and consider:

$$\begin{array}{rcl} \{z\}D\top\,[]\; p(z),\, b\langle z\rangle & \longrightarrow & \{y,z\}D\top\,[]\; z[f]y \wedge a\langle y\rangle,\, b\langle z\rangle \\ & \longrightarrow & \{y,z\}D\; z[f]y \wedge a\langle y\rangle \wedge b\langle z\rangle\,[]\;\mathsf{skip} \end{array}$$

The final configuration in this example contains a non-satisfiable constraint store. Such a configuration is called *failed* which is considered a run-time error. This is in contrast to constraint logic programming where failure is an integral part of the search control mechanism (backtracking).

If an application $p(x)$ has reduced with respect to one of multiple clauses of p, then this choice is never undone. One says that our language has *committed choice semantics*. Notice that a committed choice without guarded clauses is not overly useful in practice. For example, it is not clear how to define the length predicate on lists such that it would terminate on every finite list. As mentioned initially, we are interested in CLP(CFT) programs as *relaxations* of concurrent constraint programs (see Section 4.2).

4.1.2.3. Computations and Finite Failure

A *computation* is a maximal (and possibly infinite) sequence of configurations $(C_i)_{i=0}^n$, $n \leq \infty$, such that there exist V, D, η, and S with

$$C_1 = \quad VD\eta\,[]\; S \quad \text{and} \quad \forall i < n : C_i \longrightarrow C_{i+1}\,.$$

A computation $(C_i)_{i=0}^n$, is called *finitely failed* [105, 118] if there exists a (finite) $n' < n$ such that the constraint store in configuration $C_{n'}$ is inconsistent. A computation $(C_i)_{i=0}^n$ is called *fair* if every statement that *can* be reduced in some configuration C_i, $i < n$, *is* eventually reduced. The *finite failure set FF_D* of a given program D is defined as follows:

$$FF_D =_{def} \quad \{p \mid \{x\}\; D\top\,[]\; p(x) \text{ is finitely failed}, p \in \mathcal{P}_D, x \text{ arbitrary}\}$$

We say that a procedure $p \in \mathcal{P}_D$ *is finitely failed* if $p \in FF_D$.[26]

4.1.3. Set-based Failure Diagnosis

We formulate our analysis in terms of set constraints over feature trees as defined in Chapter 2. We also use the notation $x{=}\emptyset \leftrightarrow y{=}\emptyset$ as an abbreviation for a corresponding set constraint as defined on Page 81.

The analysis is defined in Figure 4.4 as a mapping A from CLP(CFT) programs D to existential formulas over set constraints. With every program variable x we associate a

[26] Our definition of FF_D deviates slightly from the standard literature [105], where the finite failure set is a set of "constrained atoms" $p(x)\leftarrow\eta$ rather than a set of predicates: $FF_D = \{p(x)\leftarrow\eta \mid fv(\eta) \cup \{x\}\; D\,\eta\,[]\; p(x)$ is finitely failed$\}$. We use a more coarse-grained definition that suffices for our purpose and simplifies notation.

$$
\begin{array}{lcl}
A(D) & = & \bigwedge_{p\in\mathcal{P}_D} A(Def(p,D)) \\
A(Def(p,D)) & = & \exists x_1 \ldots \exists x_n (p \subseteq x_1 \cup \ldots \cup x_n \ \wedge\ \bigwedge_{i=1}^{n} A^{x_i}(S_i)) \\
 & & \qquad \text{if } Def(p,D) = p(x_1) \leftarrow S_1,\ \ldots,\ p(x_n) \leftarrow S_n \\
 & & \qquad \text{and } x_1,\ldots,x_{n_p} \text{ pairwise distinct} \\
A^x(S) & = & \exists \overline{y}(A(S)\ \wedge \bigwedge_{y\in\{\overline{y}\}} x{=}\emptyset \leftrightarrow y{=}\emptyset) \qquad \text{if } \{\overline{y}\} = fv(S)\backslash\{x\} \\
A(q(x)) & = & x \subseteq q \\
A(\eta) & = & \eta \\
A(S_1,\ S_2) & = & A(S_1) \wedge A(S_2)
\end{array}
$$

Figure 4.4.: Set-based Failure Diagnosis for Constraint Logic Programs over Feature Trees

fresh constraint variable, and we write this constraint variable also as x. This simplifies notation and eases reading. We also use procedure names as constraint variables. The analysis interprets parallel composition "$,$ " as conjunction.

Procedures. The analysis of a procedure p defined by n_p clauses $p(x_i) \leftarrow S_i$, $1 \le i \le n_p$, considers all clause bodies S_i separately; for each S_i, a fresh variable x_i is introduced with respect to which S_i is analysed: if $A^{x_i}(S_i)$ entails $x_i = \emptyset$, then every call to this clause will be finitely failed. The constraint $p \subseteq x_1 \cup \ldots \cup x_n$ states that all possible arguments for p must be possible arguments for one of the clauses. If all the x_i are constrained to the empty set, then so is p. In other words, if all clauses of p are finitely failed, then procedure p is finitely failed.[27]

Clause. The analysis of a statement S with respect to a variable x makes the existential quantification of the variables in $fv(S)$ explicit, and then states that all these variables should denote the empty set if and only if x does:

$$\bigwedge_{y\in\{\overline{y}\}} x{=}\emptyset \leftrightarrow y{=}\emptyset$$

[27] The existential quantifiers reflect the variable scope. This is technically convenient and it avoids the need to supplement constraint simplification by a reachability analysis as for example in [173]. For the purpose of failure diagnosis alone, we could as well introduce a fresh variable wherever an existential quantifier occurs.

Essential for our purpose is the direction from right to left: if any of the $y \in fv(S)$ denotes the empty set and hence indicates a failure in statement S, then x should be forced to denote the empty set, too.

Application. The analysis of an application $q(x)$ states that the actual argument x must be valid for the procedure q. In other words, the possible values for x are bounded by the set of possible values that the formal argument of the procedure q can take without failing.

Constraints. The analysis of a constraint η is just η itself. This exploits the fact that we identify every CFT constraint with a set constraint over feature trees where $x{=}y$ is replaced by $x{\subseteq}y \wedge y{\subseteq}x$.

Our analysis reflects the fact that constraints on *all* program variables may be the reason for finite failure of a procedure, not only the formal parameters of procedures. For illustration consider the following example.

Example 24 (Analysis of Failure on Local Variables)
Consider the procedure definition

$$p(x) \leftarrow a\langle y\rangle,\ b\langle y\rangle \qquad (D_{locfail})$$

with its associated analysis (slightly simplified):

$$p{\subseteq}x \ \wedge\ \exists y(y{=}\emptyset{\leftrightarrow}p{=}\emptyset \wedge a\langle y\rangle \wedge b\langle y\rangle)$$

If $a \neq b$, then every reduction of the body $a\langle x\rangle,\ b\langle x\rangle$ will lead to failure; that is, every computation of $p(x)$ is finitely failed and thus $p \in FF_{D_{locfail}}$. The analysis detects this because $a\langle y\rangle \wedge b\langle y\rangle$ entails $y{=}\emptyset$, which, in combination with $y{=}\emptyset{\leftrightarrow}p{=}\emptyset$, entails $p{=}\emptyset$.

Example 25 (Binary Trees)
In the following program the procedure r is finitely failed.

$$\begin{array}{ll} p(x_1) \leftarrow x_1{=}a(1{:}b, 2{:}b) & q(y) \leftarrow y{=}a(1{:}b, 2{:}c) \\ p(x_2) \leftarrow x_2{=}a(1{:}c, 2{:}c) & r(z_r) \leftarrow p(z_r),\ q(z_r) \end{array} \qquad (D_{comp})$$

Finite failure of r is detected through the analysis which entails $r{=}\emptyset$:

$$r{\subseteq}z_r \ \wedge\ z_r{\subseteq}a(1{:}b, 2{:}b) \cup a(1{:}c, 2{:}c) \ \wedge\ z_r{\subseteq}a(1{:}b, 2{:}c)$$

4.1.4. Correctness

Our failure diagnosis is correct in the sense that whenever it entails emptiness of some procedure then this procedure is finitely failed.

Theorem 26 (Detection of Finite Failure)
For all CLP(CFT) programs D and all p: If $A(D) \models_{\mathcal{P}(\mathcal{FT})} p{=}\emptyset$*, then* $p \in FF_D$*.*

(Notice that a implied statement says that whenever the analysis of a CLP(CFT) program is non-satisfiable over non-empty sets of feature trees, then the finite failure set of D is non-empty: If $A(D) \models_{\mathcal{P}^{+}(\mathcal{FT})} \perp$ then $FF_D \neq \emptyset$. *Cf.* also Proposition 2.4.)
In the proof of Theorem 26 we exploit that CLP programs have a logic semantics that is closely related to their operational semantics. The logic semantics is defined in Section 4.1.4.1. The proof relies on two insights:

1. Emptiness in the greatest model $gm(D)$ of CLP(CFT) programs indeed implies finite failure: this is stated in Theorem 30 and proven in Section 4.1.4.3.

2. The greatest solution $gsol(A(D))$ of the analysis $A(D)$ is an upper approximation of the greatest model of D: this is shown as Theorem 31 in Section 4.1.4.4.

Both theorems rely on the saturation property [159] of constraint systems that is introduced in Section 4.1.4.2. Now the proof of Theorem 26 is as follows.

Proof. If $A(D) \models_{\mathcal{P}(\mathcal{FT})} p{=}\emptyset$, then $gsol(A(D))(p) = \emptyset$. Hence, by Theorem 31, it holds that $gm(D)(p) = \emptyset$, and this implies $p \in FF_D$ by Theorem 30. □

Finally notice that a corollary of Theorem 26 is the analogous statement for CLP(CFT) over *finite* feature trees: Every finitely failed computation of a CLP program over infinite trees also is a finitely failed computation over finite trees.

4.1.4.1. Logic Semantics and Consequence Operator

Every program D is associated a *logic semantics* given by a first-order formula $compl(D)$ over CFT constraints. If the definition of procedure p in a program D is

$$Def(p,D) = \quad p(x) \leftarrow S_1,\ \ldots,\ p(x) \leftarrow S_n$$

and $\{\bar{y}_i\} = fv(S_i) \setminus \{x\}$ for all i, $1 \leq i \leq n$, then $compl(p)$ is defined as the predicate

$$compl(p) =_{def} \quad \forall x\, p(x) \leftrightarrow (\exists \bar{y}_1 S_1 \ \vee\ \ldots\ \vee\ \exists \bar{y}_n S_n)$$

where parallel composition "," is interpreted as conjunction. The conjunction of these formulas for all procedures p in D is called Clark's completion [46]:

$$compl(D) =_{def} \quad \bigwedge_{p \in \mathcal{P}_D} compl(p)$$

Let D be a program. A *D-interpretation* I is a function from $\mathcal{P}_D$ to subsets of $\mathcal{FT}$ and it induces an extension $I(\mathcal{FT})$ of the structure $\mathcal{FT}$ in which every $p \in \mathcal{P}_D$ is interpreted as the predicate $I(p)$. Interpretations are ordered by pointwise set inclusion. We denote as $\mathcal{B}_D$ the greatest D-interpretation, which maps all $p \in \mathcal{P}_D$ to $\mathcal{FT}$;[28] A D-interpretation I is a *model* of D if *compl*(D) is valid in the structure $I(\mathcal{FT})$. In this case we write $I \models D$ (instead of $I(\mathcal{FT}) \models compl(D)$). Likewise, we briefly write $I, \alpha \models S$ if *compl*(S) is valid in I under a valuation α. The greatest model of a program D always exists and is denoted by *gm*(D). We write $D \models \Phi$ if the formula Φ is valid in every model of D.

The *consequence operator* $T_D : (\mathcal{P}_D \to \mathcal{P}(\mathcal{FT})) \to (\mathcal{P}_D \to \mathcal{P}(\mathcal{FT}))$ is defined as follows, for all D-interpretations I and all $p \in \mathcal{P}_D$:

$$T_D(I)(p) \quad =_{def} \quad \left\{ \tau \in \mathcal{FT} \;\middle|\; I, [\tau/x] \models \exists \bar{y} S, \;\text{ if } \begin{array}{l} p(x) \leftarrow S \in D, \text{ and} \\ \{\bar{y}\} = fv(S) \setminus \{x\} \end{array} \right\}$$

Here, $[\tau/x]$ denotes the $\mathcal{FT}$-valuation that maps x to τ and all other variables to itself. The ω-times iterated application of T_D to $\mathcal{B}_D$ is written $T_D{\downarrow}^{\omega}$.

$$T_D{\downarrow}^{\omega} \quad =_{def} \quad \bigcap_{n=1}^{\omega} {T_D}^n(\mathcal{B}_D)$$

Since T_D is a monotonic operator on the complete lattice $(\mathcal{P}(\mathcal{FT}), \subseteq)$, the Knaster-Tarski fixed point theorem guarantees the greatest fixed point *gfp*(T_D) of T_D to exist and to coincide with the greatest postfixed point of T_D (*i. e.*, the greatest I with $I \subseteq T_D(I)$).

4.1.4.2. The Saturation Property

A constraint logic program is called *canonical* [106] if the greatest fixed-point of its consequence operator can be obtained by at most ω iterations from $\mathcal{B}_D$, that is, if $gfp(T_D) = T_D{\downarrow}^{\omega}$. In general, only inclusion $gfp(T_D) \subseteq T_D{\downarrow}^{\omega}$ holds. Palmgren shows that every constraint logic program over a constraint system X is canonical if X has the saturation property ([159, Theorem 3.11], see also Theorem 29). A constraint system has the *saturation property*[29] if for all infinite number of constraints $\eta_1, \eta_2, \ldots$

$$\bigwedge_{i=1}^{\infty} \eta_i \text{ is satisfiable} \qquad \text{if and only if} \qquad \forall n < \infty : \bigwedge_{i=1}^{n} \eta_i \text{ is satisfiable}.$$

Proposition 4.1

The constraint systems CFT and $FT_{\subseteq}(ar, \cup)$ have the saturation property.

[28] The notation $\mathcal{B}_D$ alludes to (Herbrand) Base [118].

[29] The saturation property has nothing to do with the the notion of saturation used in Section 3.1.

Proof. The claim for CFT follows from the fact that the set of $\mathcal{FT}$-valuations is a compact metric space (with the order on valuations defined pointwise and in analogy to the case of constructor trees [118]). The claim for $\mathrm{FT}_{\subseteq}(ar, \cup)$ is trivial, since every (finite or infinite) selector set constraint is satisfiable (Proposition 2.1). □

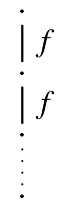
This proposition specifically holds for infinite trees. The situation is different for some other popular constraint systems: For instance, equational constraints over finite or rational constructor trees do not have the saturation property.

Example 27 (Saturation fails for Finite Trees)

Fix infinitely many $x_1, x_2, \ldots$ distinct variables and an arbitrary feature f, and define, for all i, the constraint η_i by

$$\eta_i =_{def} \quad x_i[f]x_{i+1}$$

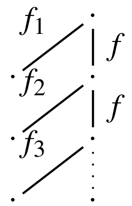
Then every finite conjunction $\bigwedge_{i=1}^{n} \eta_i$ is satisfiable over finite feature trees, while the infinite conjunction $\bigwedge_{i=1}^{\infty} \eta_i$ is not.

$x_1 =$ 

Example 28 (Saturation fails for Rational Trees)

Fix infinitely many distinct variables $y_1, y_1', y_2, y_2' \ldots$, a feature f, and infinitely many distinct features $f_1, f_2, \ldots$ different from f. Define, for all i, the constraint η_i by

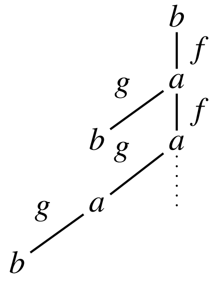
$$\eta_i =_{def} \quad y_i[f]y_{i+1} \wedge y_i[f_i]y_i'$$

$y_1 =$ 

Then every finite conjunction $\bigwedge_{i=1}^{n} \eta_i$ is satisfiable over rational feature trees, while the infinite conjunction $\bigwedge_{i=1}^{\infty} \eta_i$ is not. Note that every solution of $\bigwedge_{i=1}^{\infty} \eta_i$ must assign to y_1 an infinite tree which contains, for each of the features f_i, a subtree with feature f_i at its root.

The failure of saturation does not depend on the availability of infinitely many features. For example, given features $g \neq f$, labels $a \neq b$, and distinct variables $z_i = z_i^1, \ldots, z_i^{i-1}$ for all i, we can also define:

$$\eta_i =_{def} \quad z_i[f]z_{i+1} \wedge z_i[g]z_i^1 \wedge a\langle z_i^1 \rangle \wedge \ldots z_i[g]z_i^i \wedge b\langle z_i^i \rangle$$

$z_1 =$ 

Theorem 29 (Palmgren: Saturation and Canonicity)

Every program D over a constraint system with the saturation property is canonical.

Proof. See Palmgren [159]. □

Lemma 4.2

For all CLP(CFT) programs D: $gm(D) = T_D{\downarrow}^{\omega}$.

Proof. Proposition 4.1 and Palmgren's Theorem 29 imply that $gfp(T_D) = T_D{\downarrow}^{\omega}$. This implies the claim in combination with the fact that $gfp(T_D) = gm(D)$. □

$$V D\eta\,[]\; S \;\longrightarrow_g\; V D\,\tilde{\eta}\,[]\; S \quad \text{if} \begin{cases} \eta \text{ satisfiable and} \\ \tilde{\eta} \text{ a grounding of } \eta \text{ w.r.t. } V \end{cases} \qquad (\textsc{Ground})$$

Figure 4.5.: Ground Reduction of CLP(CFT)

4.1.4.3. Characterising Finite Failure

We show that, for CLP(CFT) programs, emptiness in the greatest solution is equivalent to finite failure in the following sense.

Theorem 30 (Greatest Models and Finite Failure)
For all CLP(CFT) programs D and all $p \in \mathcal{P}_D$: $gm(D)(p) = \emptyset$ *if and only if* $p \in FF_D$.

Proof. $gm(D)(p) = \emptyset$ is equivalent to $D \models \neg\exists x\, p(x)$ and hence to $\forall\tau \in \mathcal{FT} : D \models \neg p(\tau)$. By Proposition 4.3 this is equivalent to $\forall\tau \in \mathcal{FT} : p(\tau) \in GFF_D$ and, by Proposition 4.4, to $p \in FF_D$. □

For the proof of the necessary Propositions 4.3 and 4.4 we need some additional machinery first. Let $\tilde{\eta}$ range over possibly infinite conjunctions of CFT constraints with existential quantifiers, and note that every feature tree τ can be characterised by a formula $\tilde{\eta}$ (in the sense that $fv(\tilde{\eta}) = \{x\}$ and every solution of $\tilde{\eta}$ maps x into τ). We call $\tilde{\eta}$ *ground w.r.t. V* if all solutions of $\tilde{\eta}$ coincide on all $x \in V$. A configuration $VD\eta\,[]\; S$ is called *ground* if η is ground w.r.t. V. We call a constraint $\tilde{\eta}$ a *grounding* of η w.r.t. V if $\tilde{\eta}$ is ground w.r.t. V, entails η and has the same free variables as η.

We now generalise the notion of configuration slightly by allowing $\tilde{\eta}$ as a constraint store. A *ground computation* [105, 118] is a maximal sequence $(C_i)_{i=0}^n$, $n \le \infty$, such that there exist V, D, η, and S where η is ground,

$$C_1 = \quad VD\eta\,[]\; S, \qquad \text{and} \qquad \forall i < n : C_i \longrightarrow \circ \longrightarrow_g C_{i+1}\,.$$

A configuration C is called *[ground] finitely failed* if all fair [ground] computations issuing from C are. The *ground finite failure set* GFF_D of a program D is defined as follows

$$GFF_D =_{def} \quad \left\{ p(\tau) \;\middle|\; \begin{array}{l} \{x\}D\eta\,[]\;\, p(x) \text{ is ground finitely failed,} \\ \alpha \models \eta \text{ implies } \alpha(x) = \tau,\; p \in \mathcal{P}_D, \end{array} \right\}$$

Figure 4.5 defines a *grounding* relation between configurations C_1 and C_2 that holds if C_1 and C_2 are the same except for the constraint store, where that of C_2 is a grounding of that of C_1. *Ground reduction* is defined as the composition $\longrightarrow \circ \longrightarrow_g$ of reduction with grounding.

Ground reduction is interesting for us due to its close relation to the logic semantics of programs. By a classical result from (constraint) logic programming, the following holds (see, *e. g.*, [105, Theorem 6.1 (7)]).

$$GFF_D = \mathcal{B}_D \setminus T_D{\downarrow}^{\omega} \tag{4.1}$$

In combination with canonicity, we obtain the following proposition.

Proposition 4.3
For all CLP(CFT) programs D, $p \in \mathcal{P}_D$, and $\tau \in \mathcal{FT}$: $D \models \neg p(\tau)$ if and only if $p(\tau) \in GFF_D$.

Proof. $D \models \neg p(\tau)$ if and only if $\tau \notin gm(D)(p)$. This is equivalent to $\tau \notin T_D{\downarrow}^{\omega}(p)$ by Lemma 4.2 which is clearly equivalent to $\tau \in (\mathcal{B}_D \setminus T_D{\downarrow}^{\omega})(p)$. By Equation (4.1), however, this holds if and only if $p(\tau) \in GFF_D$. □

The next proposition states that the finite failure set and the ground finite failure set for CLP(CFT) programs coincide. While this is not a difficult result we have not been able to find it in the literature.

Proposition 4.4 (*FF* and *GFF* coincide for *CLP(CFT)*)
For all $p \in \mathcal{P}_D$: $p \in FF_D$ if and only if $\forall \tau \in \mathcal{FT} : p(\tau) \in GFF_D$.

Proof. The implication from left to right holds, since every finitely failed computation of $\{x\}D\eta\,[]\ p(x)$ induces a finitely failed computation of $\{x\}D\tilde{\eta}\,[]\ p(x)$ if $\tilde{\eta}$ is a grounding of η w.r.t. $\{x\}$. For the converse, assume $p \notin FF_D$ and let $(C_i)_{i=1}^{\infty}$ be a fair computation with $C_1 = \{x\}D\top\,[]\ p(x)$ that is not finitely failed. (The case of finite computations is simpler.) Let, for all i, η_i be the constraint store of C_i. Since the computation is unfailed, η_i is satisfiable for all *finite* $i \leq n$. Hence, $\bigwedge_{i=0}^{\infty} \eta_i$ is satisfiable by Proposition 4.1. If $V = \bigcup_{i=0}^{\infty} fv(\eta_i)$, then there exists a grounding of $\bigwedge_{i=0}^{\infty} \eta_i$ w.r.t. V; hence there exist groundings $\tilde{\eta}_i$ of all η_i w.r.t. $fv(\eta_i)$. From these we can easily construct an infinite fair and unfailed ground computation for $\{x\}D\,\tilde{\eta}_1\,[]\ p(x)$. Hence $p(\tau) \notin GFF_D$.[30] □

[30] An alternative proof can be based on Jaffar and Stuckey's result [107] which says that, for programs D over infinite trees, $T_D{\downarrow}^{\omega}$ equals $\mathcal{B}_D \setminus [FF_D]$ where $[FF_D]$ is the set of *ground instances* of FF_D

$$[FF_D] = \left\{ p(\tau) \;\middle|\; \begin{array}{l} \{x\} \cup fv(\eta) D\eta\,[]\ p(x) \text{ is finitely failed, and} \\ \text{there is a solution } \alpha \text{ of } \eta \text{ with } \alpha(x) = \tau \end{array} \right\}$$

By standard results from constraint (logic) programming, this result implies that $[FF_D] = GFF_D$. Thus, it remains to show that $[FF_D](p) = \mathcal{FT}$ if and only if $p \in FF_D$. To prove the non-trivial direction from left to right we need an argument based on saturation similar to the one that we used above.

4.1.4.4. Approximating the Greatest Model

We show that the greatest model of an arbitrary CLP(CFT) program is approximated by the greatest solution of the associated analysis.

Theorem 31 (Approximating the Greatest Model)
For all CLP(CFT) programs D: $gm(D) \subseteq gsol(A(D))$.

Proof. We apply a technique which is well-known in abstract interpretation [53]. We associate to every program D over CFT an abstract program $D^\#$ over $\mathrm{FT}_\subseteq(ar,\cup)$ and prove that the consequence operator $T_{D^\#}$ approximates T_D. Let the function sup map a $D^\#$-model to a D-model that maps every $p \in \mathcal{P}_D$ to the maximal element in $gm(D^\#)(p)$. Then the fact that $T_{D^\#}$ approximates T_D implies

$$gm(D) \subseteq \sup \circ gm(D^\#)$$

(Proposition 4.10). Then we characterise our analysis through $gm(D^\#)$ (Corrollary 32)

$$\sup \circ gm(D^\#) = gsol(A(D))$$

and conclude $gm(D) \subseteq gsol(A(D))$.[31] □

The rest of this section is devoted to the completion of this proof. For simplicity, we shall assume throughout this section that every procedure is defined by exactly two clauses

$$Def(p,D) =_{def} \quad p(x) \leftarrow S_1,\ p(x) \leftarrow S_2$$

$$\text{where } \{\bar{y}_1\} = fv(S_1)\backslash\{x\} \text{ and } \{\bar{y}_2\} = fv(S_2)\backslash\{x\}$$

Generalisation to the n-ary case is straightforward. We abstract the multiple clause definition $Def(p,D)$ by a single clause $Def(p,D)^\#$ which is defined as follows:

$$Def(p,D)^\# =_{def} \quad p(x) \leftarrow B_p$$

$$B_p =_{def} \quad \left(\begin{array}{l} x \subseteq x_1 \cup x_2 \wedge \\ \bigwedge_{y \in \{\bar{y}_1\}} x_1 = \emptyset \leftrightarrow y = \emptyset \ \wedge\ S_1[x_1/x] \wedge \\ \bigwedge_{y \in \{\bar{y}_2\}} x_2 = \emptyset \leftrightarrow y = \emptyset \ \wedge\ S_2[x_2/x] \end{array} \right)$$

The *abstract program* $D^\#$ associated with a program D is given by the set of $Def(p,D)^\#$ for all $p \in \mathcal{P}_D$. The associated operator $T_{D^\#}$ is defined like T_D with $\mathcal{B}_D$ replaced by $\mathcal{B}_{D^\#} = \mathcal{P}(\mathcal{P}(\mathcal{FT}))$.

Lemma 4.5
For all CLP($FT_\subseteq(ar,\cup)$) programs D: $gm(D^\#) = T_{D^\#}{\downarrow}^\omega$

[31] This deviates from the proof given in [171] which shows the result directly.

Proof. From Proposition 4.1 and Palmgren's Theorem 29 we obtain $gfp(D^{\#}) = T_{D^{\#}}{\downarrow}^{\omega}$. The claim follows from $gm(D^{\#}) = gfp(T_{D^{\#}})$. □

The abstraction of an $\mathcal{FT}$-valuation α to a $\mathcal{P}(\mathcal{FT})$-valuation $\alpha^{\#}$ is defined by $\alpha^{\#}(x) = \{\alpha(x)\}$, and the abstraction of a D-interpretation I to a $D^{\#}$-interpretation $I^{\#}$ is defined by $I^{\#}(p) = \mathcal{P}(I(p))$.

Lemma 4.6

For all $\mathcal{FT}$-valuations α, all $p \in \mathcal{P}_D$, and all interpretations I:

1. *If $\alpha \models_{\mathcal{FT}} \eta$ then $\alpha^{\#} \models_{\mathcal{P}(\mathcal{FT})} \eta$.*
2. *If $\alpha(x) \in I(p)$ then $\alpha^{\#}(x) \in I^{\#}(p)$.*

Proof. The first claim is proven by a simple check of all primitive constraints. The second is obvious by definition of $I^{\#}$. □

Propositions 4.7 and 4.8 establish two essential properties of the abstract program $D^{\#}$. In combination they show that every postfixed point of T_D induces one of $T_{D^{\#}}$ (Lemma 4.9).

Proposition 4.7 (Singleton Property)

For all D, I, and $p \in \mathcal{P}_D$: If $I \subseteq T_D(I)$ and $\tau \in I(p)$ then $\{\tau\} \in T_{D^{\#}}(I^{\#})$.

Proof. Let $p \in \mathcal{P}_D$ and $\tau \in I(p)$. Assume $I \subseteq T_D(I)$. Without loss of generality we assume that

$$I[\tau/x] \models_{\mathcal{FT}} \quad \exists \overline{y}_1 S_1 \,.$$

Pick $\overline{\tau}$ such that $I[\tau/x][\overline{\tau}/\overline{y}_1] \models_{\mathcal{FT}} S_1$ and define $\alpha = [\tau/x][\overline{\tau}/\overline{y}_1]$. From Lemma 4.6 one easily obtains

$$I^{\#}, \alpha^{\#} \models_{\mathcal{P}(\mathcal{FT})} \quad S_1$$

Furthermore, since $\alpha^{\#}$ maps x and all variables in $\{\overline{y}_1\}$ to a non-empty set, we have

$$I^{\#}, \alpha^{\#} \models_{\mathcal{P}(\mathcal{FT})} \quad \bigwedge_{y \in \{\overline{y}_1\}} x{=}\emptyset \leftrightarrow y{=}\emptyset \wedge S_1$$

Clearly, this implies

$$I^{\#}, [\{\tau\}/x] \models_{\mathcal{P}(\mathcal{FT})} \quad \exists x_1 \exists x_2 \begin{pmatrix} x \subseteq x_1 \cup x_2 \ \wedge \\ \exists \overline{y}_1 \bigwedge_{y \in \{\overline{y}_1\}} x{=}\emptyset \leftrightarrow y{=}\emptyset \wedge S_1[x_1/x] \ \wedge \\ \exists \overline{y}_2 \bigwedge_{y \in \{\overline{y}_2\}} x{=}\emptyset \leftrightarrow y{=}\emptyset \wedge S_2[x_2/x] \end{pmatrix}$$

because we can extend $[\{\tau\}/x]$ by mapping x_1 to $\{\tau\}$, $\bar{y}_1$ to $\alpha^{\#}(\bar{y}_1)$, and all of x_2 and $\bar{y}_2$ to the empty set. But this is just

$$I^{\#}, [\{\tau\}/x] \models_{\mathcal{P}(\mathcal{FT})} \qquad \exists x_1 \exists x_2 \exists \bar{y}_1 \exists \bar{y}_2 B_p$$

which by definition of $Def(p,D)^{\#}$ implies $\{\tau\} \in T_{D^{\#}}(I^{\#})$. □

Proposition 4.8 (Union Property)

For all D and all $D^{\#}$-interpretations $\mathcal{J}$: If $\mathcal{J}$ has a greatest element for all $p \in \mathcal{P}_D$, then $T_{D^{\#}}(\mathcal{J})(p)$ also has a greatest element for all $p \in \mathcal{P}_D$.

Proof. Let $p \in \mathcal{P}_D$ and $\sigma_1, \ldots, \sigma_n \in T_{D^{\#}}(\mathcal{J})(p)$. Let $p(x) \leftarrow S$ be the unique clause for p in $D^{\#}$ and let $\{\bar{y}\} = fv(S) \setminus \{x\}$. Define $\alpha_i(p) = \sigma_i$, and $\alpha_{max}(p) = \bigcup_{i=1}^{n} \sigma_i$. By definition of $T_{D^{\#}}$ this implies that

$$\forall i, 1 \le i \le n: \quad \mathcal{J}, \alpha_i \models \exists \bar{y} S\,.$$

It suffices to show that

$$\mathcal{J}, \alpha_{max} \models \quad \exists \bar{y} S$$

This can be shown by a structural induction over S. For the base case given by constraints η we exploit the fact that solutions of $\mathrm{FT}_{\subseteq}(ar, \cup)$ constraints are closed under unions (Lemma 2.2); for the base case given by applications $p(y)$ we use the assumption that $\mathcal{J}$ has a greatest element and, hence, is closed under union, too. □

Proposition 4.9 (Abstraction Property)

For all D and D-interpretations I: If $I \subseteq T_D(I)$ then $I^{\#} \subseteq T_{D^{\#}}(I^{\#})$.

Proof. Assume $I \subseteq T_D(I)$, and let $p \in \mathcal{P}_p$ and $\sigma \in I^{\#}(p)$. We have to show that $\sigma \in T_{D^{\#}}(I^{\#})(p)$. By definition of $I^{\#}$ we know that $\forall \tau \in \sigma$: $\tau \in I(p)$. By the Singleton Property we know that $\forall \tau \in \sigma : \{\tau\} \in T_{D^{\#}}(I^{\#})$; so $\sigma \in T_{D^{\#}}(I^{\#})$ follows from the Union Property. □

Let the function sup map every $\sigma \in \mathcal{P}(\mathcal{P}(\mathcal{FT}))$ to its greatest element if it exists. Somewhat sloppily, we also use sup as a function that maps a $D^{\#}$-interpretation $\mathcal{J}$ to a D-interpretation $\sup(\mathcal{J}) = I$ where, for all $p \in \mathcal{P}_D$, $I(p) = \sup(\mathcal{J}(p))$.[32]

Proposition 4.10 (Abstraction of Greatest Models)

For all D: $gm(D) \subseteq \sup \circ gm(D^{\#})$.

[32] sup is to counterpart of the abstraction function which, in the abstract interpretation framework, is called the *concretisation* function.

Proof. Since $gm(D) = T_D{\downarrow}^\omega$ by Lemma 4.5, $gm(D)$ is a postfixed point of T_D; *i. e.*, $gm(D) \subseteq T_D(gm(D))$. By Proposition 4.9, this implies $gm(D)^\# \subseteq T_{D^\#}(gm(D)^\#)$, so $gm(D)^\#$ is a postfixed point of $T_{D^\#}$. But since $gm(D^\#) = T_{D^\#}{\downarrow}^\omega$ is the greatest postfixed point of $T_{D^\#}$, we obtain $gm(D)^\# \subseteq gm(D^\#)$. By the Union Property, $gmD^\#$ is closed under unions, and hence $gm(D) \subseteq \sup\circ gm(D^\#)$. □

Now we show that, for all CLP(CFT) programs D, every postfixed point of $T_{D^\#}$ induces a solution induces a solution of $A(D)$ and vice versa. As a consequence, we obtain that the greatest model of $D^\#$ coincides with the greatest solution of $A(D)$ up to sup.

Lemma 4.11 (Postfixed Points and Solutions Coincide)
For all CLP(CFT) programs D, all $D^\#$-interpretations $\mathcal{J}$ and $\mathcal{FT}$-valuations α:

1. *If $\mathcal{J}$ is closed under unions, then $\mathcal{J} \subseteq T_{D^\#}(\mathcal{J})$ implies $\sup\circ\mathcal{J} \models_{\mathcal{P}(\mathcal{FT})} A(D)$.*
2. *If $\alpha \models_{\mathcal{P}(\mathcal{FT})} A(D)$ then $\alpha^\# \subseteq T_{D^\#}(\alpha^\#)$.*

Proof. Fix D, let $p \in \mathcal{P}_D$, and define $\exists\,|_x\, B_p = \exists x_1 \exists x_2 \exists \overline{y}_1 \exists \overline{y}_2 B_p$. Notice that $\exists\,|_x\, B_p$ is of the form

$$\exists x_1 \exists x_2\, (x \subseteq x_1 \cup x_2 \;\wedge\; B_p^1 \;\wedge\; B_p^2)$$

and that $A(Def(p,D))$ is of the form

$$\exists x_1 \exists x_2\, (p \subseteq x_1 \cup x_2 \;\wedge\; A^1 \;\wedge\; A^2)$$

where the B_p^i and A^i are identical, except that A^i contains a constraint of the form $y \subseteq q$ if and only if B_p^i contains an application $q(y)$.

1. If $\mathcal{J} \subseteq T_{D^\#}(\mathcal{J})$, then for all $\sigma \in \mathcal{J}(p)$: $\mathcal{J}, [\sigma/x] \models_{\mathcal{P}(\mathcal{FT})} \exists_x B_p$. By the Union Property and since $\mathcal{J}$ is closed under unions, this implies $\mathcal{J}, [\sup(\mathcal{J}(p))/x] \models_{\mathcal{P}(\mathcal{FT})} \exists\,|_x\, B_p$, so there exist σ_1, σ_2 with

   $$\mathcal{J}, [\sup(\mathcal{J}(p))/x][\sigma_1/x_1][\sigma_2/x_2] \quad \models_{\mathcal{P}(\mathcal{FT})} \quad x \subseteq x_1 \cup x_2 \;\wedge\; B_p^1 \;\wedge\; B_p^2 .$$

   Define $\alpha = [\sup(\mathcal{J}(p))/x][\sigma_1/x_1][\sigma_2/x_2]$. From $\sup(\mathcal{J}(p)) \subseteq \sigma_1 \cup \sigma_2$ we easily obtain $\sup\circ\mathcal{J}, \alpha \models_{\mathcal{P}(\mathcal{FT})} p \subseteq x_1 \cup x_2$. It remains to check that

   $$\sup\circ\mathcal{J}, [\sigma_1/x_1][\sigma_2/x_2] \models_{\mathcal{P}(\mathcal{FT})} A^1 \wedge A^2 .$$

   Let $q(y)$ be an application in B_p^i, $i \in \{1,2\}$, and let α' extend α such that $\mathcal{J}, \alpha' \models q(y)$. Then $\alpha'(y) \in \mathcal{J}(q)$ holds. Since $\mathcal{J}$ is closed under union, this implies $\alpha'(y) \subseteq \sup\circ\mathcal{J}(q)$, and hence $\sup\circ\mathcal{J}, \alpha' \models_{\mathcal{P}(\mathcal{FT})} y \subseteq p$.

2. If $\alpha \models_{\mathcal{P}(\mathcal{FT})} A(D)$, then there exist σ_1, σ_2 such that

$$\alpha[\sigma_1/x_1][\sigma_2/x_2] \quad \models_{\mathcal{P}(\mathcal{FT})} \quad p \subseteq x_1 \cup x_2 \ \wedge\ A^1 \ \wedge\ A^2$$

Let $\sigma \in \alpha^{\#}(p)$. To prove $\alpha^{\#} \subseteq T_{D^{\#}}(\alpha^{\#})$ it suffices to show that

$$\alpha^{\#}, [\sigma/x][\sigma_1/x_1][\sigma_2/x_2] \quad \models_{\mathcal{P}(\mathcal{FT})} \quad x \subseteq x_1 \cup x_2 \ \wedge\ B^1_p \ \wedge\ B^2_p$$

Define $\beta = [\sigma/x][\sigma_1/x_1][\sigma_2/x_2]$. By definition of $\alpha^{\#}$, $\sigma \in \alpha^{\#}(p)$ implies $\sigma \subseteq \alpha(p)$. From $\alpha(p) \subseteq \sigma_1 \cup \sigma_2$ we obtain that $\beta \models_{\mathcal{P}(\mathcal{FT})} x \subseteq x_1 \cup x_2$. Now let $y \subseteq q$ be an inclusion occurring in A^i, $i \in \{1,2\}$, and let α' be an extension of $\alpha[\sigma_1/x_1][\sigma_2/x_2]$ such that $\alpha' \models_{\mathcal{P}(\mathcal{FT})} y \subseteq q$. Hence $\alpha'(y) \subseteq \alpha(q)$ which implies that $\alpha'(y) \in \alpha^{\#}(q)$; so there exists an analogous extension β' of β such that $\alpha^{\#}, \beta' \models_{\mathcal{P}(\mathcal{FT})} q(y)$. □

Corollary 32 (Characterisation of Set-based Analysis)
$gsol(A(D)) = \sup \circ gm(D^{\#})$

Proof. By Lemma 4.11, case (2), $gsol(A(D))^{\#}$ is a postfixed point of $T_{D^{\#}}$ and hence $gsol(A(D))^{\#} \subseteq gm(D^{\#})$ since $gm(D^{\#})$ is the greatest postfixed point of $T_{D^{\#}}$. By Lemma 4.11, case (1), $gm(D^{\#})$ is a solution of $A(D)$ since it is closed under unions. Hence, $gm(D^{\#}) \subseteq gsol(A(D))$. □

4.1.5. Analysing Constructor Tree Equations

Our analysis naturally generalises to constructor tree equations:

$$A^p(x{=}a(y_1,\ldots,y_n)) \quad = \quad a\langle x\rangle \wedge x\{1,\ldots,n\} \wedge \bigwedge_{i=1}^{n} x[i]y_i \tag{4.2}$$

This analysis is natural since it is just our analysis of constraints up to the canonical interpretation of constructor tree equations as CFT constraints (see Page 27). The corresponding analysis of $x{=}a(\overline{y})$ in terms of co-definite set constraints would be

$$A^p(x{=}a(y_1,\ldots,y_n)) \quad = \quad x \subseteq a(y_1,\ldots,y_n) \wedge \bigwedge_{i=1}^{n} y_i \subseteq a^{-1}_{(i)}(x) \tag{4.3}$$

In terms of $\mathrm{FT}_{\subseteq}(ar, \cup)$ constraints, there is no reasonable alternative to (4.2). There seems to be an alternative to the analysis (4.3) in terms of set constraints over constructor trees, though: one might wonder whether it is possible to strengthen the analysis of equations by mapping equations to equations.

Programs $\quad D \subseteq \quad$ set of all guarded clauses $p(x) \leftarrow \eta$ then S

Figure 4.6.: Syntax of Guarded Clauses

$$V D \eta \,[]\, p(x) \longrightarrow V \cup \{\bar{z}\}\, D \eta \,[]\, S[x/y] \qquad (\textsc{GuardApply})$$
$$\text{if } \begin{cases} p(y) \leftarrow \eta' \text{ then } S \in D,\, fv(S) \backslash \{y\} = \{\bar{z}\} \\ V \cap fv(S) \subseteq \{y\},\ \eta \models_{\mathcal{FT}} \exists \bar{z}(\eta'[x/y]) \end{cases}$$

Figure 4.7.: Operational Semantics of CC(CFT)

$$A^p(x{=}a(\bar{y})) \quad \stackrel{?}{=} \quad x{\subseteq}a(\bar{y}) \wedge a(\bar{y}){\subseteq}x \tag{4.4}$$

The immediate advantage of deriving stronger equality information is that equations can be handled much more efficiently than inclusion constraints. This analysis is, however, incorrect because it invalidates Theorem 26: the program D_{cstr} below contains no finitely failed procedure, but the analysis according to (4.4) is non-satisfiable.

$$\begin{array}{ll} p(x_1) \leftarrow x_1{=}a(b,b), & q(y) \leftarrow y{=}a(u,v),\ p(y) \\ p(x_2) \leftarrow x_2{=}a(c,c), & r(z_r) \leftarrow z_r{=}a(b,b),\ q(z_r) \\ s(z_s) \leftarrow z_s{=}a(c,c),\ q(z_s) & \end{array} \tag{D_{cstr}}$$

The analysis of D_{cstr} according to (4.4) is:

$$\begin{array}{l} p{\subseteq}a(b,b) \cup a(c,c) \wedge \\ q{\subseteq}y \wedge y{\subseteq}a(u,v) \wedge a(u,v){\subseteq}y \wedge y{\subseteq}p \wedge \\ z_r{=}a(b,b) \wedge z_r{\subseteq}q \wedge z_s{=}a(c,c) \wedge z_s{\subseteq}q \end{array}$$

This constraint is non-satisfiable, because it entails

$$a(b,b), a(c,c) \quad \subseteq \quad a(u,v) \quad \subseteq \quad a(b,b) \cup a(c,c)$$

which is non-satisfiable: x and y must not denote the empty set since $a(b,b){\subseteq}f(x,y)$ implies $b{\subseteq}x$ and $b{\subseteq}y$. Since $f(x,y){\subseteq}a(b,b) \cup a(c,c)$ either $\alpha(x) = \alpha(y) = \{b\}$ or $\alpha(x) = \alpha(y) = \{c\}$ must hold In both cases, either $a(c,c){\subseteq}f(x,y)$ or $a(b,b){\subseteq}f(x,y)$ is not satisfied.

4.2. Set-based Failure Diagnosis for CC over Infinite Trees

We now consider the concurrent constraint language that we obtain by extending the language CLP(CFT) by guarded clauses. We adapt the analysis and the correctness proof. A *guarded clause* has the form defined in Figure 4.6. Call CC(CFT) the corresponding extension of CLP(CFT). Apparently, unguarded clauses are the special case of guarded clauses with trivial guards:

$$p(x) \leftarrow y{=}y \;\mathsf{then}\; S$$

The operational semantics of application with guarded clauses is adapted in Figure 4.7.

Guarded Apply. Reduction of an application $p(x)$ may pick a clause $p(y) \leftarrow \eta' \;\mathsf{then}\; S$ only if the constraint store η in the current configuration entails η'.

Define, for every program D the unguarded approximation $ug(D)$ by replacing every guarded clause with an unguarded one where then is replaced by parallel composition:

$$(\eta \;\mathsf{then}\; S) \qquad \rightsquigarrow \qquad \eta,\, S$$

Then define the analysis of a CC(CFT) program D via the analysis of its unguarded approximation: $A(D) = A(ug(D))$.

4.2.1. Blocked Reduction or Finite Failure

In contrast to unguarded clauses, it is possible that an application $p(x)$ never reduces because the guards of all clauses of p are never entailed. We say that $p(x)$ *blocks forever*. Call a procedure p *blocked forever* if in every computation issuing from $\{x\}D\top\,[]\; p(x)$ at least one application will block forever. For a trivial example consider the following statement.

$$p(x) \leftarrow a\langle x'\rangle \;\mathsf{then}\; b\langle x'\rangle$$

$$q(y) \leftarrow p(z)$$

No computation issuing from a call to q can accumulate enough information on z in the constraint store to entail or disentail $\exists x'\, a\langle x'\rangle$. Hence the call $p(z)$ will never reduce. For another example consider:

$$p(x) \leftarrow x[f]y,\; p(y)$$

$$q(y) \leftarrow z[f]z \;\mathsf{then}\; \mathsf{skip}$$

$$r(z) \leftarrow p(z),\; q(z)$$

Reduction of the application $r(u)$ will constrain u to a tree with a finite path $fff\ldots f$ of arbitrary *finite* length; yet, the guard $u[f]u$ in the clause of q (which asks whether x denotes a tree with an *infinite* path $fff\ldots$) will never be entailed.

Hence, for CC(CFT) our correctness result must be weakened with respect to the result for CLP(CFT). It still shows that and to what degree certain run-time errors are detected.

Theorem 33 (Detection of Finite Failure or Blocked Reduction)
For all CC(CFT) programs D: If $A(D) \models_{\mathcal{P}(\mathcal{FT})} p{=}\emptyset$ then p either finitely fails or blocks forever.

Proof. By contradiction. Assume $A(D) \models_{\mathcal{P}(\mathcal{FT})} p{=}\emptyset$ and suppose that there is a computation $(C_i)_{i=1}^{n}$ of $\{x\}D\top\,[]\;\; p(x)$ which does neither finitely fail nor block forever. Clearly, this computation induces a computation of $ug(D)$ that neither fails nor blocks forever, since application of an unguarded clause is possible without any side condition. Theorem 26 implies that $A(ug(D)$ does not entail emptiness for any procedure, and therefore $A(D) = A(ug(D)$ does not. □

4.2.2. Blocked Reduction and Run-time Errors

In most cases, an infinitely blocked application can be considered erroneous. Under this assumption, Theorem 33 states that our analysis indeed detects certain run-time errors in concurrent constraint programs automatically. Studies in the expressiveness of concurrent computation may take an alternative point of view, namely that a blocked application simply is not observable and thus (observationally) equivalent to the empty program skip. For example, encodings of lazy functional computation with logic variables [152, 170] block the reduction of statements that are *not needed* for the overall result. In these encodings, all blocked statements could *in principle* be woken without the danger of inducing additional run-time errors. More delicate are encodings of choice [150, 158]. For example, the program

$p(x) \leftarrow a\langle x\rangle \;\mathsf{then}\; S_1$

$p(x) \leftarrow b\langle x\rangle \;\mathsf{then}\; S_2$

$\ldots p(z) \ldots$

is, from this point of view, equivalent to the parallel composition

$p(x) \leftarrow a\langle x\rangle \;\mathsf{then}\; S_1$

$q(x) \leftarrow b\langle x\rangle \;\mathsf{then}\; S_2$

$\ldots p(z),\; q(z) \ldots$

because the involved guards are mutually exclusive so that either $p(z)$ or $q(z)$ is guaranteed to block forever. The correctness of such choice encodings crucially relies on the fact that at most one of the encoded branches (S_1 or S_2) is executed because they might perform incompatible operations.

For programs that rely on blocked reduction as a programming technique and hence do not consider it an error, Theorem 33 can still be used to explain our analysis. However, it may be necessary to interpret emptiness in the greatest solution as a warning rather than an error message. The information that some computation is unfailed *only because* it does block forever is important debugging information anyway.

4.2.3. Inevitable Failure versus Possible Failure

Apart from the *inevitability* of errors it would we useful to also obtain information about the *possibility* of errors (that is, whether there exists *at least one* finitely failed computation of a program). This is true in particular for non-deterministic languages.

Can we improve Theorem 33 accordingly? The answer is *no* since our analysis approximates conditional guards as tell statements. In most cases, this approximation yields programs that trivially have one failed computation. For illustration, notice that many programs operating on lists will contain a guarded choice of the typical form:

$$p(x_1) \leftarrow x_1{=}nil \;\mathsf{then}\; S_1$$

$$p(x_2) \leftarrow x_2[head]y \wedge x_2[tail]z \;\mathsf{then}\; S_2$$

The procedure p expects x to be constrained to a (tree modeling a) list; in this case it does not block forever and is (supposedly) unfailed. In its unguarded abstraction, however,

$$p(x_1) \leftarrow x_1{=}nil,\; S_1$$

$$p(x_2) \leftarrow x_2[head]y \wedge x_2[tail]z,\; S_2$$

the application $p(x)$ has one trivially failed computation if x is constrained at all; even if x is properly constrained to a list and has label *nil* or *cons*.

4.2.4. Inevitable Failure as a Debugging Criterion

The choice of inevitable failure as a criterion for faultiness of a program deserves some discussion. Why, in particular, is it not possible to do without the set-based analysis altogether and simply run the program for a limited amount of time?

In the CLP case, the answer is mostly positive: If a CLP program is inevitably failed, then every run of the program will eventually exhibit it. Of course, it is difficult to know beforehand whether the running program or the constraint solver will exhibit this error earlier, but in practice there is a certain chance that simply running the program is the better alternative. On the other hand, if a CLP program fails then one knows that it is *possibly* failed, which is a stronger diagnostics in the CLP case and hence clearly desirable.

In the CC case, the answer is negative: Simply running a CC program is not sufficient to detect the error "inevitable failure or blocked reduction": If one observes a blocked application during program reduction one does not know (i) whether this application will indeed block *forever*, and (ii) even if one can prove this one does not know whether an error is prevented *by the fact* that the application blocks (in other words, whether an error would occur could the application be unblocked).

A second question may come to mind: Why not simply test a CC program by running its unguarded approximation instead, *i. e.*, by ignoring synchronisation during testing? This is not useful, since possible failure of the CLP approximation is not an interesting debugging criterion as shown in the previous section.

4.3. Related Work

In the logic programming community, the status of types is more controversial than in the functional programming community. In particular, the notion of a *type error* is less clear in logic programming than it is in functional languages, since the semantics of logic programs is based on predicate logic in which every syntactically well-formed expression has a meaning. In operational terms, there is no inherent distinction between type error and a logic *failure* in logic programming. For an overview of several approaches to types in logic programming see Pfenning's collection [164], and the recent report by Meyer [122].

Yardeni and Shapiro [220] were the first to suggest that a type in logic programming should be an upper approximation of the program's least model semantics. Similarly, our failure diagnosis computes types of CC programs as upper approximations of the program's *greatest model semantics*. Such types pertaining to the denotational semantics of programs have been called *semantic types* by Heintze and Jaffar [87], in contrast to types pertaining to the operational semantics. Yardeni and Shapiro [220] gave a tuple-distributive abstraction Y_P of the consequence operator T_P. Heintze and Jaffar [85] defined an operator $\mathcal{T}_P$ based on set-substitutions and showed that is more accurately than Y_P approximated the least model semantics.

Mishra gave an analysis for logic programs in terms of constraints path-closed sets, and proven that his analysis approximated the least model semantics [132]. It has been noticed, however, that his analysis was so weak that it rather approximated the greatest model semantics [87, 171]. The characterisation of the greatest model semantics (for logic programs over infinite trees) in terms of finite failure, as we have discussed it above, is original to our paper [171]. Yardeni, Frühwirth, Vardi, and Shapiro [69] characterised the membership problem in the least model of a logic program in a subclass of so-called unary-predicate programs based on tree automata. Devienne, Tison, and Talbot present an implementation of their analysis based on tree automata [201].

Charatonik and Podelski [45] show that set-based analysis can be used to approximate temporal properties of logic programs that are intended to describe possibly non-terminating computations.

5. Set-based Failure Diagnosis for Oz

In this chapter, we extend the failure diagnosis from CC to a large sublanguage of Oz with higher-order procedures, cells, feature tree constraints, and conditionals: that is, our analysis comprises essentially the whole Oz Programming Model (OPM) according to [195], except that we fix CFT as the underlying constraint system in order to incorporate records. OPM is considerably more expressive than CC due to the presence of higher-order procedures (and cells). Nonetheless, the extension of our failure diagnosis is rather smooth. This is a desirable situation because it allows one to understand the analysis for the simpler first-order fragment first.

On the other hand, the problem of correctness of our diagnosis for OPM becomes fundamentally harder in contrast to CC due to the presence of higher-order procedures. The correctness proof for CC in the previous chapter is based on the logic semantics borrowed from constraint logic programming. An analogous argument is not available for Oz since there is no denotational semantics for Oz, even with cells put aside.[33] Notice also, that it is not easily possible to reduce a program with higher-order procedures to a first-order program. Therefore, a correctness proof for OPM probably has to argue directly on the operational semantics.

In an experimental implementation of our analysis for Oz, we have found it useful for debugging and the automated detection of errors. We illustrate the analysis by a number of examples, and we provide a set of style conventions that summarise the intuitions underlying the analysis. The material presented in this chapter has an explorative character for two reasons. First, we leave open the question of how to prove the diagnosis correct. Second, the integration of our diagnosis into a production quality

[33] It is an open research problem how such a denotational semantics would look like, in particular since it would have to subsume the denotational semantics of both CC and lambda calculus.

compiler and its feasibility and scalability needs to be explored. So the problem of automated failure diagnosis for Oz is left unsettled, but we hope that this chapter can serve as a source of inspiration for future research.

We also take a closer look at conditionals. As we shall see, a careful analysis of conditionals is crucial for the accuracy of the failure diagnosis. More specifically, the immediate extension of our analysis from CC to OPM (with higher-order procedures) yields a less accurate analysis, in particular due to the analysis of conditionals. As a remedy, we provide a simple syntactic condition on conditionals to avoid this problem. More generally, we notice that both cost and accuracy of the analysis can be drastically improved if the data flow through conditionals is statically known. This data flow information includes the knowledge which variables the conditional guards depend on, and which variables become constrained (their value is provided) during execution of a conditional clause.

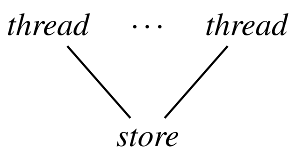
The constraint setting makes it easy to improve the analysis by annotating variables with type information. Such an annotation is a predicate that describes the possible values which a variable is *expected* or *intended* to take. Annotations nicely fit in the constraint framework as they can be interpreted as *prescriptive* constraints on the program, in addition to the constraints derived from the program by purely *descriptive* means.

5.1. The Oz Programming Model

In this section, we recall the definition of OPM, where we mostly follow Smolka's paper [195]. The reader familiar with Oz and OPM may want to skip the following section and proceed directly to Section 5.2. We do not give many examples on programming in Oz, but refer the reader instead to the other publications on Oz, the demos that are part of the Oz distribution [91, 174], as well as the examples for Plain in Section 6.1.3.

5.1.1. The Computational Setup

Concurrent computation in OPM is organised in terms of processes, called *threads*, over a shared store. Each thread represents an independent unit of concurrent computation with its own control structure. The shared store is the means through which the threads communicate and on which they synchronise.



The store contains possibly partial information about the values which a variable may take on. According to the different kinds of information, the store is partitioned in three segments, called the *constraint store*, the *procedure store* and the *cell store*.

The constraint store hosts a conjunction of first-order formulas that we call *constraints*. Typical constraints include equations between variables (*i. e.*, $x{=}y$), and between variables and atomic entities such as integers and symbolic constants. In particular we assume a class of constants called *names*. The procedure store maps names to procedures, and the cell store maps names to cells. The only way to update the constraint store is to *tell* it a new constraint, which means to add it as a new conjunct to the store. Procedure and cell store are updated whenever new procedures and cells are defined.

If the constraint store does not entail (logically imply) any constraints on a variable x apart from equality to other variables, then x is called *unbound*. If the constraint store implies equality of a variable and some data structure (such as a name), then x is called *bound to* this data structure. If x is bound to a name mapped to a procedure or a cell we also say that x is bound to a procedure or a cell. The constraint store is required to remain *satisfiable*. The attempt to tell a constraint that is inconsistent with the store is called a *failure* and is considered a run-time error. The constraint store is organised such that one can only add new information to it but never retract any: the amount of information in the constraint store grows *monotonically*.

Synchronisation between threads is through the constraint store only. A thread can block until some constraint is entailed by the constraint store. Due to the monotonicity property, this implies that synchronisation conditions are safe: if such a condition is true once, it will stay true forever. No race conditions can occur. This also yields the straightforward *fairness condition* that every reducible thread must eventually be reduced. Like the constraint store, the procedure store grows monotonically, whereas the cell store does not. This is intentional, since it is incorporated into OPM exactly *for* supporting computation with state change which is essentially non-monotonic. But a direct synchronisation on the cell store is carefully avoided.

5.1.2. The Base Language

We assume a set $\mathcal{V}$ of *variables* ranged over by x, y, z and a set $\mathcal{N}$ of *names* ranged over by n. Figure 5.1 defines the syntax of the basic OPM statements S. Typical constraints are denoted by η. The statement $(\mathsf{proc}\; x\; (\overline{y})\; S)$ defines a *procedure* with *identifier* x, formal arguments $\overline{y}$, and *body* S. The statement $(x\ \overline{y})$ applies a procedure with identifier x to the *actual arguments* $\overline{y}$. The conditional statement $\mathsf{if}\ (\exists\overline{x}_1\eta_1\ \mathsf{then}\ S_1) + (\exists\overline{x}_2\eta_2\ \mathsf{then}\ S_2)$ consists of two *guarded clauses*.[34]

[34]Notice that OPM according to [195] does not have such a choice construct, whereas Oz has one. We consider it here because it makes the embedding of CC(CFT) programs into OPM trivial. The analysis is not substantially affected by this choice.

S	$::=$	$(\mathsf{local}\ (\overline{x})\ S)$	(Variable Declaration)
	$\mid$	$S_1 \parallel S_2$	(Parallel Composition)
	$\mid$	skip	(Null Statement)
	$\mid$	$(\mathsf{proc}\ x\ (\overline{y})\ S)$	(Procedure Definition)
	$\mid$	$(x\ \overline{y})$	(Procedure Application)
	$\mid$	$\mathsf{if}\ (\exists \overline{x}_1 \eta_1\ \mathsf{then}\ S_1) + (\exists \overline{x}_2 \eta_2\ \mathsf{then}\ S_2)$	(Conditional)
	$\mid$	η	(Tell Statement)
	$\mid$	$(\mathsf{cell}\ x\ y)$	(Cell Definition)
	$\mid$	$(\mathsf{exch}\ x\ y\ z)$	(Cell Exchange)

Figure 5.1.: Syntax of OPM Statements

In a guarded clause $(\exists \overline{x} \eta\ \mathsf{then}\ S)$ we call $\exists \overline{x} \eta$ the *guard* and S the *body*. We identify $\exists \overline{x} \eta$ with η if $\overline{x}$ is an empty sequence. The statement $(\mathsf{cell}\ x\ y)$ defines a *cell* with identifier x and *initial content* y. The *exchange statement* $(\mathsf{exch}\ x\ y\ z)$ operates on a cell with identifier x and provides z as the new content of x and y as reference to the old content of x.

According to [195], OPM is parametrised with respect to the underlying constraint system. We fix constraints η to be drawn from the feature tree constraint system CFT defined in the Chapter 2. In this context we treat names simply as labels which, importantly, do not have a concrete notation in OPM.

In a procedure definition $(\mathsf{proc}\ x\ (\overline{y})\ S)$ and a declaration $(\mathsf{local}\ (\overline{y})\ S)$ the variables $\overline{y}$ are *bound* with scope S.[35] In a guarded clause $(\exists \overline{y} \eta\ \mathsf{then}\ S)$, the variables $\overline{y}$ are bound with scope η and S. Free and bound variables of a statement S are defined as usual and denoted by $fv(S)$ and $bv(S)$.

Computation in OPM proceeds by reduction on statements. It employs an *interleaving semantics*, meaning that only one statement reduces at a time; there is no simultaneous reduction of more than one statement. The atomic reduction steps are defined as follows.

[35]The fact that the notion of "bound variable" can mean both "bound in a statement" and "bound in the store" will not lead to confusion since the first one is a static concept and the second one a dynamic concept. We are aware that we could resolve the ambiguity by distinguishing between identifiers (static) and variables (dynamic) but avoid this for simplicity.

Variable Declaration. The reduction of a declaration statement $(\mathsf{local}\ (\overline{x})\ S)$ is unsynchronised; it picks fresh variables $\overline{y}$ and replaces the declaration $(\mathsf{local}\ (\overline{x})\ S)$ by the statement $S[\overline{y}/\overline{x}]$ that is obtained from S by simultaneously substituting $\overline{y}$ for all free occurrences of $\overline{x}$ in S.

Null Statement. The null statement skip does not reduce and is not observable.

Tell Statement. Reduction of a tell statement η is unsynchronised; it reduces to skip after η was successfully told to the current store.

Procedure Definition. Reduction of a procedure definition $(\mathsf{proc}\ x\ (\overline{y})\ S)$ is unsynchronised; it chooses a fresh name n, extends the procedure store by $(\mathsf{proc}\ n\ (\overline{y})\ S)$ and tells $x{=}n$ to the constraint store. Note that it is the constraint store (not the procedure store) that will exhibit the failure if another definition with identifier x has been executed before.

Procedure Application. A procedure application $(x\ \overline{y})$ synchronises on the fact that x is bound in the constraint store. If x is bound to a name n and the procedure store contains $(\mathsf{proc}\ n\ (\overline{y})\ S)$, then the application is replaced by $S[\overline{y}/\overline{x}]$; that is, by the procedure body S with the actual arguments $\overline{y}$ replacing for the formal ones $\overline{x}$. The situations that x is bound but not to a name, or that x is bound to a name that is not mapped to a procedure are run-time errors.

Conditional. A conditional $\mathsf{if}\ (\exists\overline{x}_1\eta_1\ \mathsf{then}\ S_1) + (\exists\overline{x}_2\eta_2\ \mathsf{then}\ S_2)$ synchronises on one of two conditions. If the current constraint store entails one of the guards $\exists\overline{x}_i\eta_i$ $(i = 1,2)$ then the conditional is replaced by $(\mathsf{local}\ (\overline{x}_i)\ S_i)$. If it entails *both* guards, then reduction of the conditional is *indeterminate*. This choice is never undone ("committed choice"). If the current constraint store entails the negation of both guards, a run-time error is flagged.

Cell Definition. Reduction of a cell definition $(\mathsf{cell}\ x\ y)$ is unsynchronised; it chooses a fresh name n, extends the cell $(\mathsf{cell}\ n\ y)$, and tells $x{=}n$ to the constraint store.

Cell Exchange. Reduction of a cell exchange $(\mathsf{exch}\ x\ y\ z)$ synchronises on the fact that x is bound in the constraint store. If x is bound to a name n and the cell store contains $(\mathsf{cell}\ n\ y')$, then this entry is updated to $(\mathsf{cell}\ n\ z)$, and the exchange statement is replaced by $y{=}y'$. The situations that x is bound but not to a name, or that x is bound to a name that is not mapped to a cell are run-time errors.

Synchronisation on cells is as simple as synchronisation on procedures. Once x is bound to a cell, the exchange statement does not synchronise at all. It enables communication between producer and consumer of the cell content and delegates all synchronisation to these two.

Patterns	t	$::=$	$a(\overline{f}{:}\overline{t}) \quad \vert \quad a(\overline{f}{:}\overline{t}\ldots) \quad \vert \quad (\overline{f}{:}\overline{t}) \quad \vert \quad (\overline{f}{:}\overline{t}\ldots)$
Statements	S	$::=$	$\ldots \quad \vert \quad \mathsf{case}\ x\ \mathsf{of}\ (t_1\ \mathsf{then}\ S_1) + (t_2\ \mathsf{then}\ S_2) \qquad \text{(Case)}$

Figure 5.2.: Syntax Extension for Pattern Matching

Cells add a second form of *indeterminism* to OPM; when multiple concurrent threads perform an exchange on the same cell, then the final cell content depends on reduction order. Multiple cell exchanges do not interfere with each other, though. Mutual exclusion of concurrent operations on a cell is guaranteed, since an exchange performs a read and a write operation on a cell in an atomic step.

5.1.3. Names

Names serve different purposes in OPM. First, they provide an interface between constraints, which bear a first-order logic semantics, and procedures and cells, which have none. As a consequence, they support an *untyped equality test* test at all data types (similar to `eq` in Scheme) that is particularly convenient in object oriented-programming where one can test for object identity:

$\mathsf{if}\ (obj{=}self\ \mathsf{then}\ S) + \ldots$

Third, names model *locations* at which cells are located. This is essential since there may be multiple references to the same cell, and every operation on the cell must be visible to all of them. Finally, OPM provides a primitive operation to create new names, independent of the definition of procedures and cells, as unique tokens.

Since there is no explicit notation for names, names behave just as constant symbols with the additional guarantee that they are globally unique and cannot be forged. In combination with lexical scoping, the generation of new names is a flexible mechanism to ensure privacy in an untyped setting. (In typed languages some of these services can be offered by abstract data types.)

5.1.4. Case Statements

Figure 5.2 adds some syntactic sugar to model *case statements* with *pattern matching*. A case statement like $\mathsf{case}\ x\ \mathsf{of}\ (t_1\ \mathsf{then}\ S_1) + (t_2\ \mathsf{then}\ S_2)$ provides elegant support for the decomposition of records and is common in functional programming languages. A *pattern* t is a partial description of a feature tree. More specifically, the label can be omitted if unknown, and an ellipsis is used if the available features are not completely known. In *case clauses* such as $(a(\overline{f}{:}\overline{x}\ldots)$ then $S)$ and $(a(\overline{f}{:}\overline{x})$ then $S)$, the variables $\overline{x}$ are bound with scope S. In this context, the variables $\overline{x}$ are also called *pattern variables*.

Pattern Matching. A variable x *matches* a pattern t if x is bound to a feature tree that has all the labels and features mentioned in t. For example, a feature term $a(f{:}y)$ is matched by every tree that has at least the label a and the feature f at the root.

Case Statement. Reduction of a case statement case x of $(t_1$ then $S_1) + (t_2$ then $S_2)$ synchronises on the fact that the current information on x in the constraint store matches one of the patterns t_1 or t_2. The situation that x will never match t_1 or t_2 is a run-time error.

Case statements are not primitive in OPM, since pattern matching is easily expressed as an entailment problem. For example, a variable x matches a pattern $a(f_1{:}x_1 \ldots f_n{:}x_n \ldots)$ if the constraint store entails

$$a\langle x\rangle \wedge \exists y_1 x[f_1]y_1 \wedge \ldots \wedge \exists y_n x[f_n]y_n$$

or, equivalently, $x{=}a(f_1{:}x_1,\ldots,f_n{:}x_n,\overline{g}{:}\overline{y})$ for some features $\overline{g}$ and variables $\overline{y}$. The variable x matches $a(f_1{:}x_1,\ldots,f_n{:}x_n)$ if in addition the constraint store entails $x\{f_1,\ldots,f_n\}$; that is, if the constraint store entails

$$a\langle x\rangle \wedge x\{f_1,\ldots,f_n\} \wedge \exists y_1 x[f_1]y_1 \wedge \ldots \wedge \exists y_n x[f_n]y_n\,.$$

or, equivalently $x{=}a(f_1{:}x_1,\ldots,f_n{:}x_n)$. So case statements can easily be expressed with the conditional of the base language. For example,

$$\mathsf{case}\ x\ (a(f{:}y\ldots)\ \mathsf{then}\ S_1) + (b(g{:}z)\ \mathsf{then}\ S_2)$$

is equivalent to the guarded conditional below, provided that $x \notin \mathcal{V}(t_1) \cup \mathcal{V}(t_2)$.

$$\mathsf{if}\ (\exists y(a\langle x\rangle \wedge x[f]y)\ \mathsf{then}\ S_1) + (\exists z(b\langle x\rangle \wedge x\{g\} \wedge x[g]z)\ \mathsf{then}\ S_2)$$

5.2. Set-based Failure Diagnosis

The analysis is defined in Figure 5.3 as a mapping A from OPM statements S to existential formulas $A(D)$ over $\mathrm{FT}_{\subseteq}(ar,\cup)$ constraints. The analysis uses the following criterion to mark programs as dubious and reject it.

Rejection Condition. A program D is no good if $A(D)$ is non-satisfiable over non-empty sets of trees.

In the remainder of this section we explain the analysis stepwise along with illustrating examples.[36]

[36] In addition, most of the examples from Section 4.1.1 can be easily adapted to the compositional OPM syntax. However, the analysis of a CC(CFT) program according to Figure 4.4 is more accurate

$$
\begin{array}{lcl}
A(\eta) & = & \eta \\
A(S_1 \parallel S_2) & = & A(S_1) \wedge A(S_2) \\
A((\mathsf{local}\ (x)\ S)) & = & \exists x A(S) \\
A((\mathsf{proc}\ x\ (y_1 \ldots y_n)\ S)) & = & \exists y_1 \ldots \exists y_n\, (x \subseteq \mathsf{proc}(arg_1{:}y_1, \ldots, arg_n{:}y_n) \wedge A(S)) \\
A((x\ y_1 \ldots y_n)) & = & \exists z_1 \ldots \exists z_n \left(\begin{array}{l} \mathsf{proc}\langle x \rangle \wedge x\{arg_1, \ldots, arg_n\} \wedge \\ x[arg_1]z_1 \wedge \ldots \wedge x[arg_n]z_n \wedge \\ y_1 \subseteq z_1 \wedge \ldots \wedge y_n \subseteq z_n \end{array} \right) \\
A(\mathsf{if}\ \exists \overline{x} \eta\ \mathsf{then}\ S) & = & A((\mathsf{local}\ (\overline{x})\ \eta \parallel S)) \\
\multicolumn{3}{l}{A(\mathsf{if}\ (\exists \overline{x}_1 \eta_1\ \mathsf{then}\ S_1) + (\exists \overline{x}_2 \eta_2\ \mathsf{then}\ S_2))} \\
 & = & \exists \overline{y}_1 \exists \overline{y}_2 \left(\begin{array}{l} \overline{y} \subseteq \overline{y}_1 \cup \overline{y}_2 \wedge \\ A(\exists \overline{x}_1 \eta_1\ \mathsf{then}\ S_1)[\overline{y}_1/\overline{y}] \wedge \\ A(\exists \overline{x}_2 \eta_2\ \mathsf{then}\ S_2)[\overline{y}_2/\overline{y}] \end{array} \right) \\
 & & \mathsf{if}\ \{\overline{y}\} = fv(\mathsf{if}\ (\exists \overline{x}_1 \eta_1\ \mathsf{then}\ S_1) + (\exists \overline{x}_2 \eta_2\ \mathsf{then}\ S_2)) \\
A((\mathsf{cell}\ x\ y)) & = & \mathsf{cell}\langle x \rangle \wedge x\{\} \\
A((\mathsf{exch}\ x\ y\ z)) & = & \mathsf{cell}\langle x \rangle \wedge x\{\}
\end{array}
$$

Figure 5.3.: Set-based Failure Diagnosis for OPM

5.2.1. Constraints, Parallel Composition, and Declaration

There is no surprise as to the analysis of parallel composition and of constraints: the analysis of constraints exploits again the fact that every CFT constraint is also an $\mathrm{FT}_{\subseteq}(ar)$ constraint,

Declaration. In the analysis of variable declaration note that the assertion $x{=}\emptyset \leftrightarrow p{=}\emptyset$ is dropped; it was used in Chapter 4 to relate emptiness of a local variable to emptiness of the analysis of the enclosing procedure.

In presence of global variables, this technique of localising inconsistencies at procedures does not work anymore. For example, let z be a global variable and consider the following program (where $\_$ is an arbitrary variable):

$$(\mathsf{proc}\ x\ (\_)\ a\langle z\rangle) \parallel (\mathsf{proc}\ y\ (\_)\ b\langle z\rangle) \qquad (D_{global})$$

Apparently, the procedures x and y have an inconsistent view as to the value of the global variable z. However, this does not imply that either $(x\ u)$ or $(y\ v)$ are finitely failed: none of them is. Only in combination of both indeed yields a finite failure:

$$(x\ u) \parallel (y\ v)$$

Therefore, our rejection condition takes *all* variables into account. Intuitively, this means that we consider a great number of program points in addition to the call points of procedures.

5.2.2. Procedures, Applications, and Conditionals

Since procedures are first-class in OPM, they can be referred to by variables which may also occur everywhere else in the program. Hence the analysis must be able to constrain variables to denote procedures of appropriate arity. For this purpose, we introduce a new label proc and an infinite number of (pairwise distinct) features $arg_1, arg_2, \ldots$ that may not occur everywhere else in the program.

Furthermore, we allow procedures with arbitrary arity in this chapter. For first-order procedures, a simple syntactic test can guarantee that all of them are applied with the correct number of arguments. In presence of higher-order procedures, we must explicitly reason about procedure arities.

Procedures. The analysis of $(\mathsf{proc}\ x\ (\overline{y})\ S)$ states that x is a procedure with arity n, and that all of its formal arguments must be consistent with the use of the formal arguments $y_1, \ldots, y_n$ in the procedure body S. Notice that the body of a procedure is analysed independent of whether it is applied or not.

than the analysis of the corresponding OPM program according to Figure 5.3. For further illustration, see Section 5.3.

Application. The analysis of a procedure application $(x\ \overline{y})$ states that x is a procedure that has arity n and that allows all possible values of $\overline{y}$ as arguments.

Single Clause Conditionals. A conditional that consists of only one clause is assumed to be used for synchronisation. That is, it is assumed that its guard will eventually be entailed and, therefore, that its body is executed in every fair computation.

Binary Clause Conditionals. For conditionals with two (or more) clauses we assume at least one of its clauses is eventually executed. Conditionals are analysed clause-wise. The occurring variables are renamed to avoid any unwanted dependency between the clauses to arise whose execution is mutually exclusive.

Case Statements. The analysis of case statements is defined via their translation to the basic conditional form (see Page 125).

In Section 5.3 we consider possible refinements of the analysis of conditionals.

5.2.3. Cells

The indeterminism provided by cells and conditionals is different: While conditionals make the choice between multiple clauses locally explicit, the choice between multiple exchanges on the same cell is implicit. This complicates the analysis of cells in comparison with the analysis of conditionals. In case of conditionals, the analysis of its clauses is combined in a union constraint. This is, in general, impossible for cells which leave the choice implicit. It makes the analysis of cell contents a global issue instead of a local one as the analysis of conditionals. Locally, we can only derive little information on cells.

Cell Definition. For the analysis of cells we introduce a special label cell. The analysis of a cell definition $(\mathsf{cell}\ x\ y)$ derives that x must denote a cell, but we derive no constrains on y.

Cell Exchange. From $(\mathsf{exch}\ x\ y\ z)$ we derive that x must denote a cell, but no constraints on y or z.

This analysis is fairly weak as it does not derive any information about the set of values that a cell may hold during its life time. This may be improved by a *global data flow analysis* that can delimit the set of references to a given cell. Alternatively, programmers could provide an annotation restricting the possible values that the analysis accepts for a cell.

5.2.4. Examples

To acquaint the reader with the modified syntax, we start with the example D_{fail2} from the previous chapter in OPM syntax:

$$(\mathsf{proc}\ x\ (x')\ a\langle x'\rangle)$$
$$(\mathsf{proc}\ y\ (y')\ b\langle y'\rangle)$$
$$(\mathsf{proc}\ z\ (z')\ (x\ z')\parallel(y\ z'))\,. \tag{D_{fail2}}$$

As analysis of D_{fail2} we obtain:

$$\mathsf{proc}\langle x\rangle\ \wedge\ x[arg_1]x'\ \wedge\ a\langle x'\rangle\ \wedge$$
$$\mathsf{proc}\langle y\rangle\ \wedge y[arg_1]y'\ \wedge\ b\langle y'\rangle\ \wedge$$
$$z'\subseteq x'\ \wedge\ z'\subseteq y'$$

We reject D_{fail2} since its analysis entails $z'=\emptyset$.

5.2.4.1. Procedures vs. Records

Consider a program that uses x both as a procedure and as a record identifier.

$$(\mathsf{proc}\ x\ (y)\ S)\parallel a\langle x\rangle \tag{$D_{typeerr1}$}$$

The analysis rejects the program $D_{typeerr1}$ since the associated constraint entails $x=\emptyset$.

$$\mathsf{proc}\langle x\rangle\ \wedge\ a\langle x\rangle\ \wedge\ldots$$

Similarly, field selection on a procedure is rejected:

$$(\mathsf{proc}\ x\ (y)\ S)\parallel x[f]z \tag{$D_{typeerr2}$}$$

The associated constraint is this one:

$$\mathsf{proc}\langle x\rangle\ \wedge\ x\{arg_1\}\ \wedge\ x[f]z\ \wedge\ldots$$

5.2.4.2. Arity Mismatch

The procedure D_{ar2} contains an arity mismatch.

$$(\mathsf{proc}\ x\ (z_1 z_2)\ S)\parallel(x\ y) \tag{D_{ar2}}$$

The analysis derives the following constraint, which entails $x=\emptyset$.

$$x\{arg_1,arg_2\}\ \wedge\ x\{arg_1\}\ \wedge\ldots$$

Similarly, programs that contain two applications with different arities are rejected:

$$(x\ y)\parallel(x\ z_1 z_2) \tag{D_{ar3}}$$

The failure indicated by the fact that the analysis of D_{ar3} entails $x{=}\emptyset$ is not necessarily exhibited at run-time; it is possible that no application with illegal arity will ever be *executed*. Rather, if x is never bound, both applications will block forever and are considered erroneous for this reason.

A third related example illustrates the requirement that all procedures should agree about the values of their joint global variables.

$$(\mathsf{proc}\; x\; (x')\; (u\; x')) \parallel (\mathsf{proc}\; y\; (y)\; (u\; y\; y)) \qquad (D_{ar4})$$

The procedure x in D_{ar4} expects u to be a unary procedure, while y applies u with two arguments. The analysis entails $u{=}\emptyset$ by a similar argument as for the two previous examples.

5.2.4.3. Higher-order Procedures

In the following program, x is a higher-order procedure that takes a procedure and applies it to a single argument. The application $(x\; u)$ is erroneous since u is bound to a binary procedure.

$$(\mathsf{proc}\; x\; (y)\; (y\; z)) \parallel (x\; u) \parallel (\mathsf{proc}\; u\; (v_1 v_2)\; S) \qquad (D_{ar5})$$

The error is detected because the analysis of D_{ar4} entails $u{=}\emptyset$.

$$x[arg_1]y \;\wedge\; y\{arg_1\} \;\wedge\; u{\subseteq}y \;\wedge\; u\{arg_1, arg_2\} \;\wedge \ldots$$

5.2.4.4. Multiple Procedures

Execution of two definitions for the same variable like in

$$(\mathsf{proc}\; x\; (\overline{y})\; S_1) \parallel (\mathsf{proc}\; x\; (\overline{z})\; S_2)$$

will lead to a failure due to the attempt to bind x to two different names. Our analysis detects many such situations. (i) If $|\overline{y}| \neq |\overline{z}|$, the analysis will force x to denote a unary and a binary procedure at the same time. (ii) If S_1 and S_2 use their arguments at different types, emptiness of x will be entailed. For illustration, consider $(\mathsf{proc}\; x\; (y)\; a\langle y\rangle) \parallel (\mathsf{proc}\; x\; (y')\; b\langle y'\rangle)$. Figure 5.8 on Page 137 gives a refinement of the analysis of procedures (in a different context) which can improve the accuracy of the analysis in cases like this.

5.2.4.5. Conditionals

The following conditional is considered erroneous because guard and body do not agree: the guard tests for some condition and the body contradicts it. If this clause is ever committed to, it will inevitably lead to failure.

$$\mathsf{if}\; (\exists y x[f]y \wedge a\langle x\rangle\; \mathsf{then}\; b\langle y\rangle) + \ldots$$

Program $(\mathsf{proc}\ length\ (x\ n)$
$\quad\mathsf{if}\quad x{=}nil\ \mathsf{then}\ n{=}0$
$\quad+\quad \exists y \exists z\ x{=}cons(y,z)\ \mathsf{then}\ (\mathsf{local}\ (m)\ (length\ z\ m) \parallel (inc\ m\ n))$

Analysis $length{\subseteq}\mathsf{proc}(arg_1{:}x, arg_2{:}n)\ \wedge$
$x{\subseteq}x_1 \cup x_2\ \wedge\ n{\subseteq}n_1 \cup n_2\ \wedge\ length{\subseteq}length_1 \cup length_2\ \wedge$
$x_1{=}nil\ \wedge\ n_1{=}0\ \wedge\ x_2{=}cons(y,z)\ \wedge$
$length_2[arg_1]u_1\ \wedge\ length_2[arg_2]u_2\ \wedge\ z{\subseteq}u_1\ \wedge\ m{\subseteq}u_2\ \wedge$
$inc[arg_1]v_1\ \wedge\ inc[arg_2]v_2\ \wedge\ m{\subseteq}v_1\ \wedge\ n{\subseteq}v_2$

Figure 5.4.: Analysing the Procedure Length.

The following example is similar and rejected for the same reason.

$$\mathsf{if}\ (a\langle x\rangle\ \mathsf{then}\ (x\ y)) + \ldots .$$

A particularity of our analysis is that we reject a conditional clause which is never executed because its guard is known to be inconsistent. For example:

$$\mathsf{if}\ (a\langle x\rangle \wedge b\langle x\rangle\ \mathsf{then}\ S) + \ldots$$

The Procedure Length. Figure 5.4 gives an OPM procedure that implements the function *length* and its analysis, in which we have dropped the existential quantifiers for better readability. We also assume that *inc* is statically known to be bound to a binary operation on integers; therefore we decide to treat *inc* as a constant symbol, in contrast to the variable *length*. This global type assumption can be expressed by the constraint

$$inc{\subseteq}\mathsf{proc}(arg_1{:}\mathsf{int}, arg_2{:}\mathsf{int})\,.$$

Simplification of the analysis of *length* yields the solved form

$$length{\subseteq}\mathsf{proc}(arg_1{:}x, arg_2{:}n)\ \wedge\ x{\subseteq}nil \cup cons(y,z)\ \wedge\ n \subseteq \mathsf{int}\,.$$

Its greatest solution for *length* is

$$\mathsf{proc}(arg_1{:}nil \cup cons(1,1), arg_2{:}\mathsf{int})\,.$$

where we write 1 instead of $\mathcal{P}(\mathcal{FT})$ for better legibility. The expected type of length is $\mathsf{proc}(\mathsf{list}(1), \mathsf{int})$ where $\mathsf{list}(X)$ is the greatest solution of the equation

$$L = \quad nil \cup cons(X, L)\,.$$

So the approximation is indeed correct.

Arity Constraints. We have used arity constraints at different places in the analysis. They are indispensable exactly where arity constraints syntactically occur in the program: that is, in tell statements where records are constructed and in conditional guards where records are decomposed.

Everywhere else, we could get rid of them. For example, to catch the arity mismatch in program D_{ar2}, the analysis does not necessarily require arity constraints. Rather, we could introduce another special feature ar and use the integers as labels to derive

$$A((\mathsf{proc}\ x\ (y_1 \ldots y_n)\ S)) = \quad \exists y_1 \ldots y_n\, (x \subseteq \mathsf{proc}(\mathsf{ar}{:}n, arg_1{:}y_1, \ldots, arg_n{:}y_n) \wedge A(S))$$

In this case, the analysis of D_{ar2} would entail that $\exists x' \exists x'' (x[\mathsf{ar}]x' \wedge 1\langle x' \rangle \wedge x[\mathsf{ar}]x'' \wedge 2\langle x'' \rangle)$ and hence $x{=}\emptyset$. Similarly, an additional label *none* and an analysis of feature selection of the form

$$A(x[f]y) = \quad x[f]y \wedge \neg \exists x'\, (x[\mathsf{ar}]x' \wedge none\langle x' \rangle)$$

could be used to justify rejection of $D_{typeerr2}$ without the need for arity constraints. A third option to treat this phenomenon is to derive $\neg\mathsf{proc}\langle x \rangle$ from $x[f]y$, and to assume that $a\langle x \rangle \wedge \neg\mathsf{proc}\langle x \rangle$ is satisfiable for all $a \neq \mathsf{proc}$ whereas, of course, $\mathsf{proc}\langle x \rangle \wedge \neg\mathsf{proc}\langle x \rangle$ is not. A more general treatment would assume an order on the labels, *e. g.*, a lattice (*cf.* [137]).

5.2.5. Style Conventions

We give an informal summary of the principles that underly our analysis. These principles can be seen as style conventions which a programmer should adhere when applying the analysis in order to avoid spurious error messages.

Every *good* program should

1. never reduce to a configuration with an inconsistent constraint store.
2. not contain statements that are unfailed only because they block forever.
3. contain only conditionals such that at least one of its clauses allows unfailed execution.
4. not contain procedures that are unfailed only because they are never applied.
5. contain no set of procedures which disagree about the values of their joint global variables.
6. not contain any guarded clause whose guard is inconsistent with its body.
7. contain only conditionals with consistent guards.

The first two principles should be familiar from Chapter 4 where we diagnosed failure or infinite suspension as run-time errors of CC programs. Underlying the clauses (3)–(5) is the decision not to accept any statement that is only correct because it is never executed: Conditionals with clauses that are all in conflict with the context (3) do not abide by this principle; neither do procedures with an inconsistent body (4), or multiple procedures that cannot be executed concurrently because they do not agree on their joint global variables (5). Principle (4) corresponds to our analysis of finite failure in the previous chapter: a finitely failed procedure inevitably fails *if* but *only if* it is applied. If Principle (5) is violated in a given program, then the program is unfailed only if some procedure is never applied. We want to prohibit this situation, justified by the intuition that a procedure should allow an arbitrary number of applications. Clauses (6) and (7) need further explanation.

Both Principles (6) and (7) reject a conditional clause (η then S) if the statement $\eta \| S$ is inconsistent with its context, independent of whether it is executed at all. This treatment of guards is clearly related to the unguarded approximation of Section 4.2. In addition, it implicitly introduces a program point for each conditional clause instead of just one for the complete conditional. In the CC-case, this corresponds to the introduction of auxiliary predicates as discussed in Section 4.1.1.4.

For programs that are hand-written by humans, the principles seem to be easy to obey. They may not be stable under program transformations such as partial evaluation or, more generally, they seem less convincing for automatically generated programs. In this case, however, it seems still worthwhile to report violation of these principles; in particular, since the programmer is always free to ignore the warnings and execute the program nonetheless.

5.3. Conditionals Revisited

Reconsider the *length*-example in Figure 5.4 and notice that $length_1$ is unconstrained. For this reason, the analysis cannot infer the expected type $\mathsf{list}(1)$ for the first argument. In the *length*-example, the variable z only occurs as an argument of the recursive application of *length*, z remains unconstrained, too. This weakness is due to the pessimistic operation of renaming *all* variables free in a conditional for the analysis of its clauses. It is pessimistic since it ignores the possibility of variables which have the same value during executions of all conditional clauses.

This section shows that it is important to exploit some data flow information in order to improve the accuracy of the analysis of conditionals. The next examples should illustrate this point.

Recursion and Data flow. Most recursive procedure on lists have a form similar to this one.

$$(\mathsf{proc}\ x\ (y)\ \mathsf{if}\ (y{=}nil\ \mathsf{then\ skip}) + (\exists z\ y[cdr]z\ \mathsf{then}\ (x\ z)))$$

Clearly, x is bound to a procedure whenever the second clause of the conditional is executed. This fact is crucial for the analysis of this statement to be as accurate as the

$$A(\mathsf{if}\ (\exists\overline{x}_1\eta_1\ \mathsf{then}\ S_1) + (\exists\overline{x}_2\eta_2\ \mathsf{then}\ S_2)) \quad = \quad \begin{array}{l}\exists\overline{y}_1\\ \exists\overline{y}_2\end{array}\left(\begin{array}{l}\overline{y}\subseteq\overline{y}_1\cup\overline{y}_2\ \wedge\\ A((\exists\overline{x}_1\eta_1\ \mathsf{then}\ S_1))[\overline{y}_1/\overline{y}]\ \wedge\\ A((\exists\overline{x}_2\eta_2\ \mathsf{then}\ S_2))[\overline{y}_2/\overline{y}]\end{array}\right)$$

$$\text{if } \{\overline{y}\} = non\mathcal{P}(\mathsf{if}\ (\exists\overline{x}_1\eta_1\ \mathsf{then}\ S_1) + (\exists\overline{x}_2\eta_2\ \mathsf{then}\ S_2))$$

Figure 5.5.: Analysing Conditionals with respect to Parameters

one for the corresponding first-order statement.

Procedure Calls. A similar weakness applies to all procedure calls that occur in only one of two branches of a conditional. For example, from the program

$$(\mathsf{proc}\, x\ (y)\ \mathsf{if}\ (\eta_1\ \mathsf{then}\ (p\ x\, y)) + (\eta_2\ \mathsf{then}\ (q\ x\, y)))$$

the analysis in Figure 5.3 will not deduce that x and y should denote integers – even if the analysis of the procedure definitions p and q yields that both are binary operations on integers.

The Base Case of a Recursion. The procedure *forall* applies its second argument as a unary procedure to every element in the list that it receives as the first argument.

$$\begin{array}{l}(\mathsf{proc}\, forall\ (xs, p)\\ \quad \mathsf{if}\quad xs{=}nil\ \mathsf{then\ skip}\\ \quad +\quad \exists x\exists xr\,(xs[head]x\wedge xs[tail]xr\ \mathsf{then}\ (p\ x)\parallel(forall\ xr\, p))\end{array}$$

The *intended* use of *forall* implies that p is bound to a unary procedure whenever the second clause of the conditional is executed. In general this cannot be guaranteed. In addition, if *forall* receives the empty list *nil* as the first argument it cannot fail, independent of the second argument. This is the case because the higher-order argument p is not used at the base case of a list recursion. Therefore, our analysis will not reject an application such as $(forall\ nil\ 42)$ as finitely failed.

In a first-order setting, this example is well-known to the logic programming community (*e. g.*, see [149]), where an analysis such as ours does not find out that the ternary *append*-procedure expects lists in the second and third arguments.

$$\begin{array}{l}(\mathsf{proc}\ append\ (xs, ys, zs)\\ \quad \mathsf{if}\quad xs{=}nil\ \mathsf{then}\ zs{=}ys\\ \quad +\quad \exists x\exists xr\, xs[head]x\wedge xs[tail]xr\ \mathsf{then}\\ \qquad\qquad \exists z\exists zr\,(zs[head]x\wedge zs[tail]zr\parallel(append\ xr\ ys\ zr))\end{array}$$

The analysis of conditionals can be improved a lot if (some) data flow information is statically known; that is, if it is known

$$
\begin{array}{l}
length \subseteq \mathsf{proc}(arg_1{:}x, arg_2{:}n) \wedge \\
x \subseteq x_1 \cup x_2 \ \wedge \ n \subseteq n_1 \cup n_2 \wedge \\
x_1 = nil \ \wedge \ n_1 = 0 \ \wedge \ x_2 = cons(y,z) \wedge \\
length[arg_1]u_1 \ \wedge \ length[arg_2]u_2 \ \wedge \ z \subseteq u_1 \ \wedge \ m \subseteq u_2 \ \wedge \\
inc[arg_1]v_1 \ \wedge \ inc[arg_2]v_2 \ \wedge \ m \subseteq v_1 \ \wedge \ n \subseteq v_2
\end{array}
$$

Figure 5.6.: Analysing the Procedure Length with respect to Parameters.

- on which variables it depends which clause a conditional is committed to (*tested*),
- and which variables may be constrained on execution of the clauses (*constrained*).

All other variables can be assumed constant for all executions of the conditional. Let us call these variables *parameters of the conditional*. The values of parameters are (neither conditions nor results) not related to the conditional branching of control, but they are simply accessed. Hence it is reasonable to adopt the

Parameter Principle. Parameters should not be renamed.

Figure 5.5 improves the analysis of conditionals accordingly, using $non\mathcal{P}(S)$ to denote the subset of $fv(S)$ containing all non-parameters of S. There are two important advantages in knowing conditional parameters.

1. The analysis derives fewer union constraints and more equalities instead. Hence the analysis becomes strictly more accurate.
2. The constraint solving becomes simpler. Since the treatment of inclusion constraints is substantially more expensive than that of equality constraints, the overall cost of constraint solving may drop significantly; this is in particular so, if we make the reasonable assumption that most variables occurring in a conditional are parameters.

It is clearly undecidable whether a variable in a conditional is a parameter since this depends on run-time properties of procedures. So we cannot hope for anything better in general than an approximation of parameter-hood. One obvious such approximation is this one:

Parameter Approximation. View those variables as parameters that occur only as procedure identifier in applications or as cell identifier in exchange statements.

$$A(x \text{ of } t \text{ then } S) \quad = \quad \exists\overline{x}(x{=}t \wedge A(S)) \text{if} \mathit{fv}(t) = \{\overline{x}\}$$

$$A(\text{case } x \text{ of } (t_1 \text{ then } S_1) + (t_2 \text{ then } S_2) \text{ return } \overline{y}) \quad =$$

$$\begin{array}{l}\exists x_1\\ \exists x_2\\ \exists \overline{y}_1\\ \exists \overline{y}_2\end{array} \left(\begin{array}{l} x{\subseteq}x_1 \cup x_2 \ \wedge \ \overline{y}{\subseteq}\overline{y}_1 \cup \overline{y}_2 \wedge \\ A(x_1 \text{ of } t_1 \text{ then } S_1)[\overline{y}_1/\overline{y}][x_1/x] \wedge \\ A(x_2 \text{ of } t_2 \text{ then } S_2)[\overline{y}_2/\overline{y}][x_2/x] \end{array}\right)$$

Figure 5.7.: Analysing Case Statements

Figure 5.6 shows the adapted analysis of the *length* procedure. The key improvement with respect to Figure 5.4 is shaded grey. Simplification of this constraint yields

$$\mathit{length}{\subseteq}\mathsf{proc}(\mathit{arg}_1{:}x, \mathit{arg}_2{:}n) \ \wedge \ \mathit{xs}{\subseteq}\mathit{nil} \cup \mathit{cons}(y,z) \ \wedge \ n{\subseteq}\mathsf{int} \ \wedge \ z{\subseteq}x\,.$$

The greatest solution for *length* of this constraint is

$$\mathsf{proc}(\mathit{arg}_1{:}\mathsf{list}(1), \mathit{arg}_2{:}\mathsf{int})$$

which, in this case, is exactly the expected type.

Syntactic Sugar. The parameter approximation above considers variables as non-parameters whenever they occur in constraints or in conditional guards. This is safe but rather pessimistic, and somewhat annoying given the central role that constraints play in OPM. Constraints are used both to construct and to decompose data structures. For example, the selection constraint $x[f]y$ constrains both x and y in general. Frequently, however, it is used as a selection function on x, assuming that x is bound to a record with field f. In this case, $x[f]y$ expresses a new constraint only on y. When used such, the variable x in $x[f]y$ is a parameter in a conditional like

$$\text{if } (\eta \text{ then } (\text{local } (y)\ x[f]y \parallel (y\ z))) + \dots,$$

but this is not acknowledged by the given approximation. As a second example, assume that the variables *map* and *fold* are bound to library procedures on lists. Then they behave as constants and hence are parameters in the following conditional even though they occur in an equational constraint.

$$\text{if } (x{=}\mathit{map} \text{ then } (x\ y_1\ y_2\ y_3)) + \dots$$

To further improve the analysis, we have three options.

Annotations. Enrich the syntax by special conditional forms (or other program annotations) which make intended parameters explicit. In functional programs, the data

$$
\begin{array}{lcl}
A((\mathsf{proc}\,x\;(y_1\dots y_n)\;S)) & = & \exists\overline{y}(x\subseteq\mathsf{proc}(\mathsf{n}{:}n, arg_1{:}y_1,\dots,arg_n{:}y_n)\wedge A(S))n \text{ fresh}\\
A((x\;y_1\dots y_n)) & = & \mathsf{proc}\langle x\rangle\wedge x\{\mathsf{n}, arg_1,\dots,arg_n\}\wedge\bigwedge_{i=1}^{n} y_i\subseteq x[arg_i]
\end{array}
$$

$$
A(\mathsf{if}\ (\exists\overline{x}_1\eta_1\ \mathsf{then}\ S_1)+(\exists\overline{x}_2\eta_2\ \mathsf{then}\ S_2)) =
$$

$$
\exists\overline{y}_1\exists\overline{y}_2\left(\begin{array}{l}
\overline{y}\subseteq\overline{y}_1\cup\overline{y}_2\ \wedge\\
A((\exists\overline{x}_1\eta_1\ \mathsf{then}\ S_1))[\overline{y}_1/\overline{y}]\ \wedge\\
A((\exists\overline{x}_2\eta_2\ \mathsf{then}\ S_2))[\overline{y}_2/\overline{y}]\ \wedge\\
\bigwedge_{y\in\overline{y}}\ \mathsf{isdef}(y)\ \mathsf{then}\ y{=}y_1\wedge y{=}y_2
\end{array}\right)
$$

$$
\text{where } \mathsf{isdef}(x) = \exists y(x[\mathsf{n}]y\wedge\mathsf{isname}(x))
$$

$$
\text{and } \{\overline{y}\} = fv(\mathsf{if}\ (\exists\overline{x}_1\eta_1\ \mathsf{then}\ S_1)+(\exists\overline{x}_2\eta_2\ \mathsf{then}\ S_2))
$$

Figure 5.8.: Analysing Conditionals with Automated Parameter Detection

flow through conditionals is statically clear. In OPM this is not the case; instead, one could let the programmer provide (unchecked) data flow information by marking the intended return parameters explicitly.

$$\mathsf{case}\ x\ \mathsf{of}\ (t_1\ \mathsf{then}\ S_1)+(t_2\ \mathsf{then}\ S_2)\ \mathsf{return}\ \overline{y}$$

Given such annotations, all variables except x and $\overline{y}$ can be treated as parameters. The corresponding analysis is given in Figure 5.7. Notice that the annotation $\mathsf{return}\ \overline{y}$ is essential, because the variables constrained by a conditional are not syntactically determined either.

Conditional Constraints. Detect parameters during constraint solving by means of conditional equations. A solution for the special case of procedures is given in Figure 5.8 using names. The definition of the predicate isdef assumes another predicate $\mathsf{isname}(x)$ that holds exactly for names.

Control Flow Analysis. Determine conditional parameters by a *control flow analysis*. The design of a full-fledged control flow analysis for Oz is an interesting research topic of its own.

5.4. Related Work

5.4.1. Programming Languages and Models

The history of models for concurrent computation reaches back into the 60's and 70's to the net theory of Petri [163], and to work on buffered communication between sequential processes by Dijkstra [61], Brinch-Hansen [28], and Kahn [111]. The more recent development of high-level models for concurrent computation and programming can be summarised by two main lines of research: one of them is based on process calculi and the other one on the concurrent constraint model.

5.4.1.1. Process Calculi

Process calculi and process algebras provide a *message passing* model of concurrent computation (see [126] for references). Most influential amongst them is the π-calculus by Milner, Parrow, and Walker [128, 129]. The π-calculus generalises and simplifies Milner's CCS [124, 125], the *Calculus of Communicating Systems*. It also draws intuitions from Hewitt and Agha's actor model [3, 96] that formulates the early vision of concurrent computation as organised in terms of concurrent processes (called *actors*) that communicate freely by exchanging messages. CCS is influenced by Hoare's language CSP [99, 100] of *Communicating Sequential Processes*, on which the communication models of occam [30] are based.

In the π-calculus, messages are received along channels, and channels can be passed as messages. This allows the π-calculus to express process mobility and to model dynamically reconfigurable networks of processes (that is, new processes can be created dynamically and then be communicated with). It also subsumes the λ-calculus, one of the most important models of deterministic computation [127]. Channel communication is synchronous in that both sender and receiver will block until a message has been exchanged. For the development of programming languages, asynchronous versions of the π-calculus [24, 102] (where the sender does not block) have been considered. For instance, the languages Pict and join calculus [67, 169] are based on an asynchronous versions of the π-calculus.

5.4.1.2. (Constraint) Logic Programming

Logic programming is based on the vision of computation as deduction [114], and took some of its original motivation from an application in natural language processing [49]. Logic programs are interpreted as predicate logic formulas from the Horn clause fragment and operationalised by SLD resolution and backtracking search. The language Prolog is almost synonymous with the logic programming paradigm. A key contribution of logic programming to the field of programming is the concept of *partially determined data structures*, that is data structures with embedded (logic) variables which act as place holders for unknown values. As computation proceeds, the logic variables are further instantiated via unification, so that the data structures get more and more refined.

Logic programming was developed further in two directions. Jaffar and Lassez [104] defined the *constraint logic programming* scheme CLP(X) which parametrises logic programming over a constraint system X while retaining most of its properties. In CLP(X), unification is generalised to *satisfiability* checking and solving of constraints for the constraint system X. This parameterisation made a variety of new data structures available in logic programming, by way of new constraint systems over numbers (integers and reals), booleans, trees (infinite trees, feature trees), sets, and others. This greatly enhanced the usability and the efficiency of logic programming in problem solving. For entry points into the related research see [23, 105, 208].

5.4.1.3. Concurrent Logic Programming

Another line of research took off from the insight that logic variables are an expressive concept to model complex communication and synchronisation patterns in concurrent programming. For instance, by synchronising on a logic variable to become bound one can express data driven computation as considered in data-flow languages [57]. This expressiveness was first recognised in Relational Language [47]. Subsequently, it led to the development of a plethora of concurrent logic programming languages [189], in particular with tailwind from ICOT's decision to use a concurrent logic programming language for their ambitious Fifth Generation Project.

Concurrent logic programming gave up the identification of computation and deduction. The speculative exploration of alternatives with backtracking search ("don't know" non-determinism) was replaced by synchronisation and *committed choice* ("don't care" non-determinism). Committed choice means that the choice of one alternative branch of computation cannot be retracted. A variety of synchronisation patterns were proposed and operationally specified, some of them quite involved [189]. In 1987, Maher [120] made a breakthrough in showing that *entailment* between constraints was the logic concept underlying these synchronisation schemes. This established a unifying logic view on concurrent control with logic variables and enabled a reconciliation of constraint logic programming with concurrent logic programming.

5.4.1.4. Concurrent Constraint Programming

Based on Maher's insight and influenced by process calculi such as CCS [124, 125], Saraswat developed the framework of concurrent constraint programming [180]. Syntactically, CC gave up the restrictive clausal syntax from Prolog and adopted a more flexible compositional syntax instead, which was influenced by CCS. Conceptually, it contributed the organisation of concurrent computation in terms of multiple agents which interact with each other by means of constraints imposed on *shared logic variables* and placed in the so-called *constraint store*. The basic operations on the constraint store are imposing ("telling") new constraints on the variables and testing ("asking") for the presence of constraints. The attempt to tell a constraint which is inconsistent with the constraint store is to a run-time error. Hence, the tell operation requires a satisfiability test, while the ask operation is modelled by entailment checking.

Concurrent processes synchronise on the fact that certain constraints on a variable become available in the constraint store. This allows for complex synchronisation conditions to be expressed easily and, since constraints are never deleted, it yields monotonic synchronisation conditions. Thus communication through shared variables is a reliable and high-level concept in CC. Dynamically reconfigurable networks can be expressed without reverting to the indeterministic concept of channel communication as process calculi.

Before CC arrived, research in constraint (logic) programming had led to the proposal of various delay primitives, which added a weak form of concurrency ("coroutining"). Delay primitives were pioneered by Colmerauer with Prolog II and Naish with Mu- and Nu-Prolog [51, 148] and are present in all modern Prolog systems today. Their concurrent control regime had proven beneficial for the writing of new constraint solvers. The CC framework gives a simple explanation for them and opens up additional flexibility for the development of new constraint solvers. Programming languages and notations based on the CC model include cc(FD), AKL, and Oz [108, 174, 207].

5.4.1.5. Operational Models for Oz

Various aspects of Oz have been investigated on the basis of small calculi. Smolka [194] defines the γ-calculus and relates it to the π-calculus as well as to the eager and the lazy λ-calculus calculus. Smolka also shows how to model concurrent objects in the γ-calculus. Niehren and Müller define the ρ-calculus which extends the γ-calculus by parametrising it with a constraint system, and prove that ρ properly contains the "applicative core" of the π-calculus [154]. A ρ-calculus over ordering constraints between variables has been considered in [136]. Niehren investigates the δ-calculus and proves that it can adequately embed both the eager and the lazy λ-calculus [151, 152]. He also shows how to embed the complete π-calculus into δ. All these calculi exclude constraint inference features. For entry points to these issues see [174, 182–184]. Details on the object model of Oz are presented in [91, 194]. Names in Oz have been inspired by the concept of naming in the π-calculus [128, 129]. The interaction of constraint systems with names is discussed in [154, 155]. For further details of its practical issues, notably in the object system, see [91, 184].

Recently, Victor and Parrow have proposed the fusion calculus [160, 161] as a simplification of the π-calculus that should, at the same time, allow its extension by constraint programming features such as variable equations. Research in the fusion calculus is driven by the study of program equivalences and not by programming desirables (compare also Section 6.4.1).

5.4.1.6. Concurrent Functional Languages

There are various proposals for concurrency extensions of functional languages. The ones related to logic variables include the futures in Multilisp [81] and the I-structures in Id [18]. The functional language Erlang [17] supports message-based communication (somewhat similar to the π-calculus). Futures, I-structures, and logic variables have in common that they provide a place holder for a data structure that is to be computed concurrently. They differ in how they deal with multiple assignment. Futures enforce single assignment syntactically, logic variables as in Oz combine multiple "assignments" to the same variable by unification, and I-structures raise a run-time error on second assignment (so does Plain). In contrast to a future, a logic variable can be created independently from the process that will bind it. In contrast to logic variables and similar to channels, I-structures require explicit operations to access the data.

Also, Id's M-structures [22] and OPM's cells are related. M-structures are updatable containers that can be full or empty. Reading from an empty M-structure blocks the reader, and writing to a full M-structure is a run-time error. The read and write operations on M-structures are not atomic. In contrast, cells in OPM hold logic variables. An exchange operation replaces the current content of a cell by a new one in an atomic operation. This guarantees mutual exclusion of multiple concurrent operations on the same cell which is crucial for computation with state. Once the cell is available, operations on it are unsynchronised. Thereby, the access to variables in a cell is decoupled from the synchronisation between producer and consumer of constraints on these variables. The presence of logic variables is essential here: one can put a logic variable in a cell and compute the new value afterwards.

5.4.2. Program Analysis

Set-based analysis for higher-order programming languages has received some attention recently in the context of functional languages [12, 38, 65, 84, 121, 216, 218]. Set-based analysis for constraint programming has, to the best of my knowledge, not been investigated so far. We briefly comment on the most closely related program analysis systems, and we add a remark on the constraint systems used there in contrast to the one we use.

5.4.2.1. Program Analysis Systems

Bourdoncle [26] investigates *abstract debugging* for imperative languages in the framework of abstract interpretation. Abstract debugging analyses a program with respect to so-called *invariant* and *intermittent* assertions; invariant assertions must always hold at a given program point, and intermittent assertions must hold eventually. Invariant assertions can be used to derive sufficient conditions for a program to fail at some point.

Flanagan's MrSpidey [65] is a static debugger for Scheme and part of the programming environment DrScheme [63]. MrSpidey approximates the data flow in Scheme programs and derives set expressions for every program point in order to prove that no run-time error will occur at certain program points. MrSpidey's main goal is the static detection of errors.

Wright's Soft Scheme [38, 216, 218] is the precursor of MrSpidey at Rice. Soft typing for Scheme tries to eliminate all run-time type checks which it can prove to succeed. It also reports to the programmer program points that will necessarily fail if they are reached at all. Wright shows that all run-time type errors in a checked program will be caught by one of the remaining type checks.

Aiken and Wimmers [11, 12] develop a soft type system for FL [9] based on a very expressive set constraint system with union, intersection, and complement They give an interpretation of their constraints in a domain of types, essentially the standard ideal model [119] for functional types. Aiken has developed a demonstrator version of their analysis for the experimental functional language Illyria [4].

Wadler and Marlow present a type system for the first-order fragment of Erlang [121], a functional language with server-based concurrency. Their system uses subtyping constraints based on a simplified version of Aiken and Wimmers's system [11].

Heintze [84] proposes a set-based analysis for the functional language ML. His analysis is a global program analysis and cannot be used to analyse programs module-wise.

Aiken and colleagues develop BANE [6] as a tool box for constraint-based analysis of different programming languages, including ML and Java. BANE is based on a mixture between set and tree constraints that has, for instance, beeb used for an analysis of unhandled exceptions in ML [62].

5.4.2.2. Covariant Ordering Constraints

Our failure diagnosis for OPM is based on the same set of constraints that we employed for a concurrent constraint language. More technically speaking, our analysis of OPM is based on constraints interpreted over sets of feature trees with a fully monotonic (covariant) order. This is in contrast to most other analyses for languages with higher-order procedures, in particular with all work mentioned above (except for Bourdoncle's analysis for Pascal which is first-order).

Most constraint systems used for the set-based analysis for higher-order procedures are interpreted over an ordering that is monotonic or antimonotonic depending on the top-level constructor (usually $\rightarrow$) or the tree selectors (such as *dom* and *rg*). The co-variant ordering is used to deal with output arguments of procedures, the contravariant ordering deals with input arguments.

Since the data flow through OPM procedures is not statically apparent (in contrast to procedures in functional languages) we must treat input and output arguments alike. As a consequence, we lose much information along higher-order functional data flow. On the other hand, the analysis of multiple applications of the same procedure is kept fully separate.

6. Typed Concurrent Programming with Logic Variables

This chapter presents a concurrent higher-order programming language called Plain. The main design objectives for this language have been that it should be a close relative of OPM that

- has an expressive strong type system with record-based subtyping and higher-order polymorphic types (in contrast to ML type schemes), but
- retains most of OPM's expressiveness as far as concurrent, functional, and object-oriented programming are concerned.

Higher-order polymorphic types are required for cells that contain polymorphic data structures and they are convenient for data structures that embed polymorphic procedures (*e. g.*, to describe modules). They are particularly useful in a distributed setting (*e. g.*, see [209]) since they enable one to send polymorphic procedures along a channel, an idiom which is ill-typed in ML. Higher-order polymorphic types are also required to type check certain higher-order programming abstractions, for instance in the context of typed object-oriented programming (*e. g.*, see [166]); there, the *combination* of higher-order polymorphic types with subtyping is especially convenient [1].

As it turns out, both design objectives can be met by simplifying OPM's store model such that it does not contain equality constraints between variables and hence does not require equational constraint solving (*i. e.*, unification). In Plain, the abstract constraint store of OPM is replaced by a more detailed and self-contained store model, and equational constraints are replaced by *(single) assignment*

$$x := y\,.$$

Execution of $x := y$ does not assert equality between x and y (and unify their current bindings) but blocks until y is bound to some data structure D and then *binds* x to the same D, too. The decision to give up unification makes Plain a considerable restriction of Oz as a constraint programming language, in particular with respect to tree constraints. But Plain still admits computation with partially determined data structures such as records with embedded logic variables.

$$(\mathsf{local}\ (y,\ z)\ x := \{head{:}y,\ tail{:}z\} \parallel \ldots)$$

This retains important expressiveness of logic variables, including the following: The possibility to express cyclic data structures, demand-driven computation and data flow computation [152, 170], safe (monotonic) synchronization in concurrent programming [189], and latency tolerant communication in concurrent and distributed programming [209]; it also includes the implementation of tail recursive procedures returning lists which is impossible in functional programming languages (*e. g.*, [131]).

Plain's type system employs record-based subtyping and higher-order universal polymorphism [76, 177]. It also features access modalities (modes), which have been introduced for channels by Pierce and Sangiorgi [165]; we show that one can adapt their system to a language with logic variables. Modes for logic variables are essential to make the type system work. Neither constraints on logic variables nor procedures in constraint programming impose a static distinction between input and output (even though it is often made implicitly). However, this distinction is essential for every type system that provides a non-trivial order on types, such as the subtyping order and the instantiation order on polymorphic types: it must be possible to use the procedure output of a (more specific, smaller) subtype as input of a (less specific, greater) supertype, and instantiation of polymorphic types must occur along the data flow [136].[37]

Static typing in a system with ordered types requires that the data flow is statically known. In functional languages this is ensured by the restriction to applicative syntax; it is not the case for OPM where equational constraints $x = y$ are a central computational concept; equational constraints do not have a notion of input and output. In order to solve this technical problem, Plain replaces equational constraints by an assignment statement; while assignment remains an equation semantically, it has a statically fixed input/output behaviour. Plain adapts also the operational semantics of the OPM primitives such that they use assignment instead of equational constraints; in particular, such a modification was needed for the semantics of cells. We do not consider type

[37]Note in passing that one can provide OPM with an ML-style polymorphic type system subject to Wright's restriction of polymorphic generalisation [123, 217]. This system does not require static data flow information, but it rules out many higher programming abstraction, *e. g.*, in the object system. For preliminary results on this topic see [138].

inference for Plain, which is very likely to be undecidable [214]; for further details see Section 6.4.1.3.

The changes in OPM that lead to Plain can be understood as adding a static notion of input and output to a language that, due to its nature, does not make this distinction explicit. Some of these additions affect the operational semantics (assignment) and some the type system (mode discipline). We show that these changes suffice to adapt standard strong type systems with an order on types to OPM, *i. e.*, to a language with logic variables and higher-order procedures.

Plain's expressiveness is also well-comparable with Pict [169], a recent concurrent programming language based on the π-calculus [128, 129]. So Plain's design also contributes to recent efforts [152–154, 160, 161, 194, 211] to relate the concurrent programming models based on constraints with those based on channel communication. Plain seems to be the first typed concurrent programming language with higher-order procedures and logic variables, with Id [18] being its closest relative in this respect: So Plain is also of interest as an instance of what Harper has called the HOT style of programming (higher-order, typed), extended to deal with logic variables [212].

6.1. Plain

6.1.1. Untyped Plain

Plain inherits the computational setup from the Oz Programming Model, which is given by concurrent threads that communicate and synchronise through a shared store. In contrast to OPM, Plain considerably simplifies the store model. Plain's store binds variables to data structures but does not contain explicit equations between unbound variables. The data structures may contain embedded logic variables, of which typically only some are bound. Thus, Plain accommodates *partially determined* data structures. The unbound variables in such a record can serve as the communication medium between concurrent threads.

6.1.1.1. Statements, Store, and Configurations

The abstract syntax of Plain is given in Figure 6.1. A *data structure D* is a procedure, a record, or a cell. A *record* $\{\bar{a}{:}\bar{y}\}$ has *fields* $\bar{y}$ at pairwise distinct *features* $\bar{a}$. We use the same notational conventions as for feature terms.[38]

[38]This notion of records deviates from OPM where records carry labels. This is not an essential difference but brings Plain closer to conventional programming languages with records, in particular with functional languages. It is mostly due to Oz's heritage in logic programming that records have labels in OPM.

$$
\begin{array}{llllll}
\textbf{Data Structures} & D & ::= & (\mathsf{proc}\ (\overline{y})\ S) & & \text{(Procedure)} \\
 & & | & \{\overline{a}{:}\overline{x}\} & & \text{(Record)} \\
 & & | & (\mathsf{cell}\ x) & & \text{(Cell)} \\
\textbf{Expressions} & E & ::= & x & & \text{(Variables)} \\
 & & | & D & & \text{(Data Structures)} \\
\textbf{Statements} & S & ::= & x := E & & \text{(Assigment)} \\
 & & | & (x\ \overline{y}) & & \text{(Application)} \\
 & & | & (\mathsf{case}\ x\ (\overline{a}{:}\overline{y}\ldots)\ S) & & \text{(Pattern Matching)} \\
 & & | & (\mathsf{exch}\ x\ y\ (z)\ S) & & \text{(Exchange)} \\
 & & | & (\mathsf{local}\ (\overline{x})\ S) & & \text{(Declaration)} \\
 & & | & S_1 \parallel S_2 & & \text{(Parallel Composition)} \\
 & & | & \mathsf{skip} & & \text{(Null Statement)} \\
\textbf{Configurations} & C & ::= & V\rho\sigma[]\ S & & \\
\textbf{Variables} & V & \subseteq & \mathcal{V} & (V \text{ a finite set}) & \\
\textbf{Store} & \sigma & : & \mathcal{V} \rightharpoonup \mathcal{D} \cup \mathcal{N} & & \\
\textbf{Reference Store} & \rho & : & \mathcal{N} \rightharpoonup \mathcal{D} & &
\end{array}
$$

Figure 6.1.: Syntactic and Semantics Objects of Plain

consistent renaming of bound variables (α)

$S_1 \parallel S_2 \equiv S_2 \parallel S_1 \qquad (S_1 \parallel S_2) \parallel S_3 \equiv S_2 \parallel (S_2 \parallel S_3) \qquad \mathsf{skip} \parallel S \equiv S \equiv S \parallel \mathsf{skip}$

$\{\ldots a_1{:}y_1 \ldots a_2{:}y_2 \ldots\} \;\equiv\; \{\ldots a_2{:}y_2 \ldots a_1{:}y_1 \ldots\}$

$(\mathsf{case}\; x\; (\ldots a_1{:}y_1 \ldots a_2{:}y_2 \ldots\ldots)\; S) \;\equiv\; (\mathsf{case}\; x\; (\ldots a_2{:}y_2 \ldots a_1{:}y_1 \ldots\ldots)\; S)$

Figure 6.2.: Structural Congruence of Plain

The statements are all known from OPM with two exceptions: the *assignment* $x := E$ of an expression E to a variable x, and the *cell exchange* $(\mathsf{exch}\; x\; y\; (z)\; S)$ on variable x with arguments y and z, and with continuation S. In contrast to the cell exchange statement of OPM, Plain's has a variable binder: in $(\mathsf{exch}\; x\; y\; (z)\; S)$, the variable z is bound within S.[39] For technical simplicity, we restrict case statements to contain a single clause only.[40]

We write the set of procedures, cells, and records as $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{R}$, respectively. The sets of data structures $\mathcal{D}$ is defined as their union: $\mathcal{D} = \mathcal{P} \cup \mathcal{C} \cup \mathcal{R}$. Recall that $\mathcal{V}$ and $\mathcal{N}$ denote the set of variables and names. A *store* is a pair $\rho\sigma$ of finite partial functions where σ maps variables to data structures or names, and ρ maps names to data structures. We require that $\rho(\sigma(x))$ is defined when $\sigma(x) \in \mathcal{N}$. We write $\rho_\perp$ and $\sigma_\perp$ for the totally undefined store functions. If $x \in dom(\sigma)$, then we say that x is *bound* in the store σ. If $\sigma(x) \in \mathcal{D}$, we say x is *bound to* $\sigma(x)$, if $\sigma(x) \in \mathcal{N}$ say that x is *bound to a cell* with current contents $\rho(\sigma(x))$. The free variables $fv(\sigma)$ and $fv(\rho)$ of a store are defined as follows.

$$fv(\sigma) \;=_{def}\; dom(\sigma) \cup fv(rg(\sigma))$$
$$fv(\rho) \;=_{def}\; fv(rg(\rho))$$

The *monotonic extension* of a store σ by a new binding of x to d is written $\sigma, x{\mapsto}d$ and defined by

$$\sigma, x{\mapsto}d =_{def} \begin{cases} \sigma[d/x] & \text{if } x \notin dom(\sigma) \\ \sigma & \text{otherwise} \end{cases}$$

An extension "of the empty store" such as $\sigma_\perp, x{\mapsto}d$ for $d \in \mathcal{D} \cup \mathcal{N}$ is abbreviated to just $x{\mapsto}d$ by dropping $\sigma_\perp$. Note that this is not the standard notion of extension for

[39]For further discussion on assignment and exchange, the statements in Plain that differ from OPM, see Section 6.3.1.

[40]In practice, this restriction is not possible since it restricts the expressiveness of case statements to that of field selection on records. The type checking of multiple-clause case statements can be added based on standard machinery, usign type constructors and variant types [34].

$$V\rho\sigma\,[]\ (\mathsf{local}\ (\overline{x})\ S) \ \longrightarrow\ V{\cup}\{\overline{x}\}\ \rho\sigma\,[]\ S \qquad \text{if } x \notin V \qquad (\textsc{Declare})$$

$$V\rho\sigma\,[]\ x{:=}y \ \longrightarrow\ V\rho\ \sigma, x{\mapsto}\sigma(y)\,[]\ \mathsf{skip} \qquad \text{if } y \in dom(\sigma) \qquad (\textsc{AssVar})$$

$$V\rho\sigma\,[]\ x{:=}\{\overline{a}{:}\overline{y}\} \ \longrightarrow\ V\rho\ \sigma, x{\mapsto}\{\overline{a}{:}\overline{y}\}\,[]\ \mathsf{skip} \qquad (\textsc{AssRec})$$

$$V\rho\sigma\,[]\ x{:=}D \ \longrightarrow\ V\rho[D/n]\ \sigma, x{\mapsto}n\,[]\ \mathsf{skip} \qquad (\textsc{AssData})$$
$$\text{if } D \in \mathcal{P} \cup \mathcal{C}, n \notin dom(\rho)$$

$$V\rho\sigma\,[]\ (x\ \overline{y}) \ \longrightarrow\ V\rho\sigma\,[]\ S[\overline{y}/\overline{z}] \qquad (\textsc{Apply})$$
$$\text{if } \rho(\sigma(x)) = (\mathsf{proc}\ (\overline{z})\ S)$$

$$V\rho\sigma\,[]\ (\mathsf{case}\ x\ (\overline{a}{:}\overline{y}\ldots)\ S) \ \longrightarrow\ V\rho\sigma\,[]\ S[\overline{z}/\overline{y}] \qquad \text{if } \sigma(x) = \{\overline{a}{:}\overline{z}\ldots\} \qquad (\textsc{Match})$$

$$V\rho\sigma\,[]\ (\mathsf{exch}\ x\ y\ (z)\ S) \ \longrightarrow\ V\rho[(\mathsf{cell}\ y)/\sigma(x)]\ \sigma\,[]\ S[z'/z] \qquad (\textsc{Exchange})$$
$$\text{if } \rho(\sigma(x)) = (\mathsf{cell}\ z')$$

$$\frac{V\rho\sigma\,[]\ S_1 \ \longrightarrow\ V'\rho'\sigma'\,[]\ S_2}{V\rho\sigma\,[]\ S_1 \parallel S \ \longrightarrow\ V'\rho'\sigma'\,[]\ S_2 \parallel S} \qquad (\textsc{Closure})$$

Figure 6.3.: Operational Semantics of Plain

partial functions: It is allowed to extend a store σ by a binding for a variable that σ already binds, but in this case the extension has no effect. This implies that extension preserves the bindings in the store: Hence, it is called “monotonic”.

A *configuration* is a tuple $V\rho\sigma\,[]\ S$ consisting of a statement S, a store $\rho\sigma$, and a set V of (dynamically created) variables such that $fv(S) \cup fv(\sigma) \cup fv(\rho) \subseteq V$ holds. With every configuration $V\rho\sigma\,[]\ S$ we associate a statement $S(C)$ that represents the bindings of $\rho\sigma$ in terms of assignments and extends S accordingly. The *statement $S(C)$ associated with a configuration C* is defined as follows.

$$S(V\rho\sigma\,[]\ S) =_{def} \quad S \wedge \textstyle\bigwedge_{\sigma(x)\in\mathcal{D}} x{:=}\sigma(x) \ \wedge\ \bigwedge_{\sigma(x)\in\mathcal{N}} x{:=}(\mathsf{cell}\ \rho(\sigma(x)))$$

6.1.1.2. Operational Semantics

We identify statements S, data structures D, and configurations C up to consistent renaming of bound variables and we assume once and for all that bound variables in all S, D, or C are pairwise distinct and distinct from the free variables. Furthermore, we identify S, D, and C up to the *structural congruence* given in Figure 6.2. Parallel composition of statements is commutative, associative, and has the neutral element skip. Records $\{\overline{a}{:}\overline{y}\}$ and patterns $(\overline{a}{:}\overline{y}\ldots)$ are identified up to reordering of their fields. Two configurations C and C' are congruent if their associated statements are: formally, $C \equiv C'$ if and only if $S(C) \equiv S(C')$.

The operational semantics of Plain is defined in terms of a one-step reduction relation on configurations. *Reduction* $\longrightarrow$ is defined as the smallest binary relation on configurations that satisfies the axioms in Figure 6.3 and is closed under the inference rule (CLOSURE).

Declaration. Reduction of variable declaration $(\mathsf{local}\ (\overline{x})\ S)$ is unsynchronised; provided the declared variables $\overline{x}$ do not occur in the current configuration, they are added to the set of used variables and $(\mathsf{local}\ (\overline{x})\ S)$ is replaced by S. This rule may require renaming of the declared variable before reduction.

Assignment. There are three rules for *assignment* $x{:=}E$ depending on the expression E on the right hand side. The assignment $x{:=}y$ of a *variable* y to x waits for y to be bound in the current store, and then extends the store by the binding of x to $\sigma(y)$. Reduction of an assignment $x{:=}D$ where D is a *record* $\{\overline{a}{:}\overline{x}\}$ directly extends the store by binding x to $\{\overline{a}{:}\overline{x}\}$. Reduction of $x{:=}D$ where D is a *cell* or a *procedure* first creates a fresh name m. Then the store is extended by binding x to n and n to y. In OPM, these assignment forms correspond to procedure and cell definition. The following example illustrates declaration and assignment.

$$
\begin{aligned}
&\{x\}\rho_\perp\sigma_\perp\,[]\ (\mathsf{local}\ (y)\ y{:=}\{a{:}x\} \parallel x{:=}y)\\
&\qquad\qquad\longrightarrow\quad \{x,y\}\rho_\perp\sigma_\perp\,[]\ y{:=}\{a{:}x\} \parallel x{:=}y\\
&\qquad\qquad\longrightarrow\quad \{x,y\}\rho_\perp\ y{\mapsto}\{a{:}x\}\,[]\ x{:=}y\\
&\qquad\qquad\longrightarrow\quad \{x,y\}\rho_\perp\ y{\mapsto}\{a{:}x\}, x{\mapsto}\{a{:}x\}\,[]\ \mathsf{skip}
\end{aligned}
$$

In this example, x is bound to $\{a{:}x\}$ such that a *cyclic record* is constructed.

Application. Reduction of an application $(x\ \overline{y})$ synchronises on the fact that the store binds x to a procedure $(\mathsf{proc}\ (\overline{z})\ S)$; then, it replaces the application with the procedure body $S[\overline{y}/\overline{z}]$ with the actual arguments replacing the formal ones.

Pattern Matching. A matching statement $(\mathsf{case}\, x\ (\overline{a}{:}\overline{y}\ldots)\ S)$ synchronises on the fact that the store binds x to a record that matches the pattern $(\overline{a}{:}\overline{y}\ldots)$. We say that a record *matches* a pattern $(\overline{a}{:}\overline{y}\ldots)$ if it has at least the features in $\overline{a}$, that is, if it is of the form $\{\overline{a}{:}\overline{z}\}$. A special case of matching is field selection on records. For instance, consider the following reduction.

$$
\begin{aligned}
&\{x,y\}\rho\ y{\mapsto}\{a{:}x,b{:}y\}\,[]\ (\mathsf{case}\ y\ (b{:}z\ldots)\ x:=z) \\
&\qquad\longrightarrow\quad \{x,y\}\rho\ y{\mapsto}\{a{:}x,b{:}y\}\,[]\ x:=y \\
&\qquad\longrightarrow\quad \{x,y\}\rho\ y{\mapsto}\{a{:}x,b{:}y\},x{\mapsto}\{a{:}x,b{:}y\}\,[]\ \mathsf{skip}
\end{aligned}
$$

Cell Exchange. A cell exchange $(\mathsf{exch}\ x\ y\ (z)\ S)$ synchronises on the fact that x is bound to a cell, say with current content z'. If this is the case, the store is updated at the name $\sigma(x)$ to point to the new cell content y, and then the exchange statement is replaced by the continuation S in which the former content z' is substituted for the bound variable z. For instance, let $V = \{x,y,y'\}$, fix a name n such that $\rho(n) = (\mathsf{cell}\ y')$ and consider the following example.

$$
\begin{aligned}
&V\rho\ x{\mapsto}n\,[]\ y:=x \parallel (\mathsf{exch}\ x\ y\ (z)\ (\mathsf{exch}\ x\ z\ (z')\ \mathsf{skip})) \\
&\qquad\longrightarrow\quad V\rho\ x{\mapsto}n,y{\mapsto}n\,[]\ (\mathsf{exch}\ x\ y\ (z)\ (\mathsf{exch}\ x\ z\ (z')\ \mathsf{skip})) \\
&\qquad\longrightarrow\quad V\rho[(\mathsf{cell}\ y)/n]\ x{\mapsto}n,y{\mapsto}n\,[]\ (\mathsf{exch}\ x\ y'\ (z')\ \mathsf{skip}) \\
&\qquad\longrightarrow\quad V\rho[(\mathsf{cell}\ y)/n][(\mathsf{cell}\ y')/n]\ x{\mapsto}n,y{\mapsto}n\,[]\ \mathsf{skip} \\
&\qquad=\quad V\rho\ x{\mapsto}n,y{\mapsto}n\,[]\ \mathsf{skip}
\end{aligned}
$$

6.1.2. Typed Plain

In this section we present a type system for Plain. This type system is inspired by the one that Pierce and Turner give for Pict [169] which in turn rests on a long tradition of type systems for functional languages.[41]

6.1.2.1. Types are Protocols

In the concurrent setting, types are appropriately viewed as *protocols*. The communication of concurrent threads with each other through the shared store is mediated by logic variables. For this communication to work smoothly there must be consensus between the threads on the access protocols for the shared variables. These protocols include two kinds of information:

- Structural: "Which data structures may a variable be bound to?"
- Modal: "Is it legal to read from and/or write to a variable?"

[41] For excellent overviews of type systems for programming languages see the classical paper by Cardelli and Wegner [37], and Cardelli's more recent handbook article [34].

Types are a means to describe such access protocols for variables. Typical types[42] include these ones:

$?\mathsf{int}$**:** grants the right to read a variable and guarantees that reading will yield an integer; denies write access, that is the right to bind a variable.

$!\mathsf{int}$**:** grants the right to bind the variable to an integer; denies read access.

$\hat{}\mathsf{int}$**:** grants the right to read integers from a variable and to write integers to it.

$?\{a{:}T\}$**:** grants the right to read a variable and guarantees that reading will yield a record that has at least the feature $\bar{a}$. Furthermore, it is guaranteed that the selection of the field at a will yield a variable with type T. Write access is denied.

$!\{a{:}T_1, b{:}T_2\}$**:** grants the right to bind a variable to any record that has at least the features a and b, provided their associated fields have types T_1 and T_2.

$!(\mathsf{proc}\ ?\mathsf{int})$**:** grants the right to bind a variable to a procedure that can safely be applied to variables of type $?\mathsf{int}$.

$?\forall\alpha.(\mathsf{proc}\ ?\alpha\ !\mathsf{int})$**:** grants the right to read a procedure from a variable, and apply it to all pairs of arguments of which the first provides read access, and the second one allows writing an integer.

We write $x{:}T$ for the *assumption* that variable x has type T. Type assumptions for multiple variables are grouped in so-called type environments Γ. *Type checking* is protocol validation: namely, the process of verifying that a given type environment Γ is *respected* by a configuration C and all configurations one obtains by reduction from C; we write this as $\Gamma \triangleright C$.

Subtyping defines an order $\Gamma \preceq \Gamma'$ on type environments such that C respects Γ whenever C respects Γ'; intuitively, this is the case if Γ describes the more permissive protocol in allowing more operations on the mentioned variables than Γ'. This order on type environments is obtained by lifting a corresponding order on types $T \preceq T'$ pointwise to environments. Typical subtypings include:

$?\mathsf{int} \preceq ?\mathsf{num}$**:** the protocol that grants reading of arbitrary numbers from a variable is less specific than the protocol that gives the additional guarantee that only integers will be read. Hence, every variable respecting the protocol $?\mathsf{num}$ will also respect the protocol $?\mathsf{int}$. This makes the reasonable assumption that all integers are numbers.

$\hat{}T \preceq ?T$**:** the protocol that grants read and write access to a variable for structures of type T is obviously respected if the variable is only read from.

$$V\rho\sigma[\,]\ (x\ \overline{y}) \parallel S \quad \in \mathcal{E} \qquad \text{if } \sigma(x) \notin \mathcal{N}, \text{ or } \sigma(x) \in \mathcal{N} \text{ and } \rho(\sigma(x)) \notin \mathcal{P}$$

$$V\rho\sigma[\,]\ (x\ \overline{y}) \parallel S \quad \in \mathcal{E} \qquad \text{if } \sigma(x) = (\mathsf{proc}\ (\overline{z})\ S'),\ |\overline{y}| \neq |z|$$

$$V\rho\sigma[\,]\ (\mathsf{exch}\ x\ y\ (z)\ S) \parallel S' \quad \in \mathcal{E} \qquad \text{if } \sigma(x) \notin \mathcal{N}, \text{ or } \sigma(x) \in \mathcal{N} \text{ and } \rho(\sigma(x)) \notin \mathcal{C}$$

$$V\rho\sigma[\,]\ (\mathsf{case}\ x\ (\overline{a}{:}\overline{y}\ldots)\ S) \parallel S' \quad \in \mathcal{E} \qquad \text{if } \sigma(x) \notin \mathcal{R}$$

$$V\rho\sigma[\,]\ (\mathsf{case}\ x\ (\overline{a}{:}\overline{y}\ldots)\ S) \parallel S' \quad \in \mathcal{E} \qquad \text{if } \sigma(x) = \{\overline{b}{:}\overline{z}\}, \{\overline{a}\} \not\subseteq \{\overline{b}\}$$

Figure 6.4.: Type Errors of Plain

Type checking is formalised as usual, by means of a *proof system* for *judgements* of the form $\Gamma \triangleright C$. A second auxiliary proof system is used to define subtyping $T_1 \preceq T_2$. This proof system will guarantee that C respects Γ whenever $\Gamma \triangleright C$ is derivable. Figure 6.4 defines the set $\mathcal{E}$ of configurations containing a *type error*. If C respects some environment, then *type safety* is guaranteed (see Section 6.2). This means that C will never reduce to an ill-formed configuration $C \in \mathcal{E}$.

Notice that multiple assignment to the same variable is not a type error, and the type system will not exclude the possibility of multiple assignment; neither does the type system guarantee that a variable will eventually be bound to a data structure. The type system will only guarantee that a variable is never assigned two data structures of *different type*.

6.1.2.2. Types

Figure 6.5 defines the abstract syntax of types. For technical reasons, we use *two* syntactic categories of *types* ranged over by P and T, respectively. If a distinction is necessary, we call P a proper type. There are three *modes*, a read-only mode ?, a write-only mode !, and a read/write mode ^. A *type* is a pair consisting of a mode and a proper type, or, a *proper type P* is a type with its top-level mode stripped off.

We assume two infinite sets of *type variables* $\mathcal{TV}$ ranged over by π and of *mode variables* $\mathcal{MV}$ ranged over by μ. Type and mode variables are jointly referred to by α. Types that do not contain type variables α are called *monomorphic*.

There is a proper monomorphic type per data structure. Hence there are *procedure types* $(\mathsf{proc}\ \overline{T})$, *record types* $\{\overline{a}{:}\overline{T}\}$, and *cell types* $(\mathsf{cell}\ T)$. In analogy to records, we require the features of record types to be pairwise distinct and identify record types up to reordering of fields. The only primitive monomorphic type is the *empty record type* $\{\}$. A procedure type $(\mathsf{proc}\ \overline{T})$ describes procedures that take arguments

[42] We consider only types that describe very simple protocols; see also Section 6.4.

Proper Types	P	$::=$	$(\mathsf{proc}\ \overline{T})$	(Procedure Type)
		$\mid$	$\{\overline{a}{:}\overline{T}\}$	(Record Type)
		$\mid$	$(\mathsf{cell}\ T)$	(Cell Type)
		$\mid$	π	(Proper Type Variable)
		$\mid$	$\forall\alpha.P$	(Polymorphic Type)
Modes	M	$::=$	$?$	(Read)
		$\mid$	$!$	(Write)
		$\mid$	$\hat{}$	(Read/Write)
		$\mid$	μ	(Mode Variable)
Types	T	$::=$	MP	(Moded Proper Type)
Type and Mode Variables	α	$::=$	$\pi \mid \mu$	

Figure 6.5.: Plain Types

of types $\overline{T}$, a record type $\{\overline{a}{:}\overline{T}\}$ describes records with fields $\overline{a}$ and associated types $\overline{T}$, and a cell type $(\mathsf{cell}\ T)$ describes cells that hold variables of type T.

Every type variable π is a proper type and every mode variable μ is a mode. Furthermore, there are *polymorphic types* of the form $\forall\alpha.P$ where α is either a type or a mode variable.[43] The type variable α in $\forall\alpha.P$ is *bound* in P. The *free* and *bound* type variables in a type T are defined as usual and written $ftv(T), ftv(P)$ and $btv(T), btv(P)$. This notion extends pointwise to environments. Polymorphic types of the form $\forall\alpha_1.\ldots.\forall\alpha_n.P$ are sometimes abbreviated by $\forall\overline{\alpha}.P$. If $n = 0$ then $\forall\overline{\alpha}.P$ simply means P. As for statements and data structures we assume all bound and free type variables to be pairwise distinct. Note, however, that the order of variables in the quantification prefix of a polymorphic type *does* matter. For instance, the types $\forall\pi_1.\forall\pi_2(\mathsf{proc}\ ?\pi_1\ !\pi_2)$ and $\forall\pi_2.\forall\pi_1(\mathsf{proc}\ ?\pi_1\ !\pi_2)$ are distinct.

A polymorphic procedure type $\forall\pi.(\mathsf{proc}\ \overline{T})$ describes procedures that have *every type* of the form $(\mathsf{proc}\ \overline{T})[\overline{P}/\overline{\pi}]$, obtained by substituting some proper types $\overline{P}$ for the type variables $\overline{\pi}$. For instance,

$$\forall\pi.(\mathsf{proc}\ ?\pi\ !\pi)$$

is the type of the identity procedure which assigns its first argument to its second one.

6.1.2.3. Subtyping

Subtyping is the smallest relation on types satisfying the rules given in Figure 6.6. The first eight rules define subtyping on monomorphic types, the two last ones extend subtyping to polymorphic types.

The rules (REFL) and (TRANS) require subtyping to be a preorder, and it is easy to see that it even is a partial order up to consistent renaming of bound type variables.

Modes. The six topmost rules are taken from Pierce and Sangiorgi's mode system for channels [165]. The rules (READSUB), (WRITESUB), (READ) and (WRITE) define subtyping on types in terms of subtyping of proper types. Rules (READSUB) and (WRITESUB) are obvious: a type that allows reading *and* writing is more permissive than a type that grants exclusively reading *or* writing.

Rule (READSUB) states that a read-moded type $?P$ becomes smaller in the subtyping order as the guarantees on the type P of the read expression become more specific. For instance, $?\mathsf{int} \preceq ?\mathsf{num}$ if $\mathsf{int} \preceq \mathsf{num}$ for some proper types int and num. Since there are more operations (readers) defined on integers than on numbers, $?\mathsf{int}$ is the more permissive type. Read modes are *monotonic* with respect to the subtyping order; one also says that read-moded types are ordered *covariantly*.

[43] In choosing universal polymorphism here we deviate from [144]. The existential polymorphism in [144] was inspired by Pict, and also motivated by the comparison of Plain with Pict, but universal polymorphism seems more appropriate for a language with higher-order procedures.

$$P \preceq P \quad (\textsc{Refl}) \qquad \frac{P_1 \preceq P_2 \quad P_2 \preceq P_3}{P_1 \preceq P_3} \quad (\textsc{Trans})$$

$$\hat{}P \preceq ?P \quad (\textsc{ReadSub}) \qquad \hat{}P \preceq !P \quad (\textsc{WriteSub})$$

$$\frac{P_1 \preceq P_2}{?P_1 \preceq ?P_2} \quad (\textsc{Read}) \qquad \frac{P_1 \preceq P_2}{!P_2 \preceq !P_1} \quad (\textsc{Write})$$

$$\frac{\overline{T} \preceq \overline{T'}}{(\mathsf{proc}\ \overline{T'}) \preceq (\mathsf{proc}\ \overline{T})} \quad (\textsc{ProcSub}) \qquad \frac{\overline{T} \preceq \overline{T'}}{\{\overline{a}{:}\overline{T} \ldots\} \preceq \{\overline{a}{:}\overline{T'}\}} \quad (\textsc{RecSub})$$

$$\frac{}{\forall\pi.P_1 \preceq P_1[P_2/\pi]} \quad (\textsc{Inst-P}) \qquad \frac{}{\forall\mu.P \preceq P[M/\mu]} \quad (\textsc{Inst-M})$$

$$\frac{P_1 \preceq P_2}{\forall\alpha.P_1 \preceq \forall\alpha.P_2} \quad (\textsc{Poly})$$

Figure 6.6.: Plain Subtyping

A write mode inverts the order on types; see rule (WRITESUB). A write-moded type $!P$ becomes smaller in the subtyping order as it becomes more specific with respect to the type P of the data structures to be written. For instance, $!\mathsf{num} \preceq\ !\mathsf{int}$ if $\mathsf{int} \preceq \mathsf{num}$. Since there are fewer integers than numbers, $!\mathsf{num}$ is the more permissive type and $!\mathsf{int}$ the more specific type. Write modes are *antimonotonic* respectively *contravariant* with respect to the subtyping order.

Two types with read/write mode are subtypes of each other if and only if they are equal. Since read/write moded types must simultaneously be ordered covariantly and contravariantly (as read moded types and write moded types, respectively), they are *invariant* with respect to subtyping.

Monomorphic Types. A record type T is a subtype of another record type T' if T has at least the features in T' and the corresponding fields of T and T' are in covariant subtype relation; see rule (RECSUB).

Rule (PROCSUB) states that a procedure type $(\mathsf{proc}\ T)$ is a subtype of $(\mathsf{proc}\ T')$ if $T' \preceq T$, that is, if the argument types are in contravariant subtype relationship. In other words, procedure types become *smaller* along the subtype order as their argument types become *greater*. In this case, more argument types respect the procedure type and hence the procedure is more permissive.

There is only trivial subtyping for cells. Note that every cell always supports the read operation to obtain its current content and the write operation to replace its content with another variable. Hence, in analogy to subtyping of read/write moded types, cell types must be invariant.

Polymorphic Types. Rule (INST-P) says that the polymorphic type $\forall\pi.P_1$ is a subtype of every type that is obtained by substitution of some P_2 for the proper type variable π in P_1. For instance, the polymorphic identity type is smaller than every more specific identity type:

$$\forall\pi.(\mathsf{proc}\ ?\pi\ !\pi) \preceq (\mathsf{proc}\ ?\mathsf{int}\ !\mathsf{int})$$

Rule (INST-M) is analogous for mode polymorphic types of the form $\forall\mu.P$. For instance, a procedure that is well-behaved on all arguments of types $\mu\pi$ and $(\mathsf{cell}\ \mu\pi)$ independent of the mode μ, will also be well-behaved on arguments of types $?\pi$ and $(\mathsf{cell}\ ?\pi)$ which fix the mode.

$$\forall\mu.\forall\pi.(\mathsf{proc}\ \mu\pi\ (\mathsf{cell}\ \mu\pi)) \preceq \forall\pi.(\mathsf{proc}\ ?\pi\ (\mathsf{cell}\ ?\pi))$$

Rule (POLY) defines how to compare two polymorphic types with the same quantifier prefix $\forall\alpha$. A polymorphic types $\forall\alpha.P_1$ is subtype of another one $\forall\alpha.P_2$ if P_1 is a subtype of P_2. Note that this subsumes polymorphism of both forms $\forall\pi.P$ and $\forall\mu.P$.

In Plain, unary functions are implemented as binary procedures that read their input from the first argument and write the result to the second one. The type of such procedures is $(\mathsf{proc}\ ?P_1\ \ !P_2)$. The induced subtyping rule on these types coincides with the usual subtyping rule on function types $T \to T'$ which makes the function type constructor $\to$ covariant in its range type and contravariant in its domain type.[44]

$$\dfrac{\dfrac{T_1' \preceq T_1}{?T_1' \preceq ?T_1}\ (\textsc{ReadSub}) \quad \dfrac{T_2 \preceq T_2'}{!T_2' \preceq !T_2}\ (\textsc{WriteSub})}{(\mathsf{proc}\ ?T_1\ \ !T_2) \preceq (\mathsf{proc}\ ?T_1'\ \ !T_2')}\ (\textsc{ProcSub})$$

6.1.2.4. Type Checking

A *type assumption* is a variable-type pair $x{:}T$. A *type environment* Γ is a finite set of type assumptions for distinct variables $x_1,\ldots,x_n$, written

$$x_1{:}T_1,\ldots,x_n{:}T_n\,.$$

The *extension* of an environment Γ by $x{:}T$ is written as adjunction $\Gamma, x{:}T$ and is only well-defined if Γ contains no type assumption for x yet. The notion Γ,Γ' is defined

[44] Pierce and Sangiorgi [165] have proposed this mode system for the π-calculus in order to recover subtyping as previously studied in typed functional languages. They present the analogous example (in terms of π-calculus) as one validation for their mode system.

$$\frac{}{\Gamma \triangleright x{:}T} \quad x{:}T \in \Gamma \qquad (\textsc{Var}) \qquad \frac{\Gamma \triangleright \overline{y}{:}\overline{T}}{\Gamma \triangleright \{\overline{a}{:}\overline{y}\}{:}?\{\overline{a}{:}\overline{T}\}} \qquad (\textsc{Rec})$$

$$\frac{\Gamma \triangleright E{:}T}{\Gamma \triangleright E{:}T'} \quad T \preceq T' \qquad (\textsc{Sub}) \qquad \frac{\Gamma \triangleright y{:}T}{\Gamma \triangleright (\mathsf{cell}\ y){:}?(\mathsf{cell}\ T)} \qquad (\textsc{Cell})$$

$$\frac{\Gamma, \overline{y}{:}\overline{T} \triangleright S}{\Gamma \triangleright (\mathsf{proc}\ (\overline{y})\ S){:}?\forall\overline{\alpha}.(\mathsf{proc}\ \overline{T})} \quad \{\overline{\alpha}\} \cap \mathit{ftv}(\Gamma) = \emptyset \qquad (\textsc{Proc})$$

$$\frac{\Gamma \triangleright x{:}!P \quad \Gamma \triangleright E{:}?P}{\Gamma \triangleright x := E} \qquad (\textsc{Asgn})$$

$$\frac{\Gamma \triangleright x{:}?(\mathsf{proc}\ \overline{T}) \quad \Gamma \triangleright \overline{y}{:}\overline{T}}{\Gamma \triangleright (x\ \overline{y})} \qquad (\textsc{Appl}) \qquad \frac{\Gamma, \overline{x}{:}\overline{T} \triangleright S}{\Gamma \triangleright (\mathsf{local}\ (\overline{x})\ S)} \qquad (\textsc{Local})$$

$$\frac{\Gamma \triangleright x{:}?\{\overline{a}{:}\overline{T}\} \quad \Gamma, \overline{y}{:}\overline{T} \triangleright S}{\Gamma \triangleright (\mathsf{case}\ x\ (\overline{a}{:}\overline{y}\ldots)\ S)} \qquad (\textsc{Match}) \qquad \frac{\Gamma \triangleright S_1 \qquad \Gamma \triangleright S_2}{\Gamma \triangleright S_1 \parallel S_2} \qquad (\textsc{Conc})$$

$$\frac{\Gamma \triangleright x{:}?(\mathsf{cell}\ T) \quad \Gamma \triangleright y{:}T \quad \Gamma, z{:}T \triangleright S}{\Gamma \triangleright (\mathsf{exch}\ x\ y\ (z)\ S)} \qquad (\textsc{Exch}) \qquad \frac{}{\Gamma \triangleright \mathsf{skip}} \qquad (\textsc{Skip})$$

Figure 6.7.: Typing Plain Expressions and Statements

analogously. The inference system in Figure 6.7 defines two well-typedness *judgements* of the form $\Gamma \triangleright E{:}T$ and $\Gamma \triangleright S$.

An expression E is said to *have type T* with respect to environment Γ if $\Gamma \triangleright E{:}T$ is derivable. A statement S is said to *respect an environment* Γ if $\Gamma \triangleright S$ is derivable. A configuration $V\rho\sigma\,[]\ S$ respects an environment Γ if its associated statement does: $\Gamma \triangleright C$ if and only if $\Gamma \triangleright S(C)$. An expression E is said to be *well-typed* if it has a type with respect to some environment. A statement S, a store $\rho\sigma$, and a configuration $V\rho\sigma\,[]\ S$ are said to be *well-typed* if they respect some environment. An expression, statement, or configuration that is not well-typed is called *ill-typed*.

Variables receive their type by lookup in the environment; see rule (VAR). Data structures must allow inspection, hence all of them have read-moded types; see rules (PROC), (CELL), and (REC). By rule (SUB), expressions can freely be promoted along the subtyping order. Rules (CELL) and (REC) should be fairly clear: the type of a record or a cell is straightforwardly derived from the types of the record fields and the cell content, respectively. With respect to a given environment Γ, a procedure $(\mathsf{proc}\ (\overline{y})\ S)$ has every type $(\mathsf{proc}\ \overline{T})$ such that its body S respects Γ under the additional type assumptions $\overline{y}{:}\overline{T}$ on the formal arguments. Furthermore, the procedure type can be shown polymorphic in all type variables that do not occur in Γ (but that may occur in the argument types $\overline{T}$). An assignment $x{:=}E$ is well-typed if there exists a proper type such that x has type $!P$ and E has type $?P$. For an application $(x\ y)$, an exchange $(\mathsf{exch}\ x\ y\ (z)\ S)$, or a matching $(\mathsf{case}\ x\ (\overline{a}{:}\overline{y}\ldots)\ S)$ to be well-typed, rules (APPL), (EXCH), and (MATCH), x must allow read access. The types of further arguments must match the requirements by the type of x. The rules (LOCAL), (CONC), and (SKIP) are trivial.

6.1.3. Examples

In this section we illustrate Plain by means of examples.[45] We allow for the OPM-style notation $(\mathsf{proc}\ x\ (\overline{y})\ S)$ as an alternative notation for $x{:=}(\mathsf{proc}\ (\overline{y})\ S)$. We assume new proper base types int and bool, along with the integers $1,2,3,\ldots$ as primitive data structures (constants) of type int and the booleans *true* and *false* as constants of type bool, and we freely use some basic operations such as addition $+$ on these types.

Further, we assume a minimal extension to Plain that enables type checking recursive procedures over streams (infinite lists). We assume an additional proper type $\mathsf{list}(T)$

[45] All examples have been tested by an experimental implementation of Plain compiling to Oz.

that describes lists of variables of type T and provide the following typing rules:

$$\frac{}{\Gamma \triangleright \{\}{:}?\mathsf{list}(T)} \quad (\textsc{Nil}) \qquad \frac{\Gamma \triangleright E{:}M\mathsf{list}(T)}{\Gamma \triangleright E{:}M\{hd{:}T, tl{:}M\mathsf{list}(T)\}} \quad (\textsc{Unfold})$$

$$\frac{\Gamma \triangleright x{:}T \qquad \Gamma \triangleright y{:}?\mathsf{list}(T)}{\Gamma \triangleright \{hd{:}x, tl{:}y\} : ?\mathsf{list}(T)} \quad (\textsc{Cons}) \qquad \frac{\Gamma \triangleright E{:}M\{hd{:}T, tl{:}M\mathsf{list}(T)\}}{\Gamma \triangleright E{:}M\mathsf{list}(T)} \quad (\textsc{Fold})$$

Notice that it is immediate to type check cyclic lists with these rules, for example as in the statement

$(\mathsf{local}\ (x)\ x := \{hd{:}y, tl{:}x\})$.

A generic extension of Plain by recursively defined data types à la SML, and the addition of case statements with multiple clauses is possible with standard techniques.

6.1.3.1. Basic Examples

The identity procedure has the following polymorphic type.

$(\mathsf{proc}\ (x\ y)\ y := x) : ?\forall\pi.(\mathsf{proc}\ ?\pi\ !\pi)$

This corresponds to the expected polymorphic type $\forall\pi.\pi \to \pi$ of the identity in functional languages. Notice that the identity is not the only procedure with the type $?\forall\pi.(\mathsf{proc}\ ?\pi\ !\pi)$. The other ones include the trivial procedure

$(\mathsf{proc}\ (x\ y)\ \mathsf{skip}) : ?\forall\pi.(\mathsf{proc}\ ?\pi\ !\pi)$,

as well as many procedures that side-effect variables other than its formal arguments. One such procedure is

$(\mathsf{proc}\ (x\ y)\ z_1 := z_2) : ?\forall\pi.(\mathsf{proc}\ ?\pi\ !\pi)$

provided $z_1 := z_2$ is well-typed. (Actually, this procedure has every binary procedure type, be it monomorphic or polymorphic.) More generally speaking, the type of a procedure specifies which kinds of operations it may perform on its arguments. It does not guarantee that any operations are performed on the arguments at all. Furthermore, the operations performed on global variables of a procedure are not visible in the procedure's type.

Types convey some of the synchronisation behaviour of a procedure. In particular, all input modes in procedure argument types indicate that an application of this procedure might block when it accesses the corresponding embedded variable. For example, the procedure that waits until its first argument is bound to a record before it applies its second one has the type

$(\mathsf{proc}\ (x\ y)\ (\mathsf{case}\ x\ ()\ (y))) : (\mathsf{proc}\ ?\{\}\ ?(\mathsf{proc}\ ))$

More fundamentally, the procedure *wait* that waits for its argument to be bound at all has this type:

$$(\mathsf{proc}\ (x)\ (\mathsf{local}\ (y)\ y{:=}x)) : ?\forall\pi.(\mathsf{proc}\ ?\pi)$$

Note that the procedure *wait* is not very useful in Plain so far since no statement can synchronise on an assignment being executed. It becomes extremely useful though once *sequential composition* $S_1;S_2$ of statements is added with the operational semantics to first reduce S_1 and then S_2.[46]

Pattern matching on records subsumes field selection. For example, here is a procedure selecting the field a from its first argument and assigning it to the second one.

$$(\mathsf{proc}\ (x\ y)\ (\mathsf{case}\ x\ (a{:}z\ldots)\ y{:=}z)) : ?\forall\pi.(\mathsf{proc}\ ?\{a{:}?\pi\}\ !\pi)$$

6.1.3.2. Semaphores

Semaphores are a standard mechanism for guaranteeing multiple exclusion in a concurrent setting [61]. A semaphore is a data structure with a *request* and a *release* operation. Multiple concurrent activities may request the semaphore. Once a request operation has succeeded, an option is granted to perform the corresponding release. All subsequent requests on the same semaphore are blocked until this operation has been performed to release the semaphore.

In Plain, semaphores can be implemented by a procedure *newsema* with type

$$\mathit{newsema} : ?(\mathsf{proc}\ !(\mathsf{proc}\ !(\mathsf{proc}))\ .$$

On application of *newsema*, a new cell is created; the cell is initialised with an empty record which is used as a token. Next, a unary procedure *req* is defined which implements the request operation of the semaphore as an operation on the cell. The cell is private to the request operation and thus cannot accidentally or maliciously be side-effected.

$$
\begin{array}{l}
(\mathsf{proc}\ \mathit{newsema}\ (\mathit{req})\ (\mathsf{local}\ (c) \\
\quad \mathit{tok}{:=}\{\}\ \|\ c{:=}(\mathsf{cell}\ \mathit{tok})\ \| \\
\quad\quad (\mathsf{proc}\ \mathit{req}\ (\mathit{rel})\ (\mathsf{local}\ (\mathit{new}) \\
\quad\quad\quad (\mathsf{exch}\ c\ \mathit{new}\ (\mathit{old}) \\
\quad\quad\quad\quad (\mathsf{case}\ \mathit{old}\ ()\ (\mathsf{proc}\ \mathit{rel}\ ()\ \mathit{new}{:=}\mathit{old}))))))
\end{array}
$$

On application of the procedure *req* to some argument *rel*, the current cell content *old* is replaced by a fresh unbound variable *new*. The variable *old* is then matched against the empty record pattern $()$. This operation will succeed immediately on the

[46] We do not consider sequential composition here since its addition does not affect the type system at all. Our experiences with Oz however indicate that every practical language of this family must support sequential composition [91, 174].

first application of *req*, while a subsequent request may find *old* to be an unbound variable and block on pattern matching. When the match has reduced and thus the request was successful, a release procedure *rel* is returned. On application of *rel*, *old* is assigned to *new* and thus unblocks the pattern matching of the subsequent application of *req*. The sequence of request and release operations dynamically yields a chain of assignments as follows.

$$tok \quad = \quad old_1 := new_1 \quad = \quad old_2 := new_2 \quad = \quad old_3 := new_3 \quad \ldots$$

Notice that the procedure *rel* does not operate on formal arguments at all but is meant to side effect the store.

6.1.3.3. Lazy Streams

By means of partially determined data structures Plain can conveniently express lazy streams. A *lazy stream* is a possibly infinite list whose evaluation is deferred and demand-driven. When some element of a lazy stream is requested, the stream is evaluated just up to this element and evaluation of the tail of the stream is deferred again. This means that evaluation of the stream always terminates if only a finite part is demanded. Consider a binary procedure *nat*

$nat : ?(\mathsf{proc}\ ?\mathsf{int}\ ?\mathsf{list}(!\mathsf{int}))$

that computes the lazy stream of natural numbers larger than some given n. It is interesting to consider the type of *nat* more closely, in particular the type $?\mathsf{list}(!\mathsf{int})$ of its second argument. By the rules (FOLD) and (UNFOLD) this type is equivalent to each of the finite unfoldings of the following form

$?\{hd{:}!\mathsf{int},\ tl{:}?hd{:}\{!\mathsf{int},\ tl{:}?\{hd{:}!\mathsf{int},\ \ldots\ tl{:}?\mathsf{list}(!\mathsf{int})\}\}\}$

that restrict the whole spine of the stream to be read-only. This suggests that *nat* will write the integer elements of the stream, while the stream itself will be provided from the outside. Note that these two opposite modes correspond to the

- outgoing *"functional" data flow* of *nat* that specifies to compute the infinite sequence of natural numbers, and the
- ingoing *flow of demand* that requests computation of a finite prefix of this sequence.

Here is an implementation of the procedure *nat*.

$(\mathsf{proc}\ nat\ (n\ s)\ (\mathsf{local}\ (m)\ (\mathsf{case}\ s\ (x, r \ldots)\ x := n \parallel m := n+1 \parallel (nat\ \ m\ r))))$

Assuming an unbound variable $s : \hat{}\mathsf{list}(\hat{}\mathsf{int})$, a typical application of this procedure is

$$one := 1 \;\|\; (nat\;\; one\; s)$$

Reduction of the application blocks on pattern matching since s is not bound yet. We can express demand for the smallest number in the stream by binding s to a record whose field hd serves as container for this number (assume n_1 and s' fresh):

$$s := \{hd{:}n_1, tl{:}s'\}$$

This activates the pattern matching, the assignment $n_1 := one$, and the recursive call $(nat\;\; two\; s')$ (where two is bound to 2). By binding the tail s', we can demand subsequent elements in the stream.

A second example along these lines is the procedure

$$(\mathsf{proc}\; fib\; (x_1\; x_2\; s)\; (\mathsf{case}\; s\; (x, s' \ldots)\; x := x_1 + x_2 \;\|\; (fib\;\; x_2\; x\; s')))$$

with this type:

$$fib : ?(\mathsf{proc}\; ?\mathsf{int}\; ?\mathsf{int}\; ?\mathsf{list}(\hat{}\mathsf{int}))$$

This procedure computes an infinite list of natural numbers according to the generation principle of the Fibonacci numbers: every element of the list (from the third onwards) is the sum of the two preceding ones. These two preceding numbers are passed as additional arguments of *fib* through the recursion. The following typical application of *fib* defines s as the stream of Fibonacci numbers (beginning with the third one) and binds n_1 and n_2 to its first two elements.

$$(fib\;\; one\; one\; s) \;\|\; s := \{hd{:}n_1, tl{:}s'\} \;\|\; s' := \{hd{:}n_2, tl{:}s''\}$$

Again note that the type of the list argument exposes the fact that the list elements are not only produced but also read during the recursion.

6.1.3.4. Channels

Another example for stream-based programming is the following implementation of channels with an asynchronous send and a synchronous receive. It is also an example for Plain procedures whose type is *polymorphic in the mode* of its arguments.

A *channel* for variables of type T is an abstract data type with two operations *put* and *get* of the following types:

$$\begin{array}{lcl} put & : & ?(\mathsf{proc}\; T) \\ get & : & ?(\mathsf{proc}\; ?(\mathsf{proc}\; T)) \end{array}$$

The *put* operation takes a variable of type T, puts it into ("sends it along") the channel, and then terminates. The *get* operation takes a variable of type $?(\mathsf{proc}\; ?T)$, that is, a reference to a procedure *cont* for arguments of type $?T$; then it takes ("receives") a variable from the channel and applies *cont* as a continuation to it. Combination of these operations in a record with fields *put* and *get* yields the following type of channel interfaces:

$$?\{put{:}?(\mathsf{proc}\ T),\ get{:}?(\mathsf{proc}\ ?(\mathsf{proc}\ T))\}$$

Now we proceed to implement a polymorphic procedure *newchan* that generates new channels for variables of arbitrary type *and* arbitrary mode.

$$newchan : ?\forall\mu.\forall\pi.(\mathsf{proc}\ \ !\{put{:}?(\mathsf{proc}\ \mu\pi),\ get{:}?(\mathsf{proc}\ ?(\mathsf{proc}\ \mu\pi))\})$$

Note that the procedure *newchan* is polymorphic in type *and* mode of the variables to be put in the channel. Therefore *newchan* is guaranteed not to perform any operations on the variables put into a channel since it cannot safely assume read or write permissions on them. *newchan* simply passes the variables to the receiver continuation for further processing. Here is the Plain code for the procedure *newchan*.

```
(proc newchan (chan)
    (local(s0 cput cget put get)
        cput := (cell s0) || cget := (cell s0) ||
        (proc put (z)
            (local (s2) (exch cput s2 (s1) s1 := {hd:z, tl:s2}))) ||
        (proc get (cont)
            (local (s2)
                (exch cget s2 (s1) (case s1 (hd:z, tl:s3 ...) s2 := s3 || (cont z))))) ||
        chan := {put:put, get:get))
```

We implement a channel as a variable s_0 referring to a stream and two cells *cput* and *cget* realizing the pointers into s. On creation, the stream is empty and both pointers refer to the first slot. On application of the procedure *put* on a variable z, the current content s_1 of *cput* is replaced with a fresh variable s_2 and then s_1 is bound to $\{hd{:}z, tl{:}s_2\}$. This advances the pointer *cput*. On application of the procedure *get* on a variable *cont*, the current content s_1 of *cget* is replaced with a fresh variable s_2; then s_1 is matched against the pattern $(hd{:}z, tl{:}s_3)$. When s_1 is bound to a record of this form, *cont* is applied to z and s_3, the remainder of the stream, is assigned to s_2.

6.1.3.5. Mode Polymorphism

As a final set of examples we give the types of some standard procedures on lists as further illustration on higher-order and mode polymorphism. We do not give their implementation here since they require a multiple clause conditional that we have not defined. Instead, we rely on the intuitions the reader brings along from some higher-order programming language.

The procedure *member* returns a boolean depending on whether some element in a list is bound to a given data structure. *member* has the following type:

$member : ?\forall\pi.?(\mathsf{proc}\ ?\pi\ ?\mathsf{list}(?\pi)\ !\mathsf{bool})$

To decide membership, the list must be recursively decomposed (*i. e.*, read), and its elements as well as the given data structure must be compared for equality (*i. e.*, read). The procedure *length* returning the length of a list must recursively decompose (*i. e.*, read) its list argument. However, it needs not access the list elements themselves. Thus the type of *length* is mode polymorphic.

$length : ?\forall\mu.\forall\pi.(\mathsf{proc}\ ?\mathsf{list}(\mu\pi)\ !\mathsf{int})$

Similarly, the procedure *map* that maps one finite list into another one with respect another given binary procedure need not itself perform any operation on the list elements. Rather, these are passed to the procedural argument that is responsible for further processing.

$map : ?\forall\mu_1.\forall\mu_2.\forall\pi_1.\forall\pi_2.(\mathsf{proc}\ ?\mathsf{list}(\mu_1\pi_1)\ ?(\mathsf{proc}\ \mu_1\pi_1\ \mu_2\pi_2)\ ?\mathsf{list}(\mu_2\pi_2))$

Procedures that are polymorphic in the type of some (component) of their arguments are very restricted in the operations they may perform on these arguments. For instance, a procedure of type

$?\forall\mu.(\mathsf{proc}\ \mu P\ ?(\mathsf{cell}\ \mu P))$

may perform only one interesting operation on its first argument, namely place it into the cell received as a second argument. This is safe since the type of the first and the second argument share the mode variable. Hence a procedure of the given type is this one:

$(\mathsf{proc}\ assign\ (x\ y)\ (\mathsf{exch}\ x\ y\ (z)\ \mathsf{skip})) : ?\forall\mu.\forall\pi.(\mathsf{proc}\ \mu\pi\ ?(\mathsf{cell}\ \mu\pi))$

Also observe that procedures with the following types must ignore their arguments.

$?\forall\mu.(\mathsf{proc}\ \mu P\ \ \mu P)$

$?\forall\mu.(\mathsf{proc}\ ?\{a{:}\mu P\}\ \mu P)$

A typical procedure with a higher-order polymorphic type is one implementing function composition.

$(\mathsf{proc}\ compose\ (f_1\ f_2\ f_3)\ (\mathsf{proc}\ f_3\ (x\ y)\ (\mathsf{local}\ (z)\ (f_1\ \ x\ z)\ \|\ (f_2\ \ z\ y))))$

One of its types is this one

$compose : ?\forall\pi_1.\forall\pi_2.(\mathsf{proc}\ \ ?\mathsf{listproc}(\pi_1)\ ?(\mathsf{proc}\ ?\pi_1\ !\pi_2)\ !\mathsf{listproc}(\pi_2))$

where, for all P, $\mathsf{listproc}(P) = \forall\mu.\forall\pi.(\mathsf{proc}\ ?\mathsf{list}(\mu\pi)\ !P)$. This type allows us to check the application

$(compose\ \ length\ \ iseven\ \ isevenlength)$

where *length* : ˆlistproc(int) and *iseven* : ?(proc ?int !bool).

6.2. Type Safety

The type system is sound in the sense that it excludes the type errors listed in Figure 6.4. To prove this result one first checks that no erroneous configuration can be well-typed (Proposition 6.1). Next one shows that it is an invariant of reduction for a statement to respect an environment Γ (Theorem 34). Soundness of the type system and hence type safety is then easily obtained (Corollary 35).

In this section we write judgements like $\Gamma \triangleright S$ or $T \preceq T'$ as an abbreviation for the statement that these judgements are *derivable*.

Proposition 6.1
If $\Gamma \triangleright C$ *then* $C \notin \mathcal{E}$.

Proof. See Page 169 below. □

Theorem 34 (Type Preservation)
If $\Gamma \triangleright C_1$ *and* $C_1 \longrightarrow C_2$ *then* $\Gamma \triangleright C_2$.

Proof. See Page 171 at the end of this Section. □

Corollary 35 (Type Safety)
If $\Gamma \triangleright C$ *and* $C \longrightarrow^* C'$ *then* $C' \notin \mathcal{E}$.

Proof. By induction over the length of the reduction $C \longrightarrow^* C'$. The base case $C \longrightarrow^0 C'$ (*i. e.*, $C = C'$) follows from Proposition 6.1, and the induction step with Theorem 34. □

In the remainder of this section we prove the Type Preservation Theorem 34.

The Type Preservation Proof

We first prove some standard Lemmas on well-typed statements (Lemmas 6.2–6.6). The corresponding Lemmas for configurations follow immediately, since congruence and well-typedness on configurations is defined in terms of their associated statements:

1. $\forall C_1, C_2 : \quad C_1 \equiv C_2$ iff $S(C_1) \equiv S(C_2)$.
2. $\forall \Gamma \forall C : \quad \Gamma \triangleright C$ iff $\Gamma \triangleright S(C)$.

In the sequel we shall denote with A either a proper type P or a mode M.

Lemma 6.2 (Congruence)
Well-typedness is invariant under structural congruence: If $\Gamma \triangleright S_1$ *and* $S_1 \equiv S_2$ *then* $\Gamma \triangleright S_2$.

Proof. Structural induction over S_1. □

Lemma 6.3 (Weakening)
If $x \notin fv(S)$, *then* $\Gamma, x{:}T \triangleright S$ *if and only if* $\Gamma \triangleright S$.

Proof. On inspection of the typing rules for statements one notes that in any derivation of $\Gamma \triangleright S$ only the type assumptions for the variables in $fv(S)$ matter. The proof is by induction over the derivation of $\Gamma \triangleright S$. □

Lemma 6.4 (Variable Substitution)
If $\Gamma, x{:}T, y{:}T \triangleright S$ *then* $\Gamma, x{:}T, y{:}T \triangleright S[y/x]$

Proof. Induction over the derivation of $\Gamma, x{:}T, y{:}T \triangleright S$. □

Lemma 6.5 (Type Substitution)
If $\Gamma \triangleright S$ *then* $\Gamma[P/\pi] \triangleright S$ *and* $\Gamma[M/\mu] \triangleright S$.

Proof. Induction over the derivation of $\Gamma \triangleright S$. □

Lemma 6.6 (Subtyping)
If $\Gamma, x{:}T_1 \triangleright S$, $T_2 \preceq T_1$, *and* $ftv(\Gamma, x{:}T_1) \subseteq ftv(\Gamma, x{:}T_2)$ *then* $\Gamma, x{:}T_2 \triangleright S$.

It is due to the subtyping rule (RECSUB) that the claim fails without the assumption that $ftv(\Gamma, x{:}T_1) \subseteq ftv(\Gamma, x{:}T_2)$. The additional field types which may be added on subtyping may contain additional free type variables which may conflict with the side condition of rule (PROC).

Proof. We prove the claim simultaneously with the corresponding one for expressions:

If $\Gamma, x{:}T_1 \triangleright E{:}T$, $T_2 \preceq T_1$, and $ftv(\Gamma, x{:}T_1) \subseteq ftv(\Gamma, x{:}T_2)$ then $\Gamma, x{:}T_2 \triangleright E{:}T$.

The proof is by induction over the derivation of $\Gamma, x{:}T_1 \triangleright S$ or $\Gamma, x{:}T_1 \triangleright E{:}T$. We make a case distinction over the rule that was applied last.

(VAR) In this case E must be a variable. If $E \neq x$, then the claim is trivial due to the Weakening Lemma 6.3; hence assume $E = x$. Then $\Gamma, x{:}T_1 \triangleright E{:}T$ implies that $x{:}T \in \Gamma, x{:}T_1$ and hence $T_1 = T$. Thus, we can derive $\Gamma, x{:}T_2 \triangleright x{:}T$ as follows:

$$\cfrac{\cfrac{}{\Gamma, x{:}T_2 \triangleright x{:}T_2}\; x{:}T_2 \in \Gamma, x{:}T_2}{\Gamma, x{:}T_2 \triangleright x{:}T_1}\; T_2 \preceq T_1$$

(SUB) In this case the derivation has the form

$$\dfrac{\dfrac{\vdots}{\Gamma, x{:}T_1 \triangleright E{:}T'}}{\Gamma, x{:}T_1 \triangleright E{:}T}\; T \preceq T'$$

We conclude with the induction assumption that $\Gamma, x{:}T_2 \triangleright E{:}T'$ is derivable and obtain $\Gamma, x{:}T_2 \triangleright E{:}T$ by rule (SUB).

(PROC) In this case, there exist variables $\overline{y}$ and statements S such that $E = (\mathsf{proc}\ (\overline{y})\ S)$ and $x \notin \{\overline{y}\}$, and also there exist type and mode variables $\overline{\alpha}$ and types $\overline{T}$ with $T = ?\forall\overline{\alpha}.(\mathsf{proc}\ \overline{T})$. From rule (PROC) we know that

$$\Gamma, x{:}T_1, \overline{y}{:}\overline{T} \triangleright S$$

is derivable which implies by induction assumption (and $x \notin \{\overline{y}\}$) that

$$\Gamma, x{:}T_2, \overline{y}{:}\overline{T} \triangleright S$$

is derivable. From the side condition of (PROC) we also know that

$$\{\overline{\alpha}\} \cap \mathit{ftv}(\Gamma, x{:}T_1) = \emptyset\,,$$

and since $\mathit{ftv}(\Gamma, x{:}T_2) \subseteq \mathit{ftv}(\Gamma, x{:}T_1)$ by assumption, we obtain

$$\{\overline{\alpha}\} \cap \mathit{ftv}(\Gamma, x{:}T_2) = \emptyset\,.$$

Hence $\Gamma, x{:}T_2 \triangleright (\mathsf{proc}\ (\overline{y})\ S){:}T = ?\forall\overline{\alpha}.(\mathsf{proc}\ \overline{T})$ is derivable.

The remaining cases are similar or simpler. □

Proposition 6.1
If $\Gamma \triangleright C$ then $C \notin \mathcal{E}$.

Proof. If $C \in \mathcal{E}$ then C has one of the forms defined in Figure 6.4. These are easily seen to be ill-typed. For instance, assume that C has the form $V\rho\sigma[]\ (x\ \overline{y}) \parallel S$, let $\sigma(x) \notin \mathcal{P}$, and assume $\Gamma \triangleright C$ for some Γ. Then there are types $\overline{T}$ such that

$$\Gamma \triangleright x{:}?(\mathsf{proc}\ \overline{T})$$

by rule (APPL). Furthermore, by definition of well-typed configurations (see Page 160) and the assumption that $\sigma(x) \notin \mathcal{P}$, there exists $D \in \mathcal{R} \cup \mathcal{C}$ such that $\Gamma \triangleright x := D$. Hence

$$\Gamma \triangleright x{:}!\{\} \qquad \text{or} \qquad \exists T:\ \Gamma \triangleright x{:}!(\mathsf{cell}\ T)\,.$$

Such a Γ cannot exist, since there exists no common subtype of $?(\mathsf{proc}\ \overline{T})$ and either $!\{\}$ or $!(\mathsf{cell}\ T)$. Hence C is ill-typed. $\square$

Given three sequences $\overline{\alpha_1}$, $\overline{\alpha_2}$, and $\overline{\alpha_3}$, we say that $\overline{\alpha_1}$ is a *subsequence* of $\overline{\alpha_2}$ if there is an order preserving injection from $\overline{\alpha_1}$ into $\overline{\alpha_2}$, and we say that $\overline{\alpha_1}$ is a *subsequence* of $\overline{\alpha_2}$ *with rest* $\overline{\alpha_3}$ if $\overline{\alpha_1}$ and $\overline{\alpha_3}$ are subsequences of $\overline{\alpha_2}$ and partition $\overline{\alpha_2}$.

Lemma 6.7

If $\forall\overline{\alpha_1}.P_1 \preceq \forall\overline{\alpha_2}.P_2$ is derivable then

1. *$\overline{\alpha_2}$ is a subsequence of $\overline{\alpha_1}$ with rest $\overline{\alpha_3}$, and*
2. *there is a sequence $\overline{A}$ of types and modes such that $P_1[\overline{A}/\overline{\alpha_3}] \preceq P_2$ is derivable.*

Proof. The last steps in the derivation of $\forall\overline{\alpha_1}.P \preceq \forall\overline{\alpha_2}.P'$ are determined by the sequence $\overline{\alpha_2}$ and may involve applications of rules (INST-P), (INT-M), and (POLY) only. The proof is by induction over the length of the sequence $\overline{\alpha_1}$. $\square$

Lemma 6.8 (Application)

If $\Gamma \rhd (x\ \overline{y}) \parallel x := (\mathsf{proc}\ (\overline{z})\ S)$ is derivable then there exist types $\overline{T_z}$, variables $\overline{\alpha}$ and a sequence $\overline{A}$ of types and modes such that

1. $\{\overline{\alpha}\} \cap ftv(\Gamma) = \emptyset$
2. $\Gamma, \overline{z}{:}\overline{T_z} \rhd S$
3. $\Gamma(\overline{y}) \preceq \overline{T_z}[\overline{A}/\overline{\alpha}]$

Proof. Assume $\Gamma \rhd (x\ \overline{y}) \parallel x := (\mathsf{proc}\ (\overline{z})\ S)$. From rules (PROC) and (ASGN) we then know that there are types $\overline{T_z}$ and type and mode variables $\overline{\alpha}$ such that $\{\overline{\alpha}\} \cap ftv(\Gamma) = \emptyset$, as well as

$$\Gamma, \overline{z}{:}\overline{T_z} \rhd S\,,$$

$$\Gamma \rhd (\mathsf{proc}\ (\overline{z})\ S) : ?\forall\overline{\alpha}.(\mathsf{proc}\ \overline{T_z})\,,\ \text{and}$$

$$\Gamma \rhd x{:}!\forall\overline{\alpha}.(\mathsf{proc}\ \overline{T_z})\,.$$

This implies claims (1) and (2). Further, we know from rule (APPLY) that there exist types $\overline{T_y}$ such that $\Gamma(\overline{y}) \preceq \overline{T_y}$ and

$$\Gamma \rhd x : ?(\mathsf{proc}\ \overline{T_y})\,,\ \text{and}\ \Gamma \rhd \overline{y}{:}\overline{T_y}$$

It follows from the definition of subtyping that there exist types $\overline{T_x}$ and variables $\overline{\alpha'}$ such that $\Gamma(x) = \hat{}\forall\overline{\alpha'}.(\mathsf{proc}\ \overline{T_x})$ and

$$\hat{}\forall\overline{\alpha'}.(\mathsf{proc}\ \overline{T_x}) \preceq !\forall\overline{\alpha}.(\mathsf{proc}\ \overline{T_z})\,,\ \text{and}$$

$$\hat{}\forall\overline{\alpha'}.(\mathsf{proc}\ \overline{T_x}) \preceq ?(\mathsf{proc}\ \overline{T_y})$$

From the left subtyping and contravariance of write modes we obtain

$$\forall\overline{\alpha}.(\mathsf{proc}\ \overline{T_z}) \preceq \forall\overline{\alpha'}.(\mathsf{proc}\ \overline{T_x})$$

Lemma 6.7 yields that $\overline{\alpha'}$ is a subsequence of $\overline{\alpha}$ with rest $\overline{\alpha''}$, and that there exist $\overline{A_1}$ such that $(\mathsf{proc}\ \overline{T_z})[\overline{A_1}/\overline{\alpha''}] \preceq (\mathsf{proc}\ \overline{T_x})$ and thus $\overline{T_x} \preceq \overline{T_z}[\overline{A_1}/\overline{\alpha''}]$. From the second subtyping we similarly obtain that there exist $\overline{A_2}$ such that $\overline{T_y} \preceq \overline{T_x}[\overline{A_2}/\overline{\alpha'}]$. Since $\overline{\alpha''}$ and $\overline{\alpha'}$ are disjoint, these subtypings in combination yield:

$$\Gamma(\overline{y}) \quad\preceq\quad \overline{T_y} \quad\preceq\quad \overline{T_x}[\overline{A_2}/\overline{\alpha'}] \quad\preceq\quad \overline{T_z}[\overline{A_1}/\overline{\alpha''}][\overline{A_2}/\overline{\alpha'}]$$

Merging the sequences $\overline{A_1}$ and $\overline{A_2}$ (along the subsequencing of $\overline{\alpha'}$ and $\overline{\alpha''}$ in $\overline{\alpha}$) yields the required $\overline{T_M}$ such that $\overline{T_y} \preceq \overline{T_z}[\overline{T_M}/\overline{\alpha}]$, and hence proves (3). □

Proof of Theorem 34

The proof is by rule induction [215] over the definition of the operational semantics. We assume that $\Gamma \triangleright S(C_1)$ and show that $\Gamma \triangleright S(C_2)$.

Application: In this case there exist variables $x, \overline{y}, \overline{z}$ where $\overline{y}$ and $\overline{z}$ are disjoint sequences, and statements S, S' such that

$$\begin{aligned} S(C_1) &= (x\ \overline{y}) \parallel x := (\mathsf{proc}\ (\overline{z})\ S) \parallel S' \\ S(C_2) &= S[\overline{y}/\overline{z}] \parallel x := (\mathsf{proc}\ (\overline{z})\ S) \parallel S' \end{aligned}$$

To show $\Gamma \triangleright S(C_2)$ it suffices to show that $\Gamma \triangleright S[\overline{y}/\overline{z}]$. By Lemma 6.8 we know that there exist variables $\overline{\alpha}$ and a sequence $\overline{A}$ of types and modes such that

1. $\{\overline{\alpha}\} \cap \mathit{ftv}(\Gamma) = \emptyset$
2. $\Gamma, \overline{z}{:}\overline{T_z} \triangleright S$
3. $\Gamma(\overline{y}) \preceq \overline{T_z}[\overline{A}/\overline{\alpha}]$.

From (1) and (2) we obtain with the Type Substitution Lemma 6.5 that

$$(\Gamma, \overline{z}{:}\overline{T_z})[\overline{A}/\overline{\alpha}] = \Gamma, \overline{z}{:}\overline{T_z}[\overline{A}/\overline{\alpha}] \triangleright S$$

Let $\overline{T_y} = \Gamma(\overline{y})$ and observe that $\mathit{ftv}(\Gamma, \overline{z}{:}\overline{T_z}[\overline{A}/\overline{\alpha}]) \subseteq \mathit{ftv}(\Gamma, \overline{z}{:}\overline{T_y})$ trivially holds. Hence assumption (3) and the Subtyping Lemma 6.6 yield that

$$\Gamma, \overline{z}{:}\overline{T_y} \triangleright S\,.$$

From the Weakening Lemma 6.3 (and since $\overline{y}$ and $\overline{z}$ are disjoint sequences) we obtain

$$\Gamma, \overline{y}{:}\overline{T_y}, \overline{z}{:}\overline{T_z} \triangleright S\,,$$

and by the Variable Substitution Lemma 6.4 that

$$\Gamma, \overline{y}{:}\overline{T_y}, \overline{z}{:}\overline{T_z} \triangleright S[\overline{y}/\overline{z}]\,.$$

Finally, we apply the Weakening Lemma 6.3 again to conclude

$$\Gamma, \overline{y}{:}\overline{T_y} \triangleright S[\overline{y}/\overline{z}]\,.$$

Assignment: In this case the configurations are of the following form:

$$\begin{aligned} C_1 &= V\rho\sigma\,[]\ x := E \\ C_2 &= V\rho'\sigma'\,[]\ \mathsf{skip} \end{aligned}$$

where either $E \in \mathcal{D}$, or $E \in \mathcal{V}$ and $E \in dom(\sigma)$. If $x \in dom(\sigma)$, then the assignment is ignored, hence $\rho' = \rho, \sigma' = \sigma$, and $S(C_1) = x := E \parallel S(C_2)$. If $x \notin dom(\sigma)$, then by definition of $S(C)$ it holds that $S(C_1) = S(C_2)$. In both cases, $\Gamma \triangleright S(C_1)$ trivially implies $\Gamma \triangleright S(C_2)$.

Other Cases: The cases for pattern matching (Match) and cell exchange (Exchange) are similar to the application case but simpler because they do not need a polymorphism argument. The rule for variable declaration (Declare) and the closure rule (Closure) are trivial. □

6.3. Extensions

6.3.1. Towards Oz

In contrast to OPM [195], Plain is rather restricted. Some language features have been omitted for brevity's sake and can be added and typed using standard machinery. This includes for instance boolean conditionals and boolean types, multiple-clause case-statements and variant types, and also recursive types as needed for cyclic data structures. Other features that are omitted from Plain do not occur in modern functional languages because they complicate static typing: these include first-class patterns (for instance, by abstracting over the feature of a record or a record pattern), run-time type tests ("dynamics"), as well as several aspects of Oz's object system [91].

For Plain, the most specific difference to OPM is the omission of equations and general constraint systems. A secondary difference is the fact that cell exchange comes with a continuation. The omissions of equations and the modification of cell exchange were necessary to make the type system work. In this section we explain why. We also give a brief outlook on constraint systems.

6.3.1.1. Equational Constraints

Unification is the operation to impose an equational constraint between two data structures in the store. Assignment can be seen as a restricted equational form, and its reduction as the restriction of unification binding a previously unbound variable. For a first generalisation of assignment towards unification, consider a bidirectional assignment statement of the form $x :=: y$ that behaves either as $x := y$ or as $y := x$. Since static typing requires the types of variables to be known statically, the best possible typing rule for bidirectional equations $x :=: y$ that preserves type safety is

$$\frac{\Gamma \triangleright x{:}\hat{}P \qquad \Gamma \triangleright y{:}\hat{}P}{\Gamma \triangleright x :=: y} \quad (\textsc{Bidirect})$$

Note that this allows for P to contain nested read and/or write modes. For example, the type $P = \{a{:}?\mathsf{int}\}$ yields a useful instance of (BIDIRECT).[47]

Unification of complex data structures like records subsumes the bidirectional assignment, but it also performs a recursive traversal of a given data structure while generating additional equations. The data flow is directed dynamically. Due to this recursion, a typing rule for an equational constraint must be even more restricted than (BIDIRECT):

$$\frac{\Gamma \triangleright x{:}T \qquad \Gamma \triangleright y{:}T}{\Gamma \triangleright x{=}y} \quad T \text{ does not contain ? or !} \quad (\textsc{Unif})$$

In effect, this rule trivialises subtyping on the types of all expressions that may be mentioned by an equality constraint. Hence, in a language where telling equations is a central operation, one ends up losing virtually all subtyping. For this reason, Plain uses (directed) assignment $x := y$ instead of the equations $x{=}y$ as OPM.[48]

6.3.1.2. Cell Exchange

The cell exchange $(\mathsf{exch}\ x\ y\ z)$ in OPM does not have a continuation. Its operational semantics makes use of an equation $z'{=}z$ and hence $(\mathsf{exch}\ x\ y\ z)$ suffers from the subtyping problem. In Plain style, the semantics of $(\mathsf{exch}\ x\ y\ z)$ would appear as

$$V\rho\sigma\,[]\ (\mathsf{exch}\ x\ y\ z) \quad \longrightarrow \quad V\rho\sigma[y/\sigma(x)]\,[]\ z'{=}z \qquad \text{if } \rho(\sigma(x)) = z'$$

An immediate option to get better typing is to replace the equation by an assignment. However neither $z := z'$ nor $z' := z$ is preferred over the other. Both of them

[47] It is not by accident that this corresponds to the trivial subtyping rule for cells: cells are invariant with respect to subtyping, because they support reading and writing inseparably. Similarly, the unification operation subsumes both binding (reading) and matching (writing) on logic variables.

[48] It is, of course,safely possible to have bidirectional assignment or equations in addition to directed assignment.

are needed (recall the discussion of mode polymorphism for the channel encoding in Section 6.1.3.4). Plain's modified cell exchange $(\mathsf{exch}\ x\ y\ (z)\ S)$ defers to the continuation S the decision at which mode to use the old content of x.

Another option would have been to have cells always holding records with some (arbitrary but fixed) feature a and to combine cell exchange with field selection at a. With this convention and the semantics of exchange given by

$$V\rho\sigma[]\ (\mathsf{exch}\ x\ y\ z) \quad\longrightarrow\quad V\rho\sigma[y/\sigma(x)]\,[]\ z := z' \qquad \text{if } \rho(\sigma(x)) = z'$$

we could consider Plain's exchange statement $(\mathsf{exch}\ x\ y\ (z)\ S)$ as an abbreviation of

$$(\mathsf{local}\ (y')\ y' := \{a{:}y\} \parallel (\mathsf{exch}\ x\ y'\ z) \parallel (\mathsf{case}\ z\ (a{:}z' \ldots)\ S))$$

We decided against this option to keep cells independent of the other data structures.

6.3.1.3. Constraints over Flat Domains

The rule (BIDIRECT) suffices to explain the bidirectional data flow present in constraint systems over flat domains. This includes finite domain constraints over integers but also finite sets of integers (see [73, 146, 206, 219] and references therein). Therefore, it is straightforward to integrate a statically typed version of Oz's finite domain and finite set constraint systems into Plain: to variables from these constraint systems we assign read/write-moded types ^int and ^intset, and to typical constraint propagation procedures (see [94]) we assign types $plus$: ?(proc ^int ^int ^int) or $union$: ?(proc ^intset ^intset ^intset).

6.3.1.4. Extensible Records

Record constraints allow field-wise record construction using the selection constraint. This possibility can be added to Plain as follows: First, liberalise the store by mapping variables not only to complete records, $\sigma(x) = \{\overline{a}{:}\overline{T}\}$ but to records whose fields need not all be present: write this as $\sigma(x) = \{\overline{a}{:}\overline{T} \ldots\}$. Second, introduce an assignment statement $x.a := y$ that extends the record x by the feature a and the associated field y. The extension $\sigma, x \mapsto \{a{:}y\ldots\}$ of a store σ by a feature a at x is defined by:

$$\sigma, x \mapsto \{a{:}y\ldots\} \;=\; \begin{cases} \sigma[\{a{:}y\}/x] & \text{if } x \notin dom(\sigma) \\ \sigma[\{\bar{b}{:}\bar{y}\ a{:}y\}/x] & \text{if } \sigma(x) = \{\bar{b}{:}\bar{y}\}, a \notin \{\bar{b}\} \\ \sigma & \text{otherwise} \end{cases} \quad .$$

Execution of the statement $x.a := y$ in a configuration that maps x to a procedure, a cell, or a complete record without the field a is a type error.

Note that store extension remains a monotonic operation: the binding of a variable to an extensible record can be refined to mention more fields but fields can never be retracted.

$$V\rho\sigma\,[]\; x.a := y \quad\longrightarrow\quad V\rho\sigma, x \mapsto \{a{:}y\ldots\}\,[]\;\mathsf{skip}$$

The situation that x is bound to a procedure or a cell on reduction of $x.a := y$ is a type error. The operational semantics of pattern matching need not be changed at all. Observe, however, that the side condition $\sigma(x) = \{\bar{a}{:}\bar{z}\ldots\}$ of the corresponding rule silently adapts and now requires that $\sigma(x)$ be an *extensible* record with at least the fields $\bar{a}$ being known.

Extensible records can be easily type-checked provided that the record type mentions all fields that are accessed in a program. The rule (ExtRec) accepts an extension of x by field y at feature a whenever the type of x has write mode and contains the feature a, where it allows y's type.

$$\frac{\Gamma \triangleright x : !\{a{:}T\ldots\} \qquad \Gamma \triangleright y : T}{\Gamma \triangleright x.a := y} \quad (\textsc{ExtRec})$$

6.3.2. Let-Statement

We show how to extend Plain by a let-statement as common from functional programming. This is useful to give more accurate modes in the common situation that a variable is initialised on declaration. It is also needed to adapt an ML-style polymorphic type system to Plain. While we have considered ML-polymorphism as too restrictive with respect to some common programming patterns in Oz, it may become important as part of an ongoing language design that embeds concepts of Oz into a call-by-value functional programming language [196].

In most Plain programs, the type system requires all local variables to have read/write moded types of the form $\hat{}P$. (The only exception in this thesis is the local variable in the procedure *wait* defined on Page 162 which is used solely for synchronisation purposes.) It is clear that most variables will have both, a writer and a reader.

However, there are usually only few places where a variable can be bound but many places where it is read. A frequent case is for variables to be initialised just once on declaration and to be only read everywhere else. In order to statically exclude an erroneous second assignment as in the statement

$$(\mathsf{local}\ (x)\ x := 1 \parallel x := 2)\,,$$

one should consider the following slight syntax extension. Define a new statement

$$(\mathsf{let}\ (x := E)\ S)$$

where the variable x is bound both in E and S. The operational semantics of $(\mathsf{let}\ (x := E)\ S)$ is defined as follows:

$$\frac{V\rho\sigma[]\ (\mathsf{local}\ (x)\ x := E) \quad \longrightarrow \quad V'\rho'\sigma'[]\ \mathsf{skip}}{V\rho\sigma[]\ (\mathsf{let}\ (x := E)\ S) \quad \longrightarrow \quad V'\rho'\sigma'[]\ S}$$

This statement is type-checked according to the following customised rule:

$$\frac{\Gamma, x{:}?P \triangleright E{:}P \qquad \Gamma, x{:}?P \triangleright S}{\Gamma \triangleright (\mathsf{let}\ (x := E)\ S)} \quad (\textsc{LetRec})$$

Note that $\Gamma \triangleright (\mathsf{let}\ (x := E)\ S)$ implies $\Gamma \triangleright (\mathsf{local}\ (x)\ x := E \parallel S)$ but not vice versa.

It is possible to adapt ML-style polymorphism for a language with higher-order procedures and unification. The key observation to be made here is that logic variables behave like reference cells with respect to their interaction with polymorphism.

In more detail, one needs the following insights in order to apply standard machinery:

- A let-statement is needed as defined above.
- Procedures may have a polymorphic type only if they are introduced by a let-statement and need no evaluation.

In combination, these conditions guarantee that polymorphic procedures can be instantaneously created and bound to a fresh variable: $(\mathsf{let}\ (x := (\mathsf{proc}\ (y)\ S))\ S')$. The atomicity of declaration and binding is crucial for the type soundness result.

These conditions correspond to Wright's proposal for typing polymorphic procedures in presence of reference cells [217]. Wright solves, in a very simple manner, the problem that the naïve generalisation of the Hindley/Milner system [55, 123] from a pure functional language to a language with reference cells is not sound, *i. e.*, type safety fails [202]. Wright's solution has meanwhile been adopted in the revised definition of SML [130].

6.4. Related Work

6.4.1. Pict

One of the closest relatives of Plain is Pict, a concurrent programming language based on the π calculus [169].

6.4.1.1. The Untyped Language

The π-calculus is designed as a minimal base for concurrent computation which can express concurrent versions of data structures and procedures with channel communication as its essential computational primitive. This minimality is intriguing from a foundational perspective, but of limited practical use. When designing a high-level languages, many basic programming abstractions must be encoded. The join calculus [67], a variant of the π-calculus, is superior in this respect as it directly supports a procedural form (the "join").

Following OPM, Plain provides essential programming primitives directly: Records, higher-order procedures, and cells. Due to logic variables, there is no need for a dedicated communication primitive in Plain. Once a logic variable is bound to a data structure it becomes indistinguishable from it. This is in contrast to channels which remain distinct from the data structure they receive. Channels and locks can be expressed in Plain as synchronised data structures. Our programming experience with Oz shows that concurrent threads typically communicate through custom-built synchronised objects, where the combination of data flow synchronisation with logic variables, sequential composition and locks proves essential [91, 92]. Plain can conveniently express Pict programs as our channel encoding from Section 6.1.3.4 illustrates. However, it needs considerable effort to express in Pict partial data structures and data flow synchronisation with logic variables.

6.4.1.2. The Type System

The type system of Plain is directly inspired by the one of Pict that, in turn, is firmly based on research on type systems for functional languages, more specifically on the one around the system $F^{\omega}_{\leq}$ that combines higher-order polymorphism and subtyping (see [32, 36, 76, 177] and [169] for further references). More specifically, we have applied Pierce and Sangiorgi's mode system for channels to a concurrent language with logic variables. However, the meaning of modes differs between Pict and Plain. In Pict, $?P$ is the type of a channel carrying values of type P: the mode is not separate from the channel type constructor. Hence, in contrast to Plain, nested modes do make sense in Pict. For instance, the Pict type $\hat{}?P$ describes input/output channels carrying input channels for values of type P: this reflects the fact that channels are entities separate from the data they carry and that there are explicit operations to access this data. In contrast, logic variables can be seen as once only communication channels that become indistinguishable from the data structure they eventually receive. In addition, mode polymorphism as in Plain, which allows one to abstract over a mode and then to instantiate it separately, does not suggest itself in Pict; there, mode polymorhism can be expressed with bounded polymorphic types [36].

We have used universal higher-order polymorphism [76, 177]. This is in contrast to Pict whose basic form of polymorphism is existential [169]. There, messages are the basic typed entities ("packages") about which only partial information is revealed to the receiver process (abstract types) [31, 135]. For a procedural language like Plain, universal polymorphism seems more appropriate. To facilitate a direct comparison between Plain and Pict, we have also defined a type system for Plain with existential polymorphism [144] (note, however, that existential polymorphism can be encoded by higher-order universal polymorphism [178]). Pict's type system is much larger than the one we presented for Plain, containing variant types, recursive types, kinds, etc. [169] We have kept Plain simple to focus on the language design point of view, but we do not foresee any Plain-specific difficulties in extending its type system accordingly.

6.4.1.3. Type Inference

We have not considered the type inference problem for Plain, which is very likely to be undecidable: The closely related type inference problem for the the "polymorphic λ calculus" System F [76, 177] is undecidable [214], and the addition of subtyping does not seem to make type inference any simpler. Currently, the design of type inference heuristics for type systems with subtyping and higher-order polymorphism is a challenging research topic. With Pict, Pierce and Turner have made important progress on this issue but it seems not to be settled, in particular with respect to recursive types. Initially, Pict's approach to type inference was based on an algorithm that Cardelli described for a functional language, but this is no longer the case [33, 167, 168]. We expect these experiences to be useful for a decent implementation of Plain; different in spirit, we also anticipate the usefulness of a mechanism to "bypass the type checker" as in TEL [192] which might change the game considerably.

6.4.2. Modes in Logic Programming

In logic programming, modes often describe the instantiation state of procedure arguments (ground, non-ground, free) directly before or both, before and after procedure application [29, 56, 192, 198].

In the typed Prolog dialects TEL and Mercury [192, 198], the mode of procedure arguments must be declared. For instance, input arguments must be ground on procedure application, and output arguments will be ground thereafter. Mercury strictly enforces this discipline such that computation with partially determined data structures (that is, arguments which are neither ground nor free) becomes impossible. In TEL, on the other hand, variables can be declared as "open" which enables all programming techniques developed for Prolog. This effectively bypasses the type checker which treats open variables as ground. In both systems, modes are very simple due to their restriction to ground arguments (at least during type checking). The system we present for Plain is more complex: since Plain caters for partially determined data structures, its type and mode system must deal with modes that occur on every level in the structure of a data type.

In concurrent logic programming, variable moding has always played a special role. Preceding Maher's logical characterisation of synchronisation as entailment [120], read-only annotations on variables were used to explain synchronisation patterns operationally; these could become rather complex, as for instance in Concurrent Prolog [187]. The annotations were checked during unification, and the attempt to bind an unbound read-only variable lead to a suspension. Modes were also considered as a means to exclude failure: since failure is due to disagreement between two producers for the same variable, multiple producers were excluded. Notice that this concept of modes is a *resource-sensitive* one, which contains information both on directional data flow and multiplicities.

The Relational Language [47] and its successor Parlog [79] made the declaration of input and output arguments obligatory in procedures; mode declarations were checked at run-time. The modes only referred to the top-level constructor of a record, not to its subterms. Strand [66] put away with unification altogether and disallowed multiple assignment to the same variable; the second assignment to the same variable lead to a run-time error. Directed variables [113] as in Doc and Janus [98, 181] are restricted even further in that they disallow multiple *readers*. Thus directed variables express point-to-point communication (*cf.* "linear channels") rather than a multicasting. While the write-once property can be guaranteed statically, the read-once property remains to be checked at run-time. More recently, Ueda proposed to call programs with directed variables *well-moded* [204, 205] and gave algorithms to check well-modedness in Flat GHC programs statically.

6.4.3. ML-style Polymorphism for Logic Variables

MLOG [172] is an extension of ML by logic variables due to Poirriez. In MLOG, the type system is used to separate strictly the "functional types" (such as "int") from the "logic types" (such as "unbound or int") used to describe data structures that may contain embedded logic variables. This strict seperation simplifies the implementation of logic variables as an extension of existing ML compilers, but it overlooks the expressiveness of logic variables as a synchronisation mechanism in concurrent and distributed programming. The typing of logic variables in MLOG is based on an proposal by Leroy and Weis [117] that is nowadays outdated by Wright's proposal [217].

Minamide [131] describes a type system for a restricted form of logic variables in a statically typed functional language. In Minamide's system, no data structure may contain more than one embedded logic variable (which he calls a "hole"). Although this considerably restricts the expressiveness of logic variables, it enables some programming techniques that are well-known in the logic programming community [200], such as difference lists and tail-recursive definition of procedures like *append* or *map*.

Mycroft and O'Keefe [147] have adapted the ML-type system [123] to logic programming. Their system underlies the programming languages Gödel and Mercury [97, 198]. Some preliminary results on ML-style type checking for an Oz-style language *including* feature tree constraints can be found in [138].

6.4.4. Types in Concurrent Programming

In sequential programming, the prevailing view of type checking is that it guarantees *safety* of operations on data structures. This view must be refined in concurrent computation since concurrency introduces the possibility of additional erroneous situations such as deadlock, livelock, starvation, race conditions, and the like. For example, it is desirable to guarantee the *availability* of services in a client-server system. From this point of view, it is useful to consider *types as protocols* that specify the interaction between concurrent processes, and to view type checking as *protocol verification*. Since memory and data structures can also be modelled as concurrent processes, this point of view properly generalises the traditional sequential approach.

We have taken the view that modes in type systems for concurrent languages describe (very simple) protocols. More complex protocols might account for the multiplicity of operations on resources, such as requiring "at least one reply per request" or "exactly one release per lock on a semaphore"; or they might describe more complex temporal behaviours such as the behaviour of process that offers services that vary over time. In this thesis we do not deal with multiplicity or temporal protocol properties. For some recent work on resource sensitive type system for concurrent programming languages see [25] and references therein, and for behavioural type systems see Nierstrasz [156].

7. Conclusion and Directions of Further Research

The question that motivated the research reported in this thesis was:

> *How can we provide some static type checking to the dynamically typed language Oz?*

We have approached this question from two complementary sides, and we could contribute a number of results to both of them.

First, we have shown that an expressive strong type system is possible for a language that combines key features of Oz, namely higher-order procedures, logic variables and partially determined data structures, cells, and records. The design of the corresponding language Plain marks a design option for "strongly typed Oz", and Plain is one inspiration for an ongoing language design that embeds concepts of Oz into a call-by-value functional programming language [196]. In addition, Plain provides a new link between two prominent concurrent programming models: concurrent constraints and process calculi. However, Plain is not Oz. Plain is not a constraint language anymore and, most notably, leaves open the question of strong typing for feature tree constraints.

Second, we have suggested failure diagnosis as a new class of set-based program analysis that is dual to strong typing and that does not attempt to prove the absence of run-time errors but their inevitability. We have shown how to achieve this goal for a concurrent constraint language over infinite trees. We have also proposed a set-based analysis for a large fragment of Oz. This analysis seems intuitively reasonable and an experimental implementation has been encouraging by proving its usefulness in finding errors. Unfortunately, a correctness result for this analysis has not been achieved. Such a result should, independent on the analysis, characterise in which sense a program is indeed ill-formed if the analysis rejects it.

We leave the problem open as a challenge for future research. There seem to be two options to tackle it. Either one could try to find a *denotational* semantics for (a fragment of) Oz against which to judge correctness of the analysis; we expect this to be fairly tricky, given that Oz subsumes both CC and the untyped λ-calculus. Or one could try to justify the analysis solely by reasoning about Oz's *operational* semantics.

As part of our set-based failure diagnosis, we have defined a new system of set constraints that is appropriate for the set-based analysis of languages that support record structures. We have settled many algorithmic and complexity issues for this constraint system that are relevant for their application in program analysis. We have argued the design decisions that lead to this constraint system, and we hope that it can be of independent interest to the constraint community.

The more general question that I consider still open is:

> *Which is the* best *way to provide some static type checking for a concurrent constraint language with higher-order procedures?*

The critical word "best" asks for a compromise between so diverse aspects as effectiveness in static debugging, restrictions on programming flexibility, efficiency of implementation, scalability and ease of use. The right balance between these can only be obtained by practical experimentation.

Let us mention a number of approaches that we consider worth investigating more closely next.

Flanagan's static debugger for Scheme [63, 65] provides inspiration for modular program analysis with set-constraints, as well as for the presentation of constraints.

Aiken and Fähndrich have proposed a program analysis with constraints that are interpreted over a special domain. These constraints lie "half-way between" equality constraints over trees and set inclusion constraints [62]. The hope is that this system allows one to have one's cake and eat it, too: exploit the expressiveness of set constraints where necessary, but enjoy the efficiency of solving tree constraints where possible.

We have observed more than once the need for data flow information in order to improve the program analysis. It is hence desirable to investigate data flow analysis for Oz in a more principled way, as well as its interaction with set-based analysis. Most relevant in this context appear Shivers's [191] and, once again, Flanagan's set-based data flow analysis for Scheme.

Type systems that combine higher-order polymorphic types and subtyping (as in Plain or Pict) suffer from the fact that they do not allow automated type inference [214]. Pierce and Turner propose a heuristics for automated type reconstruction that they call "local type inference" [168], which they claim to be simple and intuitive enough such that it can be part of the language definition (as opposed to being implementation specific) and as such can easily be absorbed by programmers.

Recently, Smolka has sketched a redesign of Oz that embeds constraint programming concepts into a (dynamically typed) version of ML [196]. It may be interesting to reconsider Plain's type system in this context: Independent of syntactic difference, we expect the technical insight to survive that subtyping on logic variables requires a mode discipline. However, the situation might change in a subtle way due to a different choice of primitive operations on logic variables (such as the ones sketched in the slides complementary to [196]). On the other hand, the most natural choice for a strong type system for an extension of ML is an appropriate extension of the ML type system. As for Plain, the immediate challenge for a typed Oz remains: how to treat tree constraints in a strong type system. Some preliminary results on this topic can be found in [138].

In this context, we must reconsider the restrictions of the ML type system that have led us to base Plain on a more powerful type system. In order to overcome these restrictions we could try to extend the ML type system. Language designs of interest in this context include O'Caml [72, 175], a promising attempt to integrate objects into ML, and O'Labl [71], an extension of O'Caml by polymorphic records and variants. Alternatively, we could investigate a more flexible interaction of static and dynamic type checking than usual, and allow the programmer to "bypass the type checker" for doing something "ill-typed". To my knowledge, this has not been pursued yet in the context of a functional language.

A. Mathematical Preliminaries

In this chapter we introduce some basic concepts from set theory and predicate logic, as well as some notational conventions.

A.1. Sets, Relations, and Mappings

A *set* is an unordered collection of objects called its *elements*. We write $x \in s$ if x is an element of the set σ, We write $\{x_1, \ldots, x_n\}$ for the *finite set* that contains exactly the elements x_1 through x_n, and we write $\emptyset$ for the *empty set* that has no elements. If P is a property, then we denote with $\{x \mid P(x)\}$ the set of all elements that have property P. The number of elements in σ, *i. e.*, the *cardinality of* σ, is written as $\|s\|$. A set is called *finite* or *infinite* depending whether its cardinality is finite or infinite. A set σ_1 is said to be a *subset* of a set σ_2, written $\sigma_1 \subseteq \sigma_2$, if every element of σ_1 is also an element of σ_2. If σ_1, and σ_2 are arbitrary sets, we write $\sigma_1 \cup \sigma_2$ for the *union of* σ_1 *and* σ_2, $\sigma_1 \cup \sigma_2 = \{x \mid x \in \sigma_1 \text{ or } x \in \sigma_2\}$, $\sigma_1 \cap \sigma_2$ for the *intersection of* σ_1 *and* σ_2, $\sigma_1 \cup \sigma_2 = \{x \mid x \in \sigma_1 \text{ and } x \in \sigma_2\}$, $\sigma_1 \backslash \sigma_2$ for the *difference of* σ_1 *and* σ_2, $\sigma_1 \backslash \sigma_2 \; \{x \mid x \in \sigma_1 \text{ and } x \notin \sigma_2\}$, For all sets σ, we write $\mathcal{P}(s)$ for the *powerset of* σ, *i. e.*, the set of all subsets of σ given by $\mathcal{P}(s) = \{s' \mid s' \subseteq s\}$, and as $\mathcal{P}^+(s)$ the set of all nonempty subsets of σ, *i. e.*, $\mathcal{P}^+(s) = \mathcal{P}(s) \backslash \emptyset$.

An *n-tuple* $(x_1, \ldots, x_n)$ is a finite sequence of n objects. An *n-ary relation* between sets σ_1 through σ_n is set of tuples $(x_1, \ldots, x_n)$ such that $x_1 \in \sigma_1, x_2 \in \sigma_2, \ldots$, and $x_n \in \sigma_n$. If R is a binary relation between σ_1 and σ_2 and $x_1 \in \sigma_1$ and $x_2 \in \sigma_2$, then we allow xRy as an alternative notation for $(x_1, x_2) \in R$. If R is a binary relation, then we write R^* for the *reflexive and transitive closure of* R, *i. e.*, the smallest relation containing R that is reflexive and transitive.

A *(total) function* f from σ_1 to σ_2, written $f : \sigma_1 \to \sigma_2$, is a binary relation between σ_1

and σ_2 such that for every $x_1 \in \sigma_1$ there is *exactly* one $x_2 \in \sigma_2$ such that $(x_1, x_2) \in f$. A *partial function* f from σ_1 to σ_2, written $f : \sigma_1 \rightharpoonup \sigma_2$, is a binary relation between σ_1 and σ_2 such that for every $x_1 \in \sigma_1$ there is *at most* one $x_2 \in \sigma_2$ such that $(x_1, x_2) \in f$.

Given a (total or partial) function f from σ_1 to σ_2, the *domain of f* is the subset of σ_1 on which f is defined, and the *range of f* is the subset of σ_2 whose elements are obtained as $f(x_1)$ for some $x_1 \in \sigma_1$. Formally, $dom(s) = \{x_1 \mid \text{exists } x_2 \in \sigma_2 \text{ such that } f(x_1) = x_2\}$ and $rg(s) = \{x_2 \mid \text{exists } x_1 \in \sigma_1 \text{ such that } x_2 = f(x_1)\}$. The domain of a total function $f : \sigma_1 \to \sigma_2$ is σ_1.

The *composition* of $f : \sigma_1 \rightharpoonup \sigma_1'$ and $g : \sigma_2 \rightharpoonup \sigma_2'$ where $rg(f) \subseteq dom(g)$ is the function $g \circ f : \sigma_1 \to \sigma_2'$ defined by $(g \circ f)(x_1) = g(f(x_1))$ for all $x_1 \in \sigma_1$. If f is a function from σ_1 to σ_2, and if $x_1 \in \sigma_1$ and $x_2 \in \sigma_2$, then $f[x_2/x_1]$ defines the function that coincides with f on $dom(f) \setminus \{x_1\}$ and maps x_1 to x_2: $f[x_2/x_1] = \{(x_1, x_2)\} \cup \{(x', x'') \mid (x', x'') \in f \text{ and } x_1 \neq x'\}$. We call $f[x_1/x_2]$ an *extension of f*.

A.2. Predicate Logic

A *signature* Σ is a ranked alphabet of function and predicate symbols, where every function symbol f and every predicate symbol p is associated an non-negative *arity* $\mathsf{ar}(f)$ *resp.* $\mathsf{ar}(p)$. If $\mathsf{ar}(f) = 0$ we call f a *constant symbol*. Let A be some set. An (A, Σ)-*interpretation* is a function that maps every predicate symbol $p \in \Sigma$ with $\mathsf{ar}(p) = n$ to an n-ary relation over A, and every function symbol $f \in \Sigma$ with $\mathsf{ar}(f) = n$ to a function the set of n-tuples over A to A. A Σ-*structure* is a pair $\mathcal{A} = (A, I)$ where A is a the *domain* of $\mathcal{A}$, and I is an (A, Σ)-interpretation.

A *first-order language* $\mathcal{L}$ consists of a set $\mathcal{V}$ of variables, a signature Σ, a collection of logic *connectives* such as $\wedge, \vee, \neg, \rightarrow$, and *quantifiers* $\exists$ and $\forall$, possibly an equality symbol $=$ and usually parentheses '(' and ')'. We define as usual the *first-order terms* and the *formulas* over $\mathcal{L}$, as well as the set of *variables free (resp., bound) in a formula* Φ which we write as *fv*(Φ) (*resp.*, *bv*(Φ)). A formula Φ is called *closed* if *fv*(Φ) $= \emptyset$. A *constraint system* is given by a first-order (constraint) language $\mathcal{L}$ and a structure $\mathcal{A}$. The constraint language $\mathcal{L}$ defines a set of formulas called *constraints*. All constraint languages in this thesis contain conjunction $\wedge$ as the only logic connective.

If $\mathcal{A} = (A, I)$ is a Σ-structure, then an $\mathcal{A}$-*valuation* is a function $\alpha : \mathcal{V} \to A$ from the variables into the domain of $\mathcal{A}$. We define as usual the concept of a valuation α *satisfying* a formula Φ in $\mathcal{A}$, written $\alpha \models_{\mathcal{A}} \Phi$. Given a structure $\mathcal{A}$, we say a formula Φ is *satisfiable* in $\mathcal{A}$, written $\mathcal{A} \models \Phi$, if there exists a valuation α such that $\alpha \models_{\mathcal{A}} \Phi$ and in this case α is called a *solution* of Φ in $\mathcal{A}$. We write the set of solutions of a formula A as $\mathrm{Sol}(A)$. We say that Φ is *valid* in $\mathcal{A}$, or that $\mathcal{A}$ is a *model* of Φ, if $\alpha \models_{\mathcal{A}} \Phi$ for all valuations α. We say that Φ *entails* Φ' in $\mathcal{A}$, written $\Phi \models_{\mathcal{A}} \Phi'$, if $\Phi \to \Phi'$ is valid in $\mathcal{A}$, and that Φ_1 is *equivalent* to Φ_2 (in $\mathcal{A}$) if $\Phi_1 \leftrightarrow \Phi_2$ is valid in $\mathcal{A}$. The *satisfiability problem* for a constraint system $(\mathcal{L}, \mathcal{A})$ is whether an arbitrary constraint

$\varphi \in \mathcal{L}$ is satisfiable in $\mathcal{A}$, and The *entailment* problem for a constraint system $(\mathcal{L}, \mathcal{A})$ is $\varphi_1 \models_{\mathcal{A}} \varphi_2$ holds for two constraints $\varphi_1, \varphi_2 \in \mathcal{L}$. A *theory* is a set of first-order formulas. Given a constraint system $(\mathcal{L}, \mathcal{A})$ we call the associated *first-order theory* the set of formulas over $\mathcal{L}$, extended by arbitrary first-order connectives and quantifiers, that are valid in $\mathcal{A}$.

A.3. Notational Conventions

A sequence of syntatic objects $X_1, \ldots, X_n$ is abbreviated as $\overline{X}$ if the length n of the sequence does not matter, and we denote with $|\overline{X}|$ the length of such a sequence. For two syntactic objects X and Y of the same category, we denote with $[Y/X]$ the *substitution* of Y for X in syntactic objects. The *simultaneous substitution* $[Y_1/X_1] \ldots [Y_n/X_n]$ of $Y_1, \ldots, Y_n$ for pairwise distinct $X_1, \ldots, X_n$ is abbreviated by $[\overline{Y}/\overline{X}]$. Moreover, we denote with $\overline{X}{:}\overline{Y}$ the finite sequence of pairs $X_1{:}Y_1 \ldots X_n{:}Y_n$.

"Now I declare that's too bad!" Humpty Dumpty cried, breaking into a sudden passion. "You've been listening at doors – and behind trees – and sown chimneys – or you couldn't have known it!"

"I haven't, indeed!" Alice said very gently. "It's in a book."

– Lewis Carroll, *Through the Looking Glass*

Bibliography

[1] ABADI, MARTÍN & LUCA CARDELLI (1996). *A Theory of Objects*. Springer-Verlag, Berlin. Monographs in Computer Science.

[2] ABELSON, HAROLD; GERALD JAY SUSSMAN, & JULIE SUSSMAN (1996). *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA, 2nd edn.

[3] AGHA, GUL (1986). *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, MA.

[4] AIKEN, ALEXANDER (1994). Illyria system and documentation. `http://http.cs.berkeley.edu/~aiken`.

[5] AIKEN, ALEXANDER (May 1994). Set constraints: Results, applications, and future directions. In *Proceedings of the 2nd Workshop on Principles and Practice of Constraint Programming*, edited by A. H. Borning, vol. 874 of *Lecture Notes in Computer Science*, pp. 326–335, Orcas Island, Washington. Springer-Verlag, Berlin.

[6] AIKEN, ALEXANDER; MANYEL FÄHNDRICH; JEFFREY S. FOSTER, & ZHENDONG SU (1998). A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the 2nd International Workshop on Types in Compilation*, edited by X. Leroy, pp. 165–169. Preliminary proceedings published as technical report of Research Institute for Mathematical Sciences (RIMS), Kyoto University.

[7] AIKEN, ALEXANDER; DEXTER KOZEN; MOSHE VARDI, & ED WIMMERS (Sep. 1993). The Complexity of Set Constraints. In *Proceedings of the 7th Conference on Computer Science Logic*, edited by E. Börger, Y. Gurevich, & K. Meinke, vol. 832 of *Lecture Notes in Computer Science*, pp. 1–17, Swansea, Wales. Springer-Verlag, Berlin.

[8] AIKEN, ALEXANDER; DEXTER KOZEN, & ED WIMMERS (1995). Decidability of systems of set constraints with negative constraints. *Information and Computation*, 122(1):30–44. Academic Press, New York, N.Y.

[9] AIKEN, ALEXANDER; JOHN H. WILLIAMS, & E.L. WIMMERS (1994). The FL project: Design of a functional language. Tech. rep., IBM Almaden Research Center. `http://http.cs.berkeley.edu/~aiken/pubs.html`.

[10] AIKEN, ALEXANDER & E.L. WIMMERS (Jun. 1992). Solving systems of set constraints. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*, pp. 329–340.

[11] AIKEN, ALEXANDER & ED WIMMERS (Jun. 1993). Type inclusion constraints and type inference. In *Proceedings of the 6th ACM Conference on Functional Programming and Computer Architecture*, pp. 31–41, Copenhagen, Denmark. ACM Press, New York.

[12] AIKEN, ALEXANDER; ED WIMMERS, & T.K. LAKSHMAN (Jan. 1994). Soft typing with conditional types. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, pp. 163–173, Portland, Oregon. ACM Press, New York.

[13] AIKEN, ALEXANDER; ED WIMMERS, & JENS PALSBERG (1997). Optimal representation of polymorphic types with subtyping. In *Proceedings of the 24th Annual Symposium on Theoretical Aspects of Computer Software*, edited by P. Degano, R. Gorrieri, & A. Marchetti-Spaccamela, vol. 1281 of *Lecture Notes in Computer Science*, pp. 47–76. Springer-Verlag, Berlin.

[14] AÏT-KACI, HASSAN & ROGER NASR (1986). LOGIN: A logic programming language with built-in inheritance. *The Journal of Logic Programming*, 3(3):185–215. Elsevier Science Publishers B.V. (North Holland).

[15] AÏT-KACI, HASSAN & ANDREAS PODELSKI (Jul., Aug. 1993). Towards a meaning of life. *The Journal of Logic Programming*, 16(3 – 4):195–234. Elsevier Science Publishers B.V. (North Holland).

[16] AÏT-KACI, HASSAN; ANDREAS PODELSKI, & GERT SMOLKA (Jan. 1994). A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283. Elsevier Science Publishers B.V. (North Holland).

[17] ARMSTRONG, JOE; ROBERT WILLIAMS; MIKE VIRDING, & CLAES WIKSTROEM (1996). *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, N.J., 2nd edn.

[18] ARVIND; RISHIYUR S. NIKHIL, & KESHAV K. PINGALI (1989). I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632. ACM Press, New York.

[19] BACHMAIR, LEO; HARALD GANZINGER, & UWE WALDMANN (1993). Set constraints are the monadic class. In *Proceedings of the 8th IEEE Symposium on Logic in Computer Science*, pp. 75–83. IEEE Computer Society Press.

[20] BACKOFEN, ROLF (1995). A complete axiomatization of a theory with feature and arity constraints. *The Journal of Logic Programming*, 24(1 – 2):37–71. Special Issue onComputational Linguistics and Logic Programming. Elsevier Science Publishers B.V. (North Holland).

[21] BACKOFEN, ROLF & GERT SMOLKA (Jul. 1995). A complete and recursive feature theory. *Theoretical Computer Science*, 146(1–2):243–268. Elsevier Science Publishers B.V. (North Holland).

[22] BARTH, PAUL S.; RISHIYUR S. NIKHIL, & ARVIND (Aug. 1991). M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the 5th ACM Conference on Functional Programming and Computer Architecture*, edited by J. Hughes, vol. 523 of *Lecture Notes in Computer Science*, pp. 538–568. ACM Press, New York.

[23] BENHAMOU, FRÉDÉRIC & ALAIN COLMERAUER (eds.) (1993). *Constraint Logic Programming: Selected Research*. The MIT Press, Cambridge, MA.

[24] BOUDOL, GÉRARD (May 1992). Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia Antipolis.

[25] BOUDOL, GÉRARD (Dec. 1997). Typing the use of resources in a concurrent calculus. In *3rd Asian Computing Science Conference*, edited by R. L. Shyamasundar & K. Ueda, vol. 1345 of *Lecture Notes in Computer Science*, pp. 239–253, Kathmandu, Nepal. Springer-Verlag, Berlin.

[26] BOURDONCLE, FRANÇOIS (1993). Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, edited by R. Cartwright, vol. 28 of *ACM SIGPLAN Notices*, pp. 46–55. ACM Press, New York.

[27] BOURDONCLE, FRANÇOIS & STEFAN MERZ (Jan. 1997). Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pp. 302–315, Paris, France. ACM Press, New York.

[28] BRINCH HANSEN, PER (Apr. 1970). The nucleus of a multiprogrammed system. *Communications of the ACM*, 13(4):238–250. ACM Press, New York.

[29] BRONSARD, FRANÇOIS; T.K. LAKSHMAN, & UDAY S. REDDY (1992). A framework of directionality for proving termination of logic programs. In *International Conference and Symposium on Logic Programming*, edited by K. Apt, pp. 321–335. The MIT Press, Cambridge, MA.

[30] BURNS, ALAN (1988). *Programming in occam 2*. The Instruction Set Series. Addison-Wesley, Reading, MA.

[31] CARDELLI, LUCA (1984). A semantics of multiple inheritance. In *Semantics of Data Types*, edited by G. Kahn, D. MacQueen, & G. Plotkin, vol. 173 of *Lecture Notes in Computer Science*, pp. 51–67. Springer-Verlag, Berlin. Full version in *Information and Computation*, 76(2/3):138–164, 1988.

[32] CARDELLI, LUCA (Jan. 1988). Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pp. 70–79. ACM Press, New York.

[33] CARDELLI, LUCA (Feb. 1993). An implementation of $F_{<:}$. Tech. Rep. 97, Digital Systems Research Center, CA.

[34] CARDELLI, LUCA (1996). Type systems. In *CRC Handbook of Computer Science and Engineering*, edited by A. B. Tucker. CRC Press LLC.

[35] CARDELLI, LUCA; JAMES DONAHUE; LUCILLE GLASSMAN; MICK JORDAN; BILL KALSOW, & GREG NELSON (Nov. 1989). Modula-3 report (revised). Tech. Rep. 52, Digital Systems Research Center, CA.

[36] CARDELLI, LUCA; SIMONE MARTINI; JOHN C. MITCHELL, & ANDRE SCEDROV (1994). An extension of System F with subtyping. *Information and Computation*, 109(1–2):4–56. Academic Press, New York, N.Y.

[37] CARDELLI, LUCA & PETER WEGNER (Dec. 1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522. ACM Press, New York.

[38] CARTWRIGHT, ROBERT & MICHAEL FAGAN (Jun. 1991). Soft typing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, vol. 26 of *ACM SIGPLAN Notices*, pp. 278–292. ACM Press, New York.

[39] CHARATONIK, WITOLD & LESLEK PACHOLSKI (1994). Negative set constraints with equality. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pp. 128–136. IEEE Computer Society Press.

[40] CHARATONIK, WITOLD & LESLEK PACHOLSKI (1994). Set constraints with projections are in NEXPTIME. In *Proceedings of the* 35*th* *IEEE Symposium on Foundations of Computer Science*, pp. 642–653. IEEE Computer Society Press.

[41] CHARATONIK, WITOLD & ANDREAS PODELSKI (1996). The independence property of a class of set constraints. In *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming*, edited by E. C. Freuder, vol. 1118 of *Lecture Notes in Computer Science*, pp. 76–90. Springer-Verlag, Berlin.

[42] CHARATONIK, WITOLD & ANDREAS PODELSKI (1997). Set constraints with intersection. In *Proceedings of the 12th IEEE Symposium on Logic in Computer Science*, pp. 352–361, Warsaw, Poland. IEEE Computer Society Press.

[43] CHARATONIK, WITOLD & ANDREAS PODELSKI (1997). Solving set constraints for greatest models. Technical Report MPI-I-97-2004, Max-Planck Institut für Informatik.

[44] CHARATONIK, WITOLD & ANDREAS PODELSKI (1998). Co-definite set constraints. In *International Conference on Rewriting Techniques and Applications*, edited by T. Nipkow, no. 1379 in Lecture Notes in Computer Science, pp. 211–225, Tsukuba, Japan. Springer-Verlag, Berlin.

[45] CHARATONIK, WITOLD & ANDREAS PODELSKI (1998). Set-based analysis of reactive infinite-state systems. In *Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, edited by B. Steffen, no. 1384 in Lecture Notes in Computer Science. Springer-Verlag, Berlin.

[46] CLARK, KEITH (1978). Negation as Failure. In *Logic and Databases*, edited by H. Gallaire & J. Minker, pp. 293–322. Plenum Press, New York, NY.

[47] CLARK, KEITH L. & STEVE GREGORY (1981). A relational language for parallel programming. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 171–178. ACM Press, New York.

[48] CLINGER, WILLIAM & JONATHAN REES (Jul. - Sep. 1991). The Revised^4 Report on the Algorithmic Language Scheme. *LISP Pointers*, IV(3):1–55. ACM Press, New York.

[49] COLMERAUER, ALAIN (1970). Les systèmes-q ou un formalisme pour analyser et synthétiser des phrase sur ordinateur. Publication Interne 43, Université de Montreal.

[50] COLMERAUER, ALAIN (1984). Equations and inequations on finite and infinite trees. In *Proceedings of the 2nd International Conference on Fifth Generation Computer Systems*, edited by ICOT, pp. 85–99. Omsha Ltd., Tokyo and North-Holland, Amsterdam.

[51] COLMERAUER, ALAIN; H. KANOUI, & MICHAEL VAN CANEGHEM (1983). Prolog, theoretical principles and current trends. *Technology and Science of Informatics*, 2(4):255–292.

[52] COOK, STEPHEN A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pp. 151–158. ACM Press, New York.

[53] COUSOT, PATRICK & RADHIA COUSOT (Jan. 1977). Abstract interpretation: A unified lattice model for static analysis of programs by contruction or approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238–252. ACM Press, New York.

[54] COUSOT, PATRICK & RADHIA COUSOT (Jun. 1995). Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pp. 170–181, La Jolla, California. ACM Press, New York.

[55] DAMAS, LUIS & ROBIN MILNER (Jan. 1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212. ACM Press, New York.

[56] DEBRAY, SAUMYA K. & DAVID S. WARREN (1988). Automatic mode inference for Prolog programs. *The Journal of Logic Programming*, 5(3):207–229. Elsevier Science Publishers B.V. (North Holland).

[57] DENNIS, JACK (1974). First version of a data flow procedure language. In *Proceedings Colloque sur la Programmation*, edited by B. Robinet, vol. 19 of *Lecture Notes in Computer Science*, pp. 362–376, Paris. Springer-Verlag, Berlin.

[58] DEVIENNE, PHILLIPE; JEAN-MARC TALBOT, & SOPHIE TISON (1997). Solving classes of set constraints with tree automata. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, edited by G. Smolka, vol. 1330 of *Lecture Notes in Computer Science*, pp. 62–76. Springer-Verlag, Berlin.

[59] DEVIENNE, PHILLIPE; JEAN-MARC TALBOT, & SOPHIE TISON (1998). Codefinite set constraints with membership expressions. In *Proceedings of the International Conference and Symposium on Logic Programming*, edited by J. Jaffar. The MIT Press, Cambridge, MA. to appear.

[60] DIETZFELBINGER, MARTIN; ANNA KARLIN; KURT MEHLHORN; FRIEDHELM MEYER AUF DER HEIDE; HANS ROHNERT, & ROBERT E. TARJAN (Aug. 1994). Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing*, 23(4):738–761. Society for Industrial and Applied Mathematics.

[61] DIJKSTRA, EDSGER W. (1976). Co-operating sequential processes. In *Programming Languages*, edited by F. Genys, pp. 43–112. Academic Press, New York, N.Y.

[62] FÄHNDRICH, MANUEL & ALEXANDER AIKEN (1997). Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, edited by P. Van Hentenryck, vol. 1302 of *Lecture Notes in Computer Science*, pp. 114–126, Paris, France. Springer-Verlag, Berlin.

[63] FINDLER, ROBERT BRUCE; CORMAC FLANAGAN; MATTHEW FLATT; SHRIRAM KRISHNAMURTHI, & MATTHIAS FELLEISEN (May 1997). DrScheme: A pedagogic programming environment for scheme. In *International Symposium on Programming Language Implementation and Logic Programming*, edited by H. Glaser, P. Hartel, & H. Kuchen, vol. 1292 of *Lecture Notes in Computer Science*, pp. 369–388. Springer-Verlag, Berlin.

[64] FLANAGAN, CORMAC (May 1997). *Effective Static Debugging via Componential Set-Based Analysis*. Ph.D. thesis, Rice University, Houston, Texas.

[65] FLANAGAN, CORMAC & MATTHIAS FELLEISEN (Jun. 1997). Componentional set-based analysis. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, vol. 32 of *ACM SIGPLAN Notices*, pp. 235–248. ACM Press, New York.

[66] FOSTER, IAN & STEPHEN TAYLOR (1989). *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, N.J.

[67] FOURNET, CÉDRIC & GEORGES GONTHIER (Jan. 1996). The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pp. 372–385, St. Petersburg Beach, Florida. ACM Press, New York.

[68] FRIEDMAN, DANIEL P.; MITCHELL WAND, & CHRISTOPHER T. HAYNES (1992). *Essentials of Programming Languages*. The MIT Press, Cambridge, MA.

[69] FRÜHWIRTH, THOM; EHUD SHAPIRO; MOSHE VARDI, & EYAL YARDENI (1991). Logic programs as types for logic programs. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pp. 300–309. IEEE Computer Society Press.

[70] GAREY, MICHAEL R. & DAVID S. JOHNSON (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York.

[71] GARRIGUE, JACQUES (1998). *Objective Label User's Manual*. Research Institute for Mathematical Sciences (RIMS), Kyoto University, `http://wwwfun.kurims.kyoto-u.ac.jp/soft/lsl/manual.html`.

[72] GARRIGUE, JACQUES & DIDIER RÉMY (Sep. 1997). Extending ML with semi-explicit higher-order polymorphism. In *Proceedings of the 3rd Annual Symposium on Theoretical Aspects of Computer Software*, edited by M. Abadi & T. Ito, vol. 1281 of *Lecture Notes in Computer Science*, pp. 20–46. Springer-Verlag, Berlin.

[73] GERVET, CARMEN (1997). Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244. Elsevier Science Publishers B.V. (North Holland).

[74] GILLERON, RÉMI; SOPHIE TISON, & MARC TOMMASI (1993). Solving systems of set constraints using tree automata. In *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Software*, edited by P. Enjalbert, A. Finkel, & K. W. Wagner, vol. 665 of *Lecture Notes in Computer Science*, pp. 505–514. Springer-Verlag, Berlin.

[75] GILLERON, RÉMI; SOPHIE TISON, & MARC TOMMASI (1993). Solving systems of set constraints with negated subset relationships. In *Proceedings of the 34th IEEE Symposium on Foundations of Computer Science*, pp. 372–380. IEEE Computer Society Press.

[76] GIRARD, JEAN-YVES (1972). *Interprétation fonctionelle et élimination des coupures de l'arithmetique d'ordre supérieur*. Ph.D. thesis, Université de Paris VII.

[77] GOLDBERG, ADELE & DAVID ROBSON (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA.

[78] GOSLING, JAMES & KEN ARNOLD (1996). *Java Programming Language*. Addison-Wesley, Reading, MA.

[79] GREGORY, STEVE (1987). *Parallel Logic Programming in Parlog. The Language and its Implementation*. International Series in Logic Programming. Addison-Wesley, Reading, MA.

[80] HALL, CORDELIA; KEVIN HAMMOND; SIMON PEYTON JONES, & PHILIP WADLER (1994). Type classes in haskell. In *Proceedings of the 5th European*

Symposium on Programming, edited by D. Sannella, vol. 788 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

[81] HALSTEAD, ROBERT (Oct. 1985). Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538. ACM Press, New York.

[82] HEINTZE, NEVIN (1992). Practical aspects of set based analysis. In *Proceedings of the International Conference and Symposium on Logic Programming*, edited by K. R. Apt, pp. 765–779, Washington, DC. The MIT Press, Cambridge, MA.

[83] HEINTZE, NEVIN (Oct. 1992). *Set Based Program Analysis*. Ph.D. thesis, Carnegie Mellon University.

[84] HEINTZE, NEVIN (1994). Set based analysis of ML programs. In *ACM Conference on LISP and Functional Programming*, pp. 306–317. ACM Press, New York.

[85] HEINTZE, NEVIN & JOXAN JAFFAR (1990). A decision procedure for a class of set constraints (extended abstract). In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press.

[86] HEINTZE, NEVIN & JOXAN JAFFAR (Jan. 1990). A finite presentation theorem for approximating logic programs. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pp. 197–209. ACM Press, New York.

[87] HEINTZE, NEVIN & JOXAN JAFFAR (1992). Semantics types for logic programs. In Pfenning [164], pp. 141–156.

[88] HEINTZE, NEVIN & JOXAN JAFFAR (May 1994). Set constraints and set-based analysis. In *Proceedings of the 2nd Workshop on Principles and Practice of Constraint Programming*, vol. 874 of *Lecture Notes in Computer Science*, pp. 281–298, Orcas Island, Washington. Springer-Verlag, Berlin.

[89] HENGLEIN, FRITZ & JAKOB REHOF (1997). The complexity of subtype entailment for simple types. In *Proceedings of the 12th IEEE Symposium on Logic in Computer Science*, pp. 362–372, Warsaw, Poland. IEEE Computer Society Press.

[90] HENGLEIN, FRITZ & JAKOB REHOF (1998). Constraint automata and the complexity of recursive subtype entailment. In *Proceedings of the 25th International Conference on Automata, Languages, and Programming*, edited by K. Larsen, Lecture Notes in Computer Science, Aalborg, Denmark. Springer-Verlag, Berlin. to appear.

[91] HENZ, MARTIN (Nov. 1997). *Objects for Concurrent Constraint Programming*, vol. 426 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Utrecht.

[92] HENZ, MARTIN (May 1997). *Objects in Oz*. Doctoral Dissertation. Universität des Saarlandes, Technische Fakultät, D–66041 Saarbrücken.

[93] HENZ, MARTIN; STEFAN LAUER, & DETLEV ZIMMERMANN (Nov.16–19 1996). COMPOzE — Intention-based music composition through constraint programming. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, pp. 118–121, Toulouse, Frankreich. IEEE Computer Society Press.

[94] HENZ, MARTIN; MARTIN MÜLLER; CHRISTIAN SCHULTE, & JÖRG WÜRTZ (1997). The Oz standard modules. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany.

[95] HENZ, MARTIN & JÖRG WÜRTZ (1996). Constraint-based time tabling — A case study. *Applied Artificial Intelligence*, 10(5):439–453. Hemisphere Publishing Corporation.

[96] HEWITT, CARL; PETER BISHOP, & RICHARD STEIGER (1973). A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 235–245.

[97] HILL, PAT M. & JOHN W. LLOYD (1994). *The Gödel Programming Language*. The MIT Press, Cambridge, MA.

[98] HIRATA, M. (1986). Programming language Doc and its self-description, or x=x considered harmful. In *Proceedings of the 3rd Conference of Japan Society of Software Science and Technology*, pp. 69–72.

[99] HOARE, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677. ACM Press, New York.

[100] HOARE, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J.

[101] HOFMANN, MARTIN & BENJAMIN PIERCE (Oct. 1995). A unifying type-theoretic framework for objects. *The Journal of Functional Programming*, 5(4):593–635. Cambridge University Press, Cambridge, England.

[102] HONDA, KOHEI & MARIO TOKORO (Jul. 1991). An object calculus for asynchronous communication. In *Proceedings of the European Conference on Object-Oriented Programming*, edited by P. America, Geneva, Switzerland.

[103] HOPCROFT, JOHN D. & JEFFREY D. ULLMAN (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA.

[104] JAFFAR, JOXAN & JEAN-LOUIS LASSEZ (1987). Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111–119, München. ACM Press, New York.

[105] JAFFAR, JOXAN & MICHAEL J. MAHER (May-July 1994). Constraint logic programming: A survey. *The Journal of Logic Programming*, 19/20:503–582. Special Issue: Ten Years of Logig Programming. Elsevier Science Publishers B.V. (North Holland).

[106] JAFFAR, JOXAN & PETER STUCKEY (1986). Canonical logic programs. *The Journal of Logic Programming*, 3(3):143–155. Elsevier Science Publishers B.V. (North Holland).

[107] JAFFAR, JOXAN & PETER STUCKEY (1986). Semantics of infinite tree logic programming. *Theoretical Computer Science*, 46(3):141–158. Elsevier Science Publishers B.V. (North Holland).

[108] JANSON, SVERKER (1994). *AKL — A Multiparadigm Programming Language*. Ph.D. thesis, SICS Swedish Institute of Computer Science, SICS Box 1263, S-164 28 Kista, Sweden.

[109] JOHNSON, MARK (1988). *Attribute-Value Logic and the Theory of Grammar*. No. 16 in CSLI Lecture Notes. Center for the Study of Language and Information.

[110] JONES, NEIL D. & STEVEN S. MUCHNIK (1979). Flow analysis and optimization of Lisp-like structures. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pp. 244–256. ACM Press, New York.

[111] KAHN, GILLES (1974). The Semantics of a Simple Language for Parallel Programming. In *Proceedings of the World Computer Congress of the IFIP*, pp. 471–475, Stockholm, Sweden. North-Holland, Amsterdam.

[112] KERNIGHAN, BRIAN W. & DENNIS M. RITCHIE (1988). *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 2nd edn.

[113] KLEINMAN, ALON; YAEL MOSCOWITZ; AMIR PNUELI, & EHUD SHAPIRO (1991). Communication with directed logic variables. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pp. 221–232. ACM Press, New York.

[114] KOWALSKI, ROBERT A. (1974). Predicate logic as a programming language. In *Proceedings of the World Computer Congress of the IFIP*, edited by J. Rosenfeld. North-Holland, Amsterdam, Stockholm, Sweden.

[115] LASSEZ, J. & K. MCALOON (Jun. 1990). A constraint sequent calculus. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pp. 52–61. IEEE Computer Society Press.

[116] LASSEZ, J.-L.; M.J. MAHER, & K. MARRIOT (1988). Unification revisited. In *Foundations of Deductive Databases and Logic Programming*, edited by J. Minker, pp. 1587–625. Morgan Kaufmann Publishers, Los Altos, CA.

[117] LEROY, XAVIER & PIERRE WEIS (1991). Polymorphic type inference and assignment. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pp. 291–302.

[118] LLOYD, JOHN W. (1987). *Foundations of Logic Programming*. Springer-Verlag, Berlin, 2nd edn.

[119] MACQUEEN, DAVID; GORDON PLOTKIN, & RAVI SETHI (1986). An ideal model for recursive polymorphic types. *Information and Control*, 71(1 – 2):95–130. Academic Press, New York, N.Y.

[120] MAHER, MICHAEL J. (1987). Logic semantics for a class of committed-choice programs. In *Proceedings of the International Conference on Logic Programming*, edited by J.-L. Lassez, pp. 858–876. The MIT Press, Cambridge, MA.

[121] MARLOW, SIMON & PHILIP WADLER (1997). A practical subtyping system for Erlang. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, pp. 136–149. ACM Press, New York.

[122] MEYER, GREGOR (Jun. 1996). On the use of types in logic programming. Informatik-Berichte 199, FernUniversität Hagen.

[123] MILNER, ROBIN (1978). A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17(3):348–375. Academic Press, New York, N.Y.

[124] MILNER, ROBIN (1980). *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

[125] MILNER, ROBIN (1989). *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, N.J.

[126] MILNER, ROBIN (1990). Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science*, edited by J. van Leeuwen, vol. B (Formal Models and Semantics), chap. 19, pp. 1201–1242. The MIT Press, Cambridge, MA.

[127] MILNER, ROBIN (1992). Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141. Cambridge University Press, Cambridge, England.

[128] MILNER, ROBIN (1993). The polyadic π-calculus: A tutorial. In *Proceedings of the 1991 Marktoberndorf Summer School on Logic and Algebra of Specification*, edited by F. L. Bauer, W. Brauer, & H. Schwichtenberg, NATO ASI Series. Springer-Verlag, Berlin. also available as Technical Report ECS-LFCS 91-180 from LFCS, Edinburgh, October 1991.

[129] MILNER, ROBIN; JOACHIM PARROW, & DAVID WALKER (Sep. 1992). A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40 and 41–77. Academic Press, New York, N.Y.

[130] MILNER, ROBIN; MADS TOFTE; ROBERT HARPER, & DAVID MACQUEEN (1997). *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA.

[131] MINAMIDE, YASUHIKO (1998). A functional represention of data structures with a hole. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pp. 75–84. ACM Press, New York.

[132] MISHRA, PRATEEK (1984). Towards a theory of types in Prolog. In *Proceedings of the 1st International Logic Programming Symposium*, pp. 289–298, Atlantic City, New Jersey. IEEE Computer Society Press.

[133] MISHRA, PRATEEK & UDAY REDDY (Jan. 1985). Declaration-free type checking. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pp. 7–21. ACM Press, New York.

[134] MITCHELL, JOHN C. (1996). *Foundations for Programming Languages*. The MIT Press, Cambridge, MA.

[135] MITCHELL, JOHN C. & GORDON D. PLOTKIN (Jan. 1985). Abstract types have existential type. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pp. 37–51. ACM Press, New York.

[136] MÜLLER, MARTIN (1996). Polymorphic types for concurrent constraints. Tech. rep., Programming Systems Lab, Universität des Saarlandes. `http://www.ps.uni-sb.de/~mmueller/papers/ptcc.ps.gz`.

[137] MÜLLER, MARTIN (1997). Ordering constraints over feature trees with ordered sorts. In *Computational Logic and Natural Language Understanding*, edited by P. Lopez, S. Manandhar, & W. Nutt, Lecture Notes in Artificial Intelligence, to appear. Springer-Verlag, Berlin.

[138] MÜLLER, MARTIN (1998). Type-safe programming with constraints. Tech. rep., Programming Systems Lab, Universität des Saarlandes. Draft. Available from the author: `mmueller@ps.uni-sb.de`.

[139] MÜLLER, MARTIN; TOBIAS MÜLLER, & PETER VAN ROY (Dec. 1995). Multi-paradigm programming in Oz. In *Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor of Prolog*, edited by D. Smith, O. Ridoux, & P. Van Roy, Portland, Oregon. A Workshop in Association with the International Logic Programming Symposium.

[140] MÜLLER, MARTIN & JOACHIM NIEHREN (1997). Entailment for set constraints is not feasible. Tech. rep., Programming Systems Lab, Universität des Saarlandes. `http://www.ps.uni-sb.de/Papers/abstracts/inesInfeas.html`.

[141] MÜLLER, MARTIN & JOACHIM NIEHREN (1998). Ordering constraints over feature trees expressed in second-order monadic logic. In *International Conference on Rewriting Techniques and Applications*, edited by T. Nipkow, no. 1379 in Lecture Notes in Computer Science, pp. 196–210, Tsukuba, Japan. Springer-Verlag, Berlin.

[142] MÜLLER, MARTIN; JOACHIM NIEHREN, & ANDREAS PODELSKI (Apr. 1997). Inclusion constraints over non-empty sets of trees. In *Proceedings of the Theory and Practice of Software Development*, edited by M. Bidoit & M. Dauchet, vol. 1214 of *Lecture Notes in Computer Science*, pp. 345–356, Lille, France. Springer-Verlag, Berlin.

[143] MÜLLER, MARTIN; JOACHIM NIEHREN, & ANDREAS PODELSKI (1997). Ordering constraints over feature trees. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, edited by G. Smolka, vol. 1330 of *Lecture Notes in Computer Science*, pp. 297–311, Schloß Hagenberg, Linz, Austria. Springer-Verlag, Berlin.

[144] MÜLLER, MARTIN; JOACHIM NIEHREN, & GERT SMOLKA (1998). Typed concurrent programming with logic variables. Tech. rep., Programming Systems Lab, Universität des Saarlandes. `http://www.ps.uni-sb.de/Papers/abstracts/plain-report-97.html`.

[145] MÜLLER, MARTIN; JOACHIM NIEHREN, & RALF TREINEN (1998). The first-order theory of ordering constraints over feature trees. In *Proceedings of the*

13^th IEEE Symposium on Logic in Computer Science. IEEE Computer Society Press.

[146] MÜLLER, TOBIAS & MARTIN MÜLLER (Sep. 1997). Finite set constraints in Oz. In *12. Deutscher Workshop zur Logischen Programmierung*, edited by F. Bry, B. Freitag, & D. Seipel. Available as Technical Report PMS-FB-1997-10 of the Ludwig Maximilians Universität München.

[147] MYCROFT, ALAN & RICHARD A. O'KEEFE (1984). A polymorphic type system for Prolog. *Artificial Intelligence*, pp. 295–307. Elsevier Science Publishers B.V. (North Holland).

[148] NAISH, LEE (1985). Automating control for logic programs. *The Journal of Logic Programming*, 2(3):167–184. Elsevier Science Publishers B.V. (North Holland).

[149] NAISH, LEE (1992). Types and intended meaning. In Pfenning [164], pp. 189–216.

[150] NESTMANN, UWE & BENJAMIN PIERCE (1996). Decoding choice encodings. In *Proceedings of the 7^th International Conference on Concurrency Theory*, edited by U. Montanari & V. Sassone, vol. 1119 of *Lecture Notes in Computer Science*, pp. 179–194. Springer-Verlag, Berlin.

[151] NIEHREN, JOACHIM (Dec. 1994). *Funktionale Berechnung in einem uniform nebenläufigen Kalkül mit logischen Variablen*. Ph.D. thesis, Universität des Saarlandes, Fachbereich Informatik, Stuhlsatzenhausweg, 66123 Saarbrücken, Germany.

[152] NIEHREN, JOACHIM (Jan. 1996). Functional computation as concurrent computation. In *Proceedings of the 23^rd ACM Symposium on Principles of Programming Languages*, pp. 333–343, St. Petersburg Beach, Florida. ACM Press, New York.

[153] NIEHREN, JOACHIM (1997). Uniform confluence in concurrent computation. Submitted. `http://www.ps.uni-sb.de/Papers/abstracts/Uniform-97.html`.

[154] NIEHREN, JOACHIM & MARTIN MÜLLER (Dec. 1995). Constraints for free in concurrent computation. In *Algorithms, Concurrency and Knowledge: Proceedings of the 1^st Asian Computing Science Conference*, edited by K. Kanchanasut & J.-J. Lévy, no. 1023 in Lecture Notes in Computer Science, pp. 171–186, Pathumthani, Thailand. Springer-Verlag, Berlin.

[155] NIEHREN, JOACHIM & GERT SMOLKA (Sep. 1994). A confluent relational calculus for higher-order programming with constraints. In *Proceedings of the 1st International Conference on Constraints in Computational Logics*, edited by J.-P. Jouannaud, vol. 845 of *Lecture Notes in Computer Science*, pp. 89–104, München. Springer-Verlag, Berlin.

[156] NIERSTRASZ, OSCAR (1995). Regular types for active objects. In *Object-oriented Software Construction*, edited by O. Nierstrasz & D. Tsichritzis, pp. 99–121. Prentice-Hall, Englewood Cliffs, N.J.

[157] PACHOLSKI, LESZEK & ANDREAS PODELSKI (1997). Set constraints: A pearl in research on constraints. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, edited by G. Smolka, vol. 1330 of *Lecture Notes in Computer Science*, pp. 549–561. Springer-Verlag, Berlin. Tutorial Abstract.

[158] PALAMIDESSI, CATUSCIA (Jan. 1997). Comparing the expressive power of the synchronous and asynchronous pi-calculus. In *Proceedings of the 23th ACM Symposium on Principles of Programming Languages*, pp. 256–265. ACM Press, New York.

[159] PALMGREN, ERIK (1994). Denotational semantics of constraint logic programming - a non-standard approach. In *Constraint Programming*, edited by B. H. Mayoh, E. Tyugu, & J. Penjam, vol. 131 of *NATO ASI Series F: Computer and System Sciences*, pp. 261–288. Springer-Verlag, Berlin.

[160] PARROW, JOACHIM & BJÖRN VICTOR (Dec. 1997). The update calculus. In *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, edited by M. Johnson, vol. 1349 of *Lecture Notes in Computer Science*, pp. 409–423, Sydney, Australia. Springer-Verlag, Berlin.

[161] PARROW, JOACHIM & BJÖRN VICTOR (1998). The fusion calculus: Expressiveness and summetry in mobile processes. In *Proceedings of the 13th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press.

[162] PETERSON, JOHN & KEVIN HAMMOND (1997). Report on the programming language Haskell: A non-strict, purely functional language (Version 1.4). Tech. rep., Yale University. `http://haskell.systemsz.cs.yale.edu/onlinereport/`.

[163] PETRI, CARL ADAM (1962). *Kommunikation mit Automaten*. Dissertation. Institut für Instrumentelle Mathematik, Universität Bonn.

[164] PFENNING, FRANK (ed.) (1992). *Types in Logic Programming*. Logic Programming Series. The MIT Press, Cambridge, MA.

[165] PIERCE, BENJAMIN & DAVIDE SANGIORGI (Jun. 1993). Typing and subtyping for mobile processes. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pp. 376–385. IEEE Computer Society Press.

[166] PIERCE, BENJAMIN C. & DAVID N. TURNER (Apr. 1994). Simple type-theoretic foundations for object-oriented programming. *The Journal of Functional Programming*, 4(2):207–247. Cambridge University Press, Cambridge, England.

[167] PIERCE, BENJAMIN C. & DAVID N. TURNER (1997). Pict: A programming language based on the pi-calculus. Compiler, documentation, demonstration Programs, and standard Libraries; available electronically. Version 4.0.

[168] PIERCE, BENJAMIN C. & DAVID N. TURNER (1998). Local type inference. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pp. 252–265. ACM Press, New York.

[169] PIERCE, BENJAMIN C. & DAVID N. TURNER (1998). Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, edited by G. Plotkin, C. Stirling, & M. Tofte. The MIT Press, Cambridge, MA. to appear.

[170] PINGALI, KESHAV K. (Oct. 1987). Lazy evaluation and the logic variable. Tech. rep., Cornell University. Proceedings of the the Institute on Declarative Programming. Austin, Texas.

[171] PODELSKI, ANDREAS; WITOLD CHARATONIK, & MARTIN MÜLLER (1998). Set-based error diagnosis of concurrent constraint programs. Tech. rep., Programming Systems Lab, Universität des Saarlandes. `http://www.ps.uni-sb.de/Papers/abstracts/Diagnosis-97.html`.

[172] POIRRIEZ, VINCENT (1994). MLOG: A strongly typed confluent functional language with logical variables. *Theoretical Computer Science*, 122:201–223. Elsevier Science Publishers B.V. (North Holland).

[173] POTTIER, FRANÇOIS (May 1996). Simplifying subtyping constraints. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pp. 122–133. ACM Press, New York.

[174] PROGRAMMING SYSTEMS LAB (1997). The Oz Programming System. Universität des Saarlandes: `http://www.ps.uni-sb.de/oz/`.

[175] RÉMY, DIDIER & JÉRÔME VOUILLON (1997). Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pp. 40–53. ACM Press, New York.

[176] REYNOLDS, JOHN (1969). Automatic computation of data set definitions. *Information Processing*, 68.

[177] REYNOLDS, JOHN (1974). Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*, vol. 19 of *Lecture Notes in Computer Science*, pp. 408–425. Springer-Verlag, Berlin.

[178] REYNOLDS, J.C. (Sep. 1983). Types, abstraction, and parametric polymorphism. In *IFIP Congress*, Paris.

[179] ROUNDS, WILLIAM C. (1997). Feature logics. In *Handbook of Logic and Language*, edited by J. v. Benthem & A. ter Meulen, pp. 475–533. Elsevier Science Publishers B.V. (North Holland).

[180] SARASWAT, VIJAY A. (1993). *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA.

[181] SARASWAT, VIJAY A.; KENNETH M. KAHN, & JACOB LEVY (Oct. 1990). Janus: A step towards distributed constraint programming. In *Proceedings of the North American Conference on Logic Programming*, edited by S. K. Debray & M. V. Hermenegildo, pp. 431–446, Austin, Texas. The MIT Press, Cambridge, MA.

[182] SCHULTE, CHRISTIAN (Oct. 1997). Programming constraint inference engines. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, edited by G. Smolka, vol. 1330 of *Lecture Notes in Computer Science*, pp. 519–533, Schloß Hagenberg, Linz, Austria. Springer-Verlag.

[183] SCHULTE, CHRISTIAN; GERT SMOLKA, & JÖRG WÜRTZ (May 1994). Encapsulated search and constraint programming in Oz. In *Proceedings of the 2nd Workshop on Principles and Practice of Constraint Programming*, edited by A. H. Borning, vol. 874 of *Lecture Notes in Computer Science*, pp. 134–150, Orcas Island, Washington, USA. Springer-Verlag, Berlin.

[184] SCHULTE, CHRISTIAN; GERT SMOLKA, & JÖRG WÜRTZ (1997). Constraint programming in Oz: A tutorial. DFKI Oz documentation series, DFKI, Stuhlsatzenhausweg 3, 66123 Saarbrücken.

[185] SEIDL, HELMUT (1994). Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60. Elsevier Science Publishers B.V. (North Holland).

[186] SEYNHAEVE, FRANCK; MARC TOMMASI, & RALF TREINEN (Apr. 1997). Grid structures and undecidable constraint theories. In *Theory and Practice of*

Software Development, edited by M. Bidoit & M. Dauchet, no. 1214 in Lecture Notes in Computer Science, pp. 357–368, Lille, France. Springer-Verlag, Berlin.

[187] SHAPIRO, EHUD (1983). A subset of Concurrent Prolog. Tech. Rep. CS83-06, Weizman Institute of Science.

[188] SHAPIRO, EHUD (ed.) (1987). *Concurrent Prolog. Collected Papers. Volumes 1 and 2*. The MIT Press, Cambridge, MA.

[189] SHAPIRO, EHUD (Sep. 1989). The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510. ACM Press, New York.

[190] SHIEBER, STEWARD (1986). *An Introduction to Unification-based Approaches to Grammar*. CSLI Lecture Notes No. 4. Center for the Study of Language and Information.

[191] SHIVERS, OLIN (Jun. 1988). Control flow analysis in scheme. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 164–174. ACM Press, New York.

[192] SMOLKA, GERT (1988). TEL (Version 0.9), report and user manual. SEKI-Report SR-87-17, FB Informatik, Universität Kaiserslautern.

[193] SMOLKA, GERT (1992). Feature constraint logics for unification grammars. *The Journal of Logic Programming*, 12(1 – 2):51–87. Elsevier Science Publishers B.V. (North Holland).

[194] SMOLKA, GERT (Sep. 1994). A foundation for concurrent constraint programming. In *Proceedings of the 1st International Conference on Constraints in Computational Logics*, edited by J.-P. Jouannaud, vol. 845 of *Lecture Notes in Computer Science*, pp. 50–72, München.

[195] SMOLKA, GERT (1995). The Oz Programming Model. In *Computer Science Today*, edited by J. van Leeuwen, vol. 1000 of *Lecture Notes in Computer Science*, pp. 324–343. Springer-Verlag, Berlin.

[196] SMOLKA, GERT (1998). Concurrent constraint programming based on functional programming. In *Proceedings of the 8th European Symposium on Programming*, edited by C. Hankin, no. 1381 in Lecture Notes in Computer Science, pp. 1–11, Lisbon, Portugal. Springer-Verlag, Berlin. Slides available at `http://www.ps.uni-sb.de/~smolka/drafts/etaps98.ps`.

[197] SMOLKA, GERT & RALF TREINEN (Apr. 1994). Records for logic programming. *The Journal of Logic Programming*, 18(3):229–258. Elsevier Science Publishers B.V. (North Holland).

[198] SOMOGYI, ZOLTAN; FERGUS HENDERSON, & THOMAS CONWAY (1995). Mercury, an efficient purely declarative logic programming language. In *Proceedings of the 18th Australasian Computer Science Conference*, edited by R. Kotagiri, pp. 499–512. Australian Computer Science Communications: Glenelg, South Australia.

[199] STEFANSSON, KJARTAN (1994). Systems of set constraints with negative constraints are NEXPTIME-complete. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pp. 137–141. IEEE Computer Society Press.

[200] STERLING, LEON & EHUD SHAPIRO (1986). *The Art of Prolog*. The MIT Press, Cambridge, MA, Cambridge, MA.

[201] TALBOT, JEAN-MARC; SOPHIE TISON, & PHILLIPE DEVIENNE (1997). Set-based analysis for logic programming and tree automata. In *Proceedings of the 4th International Static Analysis Symposium*, edited by P. Van Hentenryck, vol. 1302 of *Lecture Notes in Computer Science*, pp. 127–140, Paris, France. Springer-Verlag, Berlin.

[202] TOFTE, MADS (1990). Type inference for polymorphic references. *Information and Computation*, 89(1):1–34. Academic Press, New York, N.Y.

[203] TRIFONOV, VALERY & SCOTT SMITH (1996). Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium*, edited by R. Cousot & D. A. Schmidt, vol. 1145 of *Lecture Notes in Computer Science*, pp. 349–365, Aachen. Springer-Verlag, Berlin.

[204] UEDA, KAZUNORI (Jun. 1995). Strong moding in concurrent logic/constraint programming. In *Proceedings of the 12th International Conference on Logic Programming*, edited by L. Sterling, Kanagawa, Japan. The MIT Press, Cambridge, MA.

[205] UEDA, KAZUNORI & MASAO MORITA (1994). Moded Flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43. Springer-Verlag, Berlin.

[206] VAN HENTENRYCK, PASCAL (1989). *Constraint Satisfaction in Logic Programming*. Programming Logic Series. The MIT Press, Cambridge, MA.

[207] VAN HENTENRYCK, PASCAL; VIJAY SARASWAT, & YVES DEVILLE (1995). Design, implementation and evaluation of the constraint language cc(FD). In *Constraints: Basics and Trends*, edited by A. Podelski, vol. 910 of *Lecture Notes in Computer Science*, pp. 293–316. Springer-Verlag, Berlin.

[208] VAN HENTENRYCK, PASCAL & VIJAY A. SARASWAT (Dec. 1996). Strategic directions in constraint programming. *ACM Computing Surveys*, 28(4):701–726. ACM 50th Anniversary Issue. Strategic Directions in Computing Research. ACM Press, New York.

[209] VAN ROY, PETER; SEIF HARIDI; PER BRAND; GERT SMOLKA; MICHAEL MEHL, & RALF SCHEIDHAUER (Sep. 1997). Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851. ACM Press, New York.

[210] VAN ROY, PETER; MICHAEL MEHL, & RALF SCHEIDHAUER (Sep. 1996). Integrating efficient records into concurrent constraint programming. In *Proceedings of the 8th International Symposium on Programming Language Implementation and Logic Programming*, edited by H. Kuchen & S. D. Swierstra, pp. 438–453, Aachen. Springer-Verlag, Berlin.

[211] VICTOR, BJÖRN & JOACHIM PARROW (1996). Constraints as processes. In *Proceedings of the 7th International Conference on Concurrency Theory*, edited by U. Montanari & V. Sassone, vol. 1119 of *Lecture Notes in Computer Science*, pp. 389–405. Springer-Verlag, Berlin.

[212] WADLER, PHILIP (Mar. 1997). A HOT opportunity. *The Journal of Functional Programming*, 7(2):127–128. Editorial. Cambridge University Press, Cambridge, England.

[213] WAND, MITCHELL (1987). A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122. IOS Press.

[214] WELLS, JOE B. (1994). Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pp. 176–185. IEEE Computer Society Press.

[215] WINSKEL, GLYNN (1993). *The Formal Semantics of Programming Languages*. Foundations of Computing. The MIT Press, Cambridge, MA.

[216] WRIGHT, ANDREW K. (Aug. 1994). *Practical Soft Typing*. Ph.D. thesis, Rice University, Houston, Texas.

[217] WRIGHT, ANDREW K. (Dec. 1995). Simple imperative polymorphism. *Journal on Lisp and Symbolic Computation*, 8(4):343–355. Kluwer Academic Publishers, Utrecht.

[218] WRIGHT, ANDREW K. & ROBERT CARTWRIGHT (Jun. 1994). A practical soft type system for Scheme. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pp. 250–262. ACM Press, New York.

[219] WÜRTZ, JÖRG (1998). *Lösen kombinatorischer Probleme mit Constraintprogrammierung in Oz.* Doctoral Dissertation. Universität des Saarlandes, Technische Fakultät, D–66041 Saarbrücken. Submitted.

[220] YARDENI, EYAL & EHUD SHAPIRO (1991). A type system for logic programs. *The Journal of Logic Programming*, 10:125–153. Elsevier Science Publishers B.V. (North Holland).

List of Figures

List of Theorems

Index

D

E

F