Basic Research in Computer Science

The CLP(OIH) Language

Ole Ildsgaard Hougaard

BRICS Dissertation Series **DS-98-1**

ISSN 1396-7002 **February 1998**

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/98/1/`

The CLP(OIH) Language

Ole Ildsgaard Hougaard

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

The CLP(OIH) Language

Dissertation
presented to the Faculty of Science
of the University of Aarhus
in partial fulfillment of the requirements for the
Ph.D. degree

by
Ole Ildsgaard Hougaard
February 1998

Thesis

Many type inference problems are different instances of the same constraint satisfaction problem. That is, there is a class of constraints so that these type inference problems can be reduced to the problem of finding a solution to a set of constraints. Furthermore, there is an efficient constraint solving algorithm which can find this solution in polynomial time.

We have shown this by defining the appropriate constraint domain, devising an efficient constraint satisfaction algorithm, and designing a constraint logic programming language over the constraint domain. We have implemented an interpreter for the language and have thus produced a tool which is well-suited for type inference problems of different kinds.

Among the problems that can be reduced to our constraint domain are the following:

- The simply typed λ-calculus.
- The λ-calculus with subtyping.
- Arbadi and Cardelli's Object calculus.
- Effect systems for control flow analysis.
- Turbo Pascal.

With the added power of the constraint logic programming language, certain type systems with no known efficient algorithm can also be implemented — e.g. the object calculus with selftype.

The programming language thus provides a very easy way of implementing a vast array of different type inference problems.

Acknowledgements

I thank my advisor Michael I. Schwartzbach for his comments and ideas which helped me in writing this thesis.

I am grateful to Hosein Askari for his cooperation in our work on Turbo Pascal when doing our Master's thesis. Without that work this dissertation would never have been.

I also thank Gudmund Frandsen and Peter Bro Miltersen for useful suggestions for some of the algorithmical problems in this work.

This work has been supported financially by a scholarship from the Faculty of Science at Aarhus University and in part by the BRICS PhD School. I want to acknowledge the support of the people at the Department of Computer Science at the University of Aarhus.

I have made some of my work at the Department of Computer Science, University of Washington, Seattle. I extend my thanks to Alan Borning and the people at UW for their hospitality.

Contents

Chapter 1

Introduction

The first part of this chapter (until Section 1.2.1) is adapted from [32].

The subject of type inference has been widely studied. For almost any type system ever suggested there have been speculations on the feasibility of inferring the types for a program. For many of these type systems, type inference turn out to be infeasible or even impossible, but some allow polynomial-time type inference algorithms.

Many similarities exist between these algorithms and it seems reasonable to ask the question. What do these algorithms have in common? This dissertation is dedicated to that particular question.

The answer to the question will be that there is a general constraint satisfaction algorithm for a class of constraints which covers a number of these type inference problems. We use this constraint satisfaction algorithm to implement an interpreter for a constraint logic programming language. This language will then be used to implement these type inference algorithms.

The remainder of this chapter is dedicated to introducing the notion of type inference and some mathematical accessories. The remainder of the dissertation is organised as follows.

- Chapters 2 and 3 deals with constraints in general and constraint logic programming languages in particular. We give the basic definitions and some examples of their use.
- Chapter 4 motivates and defines the class of constraints which is used here for type inference — the constraint domain OIH.
- Chapters 5 and 6 deals with algorithmical problems that arise when implementing the interpreter. Chapter 5 deals with a general problem for

interpreting constraint logic programming languages — the problem of *backtracking*. Chapter 6 deals the constraint satisfaction algorithm for our class of constraints.

- Chapter 7 deals with the special features of the CLP(OIH) language. It includes a section on running the interpreter.
- Chapter 7 shows the implementation of several type inference algorithms in CLP(OIH).

1.1 Type Inference for Turbo Pascal

This section is adapted from [32].

1.1.1 Introduction

What is the task of type inference? This is a well-defined problem for any language that supports type annotations and type checking. Normally, the programmer provides type annotations for all variables and the type checker then proceeds to verify that all the type constraints are respected. Type inference is the more difficult task of accepting a program in which the type annotations are only partial or even absent and then deciding if there exists a choice of type annotations with which the program would be accepted by the normal type checker. Generally, the annotated program should be presented as well.

Note that with type inference we do not fall back to *untyped* programs, which of course are subject to all sorts of damaging type errors. Rather, our programs are *implicitly* typed, as valid types must certainly exist even if they are not supplied by the programmer. The ambition behind type inference is to obtain all the safety benefits of typed programs while avoiding the problems with verbose and cumbersome annotations.

It can certainly be argued that there are further benefits to having programs be *explicitly* typed. After all, when the programmer states his intentions up front, then certain logical errors may be caught by the element of redundancy that separates type checking from type inference.

This aspect is clearly realized by proponents of type inference who offer several arguments in reply. First of all, comparing the inferred types to those that were intended provides a similar degree of redundancy. Secondly, type inference may discover that parts of your program are more general than you had originally realized, thus widening its applicability. And finally, if parameter types are only given implicitly, then a procedure may be given more than one type in different

contexts—thus allowing polymorphism. All these benefits are major selling points for functional languages, and their potential applicability to imperative languages should be given serious consideration.

A language such as Pascal already performs a modicum of type inference. Only the types of variables must be given explicitly, and from this information the types of all expressions are inferred. For example, if x is declared to be of type *Real*, then the type of the expression x+1 is inferred as follows: the value of x is of type *Real*; there is a version of + with type $Real \times Integer \to Real$; the constant 1 has type *Integer*; thus we can conclude that x+1 has type *Real*. This sort of inference is perhaps too trivial to gain much notice, but it is there and forms a basis for using implicit types in larger parts of programs.

In the following we show how to generalise the techniques for type inference from unification to constraint solving.

1.1.2 Techniques for Type Inference

In functional languages as ML, type inference may be done by recursively going through the parse tree and assigning a type to each node. The type of a parse tree node can be derived from the types of the subexpressions. For example when we want to find the type of the expression $(e_1\, e_2)$, we first find the types of e_1 and e_2; let us call these τ_1 and τ_2 respectively. We know that the type of e_1 must be a function type, taking something of the type of e_2 as its argument. We can write this as the equation $\tau_1 = \tau_2 \to \alpha$, where α is a *type variable* corresponding to the return type of e_1 that can be instantiated with any type. In order to solve the equation we apply *unification* to the two sides of the equation. Unification finds a most general instantiation of type variables, so that the two types become equal. The type of $(e_1 e_2)$ is simply the instantiation that the unification algorithm finds for α. This technique was used by Milner for type inference of ML in [47].

The above technique allowed type variables to stand for any type. By using type variables in this manner we can represent the set of all possible types for a parse tree node (see [15]). Thus the success of this approach relies on said representation and the fact that we could compute the representation of the solutions to the *constraint* $\tau_1 = \tau_2 \to \alpha$.

In the general case we cannot expect to find a proper representation and compute the solutions to the constraints within this representation. A more general technique is that of generating and solving a set of constraints for the specific program. In this case we will not try to derive the type from those of the subexpressions. Instead we generate type variables representing the (yet unknown) types of all parse tree nodes and further generate the appropriate constraints relating these type variables. In the case of the expression $(e_1\, e_2)$ we generate

the constraint $[\![e_1]\!] = [\![e_2]\!] \to [\![(e_1\, e_2)]\!]$, where we use $[\![e]\!]$ to stand for the type variable representing the type of expression e. Now we have reduced the problem of type inference to that of finding a solution to a set of constraints. In the case of ML we can again solve the constraints by a single application of the unification algorithm. Wand [74] has used this technique for type inference of the simply typed λ-calculus (ML without polymorphic let).

In Turbo Pascal we use this constraint technique. In ML we could limit ourselves to generating constraints of the form $[\![e]\!] = [\![e']\!] \to [\![e'']\!]$, but Turbo Pascal has much more complex typing rules and we need a substantially richer class of constraints.

Consider the simple assignment, $x := e$. The typing rules of Turbo Pascal demands that the type of e is assignment compatible to the type of x, that is, $[\![x]\!] := [\![e]\!]$ is the generated constraint, where we use $:=$ to denote the assignment compatibility relation. Similarly, we generate constraints of the form $[\![e]\!]$ Tc $[\![e']\!]$ when the types of e and e' should be type compatible; $\mathrm{Op}([\![e]\!], [\![e']\!], [\![e'']\!])$ when the type of e'' should be the result type of a binary operation between e and e'; and $[\![e]\!]$ Io $[\![e']\!]$ when e' should be writable to a file which has the type of e.

For example, we generate the constraint $\mathrm{Op}([\![e]\!], [\![e']\!], [\![e\text{+}e']\!])$ for the expression $e\text{+}e'$ and the constraint $[\![f]\!]$ Io $[\![e]\!]$ for the statement $\mathsf{write}(f, e)$.

The expression $e\text{-}e'$ does not apply to strings as opposed to $e\text{+}e'$. Hence the constraint $\mathrm{Op}([\![e]\!], [\![e']\!], [\![e\text{-}e']\!])$ is too liberal. We restrict it by imposing further constraints on the types of e, e' and $e\text{-}e'$, namely $[\![e]\!], [\![e']\!], [\![e\text{-}e']\!] \in \mathcal{M}_{-}$, where $\mathcal{M}_{-}$ is the set of types on which '-' can operate.

As a further example of the use of constraints of the form $[\![e]\!] \in M$ we can regard a for-statement. Among the constraints generated for the statement:

$$\mathsf{for}\ x := e\ \mathsf{to}\ e'\ \mathsf{do}\ S$$

we have $[\![e]\!], [\![e']\!] \in \mathcal{O}$, where $\mathcal{O}$ is the set of *ordinal types*.

In connection with structured types we get the constraints $\mathrm{Rec}_\alpha([\![e]\!], [\![e']\!])$ requiring that $[\![e]\!]$ is a record with a field α that has type $[\![e']\!]$, and $[\![e]\!] = T([\![e_1]\!], \ldots, [\![e_n]\!])$, where T is a type constructor. For example, we get the constraint $\mathrm{Rec}_\alpha([\![x]\!], [\![x.\alpha]\!])$ for the expression $x.\alpha$ and $[\![x]\!] = {}^\wedge[\![x^\wedge]\!]$ for the expression $x^\wedge$.

Finally, we have the simple constraint $[\![e]\!] = [\![e']\!]$ in connection with variable parameters, where the actual type must equal the formal type, and expressions like -e, where we have the constraint $[\![\text{-}e]\!] = [\![e]\!]$.

All in all, we have the following kinds of constraints:

- $[\![e]\!] \in \mathcal{M}$, where $\mathcal{M}$ is from a fixed finite set of sets of types.
- $[\![e]\!] = T([\![e_1]\!], \ldots, [\![e_n]\!])$, where T is a type constructor.

- $\mathrm{Rec}_\alpha([\![e]\!], [\![e']\!])$
- $[\![e]\!] = [\![e']\!]$
- $[\![e]\!]$ Tc $[\![e']\!]$, $[\![e]\!] := [\![e']\!]$, $\mathrm{Op}([\![e]\!], [\![e']\!], [\![e'']\!])$, and $[\![e]\!]$ Io $[\![e']\!]$

As an example, consider the following function for computing the factorial of a number.

```
Function fac(n);
begin
    if n=0 then
      fac := 1
    else
      fac := n*fac(n-1)
end
```

We generate the following set of constraints for the function:

$[\![1]\!] \in \mathcal{I}$
$[\![0]\!] \in \mathcal{I}$
$[\![n{=}0]\!] = \mathit{Boolean}$
$[\![n]\!]$ Tc $[\![0]\!]$
$[\![n]\!], [\![0]\!] \in \mathcal{M}_=$
$[\![\mathit{fac}]\!] := [\![1]\!]$
$[\![\mathit{fac}]\!] := [\![n{*}\mathit{fac}(n{-}1)]\!]$
$\mathrm{Op}([\![n]\!], [\![\mathit{fac}(n{-}1)]\!], [\![n{*}\mathit{fac}(n{-}1)]\!])$
$[\![n]\!], [\![\mathit{fac}(n{-}1)]\!], [\![n{*}\mathit{fac}(n{-}1)]\!] \in \mathcal{M}_*$
$[\![\mathit{fac}(n{-}1)]\!] = [\![\mathit{fac}]\!]$
$[\![n]\!] := [\![\mathit{fac}(n{-}1)]\!]$
$\mathrm{Op}([\![n]\!], [\![1]\!], [\![n{-}1]\!])$
$[\![n]\!], [\![1]\!], [\![n{-}1]\!] \in \mathcal{M}_-$

where $\mathcal{I} \subsetneq \mathcal{O}$ is the set of integer types. The following is one of several solutions to the set of constraints.

$$\begin{aligned}
[\![n]\!] &= \mathit{Real} \\
[\![0]\!] &= \mathit{Integer} \\
[\![n{=}0]\!] &= \mathit{Boolean} \\
[\![\mathit{fac}]\!] &= \mathit{Real} \\
[\![1]\!] &= \mathit{Integer} \\
[\![n{*}\mathit{fac}(n{-}1)]\!] &= \mathit{Real} \\
[\![\mathit{fac}(n{-}1)]\!] &= \mathit{Real} \\
[\![n{-}1]\!] &= \mathit{Real}
\end{aligned}$$

Note, that the result may not be what we expected. In general, a type inference algorithm should infer as general types as possible.

Since all the type rules of Turbo Pascal can be expressed in this manner, we have reduced the problem of type inference to that of finding a solution to a set of certain kinds of constraints. We now have to exhibit an algorithm that solves such a set of constraints. Such an algorithm is presented in [32]. In Chapter 6 we shall look at a more general algorithm which is based on an improved version of this algorithm.

1.2 Notation and Preliminaries

In this section we introduce the notation and basic concepts which are used throughout the thesis. Section 1.2.1 deals with general mathematical terms and Section 1.2.2 deals with two important concepts for this work — terms and term automata.

1.2.1 Sets, Functions, and Relations

We use the standard notation of set theory. That is, we write $x \in S$ whenever x belongs to the set S, $\emptyset$ for the empty set, $S' \subseteq S$ when S' is a subset of S, and $S' \subsetneq S$ when S' is a true subset of S.

Some of the sets we use are the following.

- $\mathbf{R}$ — the set of real numbers.
- $\mathbf{Q}$ — the set of rational numbers.
- $\mathbf{Z}$ — the set of integer numbers.
- $\mathbf{N}$ or ω — the set of non-negative integers. We use $\mathbf{N}$ when we use the non-negative integers for arithmetic and ω when we use them for enumeration.

We define the subrange $m..n$ as the set

$$m..n = \{i \in \mathbf{Z} | m \leq i \leq n\}$$

The *power set* of a set is the set of subsets of that set.

$$\mathcal{P}(S) = \{S' | S' \subseteq S\}$$

If Σ is a set of symbols we write Σ^* for the set of finite strings of symbols from Σ. Similarly, Σ^+ is the set of non-empty, finite strings over Σ, Σ^ω is the set of infinite strings over Σ, and Σ^∞ is the set of all strings over Σ. We write ϵ for the empty string and $\alpha\beta$ for the concatenation of the strings α and β. If $\beta = \alpha\alpha'$ we say that α is a prefix of β.

We write a function, f, from A to B as $f : A \to B$. The function is *partial* if it is undefined for some of the elements of A. If f is a partial function we write $\mathrm{dom}(f)$ for the set of elements of A for which f is defined. Similarly, we write $\mathrm{codom}(f)$ for the set $\{f(x)|x \in \mathrm{dom}(f)\}$. Note, that any partial function $f : A \to B$ is also a total function $f : \mathrm{dom}(f) \to \mathrm{codom}(f)$. We write $x \mapsto e$ for the function f defined by $f(x) = e$ (e.g. $x \mapsto x^2$ is the function which squares its input). If f is a function, $f[x \leftarrow v]$ is the function defined as

$$f[x \leftarrow v](x') = \begin{cases} v & \text{if } x' = x \\ f(x') & \text{otherwise} \end{cases}$$

A partial ordering is a binary relation which is reflexive, transitive, and anti-symmetric. That is, $\leqslant \subseteq A^2$ is a partial ordering, iff the following hold

- $\forall x \in A : x \leqslant x$ (reflexivity)
- $\forall x, y, z \in A : x \leqslant y \wedge y \leqslant z \Rightarrow x \leqslant z$ (transitivity)
- $\forall x, y \in A : x \leqslant y \wedge y \leqslant x \Rightarrow x = y$ (antisymmetry)

We use $\leqslant$, $\sqsubseteq$, and $\preceq$ for partial orderings.

Let $\leqslant$ be a partial ordering. If $z \leqslant x$ and $z \leqslant y$ and for all other z' such that $z' \leqslant x$ and $z' \leqslant y$ we have $z' \leqslant z$ we say that z is the *greatest lower bound* (or glb) of z and write $z = x \wedge y$. Similarly, z is the *least upper bound* (or lub) of x and y if $x \leqslant z$ and $y \leqslant z$ and $x \leqslant z' \wedge y \leqslant z' \Rightarrow z \leqslant z'$. In this case we write $z = x \vee y$. Dependent on the symbol used for the ordering, we use $\wedge$, $\sqcap$, and $\curlywedge$ to stand for greatest lower bounds and $\vee$, $\sqcup$, and $\curlyvee$ to stand for least upper bounds.

Let $\leqslant$ be an ordering over A. We say that $(A, \leqslant)$ form a *lattice*, iff all pairs of elements from A has a greatest lower bound and a least upper bound. We say that $(A, \leqslant)$ form a lower (upper) *semi-lattice*, iff all pairs of elements has a greatest lower bound (least upper bound).

The definition of greatest lower bounds and least upper bounds extend readily to arbitrary sets of elements. We write $\bigwedge S$ and $\bigvee S$ for the greatest lower bound and least lower bound of $S \subseteq A$, respectively. If all *subsets* of A has a greatest lower bound and a least upper bound we say that $(A, \leqslant)$ form a *complete lattice*. We define complete lower and upper semi-lattices similarly.

1.2.2 Finite and Infinite Terms

Type expressions play an important role in this work. Type expressions are expressions like $(\alpha \to \beta) \to \beta$. In this type expression we have that $\to$ takes two

arguments, where α and β takes no arguments. We say that the *rank* of $\rightarrow$ is 2, and the ranks of α and β are 0.

In general, a *ranked alphabet* is a set $\Sigma = \bigcup_{k=0}^{\infty} \Sigma_k$, where the Σ_k are mutually disjoint. Thus, we have that if $\sigma \in \Sigma$ there is a unique k such that $\sigma \in \Sigma_k$. We call this k the rank of σ and write it as $\mathrm{rank}(\sigma)$.

Let $\Sigma = \bigcup_{k=0}^{\infty} \Sigma_k$ be a ranked alphabet. A finite or infinite Σ-term is of the form $t = \sigma(t_1, t_2, \ldots, t_k)$, where $\sigma \in \Sigma_k$ and $t_1, \ldots, t_k$ are Σ-terms. If t is as above we call σ the *root label* of t and write t/i for t_i where $1 \leq i \leq n$. We shall write $t/i_1 i_2 \ldots i_n$ for $t/i_1/i_2/\ldots/i_n$ and refer to $i_1 i_2 \ldots i_n$ as the *address* of $t/i_1/i_2/\ldots/i_n$ in t. We write ϵ for the empty address and set $t/\epsilon = t$. An address, α, is *valid* in t if t/α denotes a subterm as above. If α is a valid address in t we denote by $t(\alpha)$ the root label of t/α.

We can view the term as a function from the set of valid addresses to Σ as in the following definition due to Courcelle [14].

Definition 1.2.1 *Let Σ be a ranked alphabet. A* Σ-term *is a partial function $t : \omega^* \rightarrow \Sigma$ whose domain is nonempty, prefix-closed, and respects rank in the sense that if $\alpha \in \mathrm{dom}(t)$ then*

$$\{i \in \omega | \alpha i \in \mathrm{dom}(t)\} = 1..\mathrm{rank}(t(\alpha))$$

We shall write T_Σ for the set of Σ-terms.

A finite or infinite Σ-term is *regular* iff it has only finitely many *different* subterms. We shall denote the set of regular Σ-terms as Reg_Σ and the set of finite Σ-terms as Fin_Σ. Formally, $\mathsf{Reg}_\Sigma = \{t \in \mathsf{T}_\Sigma | \ |\{t/\alpha | \alpha \in \mathrm{dom}(t)\}| < \infty\}$ and $\mathsf{Fin}_\Sigma = \{t \in \mathsf{T}_\Sigma | \ |\mathrm{dom}(t)| < \infty\}$.

A *path* in a Σ-term t is a set of addresses $p \subseteq \mathrm{dom}(t)$, such that

- p is prefix-closed.
- For every $\alpha, \alpha' \in p$, either α is a prefix of α' or vise versa. That is, the prefix ordering is *total* on p.

If p is finite it is of the form $[\alpha]$, where $[\alpha] = \{\alpha' | \alpha' \text{ is a prefix of } \alpha\}$. An infinite path is of the form $\{\epsilon, i_1, i_1 i_2, i_1 i_2 i_3, \ldots\}$. We extend the notation above and write the infinite path as $[\alpha^\infty]$ where $\alpha^\infty = i_1 i_2 i_3 \ldots$.

Now, let us give a formal meaning to the expression $\sigma(t_1, \ldots, t_k)$. For each $\sigma \in \Sigma_k$ we can define a function $\sigma^{\mathsf{T}_\Sigma} : {\mathsf{T}_\Sigma}^k \rightarrow \mathsf{T}_\Sigma$ as follows

$$\sigma^{\mathsf{T}_\Sigma}(t_1, \ldots, t_k)(\alpha) = \begin{cases} \sigma & \text{if } \alpha = \epsilon \\ t_i(\alpha') & \text{if } \alpha = i\alpha' \end{cases}$$

We shall omit the T_Σ and write $\sigma(t_1, \ldots, t_{\mathrm{rank}(\sigma)})$ for $\sigma^{\mathsf{T}_\Sigma}(t_1, \ldots, t_{\mathrm{rank}(\sigma)})$, when no ambiguity can arise. It follows from the definition that $\sigma^{\mathsf{T}_\Sigma}$ on regular terms gives a regular term, and that $\sigma^{\mathsf{T}_\Sigma}$ on finite terms gives finite terms. Hence we have also $\sigma^{\mathsf{T}_\Sigma} : \mathsf{Reg}_\Sigma{}^k \to \mathsf{Reg}_\Sigma$ and $\sigma^{\mathsf{T}_\Sigma} : \mathsf{Fin}_\Sigma{}^k \to \mathsf{Fin}_\Sigma$.

There are two important classes of representations of Σ-terms: *sets of term equations* and *term automata*.

Definition 1.2.2 *Let $\Sigma = \bigcup_{k=0}^{\infty} \Sigma_k$ be a ranked alphabet, and let $\mathcal{V}$ be a recursively enumerable set of variables. The* term expressions *over Σ is the smallest set such that*

- *Any variable $v \in \mathcal{V}$ is a term expression.*
- *If $\sigma \in \Sigma_k$ and for all i, t_i is a term expression, then $\sigma(t_1, \ldots, t_k)$ is a term expression.*

We write Expr_Σ for the set of term expressions over Σ.

A *term equation* is an expression of the form $t = t'$ where t and t' are term expression. A *Solution* to $t = t'$ is a function, $\varphi : \mathcal{V} \to \mathsf{T}_\Sigma$, such that $\varphi_\Sigma(t) = \varphi_\Sigma(t')$, where φ_Σ is inductively defined by

- $\varphi_\Sigma(v) = \varphi(v)$
- $\varphi_\Sigma(\sigma(t_1, \ldots, t_n)) = \sigma^{T_\Sigma}(\varphi_\Sigma(t_1), \ldots, \varphi_\Sigma(t_n))$

A solution to a set of term equation is a function, which is a solution to all the equations in the set.

A set of term equations is *cyclic*, iff there are variables $v_1, \ldots, v_n$ and term equations

$$t_1 = t'_1, t_2 = t'_2, \ldots, t_n = t'_n$$

such that $n > 0$, v_1 is contained in t_1 and t'_n, and for each $i \in 1..n-1$, v_i is contained in t'_i and t_{i+1}. Furthermore, for all the equations, $t_i = t'_i$, either the equation itself or $t'_i = t_i$ is in the set of equations. The cyclic sets of equations include

$$\begin{aligned} x &= f(y) \\ g(x) &= y \end{aligned}$$

and

$$f(x) = f(f(x))$$

whereas the following set of equations is acyclic:

$$x = g(y, z)$$

$$\begin{aligned} f(y) &= f(z) \\ z &= c \end{aligned}$$

The equivalence between terms and equations is stated in the following proposition.

Proposition 1.2.1 (Courcelle [13]) *Let Σ be a ranked alphabet, and $\mathcal{V}$ a recursively enumerable set of variables. We have the following:*

1. *$t \in \mathsf{T}_\Sigma$, iff there is a variable $v \in \mathcal{V}$ and a set of term equations, such that for all solutions, φ, we have that $\varphi(v) = t$.*
2. *$t \in \mathsf{Reg}_\Sigma$, iff there is a variable $v \in \mathcal{V}$ and a* finite *set of term equations, such that for all solutions, φ, we have that $\varphi(v) = t$.*
3. *$t \in \mathsf{Fin}_\Sigma$, iff there is a variable $v \in \mathcal{V}$ and a* finite, acyclic *set of term equations, such that for all solutions, φ, we have that $\varphi(v) = t$.*

The other representation of terms is by term automata. Term automata are defined in [38] as the following.

Definition 1.2.3 *Let $\Sigma = \bigcup_{k=0}^{\infty} \Sigma_k$ be a ranked alphabet. A* Σ-term automaton *is a quintuple $M = (Q, \Sigma, q_0, l, \delta)$, where:*

- *Q is a non-empty set of* states.
- *$q_0 \in Q$ is the* initial state.
- *$l : Q \to \Sigma$ is a* labelling function.
- *The* transition function, *$\delta : Q \times \omega \to Q$, is a partial function such that for all $q \in Q$,*
$$\{i \in \omega | (q, i) \in \mathrm{dom}(\delta)\} = 1..\mathrm{rank}(l(q))$$

We shall write A_Σ for the set of Σ-term automata.

We say that an automaton is finite if Q is finite. Given an automaton as above we define δ^* to be the function $\delta^*(q, i_1 i_2 \ldots i_n) = \delta(\ldots \delta(\delta(q_0, i_1), i_2), \ldots, i_n)$ and $\delta^*(q, \epsilon) = q$. A state q is *reachable*, if there is an address, α, such that $\delta^*(q_0, \alpha) = q$. An address, α, is *accepted* by the automaton, iff $\delta^*(q_0, \alpha)$ is well-defined. An automaton is *cyclic* if for some reachable $q \in Q$ and $\alpha \in \omega^* \setminus \{\epsilon\}$, we have that $\delta^*(q, \alpha) = q$.

Lemma 1.2.2 *The set of addresses accepted by a term automaton is non-empty and prefix-closed.*

Proof: Let $M = (Q, \Sigma, q_0, l, \delta)$ be a Σ-term automaton. By definition $\delta^*(q_0, \epsilon)$ is defined, so the set of accepted addresses is non-empty. Now let $\alpha = i_1 \dots i_n$ where $n > 0$ and let α' be a non-trivial prefix of α. That is, $\alpha' = i_1 \dots i_m$ for some m, such that $0 < m < n$. Assume that $\delta^*(q_0, \alpha)$ is defined. By definition,

$$\delta^*(q_0, \alpha) = \delta(\dots\delta(\dots\delta(\delta(q_0, i_1), i_2), \dots, i_m), \dots, i_n)$$

Since this is defined, $\delta(\dots\delta(\delta(q_0, i_1), i_2), \dots, i_m) = \delta^*(q_0, \alpha')$ must be as well. □

Lemma 1.2.2 allows us to make the following definition.

Definition 1.2.4 *Let $M = (Q, \Sigma, q_0, l, \delta)$ be a term automaton. The term $t^M \in T_\Sigma$ is the function with domain $\mathrm{dom}(t^M) = \{\alpha | M \text{ accepts } \alpha\}$ defined such that $t^M(\alpha) = l(\delta^*(q_0, \alpha))$.*

And thus we arrive at the following proposition.

Proposition 1.2.3 *Let Σ be a ranked alphabet. We have the following*

1. *$t \in \mathsf{T}_\Sigma$ iff there is an $M \in \mathsf{A}_\Sigma$, such that $t = t^M$.*
2. *$t \in \mathsf{Reg}_\Sigma$ iff there is a* finite *$M \in \mathsf{A}_\Sigma$, such that $t = t^M$.*
3. *$t \in \mathsf{Fin}_\Sigma$ iff there is a* finite, acyclic *$M \in \mathsf{A}_\Sigma$, such that $t = t^M$.*

Proof:

1. We need to prove that all terms are of the form t^M.

   Given a Σ-term, t, we construct a term automaton, $M = (Q, \Sigma, q_0, l, \delta)$ the following way: $Q = \{t/\alpha | \alpha \in \mathrm{dom}(t)\}$, $q_0 = t$, for each $t' \in Q$, $l(t') = t'(\epsilon)$, and the transition function, δ, is defined as $\delta(t', i) = t'/i$.

   It follows by straightforward induction that $\delta^*(q_0, \alpha) = t/\alpha$, and thus that the set of addresses accepted by M is exactly $\mathrm{dom}(t)$. Furthermore, we have that

   $$t^M(\alpha) = l(\delta^*(q_0, \alpha)) = l(t/\alpha) = t/\alpha(\epsilon) = t(\alpha)$$

   We conclude that $t = t^M$.

2. Proven in [41].

3. We show that if $M = (Q, \Sigma, q_0, l, \delta)$ is finite, then t^M is infinite iff M is cyclic.

   Assume that M is cyclic, that is that $\delta^*(q, \alpha) = q$ for some α and reachable state q. Since q is reachable there is an α' such that $q = \delta^*(q_0, \alpha')$. By straightforward induction we have that M accepts the addresses $\alpha'\alpha^n$ for all $n \geq 0$. Hence the domain of t^M is infinite.

If t^M is infinite it contains an infinite path, $[\alpha^\infty]$. Look at the set of states $S = \{\delta^*(q_0, \alpha) | \alpha \in [\alpha^\infty]\}$ Since $S \subseteq Q$ and Q is finite, S is finite. Hence, for some $\alpha, \alpha' \in [\alpha^\infty]$ we have that $\delta(q_0, \alpha) = \delta(q_0, \alpha')$. Assume without loss of generality that α is a prefix of α' (from the definition of a path). Then $\alpha' = \alpha\beta$ for some address β. Let $q = \delta^*(q_0, \alpha)$. Then q is reachable and $\delta^*(q, \beta) = q$.

□

Chapter 2

Constraints and Constraint Programming

A constraint describes that a certain relationship should hold. A constraint may express that the distance between two points should be invariant, that the input of a electronic gate should be 1, or that two expressions should have the same type.

Constraints have been used as a tool to solve many different problems. In each application the domain of the constraints depends on the problem that are being solved. The domains used are as different as the set of real numbers, $\mathcal{R}$ [35], an arbitrary finite domain [50, 25], finite terms [74], sets of strings [52], and regular terms [65].

2.1 Introduction

Constraints arise all around us. There are constraints implicit in the laws of physics. We know also of economical constraints, time constraints, and many other kinds of constraints. A constraint may be an equation, it may be an inequality, or it may be some abstract relation over some abstract domain.

Let us turn to the world of physics. Assume that we have a rigid rod of length l, and assume that we have fixed one end of the rod at a non-movable point, (x_0, y_0, z_0). The other end of the rod moves freely in three-dimensional space. The position of this end can be described by the coordinates x, y, and z. Now, the distance between the two ends of the rod is $\sqrt{(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2}$, and hence the movement of the rod is *constrained* by the equation

$$(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2 = l^2$$

Suppose that we replace the rod with a ball on a string. Again, we fix the one end (the one opposite the ball) at (x_0, y_0, z_0). If the centre of the ball has coordinates (x, y, z) we have that the movement of the ball is constrained by the inequality

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 \leq l^2$$

So far, we have dealt with variables which range over the real numbers, but the domain of the constraints can be anything. Consider the problem of giving correct change. Suppose we have three kinds of coins, which has values 1, 5 and 10, respectively. Now, the problem of paying the amount of 37 can be expressed as the constraint

$$n_1 + 5n_5 + 10n_{10} = 37$$

This constraint should be solved over the domain of integer numbers. For example we have that $n_{10} = 3$, $n_5 = 1$, and $n_1 = 2$ is a solution, but we should not consider $n_{10} = 3.7$ and $n_5 = n_1 = 0$ one. Instead of adding further constraint to the effect that n_1, n_5, and n_{10} are integers it is simpler to say that we use the integers as our *constraint domain*.

There is in principle no limits to what could be considered constraint domains. For example, we can view systems of differential equations as sets of constraints over the domain of functions, or we can view the rules of chess as a set of constraints over the domain of chess positions. In the next section we shall look at the general structure of constraint domains.

2.2 Constraint Domains

Let us look at the elements comprising a constraint. As an example of a constraint consider the equation $A + B = C$. There are three elements of this constraint: The *variables* A, B, and C; the *function* $+$; and the *relation* $=$.

We need to give meaning to these elements before we can make sense of the above constraint. One such meaning could be that the *universe* is $\mathbf{R}$, that is A, B, and C take on real numbers; $+ : \mathbf{R}^2 \to \mathbf{R}$ is addition; and $= \subseteq \mathbf{R}^2$ is the relation that is true, iff the left-hand side and the right-hand side are the same number.

Formally, the elements of a constraint is a triple, $(\mathcal{V}, \Phi, \Gamma)$, where

- $\mathcal{V}$ is a recursively enumerable set of variables.
- $\Phi = \bigcup_{i=0}^{\infty} \Phi_i$ is a ranked alphabet, where for each $f \in \Phi$ we have $f \in \Phi_{\text{rank}(f)}$. We call Φ a *function alphabet*.
- $\Gamma = \bigcup_{i=0}^{\infty} \Gamma_i$ is a ranked alphabet as above. We call Γ a *relation alphabet*.

We can define the expression over a set of variables, $\mathcal{V}$, and a function alphabet, Φ, inductively as follows:

- v is an expression over $\mathcal{V}$ and Φ if $v \in \mathcal{V}$.
- $f(e_1, \ldots, e_k)$ is an expression over $\mathcal{V}$ and Φ, if $f \in \Phi_k$ and e_1, $\ldots$, e_k are expressions over $\mathcal{V}$ and Φ.

With this we can see what the form of the constraints are.

Definition 2.2.1 *Given a set of variables, $\mathcal{V}$, a function alphabet, Φ, and a relation alphabet, Γ, a constraint over $\mathcal{V}$, Φ, and Γ is on the form $R(e_1, \ldots, e_n)$ where $R \in \Gamma_n$ and e_1, $\ldots$, e_n are expressions over $\mathcal{V}$ and Φ.*

We shall write $\Gamma[\mathcal{V}, \Phi]$ for the set of constraints over $\mathcal{V}$, Φ, and Γ.

The following defines the meaning of functions and relations in a universe.

Definition 2.2.2 *Let $\mathcal{U}$ be a universe.*

1. *Let Φ be a function alphabet. An interpretation of Φ is a mapping $\mathcal{J} : \Phi \to \bigcup_{i=0}^{\infty}(\mathcal{U}^i \to \mathcal{U})$ such that $\mathcal{J}(f) : \mathcal{U}^{\mathrm{rank}(f)} \to \mathcal{U}$.*
2. *Let Γ be a relation alphabet. An interpretation of Γ is a mapping $\mathcal{I} : \Gamma \to \bigcup_{i=0}^{\infty} \mathcal{P}(\mathcal{U}^i)$ such that $\mathcal{I}(R) \subseteq \mathcal{U}^{\mathrm{rank}(R)}$.*

We now have the elements necessary to define a *constraint domain*.

Definition 2.2.3 *A constraint domain is a quintuple $\mathcal{D} = (\mathcal{U}, \Gamma, \Phi, \mathcal{I}, \mathcal{J})$, where*

- *$\mathcal{U}$ is a universe.*
- *Γ is a relation alphabet.*
- *Φ is a function alphabet.*
- *$\mathcal{I}$ is an interpretation of Γ.*
- *$\mathcal{J}$ is an interpretation of Φ.*

Let $\mathcal{V}$ be a recursively enumerable set of variables. The set of constraints over $\mathcal{D}$ and $\mathcal{V}$ is the set $\mathcal{D}[\mathcal{V}] = \Gamma[\mathcal{V}, \Phi]$.

That is, a constraint domain is a first-order structure, and the constraints are atomic formulae over the structure.

For any constraint domain, $\mathcal{D}$, as above we shall write $\mathcal{D} = (\mathcal{U}^{\mathcal{D}}, \Gamma^{\mathcal{D}}, \Phi^{\mathcal{D}}, \mathcal{I}^{\mathcal{D}}, \mathcal{J}^{\mathcal{D}})$.

The evaluation of an expression in an assignment, $\varphi : \mathcal{V} \to \mathcal{U}$, is inductively defined as:

- $\mathcal{J}(v)(\varphi) = \varphi(v)$ if $v \in \mathcal{V}$.
- $\mathcal{J}(f(e_1, \ldots, e_k))(\varphi) = (\mathcal{J}(f))(\mathcal{J}(e_1)(\varphi), \ldots, \mathcal{J}(e_k)(\varphi))$.

The most important aspect of constraints is that of finding a *solution* to the constraint. That is, an assignment of values to the variables such that the relationship holds.

Definition 2.2.4 *Let $\mathcal{D} = (\mathcal{U}, \Gamma, \Phi, \mathcal{I}, \mathcal{J})$ be a constraint domain, $\mathcal{V}$ a set of variables, and $R(e_1, \ldots, e_d) \in \mathcal{D}[\mathcal{V}]$. An assignment, $\varphi : \mathcal{V} \to \mathcal{U}$,* satisfies *$R(e_1, \ldots, e_d)$, iff $(\mathcal{J}(e_1)(\varphi), \ldots, \mathcal{J}(e_d)(\varphi)) \in \mathcal{I}(R)$. We write $\varphi \models_{(\mathcal{I},\mathcal{J})} C$ when φ satisfies C.*

If the context is clear we shall omit the $(\mathcal{I}, \mathcal{J})$ and write $\varphi \models C$. Similarly, we shall write $\varphi \models_{\mathcal{I}} C$ if it is clear what the interpretation of the function symbols is.

This extends to sets of constraints as below.

Definition 2.2.5 *Let $\mathcal{D}$ be a constraint domain, $\mathcal{V}$ a recursively enumerable set of variables, and $\mathcal{C} \subseteq \mathcal{D}[\mathcal{V}]$. An assignment, $\varphi : \mathcal{V} \to \mathcal{U}$, is a* solution *to $\mathcal{C}$, iff φ satisfies all constraints in $\mathcal{C}$. We write this as $\varphi \models_{(\mathcal{I},\mathcal{J})} \mathcal{C}$.*

We say that $\mathcal{C}$ is satisfiable *whenever a solution to $\mathcal{C}$ exists.*

Definition 2.2.6 *Let $\mathcal{C} \subseteq \mathcal{D}[\mathcal{V}]$ be a set of constraints. The* solution-space *of $\mathcal{C}$ is the set*

$$\mathcal{L}(\mathcal{C}) = \{\varphi : \mathcal{V} \to \mathcal{U}^{\mathcal{D}} | \varphi \models_{(\mathcal{I}^{\mathcal{D}},\mathcal{J}^{\mathcal{D}})} \mathcal{C}\}$$

2.3 Constraints in Programming

The definition in the previous section viewed constraint sets and solutions as something static. A constraint set is something given and an assignment is either a solution or it is not. For programming purposes this is not always practical. Here we often have that constraint sets evolve dynamically, and the question then arises how to find a new solution to the new constraint set.

In some views of constraints (e.g. [20, 61, 62]) constraints are viewed mostly as a description of how to obtain a new solution from an old one. That is, both the constraint set as well as the solution are dynamic.

2.3.1 General Techniques

In general, we can distinguish between two schools of thought:

1. The *denotational* view. The denotational view is the one shown in Section 2.2. Here a set of constraints, $\mathcal{C}$, denotes a solution-space, $\mathcal{L}(\mathcal{C})$. As seen above the solution-space is something static, it depends on nothing but the actual constraint set $\mathcal{C}$.

2. The *operational* view. Here a set of constraints denotes a set of methods for transforming one solution into another. Often it is not a question of whether a solution satisfies the constraint set or not, it is more a question of how badly it fails to satisfy it (see [61, 62]). In this view the solution depends not only on the constraint set but also on the history of constraint sets and solutions. This means that we can introduce things like *read-only constraints* where we can demand that the introduction of the constraint should leave certain variables unchanged.

One of the areas where constraints have been used in many applications is the area of user interface design [62, 46]. In this area it is costumary to use the operational view on constraints. There are several reasons for this. Among the most important is that it is not an option just to output "Unsatisfiable" when we are unable to find a solution. The user interface most work at any time, and we simply have to find a sufficiently good solution to the problem. This is dealt with by Borning et al. [7] by introducing a hierarchy between the constraints so we can define which of the constraints are most important.

One way of programming with constraints is to use the constraint set as a data structure, where we can add (and possibly remove) constraints and extract solutions and use the results in our ordinary programming languages which uses constraints in no other way. A special case of this is when our problems can be *reduced* to a constraint satisfaction problem. That is, when we can generate offline a set of constraints so that the set of constraints is satisfiable iff our original problem has a solution. In this special case the constraint *programming* aspect is all but gone.

A more ambitious project is to integrate the constraints into the programming languages so that the variables in the programming language and the variables of the constraints become the same variables. One language which does this is Kaleidoscope'90 [20], which integrates imperative programming with constraint programming. As a consequence temporal aspects are introduced into the constraint set, and hence we have to use the operational view of constraints. If we try to look at the constraint set with a denotational frame of mind we put ourselves in a situation where the present can influence the past (since we do not have read-only constraints).

Another successful integration is that of constraint programming and logic programming — called *constraint logic programming*. Since logic programming already takes a denotational view of the world, it is no surprise that constraint logic

programming languages use the denotational view on constraints (an exception is the hierarchical constraint logic programming language of Borning et al. [8]). We shall look more closely at constraint logic programming languages in Chapter 3 and indeed in the remainder of this dissertation.

2.3.2 Type Inference

For more than a decade there has been widespread use of constraint programming for type inference (e.g. [74]). The basic idea is that given an expression e, we can write the typing rules for e down as a set of constraints for the type of e, which we write as the variable $[\![e]\!]$. Similarly, each subexpression e' gives rise to a variable $[\![e']\!]$ and a number of constraints over this variable.

As an example let us look at the λ-calculus. In the λ-calculus we have the expression $(e\ e')$. This expression is an application of the function e on the argument e'. It follows then that e should have some function type, that e' has a type matching the argument type of e, and that the result of the function application should be a value which has the return type of e. We combine these insights into the constraint

$$[\![e]\!] = [\![e']\!] \to [\![(e\ e')]\!]$$

In 1978, Milner [47] devised the first type inference algorithm for the λ-calculus with polymorphic let. This algorithm was not constraint based. Instead, Milner computed a *principal type scheme* (see [15]) for each subexpression. The principal type scheme represented all types the expression could possibly take. From the principal type scheme of the subexpressions it is possible to compute the principal type scheme of the full expressions.

From the point of view of constraint programming we can say that the principal type scheme is a representation of the solution-space of the constraints. Clearly, it is language-dependent whether it is possible to represent the solution-space compactly. Hence, in the general case we have to keep the entire constraint set for the subexpressions. In this way we reduce the type inference problem to a problem of solving a constraint set.

Almost all constraint based type inference algorithms (e.g. [74, 32, 65, 52, 57]) works by a reduction like the one described above. Indeed, this method is considered standard to the degree that in [53], Palsberg generates constraints of the form

$$\mathsf{if}\ U = \mathsf{selftype}\ \mathsf{then}\ W \leqslant W'\ \mathsf{else}\ W'' \leqslant U$$

to be able to make the reduction to a set of constraints, rather than dynamically interact with the constraint set to choose between the constraints $W \leqslant W'$ and $W'' \leqslant U$.

Similarly, Palsberg and Schwartzbach [56] generates constraints on the form

$$A_1 \in [\![e_1]\!], \ldots, A_n \in [\![e_n]\!] \Rightarrow \mathcal{C}$$

where $\mathcal{C}$ is a constraint set. This example shows why it is not always right to reduce typeability to satisfiability of a constraint set, since the algorithm has a running time of $O(2^{n^2})$. Together with Oxhøj, the authors provides a better algorithm in [51]. This algorithm gives up on the idea of a reduction and instead generates constraints dynamically based on the constraint set generated thus far. The result is a dramatic decrease in complexity — the new algorithm has a polynomial running time.

Even though there are examples of type inference algorithms which does not work by reduction to satisfiability of a constraint set, so far there has not been attempts to implement type inference algorithms in a language which integrates constraints into a programming language. Chapter 8 is devoted to this subject.

Chapter 3

Constraint Logic Programming

The constraint logic programming paradigm has been the focus of much research since its introduction by Jaffar and Lassez in 1987 (see e.g. [8, 17, 23, 33, 40]). The paradigm extends Prolog and other programming languages with constraints so that it gives the opportunity to write a constraint program in a declarative fashion. In this chapter we look at the syntax and semantics of constraint logic programming and look at some important examples of constraint logic programming languages.

3.1 Logic Programming and CLP

The family of constraint logic programming languages is a generalisation of ordinary logic programming languages. The purpose of constraint logic programming languages is to extend the pure logic of languages like Prolog with a more useful domain such as the real numbers or the integers.

3.1.1 Logic Programming in Prolog

Logic programming languages such as Prolog [29] are developed as an attempt to make programming languages, which are closer to normal human understanding than the imperative programming languages such as Pascal. They allow the programmer to deal with the logical description of the program rather than the intrinsic details of program control.

The logical description used in logic programming languages are first-order formulae of the form

$$\forall x_1.\forall x_2.\ldots\forall x_l.Q_1(t_1^1,\ldots,t_{m_1}^1)\wedge\ldots\wedge Q_n(t_1^n,\ldots,t_{m_n}^n)\Rightarrow P(u_1,\ldots,u_k)$$

In Prolog notation this becomes

$$\mathtt{P}(u_1, \ldots, u_k) \text{ :- } \mathtt{Q_1}(t_1^1, \ldots, t_{m_1}^1), \ldots, \mathtt{Q_n}(t_1^n, \ldots, t_{m_n}^n).$$

or, if $n = 0$,

$$\mathtt{P}(u_1, \ldots, u_k).$$

A formula written in one of these ways is called a *rule* and the latter of the rules is called a *fact*. These rules can be understood as the axioms describing the problem. Understood in this way a Prolog program defines a set of atomic first-order formulae, namely the set of formulae that are the logical consequences of the set of rules in the program.

From a programming point of view the interesting aspect of such a program is the question of *answer generation*. That is, given a clause containing some indeterminate variables are there any values for the variables such that the clause becomes a consequence of the program?

Let us look at a simple example. (This and the following examples are from [63]):

$\mathtt{likes}(\mathsf{Eve}, \mathsf{apples})$.
$\mathtt{likes}(\mathsf{Eve}, \mathsf{wine})$.
$\mathtt{likes}(\mathsf{Adam}, x)$:- $\mathtt{likes}(x, \mathsf{wine})$.

In first-order logic the last rule reads

$$\forall x.\text{likes}(x, \text{wine}) \Rightarrow \text{likes}(\text{Adam}, x)$$

That is, Adam likes anybody who likes wine. Obviously, a logical consequence is that Adam likes Eve, but we could also use the program for answer generation. We could ask the question "is there anyone that Adam likes?", or in Prolog terms

?- $\mathtt{likes}(\mathsf{Adam}, x)$

And the result is that yes, for $x = \mathsf{Eve}$, Adam does in fact like x. This works both ways so we can also start with the *goal*

?- $\mathtt{likes}(x, \mathsf{Eve})$

That is, "is there anyone that likes Eve?", and the result is of course that for $x = \mathsf{Adam}$ we have that x likes Eve.

As a more realistic example (for programming purposes) let us consider the following rules dealing with the concatenation of lists:

$\mathtt{concat}(\mathsf{nil}, y, y)$.
$\mathtt{concat}(\mathsf{cons}(e, x), y, \mathsf{cons}(e, z))$:- $\mathtt{concat}(x, y, z)$.

The predicate $\texttt{concat}(x, y, z)$ is true, iff z is the concatenation of x and y. (In Prolog there is several pieces of syntactical sugar dealing with lists but these are beyond the scope of this work.)

Answer generation can be used to compute the concatenation of two lists as in the goal

?- $\texttt{concat}(\mathsf{cons}(a, \mathsf{cons}(b, \mathsf{nil})), \mathsf{cons}(c, \mathsf{nil}), z)$

We obtain the result $z = \mathsf{cons}(a, \mathsf{cons}(b, \mathsf{cons}(c, \mathsf{nil}))$. Alternatively, we could extract the "rest" of the list given a certain prefix as in the goal

?- $\texttt{concat}(\mathsf{cons}(a, \mathsf{cons}(b, \mathsf{nil})), y, \mathsf{cons}(a, \mathsf{cons}(b, \mathsf{cons}(c, \mathsf{nil}))))$.

This goal *succeeds* with the result $y = \mathsf{cons}(c, \mathsf{nil})$. Of course, this only works if the first argument is a prefix of the third argument. That is, the goal

?- $\texttt{concat}(\mathsf{cons}(a, \mathsf{nil}), y, \mathsf{nil})$

fails. A goal fails if no instantiation of the variables will make the goal a consequence of the program. Note that a predicate which defines a function also defines the inverse of that function.

The notions of success and failure in Prolog programs corresponds to some extend to the notions of true and false in ordinary programming languages (see [42]). By this analogy we can use the predicate `concat` to define new predicates

$\texttt{prefix}(x, z)$:- $\texttt{concat}(x, y, z)$.
$\texttt{suffix}(y, z)$:- $\texttt{concat}(x, y, z)$.

The first predicate reads "x is a prefix of z, if z is the concatenation of x and y for *some* y". So variables that appear on the right-hand side only is existentially quantified. This corresponds to the logical equivalence

$$\forall x : F \Rightarrow G \equiv (\exists x : F) \Rightarrow G$$

if x does not appear freely in G.

As a perhaps more interesting example of how a predicate defines the inverse as well as the original function, let us look at the `sum` predicate:

$\texttt{sum}(\mathsf{zero}, y, \mathsf{zero})$.
$\texttt{sum}(\mathsf{s}(x), y, \mathsf{s}(z))$:- $\texttt{sum}(x, y, z)$

where zero is 0 and s is the successor function. These rules defines in one predicate addition as well as subtraction. For example the goal

?- $\texttt{sum}(\mathsf{s}(\mathsf{s}(\mathsf{zero})), \mathsf{s}(\mathsf{s}(\mathsf{zero})), z)$

gives the result $z = \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{zero}))))$. And the goal

?- sum(s(s(zero)), y, s(s(s(zero))))

gives y = s(zero). These results correspond to 2+2 = 4 and 3−2 = 1, respectively.

3.1.2 From Prolog to CLP(X)

It is important that the results from the previous section *correspond* to 4 and 1, they do not *equal* 4 and 1. The reason is that Prolog looks for formulae that are logical consequences of the program. That is, they should be true in *every model* of the program, so that whatever meaning we give to the predicate sum and the function symbols s and zero — as long as it renders the rules valid — the results of the answer generation will be valid.

For this reason Prolog always makes *the most general substitution* of values for the variables, hence the results of computations will always be some finite term (like s(zero)) and computation in Prolog is in fact restricted to the *Herbrand Universe*.

The family of *constraint logic programming languages* [33] was created in order to introduce other universes than the Herbrand Universe into the logic programming languages. The way these universes are introduced are by means of *constraint domains* as defined in section 2.2.

For example let us look at the constraint domain, $\mathcal{Z}$. The universe of $\mathcal{Z}$ is the set of integers, the relations include $\leq$ and $=$, and the functions include $+$. The constraint logic programming language thus obtained is called CLP($\mathcal{Z}$). In CLP($\mathcal{Z}$) we can define the sum predicate above as

sum(x, y, z) :- $x + y = z$.

Clearly, a much more satisfying definition. Furthermore, the universe of discourse is the "right" one so that we can use the numbers as we are used to for example in the goals

?- sum(2, 2, z)
?- sum(2, y, 3)

which yield the results $z = 4$ and $y = 1$, respectively.

A further advantage of using CLP(X) is that X could be a constraint domain which is not easily encoded in the Herbrand Universe — for instance the real numbers, $\mathcal{R}$. CLP($\mathcal{R}$) [35] is the most publicised and most widely used of the family of constraint logic programming languages. Here we will look at a single example (taken from [33]). The language is described further in section 3.4.1.

The example we shall look at is that of complex multiplication. We define a predicate zmul, such that zmul($r_1, t_1, r_2, t_2, r_3, t_3$) iff $r_3 + i \cdot t_3 = (r_1 + i \cdot t_1) \cdot (r_2 + i \cdot t_2)$.

$\texttt{zmul}(r_1, t_1, r_2, t_2, r_3, t_3)$:- $r_3 = r_1 \cdot r_2 - t_1 \cdot t_2$, $t_3 = r_1 \cdot t_2 + r_2 \cdot t_1$.

This predicate can be used in multiplication as well as division of complex numbers. The goal

?- $\texttt{zmul}(1, 2, 3, 4, r_3, t_3)$

yields the result $r_3 = -5, t_3 = 10$.

Sometimes the goal does not yield a definite answer: The goal

?- $\texttt{zmul}(1, 2, r_2, t_2, r_3, t_3)$

succeeds with the *constraint set* $i_2 = 0.2 \cdot i_3 - 0.4 \cdot r_3, r_2 = 0.4 \cdot i_3 + 0.2 \cdot r_3$.

In general, if the computation of a CLP(X) program succeeds, the result is a satisfiable constraint set. How to obtain a useful solution from this set is quite domain-dependent [34]. As we shall see, in this work we obtain the *smallest* solution to the constraint set.

3.2 The CLP(X) Scheme

Joxan Jaffar and Jean-Louis Lassez introduced the CLP(X) scheme in [33]. The 'X' in CLP(X) stands for a constraint domain. In the notation introduced in section 2.2 the scheme would be called CLP($\mathcal{D}$). We will adopt the latter notation throughout this section.

3.2.1 Preliminaries

In this subsection we will describe the syntax of the CLP($\mathcal{D}$) language. This language is a well-defined language only if $\mathcal{D}$ is a *solution-compact* constraint domain containing the constraint $=$. Solution-compactness is defined in [33] as the following requirements:

1. Every element of $\mathcal{U}^{\mathcal{D}}$ is uniquely definable by a finite or infinite set of constraints. That is, for each $x \in \mathcal{U}^{\mathcal{D}}$ there is a set of constraints $\mathcal{C} \subseteq \mathcal{D}[\mathcal{V}]$ and a variable $v \in \mathcal{V}$, such that for all solutions, φ, of $\mathcal{C}$ we have that $\varphi(v) = x$. The elements of $\mathcal{U}^{\mathcal{D}}$ that are definable by an infinite set of constraints only are called the *limit elements* of $\mathcal{D}$.
2. Every assignment not satisfying some constraint, $C \in \mathcal{D}[\mathcal{V}]$, satisfies another constraint, $C' \in \mathcal{D}[\mathcal{V}]$, such that $\{C, C'\}$ is unsatisfiable.

That $\mathcal{D}$ contains $=$ can be written formally as:

$$= \in \Gamma_2^{\mathcal{D}} \quad \text{and} \quad \mathcal{I}^{\mathcal{D}}(=) = \{(x, x) | x \in \mathcal{U}^{\mathcal{D}}\}$$

The syntax of $\mathsf{CLP}(\mathcal{D})$ is much like the syntax of Prolog described above. Let $\Pi = \bigcup_{i=0}^{\infty} \Pi_i$ be a ranked alphabet of *relation symbols* such that $\Pi \cap \Gamma^{\mathcal{D}} = \emptyset$, and let $\mathcal{V}$ be a recursively enumerable set of variables. An *atomic formula* is a formula of the form $P(e_1, \ldots, e_k)$, where $P \in \Pi_k$ and for all i, e_i is an expression over $\mathcal{V}$ and $\Phi^{\mathcal{D}}$. A *rule* is one of the following:

$\mathtt{P}(u_1, \ldots, u_k).$
$\mathtt{P}(u_1, \ldots, u_k)$:- $B_1, \ldots, B_n, C_1, \ldots, C_{n'}.$

where $P \in \Pi_k$, the u_i are expressions over $\mathcal{V}$ and $\Phi^{\mathcal{D}}$, the B_i are atomic formulae, and the C_i are constraints from $\mathcal{D}[\mathcal{V}]$. In the latter rule we can have $n = 0$ or $n' = 0$, but not both. $\mathtt{P}(u_1, \ldots, u_k)$ is called the *conclusion* of the rule, and the B_i and C_i are called the *antecedents*. A *program* is a (usually finite) set of rules.

A *goal* is a non-empty set of atomic formulae and constraints. We write a goal as the following:

?- $B_1, \ldots, B_n, C_1, \ldots, C_{n'}.$

where the B_i and C_i are as above.

3.2.2 Declarative Semantics

Given a constraint domain, $\mathcal{D}$, and a relation alphabet, Π, we consider first-order structures, $\mathcal{M}$, with carrier $\mathcal{U}^{\mathcal{D}}$, the function symbols $f \in \Phi^{\mathcal{D}}$ with interpretation $\mathcal{J}^{\mathcal{D}}(f)$, and the relation symbols $R \in \Gamma^{\mathcal{D}}$ and $P \in \Pi$, the former with interpretation $\mathcal{I}^{\mathcal{D}}(R)$ the latter with some interpretation specified by a set $\mathcal{M}_{\text{base}} \subseteq \mathcal{D}_{\text{base}}$, where $\mathcal{D}_{\text{base}}$ is the set

$$\mathcal{D}_{\text{base}} = \{P(d_1, \ldots, d_k) | P \in \Pi_k \wedge \forall i \in 1..k : d_i \in \mathcal{U}^{\mathcal{D}}\}$$

We shall say that $\mathcal{M}$ is a *model* of the program π, iff $\mathcal{M}$ satisfies all the rules in π considered as universally quantified first-order formulae. We write this as $\mathcal{M} \models \pi$.

3.2.3 Fixpoint semantics

The fixpoint semantics is described by using the function T_π defined as

$$T_\pi(S) = \{d \in \mathcal{D}_{\text{base}} | \exists A \text{ :- } \mathcal{B}, \mathcal{C} \in \pi, \varphi : \mathcal{V} \to \mathcal{U}^{\mathcal{D}}. \varphi(A) = d \wedge \varphi \models \mathcal{C} \wedge \varphi(B) \subseteq S\}$$

The function maps a set of statements, S, to the set of statements that can be proven from S using one of the implications in the program π. The following theorem is a consequence of this.

Theorem 1 (Jaffar & Lassez [33]) *Let π be a $\mathsf{CLP}(\mathcal{D})$ program and let $\mathcal{M}$ be a model of π. We have that $\mathcal{M}$ is the least model of π (w.r.t set inclusion), iff $\mathcal{M}_{\text{base}}$ is the least fixed point of T_π.*

This theorem establishes the equivalence of the declarative and fixpoint semantics.

3.2.4 Operational Semantics

The operational semantics of a $\mathsf{CLP}(\mathcal{D})$ program is actually a (non-deterministic) proof search for the goal. The semantics start with a goal and uses T_π^{-1} until reaching the empty set. A bit more information is needed throughout. What we need is a goal containing constraints as well as atomic formulae. We write a goal like $(\mathcal{B};\mathcal{C})$ where $\mathcal{B}$ is a set of formulae and $\mathcal{C}$ is a set of constraints. The goal $(\{B_1,\ldots,B_n\};\{C_1,\ldots,C_{n'}\})$ corresponds to

$$\text{?- } B_1,\ldots,B_n,C_1,\ldots,C_{n'}$$

For convenience we shall write x, M as shorthand for $\{x\} \cup M$.

Consider a goal $(P(s_1,\ldots,s_k),\mathcal{B};\mathcal{C})$, where $P \in \Pi_k$ and s_1, …, s_k are expressions. And assume that there is a rule,

$$P(t_1,\ldots,t_k) \ :- \ \mathcal{B}',\mathcal{C}'.$$

which is a fresh instantiation of a rule in π. That is the rule above is α-equivalent with some rule from π, and the variables in the rule does not occur anywhere in the goal. We have that

$$(P(s_1,\ldots,s_k),\mathcal{B};\mathcal{C}) \rhd_\pi (\mathcal{B}\cup\mathcal{B}';\mathcal{C}\cup\mathcal{C}'\cup\{s_i=t_i|i\in 1..n\})$$

if the constraint set $\mathcal{C}\cup\mathcal{C}'\cup\{s_i=t_i|i\in 1..n\}$ is satisfiable. We shall omit the π and write $G \rhd G'$ whenever the context is clear.

Let $\rhd_\pi^*$ be the reflexive, transitive closure of $\rhd_\pi$. We say that the goal $(\mathcal{B};\mathcal{C})$ *succeeds*, iff there is a satisfiable constraint set, $\mathcal{C}'$, such that

$$(\mathcal{B};\mathcal{C}) \rhd_\pi^* (\emptyset;\mathcal{C}')$$

If the goal does not succeed we say that it *fails*, and if all derivation sequences are finite we say that it *fails finitely*.

The following theorem establishes the equivalence between the operational and declarative semantics (and hence also between the operational and fixpoint semantics).

Theorem 2 (Kozen [40]) *Let π be a program and let $\mathcal{M}$ be the least model of π. For a set of atomic formulae, $\mathcal{B}$, and sets of satisfiable constraint sets, $\mathcal{C}$ and $\mathcal{C}'$, we have that*

$$(\mathcal{B};\mathcal{C}) \rhd_\pi^* (\emptyset;\mathcal{C}') \Leftrightarrow \mathcal{M} \models \mathcal{C}' \Rightarrow \mathcal{B} \wedge \mathcal{C}$$

3.3 Interpretation of CLP

The operational semantics of section 3.2.4 describes how an interpreter should work, but it leaves open two algorithmical problems: constraint solving and proof search. In this section we look at these issues.

3.3.1 Constraint Solvers

It is clear from the operational semantics that the algorithmical problem of maintaining a set of constraints becomes very important in CLP. The set of constraints should be maintained in such a way that we can perform the step involved in $G \rhd G'$.

A *constraint solver* is an algorithm for finding solutions to sets of constraints. Given a constraint set, $\mathcal{C}$, the constraint solver must decide whether $\mathcal{C}$ is satisfiable. If $\mathcal{C}$ is satisfiable it should be possible to extract a satisfying assignment from the answer provided by the constraint solver.

In order to be well-suited for the interpretation the constraint solver should satisfy the following requirements:

- It should be *incremental*. That is, it should be able to efficiently solve the set $\mathcal{C} \cup \mathcal{C}'$ when it have already solved $\mathcal{C}$.
- It should be *complete*. That is, it should be able to solve any set of constraints.

As we shall see, the latter of these requirements is not always met.

3.3.2 Proof Search

The operational semantics from section 3.2.4 constitutes a non-deterministic algorithm for interpreting CLP($\mathcal{D}$). The problem in this section is to find a deterministic version of this algorithm.

To see that there is in fact non-determinism involved consider the following CLP($\mathcal{Z}$) program, π_1:

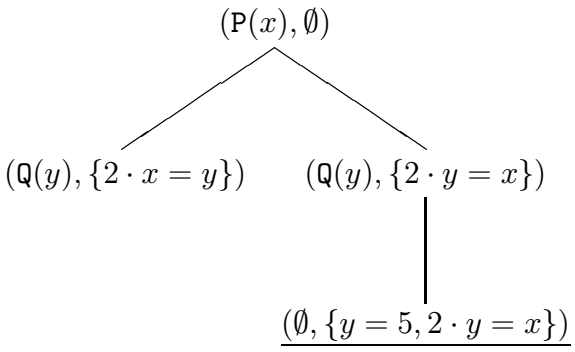


Figure 3.1: The computation tree for the goal $(\mathtt{P}(x), \emptyset)$ in π_1.

$\mathtt{P}(x)$:- $2 \cdot x = y$, $\mathtt{Q}(y)$.
$\mathtt{P}(x)$:- $2 \cdot y = x$, $\mathtt{Q}(y)$.
$\mathtt{Q}(5)$.

Starting with the goal $(\mathtt{P}(x); \emptyset)$ and choosing the first of the rules we obtain the new goal $(\mathtt{Q}(y); \{2 \cdot x = y\})$. The only rule we can conceivably choose at this point is the third one, but the constraint set $\{5 = y, 2 \cdot x = y\}$ is unsatisfiable in $\mathcal{Z}$ since $2 \nmid 5$. We can see that we have chosen the wrong rule. The second rule leads to $(\mathtt{Q}(y); \{2 \cdot y = x\})$ which again leads to $(\emptyset; \{y = 5, 2 \cdot y = x\})$. The latter constraint set is clearly satisfiable.

Barring the option of psychic computer programs, the only possibility is to *search* for a successful computation. There are two ways of doing this: *breadth-first search* and *depth-first search*. These correspond to a breadth-first search and a depth-first search of the *computation tree* shown in figure 3.1. That is, breadth-first search proceeds by first looking at all computations of length 1, then all computations of length 2, then length 3, and so on. Depth-first search tries to follow a single computation as far as possible until it reaches either a goal of the form $(\emptyset; \mathcal{C})$ in which case it is done, or a goal from which there is no possible transitions in which case it will have to *backtrack* to the last place where a decision was made and undo that decision.

The problem with breadth-first search is its lack of efficiency. A full binary tree of depth n has as many as $2^n - 1$ nodes. So, while the breadth-first search will have to search through an exponential number of nodes in order to find the right leaf, depth-first search *may* find the right leaf in n steps. Of course, in general the likelihood of finding the right leaf in the first attempt is very low $(2^{-(n-1)})$ but in connection with a CLP program it could be very high in practice — especially if the programmer is aware of the search strategy and designs the program accordingly. Furthermore, we have the possibility that several of the leaves are successful.

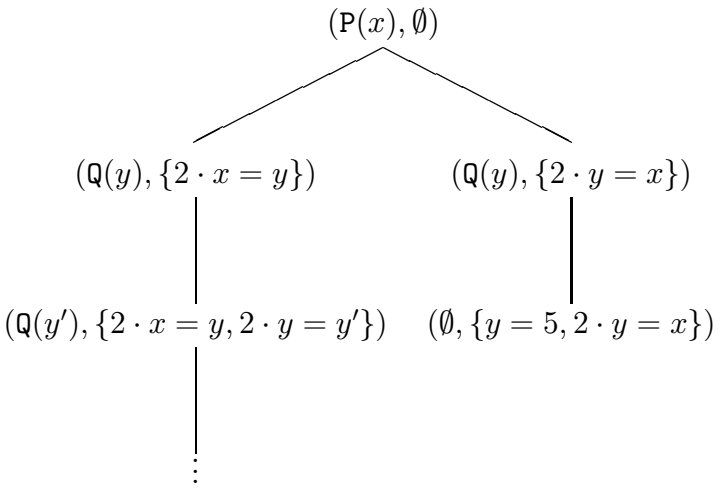


Figure 3.2: The computation tree for the goal $(\mathtt{P}(x), \emptyset)$ in π_2.

It seems thus that the balance swings in the direction of depth-first search, but there is a grave problem with this. The depth-first search may traverse a longer path than actually necessary because of poor choices. This can be illustrated quite dramatically by making a small change to π_1 obtaining the program π_2:

$\mathtt{P}(x)$:- $2 \cdot x = y$, $\mathtt{P}(y)$.
$\mathtt{P}(x)$:- $2 \cdot y = x$, $\mathtt{Q}(y)$.
$\mathtt{Q}(5)$.

The computation tree for the goal $(\mathtt{P}(x), \emptyset)$ can be seen in figure 3.2. Note that the tree contains the *infinite* path

$$\begin{array}{lllll} (\mathtt{P}(x), \emptyset) & \rhd_{\pi_2} & (\mathtt{P}(y), & \{x = 2 \cdot y\}) \\ & \rhd_{\pi_2} & (\mathtt{P}(y'), & \{x = 2 \cdot y, y = 2 \cdot y'\}) \\ & \rhd_{\pi_2} & (\mathtt{P}(y''), & \{x = 2 \cdot y, y = 2 \cdot y', y' = 2 \cdot y''\}) \\ & \rhd_{\pi_2} & \cdots & \end{array}$$

All the constraint sets are satisfiable in $\mathcal{Z}$ so there is no way for the depth-first strategy to avoid diverging down this infinite path, while the breadth-first strategy finds the successful computation quite rapidly.

Note that there are programs and goals where both kinds of searches diverge — for instance the program consisting of only the first rule from π_2. In fact, in general the question of whether there is a successful computation for a certain goal is undecidable [63]. The real difference is that breadth-first search will find a successful computation whenever such a computation exists, whereas depth-first search may diverge.

The choice between depth-first and breadth-first search is thus not clear. Depth-first search gives efficiency and breadth-first search gives completeness. From a

pragmatic point of view we must say that the overhead involved in using breadth-first search renders the programming language impossible to use in practice, and as a consequence most implementations of CLP languages as well as the Prolog family uses the depth-first search strategy.

In Prolog non-logical operators are introduced to give the programmer better control over the search process. This has the drawback of compromising the spirit of logic programming and constraint logic programming, where the programmer is supposed to deal with the logical description of the programming only and let the interpreter deal with the control.

In conclusion, the choice of search strategy for (constraint) logic programming languages is not clear, but the most promising strategy in practice is depth-first search.

3.4 Constraint Logic Programming Languages

There have been several instances of the general CLP(X)-scheme. We shall take a look at some of the most important and most relevant in connection with this work.

3.4.1 CLP($\mathcal{R}$)

The CLP($\mathcal{R}$) [35] is the first and best known constraint logic programming language that was designed to be an instance of the CLP(X) scheme. The computation domain of CLP($\mathcal{R}$) is the set of real numbers, and it is for this feature it has attracted much attention. There are many applications for the domain of real numbers and CLP($\mathcal{R}$) is well suited for many difficult problems, such as linear programming, circuit analysis, economic theories, and more.

Formally, the domain $\mathcal{R}$ is

$$(\mathbf{R}, \{=, \leq, <, \neq, \geq, >\}, \{+, -, \cdot, /\} \cup \mathbf{Q}, \mathcal{I}^{\mathcal{R}}, \mathcal{J}^{\mathcal{R}})$$

where

$$\begin{aligned}
\mathcal{I}^{\mathcal{R}}(=) &= \{(a,a) | a \in \mathbf{R}\} \\
\mathcal{I}^{\mathcal{R}}(\leq) &= \{(a,b) \in \mathbf{R}^2 | a \leq b\} \\
\mathcal{I}^{\mathcal{R}}(<) &= \{(a,b) \in \mathbf{R}^2 | a < b\} \\
\mathcal{I}^{\mathcal{R}}(\neq) &= \mathbf{R}^2 \setminus \mathcal{I}^{\mathcal{R}}(=) \\
\mathcal{I}^{\mathcal{R}}(\geq) &= \mathbf{R}^2 \setminus \mathcal{I}^{\mathcal{R}}(<) \\
\mathcal{I}^{\mathcal{R}}(>) &= \mathbf{R}^2 \setminus \mathcal{I}^{\mathcal{R}}(\leq)
\end{aligned}$$

$$\begin{aligned}
\mathcal{J}^{\mathcal{R}}(+)(a,b) &= a+b \\
\mathcal{J}^{\mathcal{R}}(-)(a,b) &= a-b \\
\mathcal{J}^{\mathcal{R}}(\cdot)(a,b) &= ab \\
\mathcal{J}^{\mathcal{R}}(/)(a,b) &= \frac{a}{b} \\
\mathcal{J}^{\mathcal{R}}(q) &= q
\end{aligned}$$

where $q \in \mathbf{Q}$. We can see that the domain contains the relation $=$ as required. To see that $\mathcal{R}$ is solution-compact we must consider the two requirements.

1. To see that the real numbers are uniquely definable we start by observing that any element $q \in \mathbf{Q}$ is the only solution to the constraint $v = q$. Now, consider a number, $r \in \mathbf{R} \setminus \mathbf{Q}$. All such numbers can be written as the limit of an infinite sequence of rational numbers, so let

   $$r = \lim_{n \to \infty} q_n$$

   where $q_n \in \mathbf{Q}$ for all n. Then r is the only solution for v in the constraint set

   $$\{-\epsilon < v - q < \epsilon | \epsilon \in \mathbf{Q}, \epsilon > 0, q \in \{q_i | i \in \mathbf{N}_0\}, |r - q| < \epsilon\}$$

   The limit elements of $\mathcal{R}$ are the transcendental numbers [33].

2. The second requirement follows directly from the fact that the negation of every relation is also in the relation set.

The description above is of a *pure* CLP($\mathcal{R}$) language. The actual language contains uninterpreted functors as well (as in Prolog). That is, it is possible to combine symbolic computation with computation in the domain of reals as in the following example.

$$\texttt{zmul}(\mathsf{c}(r_1,t_1),\mathsf{c}(r_2,t_2),\mathsf{c}(r_3,t_3)) \text{ :- } r_3 = r_1 \cdot r_2 - t_1 \cdot t_2,\ t_3 = r_1 \cdot t_2 + r_2 \cdot t_1.$$

This is simply a reformulation of the complex multiplication example but here we have coded a complex number as $\mathsf{c}(r,t)$. In this version we can write

?- $\texttt{zmul}(\mathsf{c}(1,2),\mathsf{c}(3,4),z)$

and get the result $z = \mathsf{c}(-5,10)$.

The full language still belongs to the CLP(X) scheme but we shall not be concerned with the details in this work.

The CLP($\mathcal{R}$) language uses the Simplex method to solve the arithmetical constraints. This introduces a peculiarity because the Simplex method is not complete: it can only solve linear constraints — that is, constraints that do not contain products of two (or more) variables. As a result CLP($\mathcal{R}$) has two constraint stores: one for the linear constraints and one for the non-linear constraints.

The constraints in the linear store are solved immediately but the constraints in the non-linear store are *delayed* until information from the linear store can make them linear. Sometimes, CLP($\mathcal{R}$) is unable to linearise all constraints and is thus unable to tell whether the goal succeeds.

As an example of this we can look at the goal

?- $\texttt{zmul}(\texttt{c}(r_1, 2), \texttt{c}(r_2, 4), -5, 10), r_2 < 3$

This results in the constraint set

$$\{r1 = -.5 \cdot r_2 + 2.5, 3 = r_1 \cdot r_2, r_2 < 3\}$$

and the answer that this constraint set *may* be satisfiable (it is, with $r_1 = 1.5$ and $r_2 = 2$). The non-linear constraint, $3 = r_1 \cdot r_2$, makes it impossible for CLP($\mathcal{R}$) to tell whether or not the constraint set is satisfiable.

3.4.2 CHIP

The constraint logic programming language CHIP (Constraint Handling In Prolog) [17] contains three kinds of domains: Finite domains, The Boolean domain, and the domain of rational numbers.

Finite domains. For any finite universe, Σ, the finite constraint domain is

$$\mathcal{F}_\Sigma = (\Sigma, \{=, \neq\} \cup \{\text{element}_n |, n \geq 1\} \cup \{\text{alldistinct}_n | n \geq 1\} \cup \ldots\}, \Sigma, \mathcal{I}^\mathcal{F}, \mathcal{J}^\mathcal{F})$$

where the interpretation $\mathcal{I}^\mathcal{F}$ is

$$\begin{aligned}
\mathcal{I}^\mathcal{F}(=) &= \{(\sigma, \sigma) | \sigma \in \Sigma\} \\
\mathcal{I}^\mathcal{F}(\neq) &= \Sigma^2 \setminus \mathcal{I}^\mathcal{F}(=) \\
\mathcal{I}^\mathcal{F}(\text{element}_n) &= \{(i, \sigma_1, \ldots, \sigma_n, \sigma) | 1 \leq i \leq n, \sigma_i = \sigma\} \\
\mathcal{I}^\mathcal{F}(\text{alldistinct}_n) &= \{(\sigma_1, \ldots, \sigma_n) | \forall i, j \in 1..n : i \neq j \Rightarrow \sigma_i \neq \sigma_j\}
\end{aligned}$$

The interpretation $\mathcal{J}^\mathcal{F}$ is the identity function. The expressions

$$\text{element}_n(i, \sigma_1, \ldots, \sigma_n, \sigma) \qquad \text{and} \qquad \text{alldistinct}_n(\sigma_1, \ldots, \sigma_n)$$

are written in CHIP as

$$\text{element}(i, [\sigma_1, \ldots, \sigma_n], \sigma) \qquad \text{and} \qquad \text{alldistinct}([\sigma_1, \ldots, \sigma_n])$$

respectively.

On top of the predefined constraints CHIP also allows *user-defined constraints* (the '...' in the definition above). Any relation can be defined by the user to be a constraint. Hence the actual relation alphabet is in fact $\bigcup_{i=1}^{\infty} \mathcal{P}(\Sigma^i)$, and we can view the relations element_n and alldistinct_n as syntactical sugar.

If $\Sigma \subseteq \mathbf{Z}$, the predefined relations also include $<$, $>$, $\leq$, and $\geq$.

The Boolean domain. The Boolean domain is the domain of boolean arithmetic,

$$\mathcal{B} = (\{ff, tt\}, \{\Leftrightarrow, \not\Leftrightarrow\}, \{\&, \text{not}, 0\}, \mathcal{I}^{\mathcal{B}}, \mathcal{J}^{\mathcal{B}})$$

where $\mathcal{I}^{\mathcal{B}}$ and $\mathcal{J}^{\mathcal{B}}$ are

$$\begin{aligned} \mathcal{I}^{\mathcal{B}}(\Leftrightarrow) &= \{(ff, ff), (tt, tt)\} \\ \mathcal{I}^{\mathcal{B}}(\not\Leftrightarrow) &= \{(ff, tt), (tt, ff)\} \end{aligned}$$

$$\begin{aligned} \mathcal{J}^{\mathcal{B}}(\&)(a, b) &= \begin{cases} \text{tt} & \text{if } a = b = \text{tt} \\ \text{ff} & \text{otherwise} \end{cases} \\ \mathcal{J}^{\mathcal{B}}(\text{not})(a) &= \begin{cases} \text{tt} & \text{if } a = \text{ff} \\ \text{ff} & \text{otherwise} \end{cases} \\ \mathcal{J}^{\mathcal{B}}(0) &= \texttt{ff} \end{aligned}$$

The usual function symbols are derived:

$$\begin{aligned} 1 &\equiv \text{not}(0) \\ a!b &\equiv \text{not}(\text{not}(a)\&\text{not}(b)) \\ a\#b &\equiv (a\&\text{not}(b))!(\text{not}(a)\&b) \\ a \text{ nand } b &\equiv \text{not}(a\&b) \\ a \text{ nor } b &\equiv \text{not}(a!b) \end{aligned}$$

The domain of rational numbers. The domain of rational numbers, $\mathcal{Q}$, is like $\mathcal{R}$ except that the universe consists of rational numbers only. In CHIP there is another restriction: all constraints should be linear. The latter requirement is a deviation from the CLP($\mathcal{D}$) scheme, but it makes sense from a pragmatic point of view since there is no known algorithm for dealing with non-linear constraints. As a result of the linearity there is a complete constraint solver for CHIP.

Extensions to CHIP The CHIP language is extended with a number of non-CLP relations and operations. Among the former we have higher-order relations for minimising and maximising functions, among the latter we have several operations dealing with the evaluation order of constraints.

These extensions are included to improve efficiency and practical applicability, and indeed CHIP has many practical applications in areas such as operations research and circuit design (see [17]).

3.4.3 The Prolog Family

The constraint logic programming languages were designed as an extension of Prolog. Thus, Prolog is also an instance of the CLP($\mathcal{D}$) scheme. The constraint domain of Prolog is the Herbrand Universe defined as

$$\mathsf{H}_\Sigma = (\mathsf{Fin}_\Sigma, \{=\}, \Sigma, \mathcal{I}^\mathsf{H}, \mathcal{J}^\mathsf{H})$$

where

$$\begin{aligned} \mathcal{I}^\mathsf{H}(=) &= \{(a,a) | a \in \mathsf{Fin}_\Sigma\} \\ \mathcal{J}^\mathsf{H}(\sigma) &= \sigma^{\mathsf{T}_\Sigma} \end{aligned}$$

Since any finite term is definable as a finite set of acyclic constraints, H_Σ has no limit elements, and is thus solution-compact [33].

The constraint solving algorithm used in Prolog is an incremental version of Robinson's *unification* algorithm [60].

Prolog can only work with finite trees, hence the cyclic constraint $x = f(x)$ has no solution. That is, the goal $(P(x), \emptyset)$ fails (finitely) for the program

$P(x)$:- $Q(x, f(x))$.
$Q(y, y)$.

Colmerauer [10, 11] introduced the Prolog II programming language designed to overcome shortcomings of this sort in Prolog. The Prolog II language can work with infinite trees and is thus able to find a successful computation in the case above.

The domain of Prolog II is the *Infinitary Herbrand Universe*,

$$\mathsf{IH}_\Sigma = (\mathsf{T}_\Sigma, \{=, \neq\}, \Sigma, \mathcal{I}^\mathsf{IH}, \mathcal{J}^\mathsf{IH})$$

The interpretations are

$$\begin{aligned} \mathcal{I}^\mathsf{IH}(=) &= \{(a,a) | a \in \mathsf{T}_\Sigma\} \\ \mathcal{I}^\mathsf{IH}(\neq) &= \mathsf{T}_\Sigma^2 \setminus \mathcal{I}^\mathsf{IH}(=) \\ \mathcal{J}^\mathsf{IH}(\sigma) &= \sigma^{\mathsf{T}_\Sigma} \end{aligned}$$

Note that although Prolog II can sometimes succeed when Prolog fails finitely, it cannot succeed when Prolog fails infinitely as in the following program

$$P(f(x)) \text{ :- } P(x).$$

We might expect this to yield the solution $x = f(x)$, but in fact it diverges (in accord with the semantics of section 3.2.4) along the infinite computation

$$(P(x), \emptyset) \rhd (P(x'), \{x = f(x')\}) \rhd (P(x''), \{x = f(x'), x' = f(x'')\}) \rhd \cdots$$

This corresponds to the fact that the fixpoint semantics defines the *smallest* fixpoint of T_π.

IH_Σ is quite easily seen to be solution-compact:

1. Every term is the unique solution of some set of equations. Since the terms definable by a finite set of equations is exactly the regular terms, we see that the limit elements of IH_Σ are the irregular terms.
2. Since the relation alphabet includes $=$ as well $\neq$, the negation of every constraint is also a constraint.

Prolog II uses unification of regular terms [58] in the constraint solver. This is complete since it can find the solution for any finite set of $=$-constraints. For $\neq$-constraints it is sufficient to check whether the solution found by unification violates any of these constraints [10]. See section 6.1 for a detailed treatment of the unification algorithm for regular terms.

Prolog III [12] is an extension to Prolog II with rational numbers, the Boolean domain, and the string domain,

$$\text{Str} = (\mathcal{U}^*, \{::, =\}, \{.\} \cup \{\langle \underbrace{\cdot, \ldots, \cdot}_{n} \rangle | n \geq 0\}, \mathcal{I}^{\text{Str}}, \mathcal{J}^{\text{Str}})$$

where the interpretations are

$$\begin{aligned} \mathcal{I}^{\text{Str}}(=) &= \{(a, a) | a \in \mathcal{U}^*\} \\ \mathcal{I}^{\text{Str}}(::) &= \{(\langle a_1, \ldots, a_n \rangle, n) | a_i \in \mathcal{U}*, n \geq 0\} \end{aligned}$$

$$\begin{aligned} \mathcal{J}^{\text{Str}}(.)(\langle a_1, \ldots, a_n \rangle, \langle b_1, \ldots, b_m \rangle) &= \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle \\ \mathcal{J}^{\text{Str}}(\langle \cdot, \ldots, \cdot \rangle)(a_1, \ldots, a_n) &= \langle a_1, \ldots, a_n \rangle \end{aligned}$$

As an example of using the domain of strings let us look at the following example from [12]:

$$P(z) \text{ :- } \langle 1, 2, 3 \rangle . z = z . \langle 2, 3, 1 \rangle .$$

The goal $(P(z), \emptyset)$ succeeds with the constraint $\langle 1, 2, 3 \rangle . z = z . \langle 2, 3, 1 \rangle$ which has an infinite number of solutions. Prolog III delays evaluation of such constraints

until the length of the constraint is known. The goal $(P(z), \{z :: 10\})$ succeeds with the definite answer

$$z = \langle 1, 2, 3, 1, 2, 3, 1, 2, 3, 1 \rangle$$

For the full treatment of Prolog III we refer the reader to [12].

3.4.4 CLP(SC)

The CLP(SC) language was introduced by Dexter Kozen in [40, 39]. Its domain is the domain of *set constraints*. Set constraints are set inclusions over the universe of sets of terms. Such a set constraint could be of the form

$$x \subseteq f(y)$$

The above constraint means that all terms in x are of the form $f(t)$, where $t \in y$. The sets that can be defined in this way are the *regular sets* — the sets of terms that can be described by a finite tree automaton. A set, $S_1 \subseteq \mathcal{U}^{\mathsf{H}}$, is regular, iff it can be described by a finite set of set equations like this:

$$\begin{array}{rcl} S_1 & = & e_1 \\ S_2 & = & e_2 \\ & \vdots & \\ S_m & = & e_m \end{array} \tag{3.1}$$

where e_i is of the form $\bigcup_{j=1}^{n} \sigma_j(y_1^j, \ldots, y_{k_j}^j)$ where σ_j is a function symbol of rank k_j and $y_l^j \in \{S_1, \ldots, S_m\}$. If the ranked alphabet containing the allowed function symbols is Σ, we write RS_Σ for the universe of regular sets definable this way.

Given a ranked alphabet, $\Sigma = \bigcup_{k=0}^{\infty} \Sigma_k$, we can define the domain of set constraints as

$$\mathsf{SC}_\Sigma = (\mathsf{RS}_\Sigma, \{\subseteq, \not\subseteq\}, \{\cup, \neg, 0\} \cup \Sigma, \mathcal{I}_{\mathsf{SC}}, \mathcal{J}_{\mathsf{SC}})$$

where the interpretations are

$$\begin{array}{rcl} \mathcal{I}^{\mathsf{SC}}(\subseteq) & = & \{(S, S') \in \mathsf{RS}_\Sigma^2 | \forall t \in S : t \in S'\} \\ \mathcal{I}^{\mathsf{SC}}(\not\subseteq) & = & \mathsf{RS}_\Sigma^2 \setminus \mathcal{I}^{\mathsf{SC}}(\subseteq) \end{array}$$

$$\begin{array}{rcl} \mathcal{J}^{\mathsf{SC}}(\cup)(S, S') & = & \{t \in \mathcal{U}^{\mathsf{H}} | t \in S \vee t \in S'\} \\ \mathcal{J}^{\mathsf{SC}}(\neg)(S) & = & \{t \in \mathcal{U}^{\mathsf{H}} | t \notin S\} \\ \mathcal{J}^{\mathsf{SC}}(0) & = & \emptyset \\ \mathcal{J}^{\mathsf{SC}}(\sigma)(S_1, \ldots, S_k) & = & \{\sigma^{T_\Sigma}(t_1, \ldots, t_k) | t_1 \in S_1, \ldots, t_k \in S_k\} \end{array}$$

where $\sigma \in \Sigma_k$. A constraint using the relation $\subseteq$ is called a *positive set constraint* and a constraint using $\not\subseteq$ is called a *negative set constraint*. Sometimes, only the positive set constraints are allowed.

We define the usual functions and relations on sets as follows

$$\begin{aligned} S \cap S' &\doteq \neg(\neg S \cup \neg S') \\ S \setminus S' &\doteq S \cap \neg S' \\ S \oplus S' &\doteq S \setminus S' \cup S' \setminus S \\ 1 &\doteq \neg 0 \end{aligned}$$

$$\begin{aligned} S = S' &\doteq S \oplus S' \subseteq 0 \\ S \neq S' &\doteq S \oplus S' \not\subseteq 0 \end{aligned}$$

Let us turn to the solution-compactness of SC_Σ.

1. The unique definability follows directly from the fact that every regular set can be defined by a finite set of equations of the form (3.1). Note that there is no limit elements of SC_Σ.

2. Assume that C is a constraint and φ is an assignment not satisfying C. If negative constraints are allowed we can let C' be the negation of C. If only positive constraints are allowed we exploit the fact that φ can be uniquely defined by a set of constraints of the form (3.1). The constraint C' is then

$$\bigcup_{i=1}^{m} S_i \oplus e_i \subseteq 0$$

   which is simply another way of writing (3.1).

The satisfiability problem of set constraints is decidable [3, 24] but complete for non-deterministic exponential time [2]. For some alphabets, though, the problems is easier and a particular CLP(SC)-program may have a feasible running time. Kozen [39] uses an incremental version of the algorithm in [3] as a constraint solver for SC. This constraint solver is complete.

CLP(SC) extends ordinary logic programming with sets of terms. The terms of Prolog can be encoded in CLP(SC) as singleton sets. If t is a Σ-term then for any assignment, φ, we have that $\mathcal{J}^{\mathsf{SC}}(t)(\varphi) = \{t\}$. The membership relation is thus encoded as set inclusion. This can be used to make a Prolog-like program where the non-determinism shows up as non-singleton sets rather than multiple solutions.

The main uses for set constraints are in program analysis and type inference [4, 26, 27, 48]. The CLP(SC) language thus provides a way of implementing

these forms of program analysis simply by writing the specification directly in the program. An example is given in [39] of the monadic approximation of the collecting semantics of a simple programming language (see [26] for a thorough treatment of this). For example we have that the monadic approximation to the collecting semantics of '=' is

$$\widehat{\Psi}[x = y] = \begin{cases} \widehat{\Psi}[x \leftarrow \widehat{\Psi}(x) \cap \widehat{\Psi}(y), y \leftarrow \widehat{\Psi}(x) \cap \widehat{\Psi}(y)] & \text{if } \widehat{\Psi}(x) \cap \widehat{\Psi}(y) \neq \emptyset \\ x \mapsto \emptyset & \text{otherwise} \end{cases}$$

In CLP(SC) this becomes

$$\texttt{test}(s_1, \ldots, s_n, \text{“}x_i = x_j\text{”}, \ldots, s_i \cap s_j, \ldots, s_i \cap s_j, \ldots) \text{ :- } s_i \cap s_j \neq \emptyset.$$
$$\texttt{test}(s_1, \ldots, s_n, \text{“}_i = s_j\text{”}, 0, \ldots, 0) \text{ :- } s_i \cap s_j = \emptyset.$$

where "$x_i = x_j$" is some suitable encoding of the expression. Here the function $\widehat{\Psi}$ is encoded by a finite sequence of sets — one for each variable. This has the problem that there is a new program for each n. We shall look closer into this subject in Chapter 7.

Chapter 4

Constraint Domains for Type Inference

In this chapter we will design a constraint domain which is suited for type inference. We have two important considerations when defining this domain.

1. It should be *general*.
2. It should be *efficient*.

The first of the considerations means that we are looking for a constraint domain that is suited for as many type inference problems as possible. The other of the considerations say that there should nevertheless be an efficient constraint solver for the domain.

Clearly, the two goals are conflicting. For goal number 1, we would like to be able to use the domain for every type inference problem whatsoever. But many type inference problems are very hard: they may be *NP*-complete [54], DEXPTIME-complete [45, 37], or even undecidable [28, 36].

We will define a constraint domain which is efficient (as we shall see in Chapter 6) and sufficiently general to be of use in many kinds of type inference (as we shall see in Chapter 8). This constraint domain is the *Ordered Infinitary Herbrand Universe*, OIH.

4.1 Constraints over Finite and Regular Terms

In the majority of type systems, the types are defined as a set of terms following a particular grammar. According to the type system the terms may be either

finite or regular. In this section we explore these type systems and look at a way to have user-defined relations over types of this form.

4.1.1 Type Systems and Constraints

The type expressions used in a programming language are usually finite or regular terms over some ranked alphabet. As an example we could look at the ML types [47, 15]. The ML types are the terms generated by the following grammar:

$$\tau ::= \mathsf{Bool}|\mathsf{Int}|\mathsf{Real}|\tau \to \tau|\alpha|\beta|\gamma| \ldots$$

The set of ML types is $\mathsf{Fin}_{\Sigma^{\mathsf{ML}}}$, where $\Sigma_0^{\mathsf{ML}} = \{\mathsf{Bool}, \mathsf{Int}, \mathsf{Real}\} \cup \mathcal{V}^{\mathsf{ML}}$ and $\Sigma_2^{\mathsf{ML}} = \{\to\}$, and for all other k, $\Sigma_k^{\mathsf{ML}} = \emptyset$. $\mathcal{V}^{\mathsf{ML}}$ is a (usually finite) set of *type variables*. We use α, α', β, etc. to range over type variables.

The constraints in ML are term equations over Σ^{ML} and $\mathcal{V}^{\mathsf{ML}}$, and *unifiability* constraints. Two terms, τ_1 and τ_2, are unifiable, iff there is a function $\varphi : \mathcal{V}^{\mathsf{ML}} \to \mathsf{Fin}_{\Sigma^{\mathsf{ML}}}$, such that $\varphi_{\Sigma^{\mathsf{ML}}}(\tau_1) = \varphi_{\Sigma^{\mathsf{ML}}}(\tau_2)$. We write $\tau_1 \approx \tau_2$, iff τ_1 and τ_2 are unifiable.

Unifiability constraints are not the same as term equations: The constraint set $\{v \approx \mathsf{Int}, v \approx \mathsf{Real}\}$ is satisfiable (with e.g. $\phi(v) = \alpha$), but the constraint set $\{v = \mathsf{Int}, v = \mathsf{Real}\}$ is not. Note that we have introduced two levels of solution here: the one from the definition of solution (Definition 2.2.5), and the one from the definition of unifiability.

The difference between the two kinds of constraints is exploited in defining polymorphism in ML. If a function, f, is defined to be polymorphic then the expression (fx) result in the constraint $[\![f]\!] \approx [\![x]\!] \to [\![(fx)]\!]$, otherwise the constraint is $[\![f]\!] = [\![x]\!] \to [\![(fx)]\!]$, as in the simply typed λ-calculus (see Section 8.1).

Our next example is a subtyping system from [59]. The types in [59] are defined by

$$\tau ::= \mathsf{Int}|\mathsf{Bool}|\mathsf{Prod}(a_1 : \tau, \ldots, a_n : \tau)|\mathsf{Sum}(a_1 : \tau, \ldots, a_n : \tau)|\mathsf{List}(\tau)|\tau \to \tau$$

(Here we have omitted the somewhat peculiar unnamed products and sums.) In the constructs $\mathsf{Prod}(a_1 : \tau, \ldots, a_n : \tau)$ and $\mathsf{Sum}(a_1 : \tau, \ldots, a_n : \tau)$ it is required that the a_i are different. These constructs correspond to an infinite number of labels in the alphabet. This will be the subject of Section 4.3.

The typing rules for the language with subtyping uses the *subtyping order*, $\leqslant_{\mathsf{st}}$, defined as the smallest reflexive and transitive relation, such that

(1) $\tau_1 \to \tau_2 \leqslant_{\mathsf{st}} \tau_1' \to \tau_2'$, if $\tau_1' \leqslant_{\mathsf{st}} \tau_1$ and $\tau_2 \leqslant_{\mathsf{st}} \tau_2'$.

(2) $\mathsf{Prod}(a_1 : \tau_1, \ldots, a_n : \tau_n) \leqslant_{\mathsf{st}} \mathsf{Prod}(a_1 : \tau_1', \ldots, a_n : \tau_n')$, if $\forall i \in 1..n : \tau_i \leqslant_{\mathsf{st}} \tau_i'$.

(3) $\mathsf{Prod}(a_1 : \tau_1, \ldots, a_n : \tau_n) \leqslant_{\mathsf{st}} \mathsf{Prod}(a'_1 : \tau'_1, \ldots, a'_n : \tau'_m)$, if

$$\{a_i | i \in 1..n\} \supseteq \{a'_i | i \in 1..m\} \wedge \forall i \in 1..n, j \in 1..m : a_i = a'_j \Rightarrow \tau_i = \tau'_j$$

(4) $\mathsf{Sum}(a_1 : \tau_1, \ldots, a_n : \tau_n) \leqslant_{\mathsf{st}} \mathsf{Sum}(a_1 : \tau'_1, \ldots, a_n : \tau'_n)$, if $\forall i \in 1..n : \tau_i \leqslant_{\mathsf{st}} \tau'_i$.

(5) $\mathsf{Sum}(a_1 : \tau_1, \ldots, a_n : \tau_n) \leqslant_{\mathsf{st}} \mathsf{Sum}(a'_1 : \tau'_1, \ldots, a'_n : \tau'_m)$, if

$$\{a_i | i \in 1..n\} \subseteq \{a'_i | i \in 1..m\} \wedge \forall i \in 1..n, j \in 1..m : a_i = a'_j \Rightarrow \tau_i = \tau'_j$$

(6) $\mathsf{List}(\tau) \leqslant_{\mathsf{st}} \mathsf{List}(\tau')$, if $\tau \leqslant_{\mathsf{st}} \tau'$.

This is the same ordering that is used in [9]. There are two kinds of rules above. The rules (1), (2), (4), and (6) are of the first kind. These are the *congruence rules* or *inductive rules*. They say that the labels in the alphabet preserve the relation between the types (an exception to this is the contravariance rule (1) which has the order reversed in the first argument). The rules (3) and (5) are of the second kind. These rules are the *root label rules*. Where the congruence rules have the same root label at either side, the root label rules relates types with different root labels.

In addition to the rules above we also have the implicit rules

(7) $\mathsf{Int} \leqslant_{\mathsf{st}} \mathsf{Int}$

(8) $\mathsf{Bool} \leqslant_{\mathsf{st}} \mathsf{Bool}$

We shall treat these rules as root label rules since they are defined over the rule labels only.

A more involved example is that of Turbo Pascal [31, 32]. In Turbo Pascal there are many different types and relations. Let us look at one of the central relations, assignment compatibility. For simplicity, we shall look at the relation restricted to the alphabet $\Sigma^{\mathsf{TP}} = \{\mathsf{Integer}, \mathsf{Real}, \mathsf{Set}, {}^{\wedge}\}$. The relation Ac is the relation defined such that $Ac(\tau, \tau')$, iff one of the following holds

(1) $\tau = \mathsf{Integer}$ and $\tau' = \mathsf{Integer}$

(2) $\tau = \mathsf{Real}$ and $\tau' = \mathsf{Integer}$

(3) $\tau = \mathsf{Real}$ and $\tau' = \mathsf{Real}$

(4) $\tau = \mathsf{Set}(\tau_1)$, $\tau' = \mathsf{Set}(\tau'_1)$, and $Tc(\tau_1, \tau'_1)$.

(5) $\tau = \tau' = {}^{\wedge}(\tau'')$

where Tc is the type compatibility relation. Note that rule (4) above is not truly a congruence rule because it does not refer to the assignment compatibility relation. We will however treat rule (4) as a congruence rule since it is structurally similar

to the true congruence rules. We shall see that (5) is a congruence rule as well in this sense.

Our goal is to define a constraint domain, which allows constraints to be defined by such rules but still allows an efficient constraint solver.

4.1.2 Defining Relations

In this section we explore the possibilities for a uniform notation for defining relations by congruence and root label rules. The goal is to be able to write down for example the definition of $\leqslant_{\mathsf{st}}$ as the *greatest* relation satisfying the following description.

$$
\begin{array}{l}
\tau \leqslant_{\mathsf{st}} \tau' \leftrightarrow \\
\quad (\mathsf{Int}, \mathsf{Int}); \\
\quad (\mathsf{Bool}, \mathsf{Bool}); \\
\quad (\rightarrow, \rightarrow) \qquad \text{if } \tau'/1 \leqslant_{\mathsf{st}} \tau/1 \text{ and } \tau/2 \leqslant_{\mathsf{st}} \tau'/2; \\
\quad (\mathsf{Prod}[a_1, \ldots, a_n], \mathsf{Prod}[b_1, \ldots, b_n]) \\
\qquad\qquad \text{if } \{a_i\} \supseteq \{b_j\} \text{ and } \forall i, j : a_i = b_j \Rightarrow \tau/i \leqslant_{\mathsf{st}} \tau'/j; \\
\quad (\mathsf{Sum}[a_1, \ldots, a_n], \mathsf{Sum}[b_1, \ldots, b_n]) \\
\qquad\qquad \text{if } \{a_i\} \subseteq \{b_j\} \text{ and } \forall i, j : a_i = b_j \Rightarrow \tau/i \leqslant_{\mathsf{st}} \tau'/j; \\
\quad (\mathsf{List}, \mathsf{List}) \qquad \text{if } \tau/1 \leqslant_{\mathsf{st}} \tau'/1.
\end{array}
$$

In the example above we have introduced *conditional constraints* in e.g. the case $(\rightarrow, \rightarrow)$. The meaning of this is the straightforward one that types with root labels $\rightarrow$ are related only if their subterms are related in the right way. We know the constraints make sense because $\rightarrow$ has rank 2, and so the expressions $\tau/1$ and $\tau/2$ are well-defined whenever $\tau(\epsilon) = \rightarrow$. We cannot allow expressions of the form τ/α where $|\alpha| > 1$ since it cannot be known whether the expression is well-defined. It is however meaningful to write τ/ϵ (or just τ) since this is always well-defined.

Note how we have made the field names a part of the root label. This means that the alphabet is infinite. It also means that the expression $\{a_i\} \subseteq \{b_j\}$ is a part of the root label rule rather than the congruence rule. The order of the field names is significant only when we refer to the subterms. It is needed to assure that we get the right i and j in the expressions of the form $\forall i, j : a_i = b_j \Rightarrow \ldots$. It is convenient to treat the labels as being on the form $\mathsf{Prod}[\{a_1, \ldots, a_n\}]$, so we introduce a special notation for the expressions above to retain the access to the right subterms. We rewrite the rule as the following.

$$
\begin{array}{l}
\tau \leqslant_{\mathsf{st}} \tau' \leftrightarrow \\
\quad (\mathsf{Int}, \mathsf{Int}); \\
\end{array}
$$

$(\mathsf{Bool}, \mathsf{Bool})$;
$(\rightarrow, \rightarrow)$ if $\tau'/1 \leqslant_{\mathsf{st}} \tau/1, \tau/2 \leqslant_{\mathsf{st}} \tau'/2$;
$(\mathsf{Prod}[\alpha], \mathsf{Prod}[\beta])$ if $\beta \subseteq \alpha, \forall l \in \alpha\&\beta : \tau/l \leqslant_{\mathsf{st}} \tau'/l$;
$(\mathsf{Sum}[\alpha], \mathsf{Sum}[\beta])$ if $\alpha \subseteq \beta, \forall l \in \alpha\&\beta : \tau/l \leqslant_{\mathsf{st}} \tau'/l$;
$(\mathsf{List}, \mathsf{List})$ if $\tau/1 \leqslant_{\mathsf{st}} \tau'/1$.

This is the final form of the definition.

In the definition above we had that the relation was recursively defined. This is not the case for assignment compatibility which is defined as

$Ac(\tau, \tau') \leftrightarrow$
$(\mathsf{Integer}, \mathsf{Integer})$;
$(\mathsf{Real}, \mathsf{Integer})$;
$(\mathsf{Real}, \mathsf{Real})$;
$(\mathsf{Set}, \mathsf{Set})$ if $Tc(\tau/1, \tau'/1)$;
$(\hat{}, \hat{})$ if $\tau/1 = \tau'/1$.

Where Tc is previously user-defined. The $=$ relation will not be user-defined, however, because it will not comply with the restrictions we will later put on the user-defined relations to allow for an efficient constraint solver (the definition of stability in Section 4.6). Instead $=$ is a special *pre-defined* relation and will be dealt with as a special case throughout.

The non-recursive relations are in general easier to handle than the recursive ones, so we distinguish these two kinds of constraints.

Let us recapitulate the elements that appear in a user-defined constraint.

The root label rules. In the finite case these rules can be any relation over finite universes. We look at the properties of such relations in Section 4.2.

Root label rules over infinite alphabets. Section 4.3 is devoted to the study of these. We define the general root label rules as relations of the form R_Q where R is any relation over a finite universe and Q defines the inequalities of the form $\alpha \subseteq \beta$.

Another problem considered in this section is how to retain the information about which subterms correspond to which field names now that we no longer have the ordering explicitly. This is dealt with by the introduction of *relabeling functions*.

The conditional constraints. These are defined in Section 4.4. The bulk of the section is about defining the semantics of constraints of the form $\forall j \in \alpha\&\beta : \ldots$

Section 4.5 is concerned with combining these elements and provide recursive and non-recursive user-defined constraints.

These constraints are quite general, but they suffer an efficiency problem. We first encounter this problem in connection with the root label rules. We show in Section 4.2.2 that the constraint satisfaction problem for these constraints is *NP*-complete. We solve this problem by introducing an ordering such that we can solve the constraints efficiently under certain conditions known as *stability*.

Section 4.6 shows how we can extend the requirements to the general user-defined relations. More precisely, we define the notions of *monotonicity* and *strictness* for side conditions. These are necessary and sufficient conditions for the efficiency of the constraint solver in Chapter 6.

The ordering that is necessary for stability well play an important role in the definition of infinite alphabets (in Section 4.3).

4.2 The Constraint Satisfaction Problem

The constraint satisfaction problem is the problem of finding a satisfying assignment to a set of general constraints over a finite universe. That the constraints are general means that any relation over the finite universe can be used. For this reason the constraint satisfaction problem has many applications. Unfortunately, it also means that it has very poor complexity properties.

4.2.1 CSP and User-defined Constraints

The constraint satisfaction problem is a constraint domain that arises when we allow user-defined constraints. Let us look at the root label rule part of the definition of Ac in the previous section. That is the definition without the side conditions. [1]

$$
\begin{aligned}
&Ac'(\tau, \tau') \leftrightarrow \\
&\quad (\mathsf{Integer}, \mathsf{Integer}); \\
&\quad (\mathsf{Real}, \mathsf{Integer}); \\
&\quad (\mathsf{Real}, \mathsf{Real}); \\
&\quad (\mathsf{Set}, \mathsf{Set}); \\
&\quad (\hat{}, \hat{}).
\end{aligned}
$$

This no longer defines assignment compatibility, but it serves to illustrate a user-defined relation on root labels. The relation defined is a binary relation over Σ^{TP}, that is it is a subset of $\Sigma^{\mathsf{TP}} \times \Sigma^{\mathsf{TP}}$, namely the set

$$Ac' = \{(\mathsf{Integer}, \mathsf{Integer}), (\mathsf{Real}, \mathsf{Integer}), (\mathsf{Real}, \mathsf{Real}), (\mathsf{Set}, \mathsf{Set}), (\hat{}, \hat{})\}$$

[1] In this special case τ and τ' are redundant.

This leads to a general definition of the constraint domain, CSP.

Definition 4.2.1 *Let Σ be a finite set of values. The constraint satisfaction problem domain over Σ is the constraint domain*

$$\mathsf{CSP}_\Sigma = (\Sigma, \bigcup_{k=0}^{\infty} \mathcal{P}(\Sigma^k), \emptyset, R \mapsto R, f \mapsto f)$$

That is, the CSP domain has all possible relations, but no function symbols or constants. The reason for this is that every constraint including functions or constants can be rewritten with a constraint of the form $R(v_1, \ldots, v_k)$ as long as we have all possible relations in the relation alphabet.

Often in the literature (e.g. [22, 50]) only unary and binary constraints are considered. That is, all constraints are of the form $R(v)$ or $R(v, v')$ (where $v \neq v'$). In this case the former constraints are called *node constraints* and the latter *arc constraints*.

4.2.2 The Time Complexity of CSP

The *constraint satisfaction problem* for Σ is the satisfiability problem for the domain CSP_Σ. If the constraints are restricted to unary and binary relations we shall refer to the problem as the *binary constraint satisfaction problem* for Σ. Both problems are well-known to be *NP*-complete as the following theorem shows.

Theorem 3 *Let Σ be a finite set of values.*

1. *The constraint satisfaction problem for Σ is NP-complete if $|\Sigma| > 1$.*
2. *The binary constraint satisfaction problem for Σ is NP-complete if $|\Sigma| > 2$.*

Proof: It is easy to test whether a given assignment satisfies a set of constraints. Hence both problems are in *NP*. It remains to be seen that they are *NP*-hard. We show this by reduction from the *NP*-complete problems 3-SAT and 3-colouring, respectively.

1. By assumption Σ has at least 2 distinct values. Call these values *tt* and *ff*, respectively. We define the relations $B = \{tt, ff\}$ and $D = B^3 \backslash \{(ff, ff, ff)\}$. Now, given a formula from 3-SAT we generate for each propositional variable, two variables $[\![A]\!]$ and $[\![\neg A]\!]$ and the constraints $B([\![A]\!])$, $B([\![\neg A]\!])$, and $[\![A]\!] \neq [\![\neg A]\!]$. For each conjunct $L_1 \vee L_2 \vee L_3$ we make the constraint $D([\![L_1]\!], [\![L_2]\!], [\![L_3]\!])$. It is clear that the size of the constraint set is polynomial (in fact, linear) in the size of the formula.

Now, let φ be a solution to the constraint set. We have that $\tilde{\varphi}$ defined as $\tilde{\varphi}(A) = \varphi([\![A]\!])$ satisfies the formula. Conversely, let $\tilde{\varphi}$ be an assignment satisfying the formula, then φ defined as $\varphi([\![A]\!]) = \tilde{\varphi}(A)$ and $\varphi([\![\neg A]\!]) = \neg\tilde{\varphi}(A)$ is a solution to the constraint set.

2. We have three distinct values, *red*, *green*, and *blue* in Σ. We define the relation $C = \{red, green, blue\}$. Now, assume that $G = (V, E)$ is an undirected graph. For each node $v \in V$ we make the variable $[\![v]\!]$ and the constraint $C([\![v]\!])$; and for each edge $(v, v') \in E$ we make the constraint $[\![v]\!] \neq [\![v']\!]$. Again we have a constraint set of linear size.

   We define $\tilde{\varphi}(v) = \varphi([\![v]\!])$. Then φ is a solution to the constraint set, iff $\tilde{\varphi}$ is a valid colouring of the graph.

□

If $|\Sigma| = 1$ both constraint satisfaction problems are of course trivial. If $|\Sigma| = 2$ the binary constraint satisfaction problem is equivalent to 2-SAT which is known to be solvable in linear time [19].

The bad complexity of the constraint satisfaction problems has given rise to many suggestions on how to resolve that situation. A common method is to look for an algorithm which has an exponential worst-case running time, but is fast on many practical problems [25, 68].

Another approach is that of Freuder [21] who looks at the binary constraint satisfaction problem. He looks on the structure of the constraint set and define the property *strong consistency* such that backtracking is not needed, if the constraint set is strongly consistent. The problem then becomes how to make the constraint set strongly consistent.

Following Freuder, Detcher and Pearl [16] devises an efficient backtracking-free algorithm. Basically, we can look at a constraint set as a graph which has as nodes the variables in the set, and has an edge between two nodes iff the set contains a constraint over the variables. If this *constraint graph* is a tree, no backtracking is needed if we have *arc consistency*. As we shall see in Section 6.2, we can make a constraint set arc consistent in linear time (linear in the sum of the sizes of the relations that is).

The approach we shall be taking here also avoids backtracking, but is does so by restricting the relation alphabet rather than the structure of the graph. We look at a certain partial ordering on the values and look only at relation which is *stable* with respect to this ordering as defined in the following.

Definition 4.2.2 *Let Σ be a finite set of values and let $\sqsubseteq$ be a partial ordering such that $(\Sigma, \sqsubseteq)$ forms a lower semi-lattice. We say that a relation $R \subseteq \Sigma^k$ is*

$\sqsubseteq$-stable, *iff it is closed under* $\sqcap$. *That is, for every* $\sigma_1, \ldots, \sigma_k, \sigma'_1, \ldots, \sigma'_k \in \Sigma$ *such that* $R(\sigma_1, \ldots, \sigma_k)$ *and* $R(\sigma'_1, \ldots, \sigma'_k)$ *we have that* $R(\sigma_1 \sqcap \sigma'_1, \ldots, \sigma_k \sqcap \sigma'_k)$.

Note that since Σ is finite, all non-empty subsets of Σ has a greatest lower bound (by induction on the size of the subset), and hence all satisfiable constraint sets have a $\sqsubseteq$-least solution. We say that a constraint satisfaction problem is a *stable constraint satisfaction problem* if all the relations are stable.

We show in Section 6.2 that a constraint satisfaction problem where all the constraints are stable is solvable in linear time. The idea is that the ordering should guide our search for a proof and the stability of the constraints ensures that backtracking is unnecessary. For now, let us note that the reductions in the proof of Theorem 3 can no longer work because the $\neq$ relations are unstable with respect to *all* orderings. To see this assume that $\sigma \neq \sigma'$. Then we have $\sigma' \neq \sigma$, but due to the definition of $\sqcap$ it is never the case that $\sigma \sqcap \sigma' \neq \sigma' \sqcap \sigma$.

By contrast to the above, the Ac' relation defined in the beginning of this section is stable with respect to the ordering

$$\mathsf{Integer} \sqsubseteq \mathsf{Real} \sqsubseteq \mathsf{Set} \sqsubseteq {}^{\wedge}$$

But this is not the only ordering the relation is stable with respect to. It is also stable with respect to the ordering

- $\mathsf{Real} \sqsubseteq \mathsf{Integer}$
- $\mathsf{Real} \sqsubseteq \mathsf{Set}$
- $\mathsf{Real} \sqsubseteq {}^{\wedge}$

The choice of ordering makes no difference regarding the definition of the Ac-relation. The only difference is in the choice of a solution among the many possible ones.

Another example of stable constraint sets are *Horn clauses*. A Horn clause is an expression of the form $\bigvee_{i=1}^{n} L_i$, where all the L_i is either A_i or $\neg A_i$, and at most one of the L_i is A_i. A Horn Clause can be written in one of the following forms.

- A
- $A_1 \wedge \cdots \wedge A_n \Rightarrow A$
- $\neg(A_1 \wedge \cdots \wedge A_n)$

It is easy to check that each of these can be written as a $\Rightarrow$-stable relation ($\Rightarrow$ is easily seen to be a complete lattice and hence a lower semi-lattice). We have then that any *Horn formula* can be written as a $\Rightarrow$-stable satisfaction problem, where a Horn formula is an expression of the form $\bigwedge_{j=1}^{m} C_j$ where the C_j are

Horn clauses. In fact, the Horn formulae was the motivation for the definition of stability.

4.3 Infinite Alphabets

Type constructors such as Prod and Sum are not like the normal root labels. They introduce *field names* into the type expression and they do not have a fixed arity. In order to put these expressions into the normal scheme for making terms we introduce an *infinite number* of type constructors

$$\{\mathsf{Prod}[\alpha] | \alpha \text{ is a set of field names}\}$$

where $\text{rank}(\mathsf{Prod}[\alpha]) = |\alpha|$. So the term $\mathsf{Prod}(a_1 : \tau_1, \ldots, a_k : \tau_k)$ is written as $\mathsf{Prod}[\{a_1, \ldots, a_k\}](\tau_1, \ldots, \tau_k)$ instead. This introduces the problem of which a_i belongs to which τ_j. Instead we will write $\mathsf{Prod}[a_1, \ldots, a_k](\tau_1, \ldots, \tau_k)$. This notation will be formalised at the end of this section.

4.3.1 Ordered Infinite Alphabets

All labels in the infinite alphabet induced by Prod are of the form $\mathsf{Prod}[\alpha]$. We call Prod the *generic label* and α the *extension* of the label. Using this we can define an infinite alphabet as generated by a finite set of generic and non-generic labels and their extensions. This provides some structure to the infinite alphabets which will turn out to be useful when designing the constraint solver in Chapter 6.

Definition 4.3.1 *Let Σ be a finite ranked alphabet, $\mathcal{E}$ a partial function such that $\text{dom}(\mathcal{E}) \subseteq \Sigma$ and for each $\sigma \in \text{dom}(E)$, $\mathcal{E}(\sigma)$ is a recursively enumerable set of extensions equipped with a size function, $|\cdot|_\sigma$. The infinite alphabet generated by Σ, and $\mathcal{E}$ is the set*

$$\Sigma_\mathcal{E} = \Sigma \setminus \text{dom}(\mathcal{E}) \cup \bigcup_{\sigma \in \text{dom}(\mathcal{E})} \{\sigma[e] | e \in \mathcal{E}(\sigma)\}$$

where the rank of $\sigma[e]$ is $\text{rank}(\sigma) + |e|_\sigma$.

We call the labels from $\text{dom}(\mathcal{E})$ the *generic labels* of Σ. It is clear that for a finite alphabet Σ we have that $\Sigma = \Sigma_\mathcal{E}$, where $\text{dom}(\mathcal{E}) = \emptyset$.

We also define an ordering over this alphabet given orderings over Σ and e.

Definition 4.3.2 *Let Σ and $\mathcal{E}$ be as above. Furthermore let $\sqsubseteq$ be a lower semi-lattice on Σ and $\preceq_\sigma$ a lattice on $\mathcal{E}(\sigma)$ such that all non-empty subsets have greatest lower bounds. The ordering generated by $\sqsubseteq$ and $\preceq$ is the least ordering, $\leqslant$, over the infinite alphabet generated by Σ and $\mathcal{E}$, such that*

- $\sigma \leqslant \sigma'$, *if* $\sigma \sqsubseteq \sigma'$.
- $\sigma \leqslant \sigma'[\bot_{\mathcal{E}(\sigma')}]$, *if* $\sigma \sqsubseteq \sigma'$.
- $\sigma[\bot_{\mathcal{E}(\sigma)}] \leqslant \sigma'$, *if* $\sigma \sqsubseteq \sigma'$.
- $\sigma[\bot_{\mathcal{E}(\sigma)}] \leqslant \sigma'[\bot_{\mathcal{E}(\sigma')}]$, *if* $\sigma \sqsubseteq \sigma'$.
- $\sigma[e] \leqslant \sigma[e']$, *if* $e \preceq_\sigma e'$.

If $\leqslant$ *is as above we shall write* $\leqslant = \sqsubseteq_\preceq$.

The $\bot_{\mathcal{E}(\sigma)}$ in the definition above is the least element of $\mathcal{E}(\sigma)$. We call an element of the form $\sigma[\bot_{\mathcal{E}(\sigma)}]$ a *ground instance* of σ, and write a ground instance of σ as $\sigma[]$. It follows from the definition that all non-empty subsets of $\Sigma_\mathcal{E}$ has a greatest lower bound in $\Sigma_\mathcal{E}$.

The motivating example for this was the alphabet for subtyping from [59]. This can be written as $\Sigma^{\mathsf{st}} = \Sigma_\mathcal{E}$ where

$$\Sigma = \{\mathsf{Int}, \mathsf{Bool}, \mathsf{Prod}, \mathsf{Sum}, \mathsf{List}, \to\}$$
$$\mathcal{E}(\mathsf{Prod}) = \mathcal{E}(\mathsf{Sum}) = \mathcal{P}(\{a | a \text{ is a field name}\})$$

The orderings $\preceq_{\mathsf{Prod}}$ and $\preceq_{\mathsf{Sum}}$ are both the set inclusion ordering.

Definition 4.3.2 says that only ground instances of different generic labels are comparable. We can illustrate this as in Figure 4.1, where $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5\}$, $\mathrm{dom}(\mathcal{E}) = \{\sigma_2, \sigma_4, \sigma_5\}$, and $\sqsubseteq$ is the smallest ordering such that

- $\sigma_1 \sqsubseteq \sigma_2$
- $\sigma_1 \sqsubseteq \sigma_3$
- $\sigma_1 \sqsubseteq \sigma_4$
- $\sigma_4 \sqsubseteq \sigma_5$

The cones marked '$\mathcal{E}(\sigma_i)$' in the figure are the infinite subalphabets generated by the generic labels.

4.3.2 Constraints over Infinite Alphabets

Let us now turn to relations over infinite alphabets. Clearly, this is more difficult than the finite alphabets. The problem is that there are uncountably many relations over infinite alphabets, and hence we have a problem with representing them in a computer.

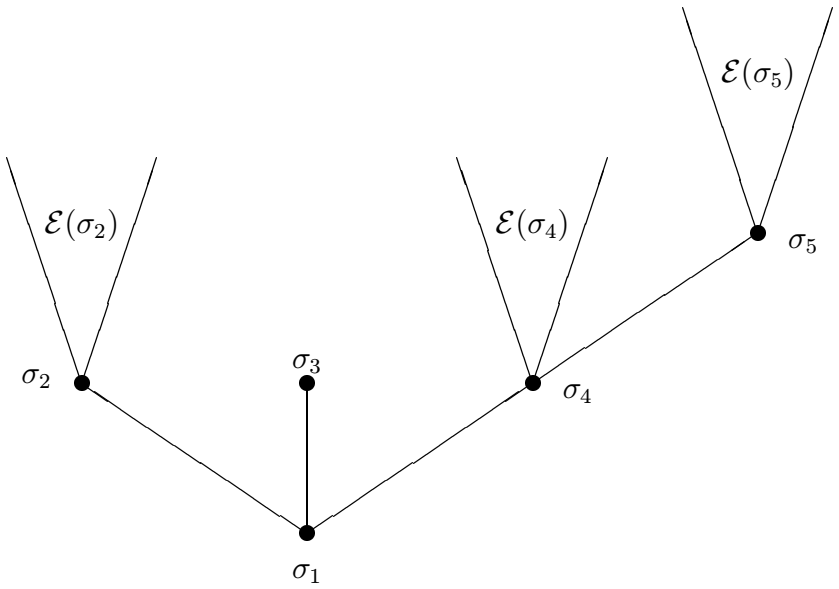


Figure 4.1: The lower semi-lattice generated by $\sqsubseteq$ and $\preceq$.

We will first consider how to represent subsets of infinite alphabets finitely. The set $\mathcal{P}(\Sigma_{\mathcal{E}})$ is uncountably infinite so we must decide which sets we want to represent. Two kinds of sets come immediately to mind: Singleton sets and cone sets. A singleton set is a set of the form $\{e\}$ and a cone set is a set of the form $\{e \in \mathcal{E}(\sigma) | e \geqslant e'\}$.

We want to find relations which preserve the kinds of sets described above — that is, if S is representable and R is such a relation, the set $\{e' | \exists e \in S : R(e, e')\}$ should also be representable and vice versa. An obvious (and necessary) choice is $=$. Another relation that preserves representability is $\preceq$. These two relations are sufficient for our purpose, and since the constraint sets $\{v = v'\}$ and $\{v \preceq v', v' \preceq v\}$ are equivalent we can concentrate our efforts on the $\preceq$ relation.

Definition 4.3.3 *Let $R \subseteq \Sigma^n$ and let Q be a function*

$$Q : R \to \mathcal{P}((\{i \preceq_\sigma j | i, j \in 1..n, \sigma \in \mathrm{dom}(\mathcal{E})\}))$$

such that if $i \preceq_\sigma j \in Q(\sigma_1, \ldots, \sigma_n)$ then $\sigma_i = \sigma_j = \sigma$. We define the relation $R_Q \subseteq {\Sigma_{\mathcal{E}}}^n$ as

> *$R_Q(\sigma_1, \ldots, \sigma_n)$, iff there are $\sigma'_1, \ldots, \sigma'_n$ and $e_1, \ldots, e_n$ such that the following hold*
>
> - $\sigma_i = \begin{cases} \sigma'_i[e_i] & \text{if } \sigma'_i \in \mathrm{dom}(\mathcal{E}) \\ \sigma'_i & \text{otherwise} \end{cases}$
> - $R(\sigma'_1, \ldots, \sigma'_n)$

- $\varphi \models Q(\sigma_1, \ldots, \sigma_n)$, *where* φ *is defined by* $\varphi(i) = e_i$.

We say that a relation over $\Sigma_{\mathcal{E}}$ *is* finitely definable, *iff it is of the form* R_Q *for some* R *and* Q *as above.*

If we write down the part of the definition of $\leqslant_{\mathsf{st}}$ that has to do with the root labels only we get the following relation.

$$
\begin{array}{l}
R^{\mathsf{st}}(\tau, \tau') \leftrightarrow \\
\quad (\mathsf{Int}, \mathsf{Int}); \\
\quad (\mathsf{Bool}, \mathsf{Bool}); \\
\quad (\rightarrow, \rightarrow); \\
\quad (\mathsf{Prod}[\alpha], \mathsf{Prod}[\beta]) \quad \text{if } \beta \subseteq \alpha; \\
\quad (\mathsf{Sum}[\alpha], \mathsf{Sum}[\beta]) \quad \text{if } \alpha \subseteq \beta; \\
\quad (\mathsf{List}, \mathsf{List}).
\end{array}
$$

In the notation of Definition 4.3.3, we have that $R^{\mathsf{st}} = R_Q$ where

$$
\begin{array}{rcl}
R & = & \{(\mathsf{Int}, \mathsf{Int}), (\mathsf{Bool}, \mathsf{Bool}), (\rightarrow, \rightarrow), \\
 & & \quad (\mathsf{Prod}, \mathsf{Prod}), (\mathsf{Sum}, \mathsf{Sum}), (\mathsf{List}, \mathsf{List})\} \\
Q(\mathsf{Prod}, \mathsf{Prod}) & = & \{2 \preceq_{\mathsf{Prod}} 1\} \\
Q(\mathsf{Sum}, \mathsf{Sum}) & = & \{1 \preceq_{\mathsf{Sum}} 2\}
\end{array}
$$

4.3.3 Relabeling Functions

Let us return to the definition of the ordering $\leqslant_{\mathsf{st}}$. From the root label rule (3) we get that $\mathsf{Prod}(a : \tau) \leqslant_{\mathsf{st}} \mathsf{Prod}(a : \tau, b : \tau')$ but $\mathsf{Prod}(b : \tau) \not\leqslant_{\mathsf{st}} \mathsf{Prod}(a : \tau, b : \tau')$ (if we assume that $\tau \not\leqslant_{\mathsf{st}} \tau'$). If we rewrite this with the notation from this section we get $\mathsf{Prod}[a](\tau) \leqslant_{\mathsf{st}} \mathsf{Prod}[a, b](\tau, \tau')$ and $\mathsf{Prod}[b](\tau) \not\leqslant_{\mathsf{st}} \mathsf{Prod}[a, b](\tau, \tau')$. That is, we have three labels σ_a, σ_b, σ_{ab}, such that $\sigma_a \leqslant \sigma_{ab}$ and $\sigma_b \leqslant \sigma_{ab}$, but $\sigma_a(\tau) \leqslant_{\mathsf{st}} \sigma_{ab}(\tau, \tau')$ and $\sigma_b(\tau) \not\leqslant_{\mathsf{st}} \sigma_{ab}(\tau, \tau')$. This is at odds with the goal of writing the rules so that the root labels are the significant factor.

The problem above is that τ is compared to different subtrees. In the definition of $\leqslant$ we lose this information — we cannot see that σ_a and σ_b are related to σ_{ab} in different ways. We can retain this information by defining *relabeling functions*, that shows which subtrees are supposed to be compared. The relabeling function from σ_a to σ_{ab} is defined by $\rho_{\sigma_a \leqslant \sigma_{ab}}(1) = 1$, and the one from σ_b to σ_{ab} by $\rho_{\sigma_b \leqslant \sigma_{ab}}(1) = 2$. That is, in the first case the first (and only) subtree of $\sigma_a(\tau)$ corresponds to the first subtree of $\sigma_{ab}(\tau_a, \tau_b)$, in the second case the first subtree of $\sigma_b(\tau)$ corresponds to the second subtree of $\sigma_{ab}(\tau_a, \tau_b)$.

Definition 4.3.4 *Let* Σ *be a (finite or infinite) ranked alphabet, and* $\leqslant$ *be a lower semi-lattice over* Σ. *A* relabeling function *is a partial function* $\rho : \Sigma^2 \to (\omega \to \omega)$, *such that*

1. $\mathrm{dom}(\rho) = \{(\sigma, \sigma') \in \Sigma^2 | \sigma \leqslant \sigma'\}$
2. *$\rho(\sigma, \sigma')$ is a total injective function* $\rho(\sigma, \sigma') : 1..\mathrm{rank}(\sigma) \to 1..\mathrm{rank}(\sigma')$
3. *(Reflexivity)* $\rho(\sigma, \sigma) = i \mapsto i$
4. *(Transitivity) If* $\sigma \leqslant \sigma'$ *and* $\sigma' \leqslant \sigma''$, $\rho(\sigma, \sigma'') = \rho(\sigma', \sigma'') \circ \rho(\sigma, \sigma')$

We write $\rho_{\sigma \leqslant \sigma'}$ for the function $\rho(\sigma, \sigma')$.

If the alphabet is an infinite alphabet of the form $\Sigma_{\mathcal{E}}$ and Σ and each $\mathcal{E}_\sigma$ are equipped with relabeling functions there is a unique relabeling function that extends these functions while keeping with the transitivity requirement from the definition above. Note that due to the totality and injectivity of $\rho_{\sigma \leq \sigma'}$ we must have that $\mathrm{rank}(\sigma) \leq \mathrm{rank}(\sigma')$. Hence there can be a relabeling function only if rank is monotone.

We assume that some relabeling function, ρ, is defined over $\mathcal{E}(\mathsf{Prod})$, such that $\rho_{\{a\} \subseteq \{a,b\}}(1) \neq \rho_{\{b\} \subseteq \{a,b\}}(1)$. Hence ρ defines an ordering between the elements in the set $\{a, b\}$. With this we can define our notation $\mathsf{Prod}[a_1, \ldots, a_n]$. Let α be a set of field names, $n = |\alpha|$, and let a_i be the (unique) element of α for which $\rho_{\{a_i\} \subseteq \alpha}(1) = i$. Then we write $\mathsf{Prod}[a_1, \ldots, a_n]$ for $\mathsf{Prod}[\{a_1, \ldots, a_n\}]$.

4.4 Conditional Constraints

The conditional constraints are the side conditions that are not on the form $\alpha \subseteq \beta$. The definition of the conditional constraints will be in three steps.

1. The definition of the expressions $\tau/1$ and $\tau'/2$ as the *pseudovariables* $\langle 1/1 \rangle$ and $\langle 2/2 \rangle$.
2. The definition of conditional constraints as constraints over pseudo-variables.
3. The definition of expressions of the form $\forall j \in \alpha \& \beta : \ldots$ as *quantified constraints*. The definition of this is divided into the syntax and semantics of the expressions.

Let us now turn to the formal definition of the constraints. The first step is to define the pseudovariables.

Definition 4.4.1 *Let $n \geq 1$ be an integer and let $\underline{k} = (k_1, \ldots, k_n)$ be a sequence of non-negative integers. A* pseudovariable *of rank n and $\underline{k}$ is an expression of the form $\langle i/\alpha \rangle$ where $1 \leq i \leq n$ and $\alpha \in \{\epsilon\} \cup 1..k_i$.*

We write $\Psi^n_{\underline{k}}$ for the set of pseudovariables of rank n and $\underline{k}$ and Ψ for the set of all pseudovariables.

Now, a conditional constraint can be defined as a constraint over the pseudovariables. That is, if the constraint alphabet is Γ the set of conditional constraints is $\Gamma[\Psi^n_{\underline{k}}, \emptyset]$, where n is the rank of the relation being defined and $\underline{k}$ is the sequence of ranks of the root labels. This is formalised in the following definition.

Definition 4.4.2 *Let Σ be a ranked alphabet, Γ a relation alphabet, and $R \subseteq \Sigma^n$ a relation. A function $\chi : R \to \mathcal{P}(\Gamma[\Psi, \emptyset])$ is a* condition function*, iff for each tuple $(\sigma_1, \ldots, \sigma_n) \in R$, we have that $\chi(\sigma_1, \ldots, \sigma_n) \subseteq \Gamma[\Psi^n_{\underline{k}}, \emptyset]$, where $\underline{k} = (\mathrm{rank}(\sigma_1), \ldots, \mathrm{rank}(\sigma_n))$.*

With this definition we have for instance that

$$\chi(\rightarrow, \rightarrow) = \{\langle 2/1\rangle \leqslant \langle 1/1\rangle, \langle 1/2\rangle \leqslant \langle 2/2\rangle\}$$

in the definition of $\leqslant_{\mathsf{st}}$.

Let us turn to the definition of the conditional constraint $\forall l \in \alpha\&\beta : \tau/l \leqslant_{\mathsf{st}} \tau'/l$. The α and β refers to the subtrees of the first and the second term respectively, and we shall write them as [1] and [2]. In this notation the expression becomes $\forall l \in [1]\&[2] : \langle 1/l\rangle \leqslant_{\mathsf{st}} \langle 2/l\rangle$. We call expressions of this form *quantified constraints*.

Definition 4.4.3 *A* quantified constraint *is an expression of the form*

$$\forall l_1 \in [p^1_1]\& \cdots \&[p^1_{m_1}], l_2 \in [p^2_1]\& \cdots \&[p^2_{m_2}], \ldots, l_n \in [p^n_1]\& \cdots \&[p^n_{m_n}] : C$$

where p^i_j, m_i and n are positive integers, $l_i \notin \omega \cup \{\epsilon\}$ are subtree variables*, and C is a conditional constraint over pseudovariables of the form $\langle i/\epsilon\rangle$, $\langle i/j\rangle$, or $\langle p^i_j/l_i\rangle$.*

What we want to do is to define the semantics of quantified constraint as a condition function. Such a condition function has the form $\chi : R_Q \to \mathcal{P}(\Gamma[\Psi, \emptyset])$, since the quantified constraints are meaningful only when dealing with infinite alphabets. For the quantified constraint $\forall l \in [1]\&[2] : \langle 1/l\rangle \leqslant_{\mathsf{st}} \langle 2/l\rangle$ we saw that the corresponding condition function is

$$\chi(\mathsf{Prod}[\alpha], \mathsf{Prod}[\beta]) = \{\langle 1/\rho_{\alpha\cap\beta\leq\alpha}(l)\rangle \leqslant_{\mathsf{st}} \langle 2/\rho_{\alpha\cap\beta\leq\beta}(l)\rangle | l \in 1..|\alpha \cap \beta|\}$$

This is what we need to generalise to all quantified constraints. The semantics of the quantified constraints is defined in the following.

Definition 4.4.4 *Let QC be the quantified constraint*

$$\forall l_1 \in [p^1_1]\& \cdots \&[p^1_{m_1}], l_2 \in [p^2_1]\& \cdots \&[p^2_{m_2}], \ldots, l_n \in [p^n_1]\& \cdots \&[p^n_{m_n}] : C$$

and $\underline{\sigma} = (\sigma_1, \ldots, \sigma_k) \in {\Sigma_\mathcal{E}}^k$ *a tuple such that there exist a generic label* σ' *and extensions* e_i *so that for all* p^i_j *we have that* $\sigma_{p^i_j} = \sigma'[e_{p^i_j}]$. *A conditional constraint* C' *is* defined *by* QC *and* $\underline{\sigma}$, *iff there is a substitution*

$$sub : \{\langle i/l_j\rangle | i \in 1..k, j \in 1..n\} \to \{\langle i/j\rangle | i \in 1..k, j \in 1..n\}$$

such that $C' = C sub$ *and the following holds:*

> *Let* $\eta_j = e_{p^j_1} \curlywedge_{\sigma'} \ldots \curlywedge_{\sigma'} e_{p^i_{m_i}}$, *and* $\tilde{e}^j_i = e_{p^j_i}$, *then for all* j *there is a* $k \in 1..|\eta_j|$, *such that for all* i, $sub(\langle i/l_j\rangle) = \langle i/\rho_{\eta_j \preceq_{\sigma'} \tilde{e}^j_i}(k)\rangle$.

We write $\mathrm{Def}(QC, \underline{\sigma})$ *for the set of constraints defined by* QC *and* $\underline{\sigma}$.

Let us return to the example

$$QC = \forall l \in [1]\&[2] : \langle 1/l\rangle \leqslant_{\mathsf{st}} \langle 2/l\rangle$$

Let $\underline{\sigma} = (\mathsf{Prod}[a, b], \mathsf{Prod}[b, c])$. Then by the definition $\sigma' = \mathsf{Prod}$, $e_1 = \{a, b\}$, and $e_2 = \{b, c\}$. The conditional constraint $\langle 1/2\rangle \leqslant_{\mathsf{st}} \langle 2/1\rangle$ is definable by QC and $\underline{\sigma}$.

To see this let *sub* be defined as $sub(\langle 1/l\rangle) = \langle 1/2\rangle$ and $sub(\langle 2/l\rangle) = \langle 2/1\rangle$. We have that $\eta_1 = \{a, b\} \cap \{b, c\} = \{b\}$. Let $k = 1$. Then $k \in 1..|\eta_1|$ and the following hold as requested:

- $sub(\langle 1/l\rangle) = \langle 1/\rho_{\{b\}\subseteq\{a,b\}}(k)\rangle$
- $sub(\langle 2/l\rangle) = \langle 2/\rho_{\{b\}\subseteq\{b,c\}}(k)\rangle$

Since $|\eta_1| = 1$, we have $k = 1$ as the only possible choice for k. So the above is the only conditional constraint defined by QC and $\underline{\sigma}$. Hence we have

$$\mathrm{Def}(QC, \underline{\sigma}) = \{\langle 1/2\rangle \leqslant_{\mathsf{st}} \langle 2/1\rangle\}$$

By similar reasoning we have

$$\mathrm{Def}(QC, (\mathsf{Prod}[a, b, c], \mathsf{Prod}[a, c, d])) = \{\langle 1/1\rangle \leqslant_{\mathsf{st}} \langle 2/1\rangle, \langle 1/3\rangle \leqslant_{\mathsf{st}} \langle 2/2\rangle\}$$

and so on.

In Definition 4.4.4 above $\mathrm{Def}(QC, (\mathsf{Prod}[\alpha], \mathsf{Sum}[\beta]))$ is not well-defined because there is no appropriate σ'. We will make sure that this cannot possibly happen by saying that a quantified constraint

$$\forall l_1 \in [p^1_1]\& \cdots \&[p^1_{m_1}], l_2 \in [p^2_1]\& \cdots \&[p^2_{m_2}], \ldots, l_n \in [p^n_1]\& \cdots \&[p^n_{m_n}] : C$$

is *well-defined* with respect to a tuple $(\sigma_1, \ldots, \sigma_k) \in \Sigma^k$, iff all the $\sigma_{p^i_j}$ are equal.

Now, an *extended condition function* is a condition function where $\chi(\underline{\sigma})$ contains both conditional constraints and quantified constraints which are well-defined with respect to $\underline{\sigma}$. This leads to the following definition.

Definition 4.4.5 *Let $R_Q \subseteq {\Sigma_{\mathcal{E}}}^n$ be a finitely definable relation and χ an extended condition function with domain R. The condition function $\chi_Q : R_Q \to \mathcal{P}(\Gamma[\Psi, \emptyset])$ is defined by the following. Let $\underline{\sigma} = (\sigma_1, \ldots, \sigma_n)$ and let σ_i' and e_i be such that $\sigma_i = \sigma_i'[e_i]$, if $\sigma_i' \in \mathrm{dom}(\mathcal{E})$ and $\sigma_i = \sigma_i'$, otherwise. Then we have that*

$$\chi_Q(\underline{\sigma}) = (\chi(\underline{\sigma}') \cap \Gamma[\Psi, \emptyset]) \cup \bigcup_{QC \in \chi(\underline{\sigma}') \setminus \Gamma[\Psi,\emptyset]} \mathrm{Def}(QC, \underline{\sigma})$$

If χ_Q is like above we say that χ_Q is uniform.

4.5 User-defined Constraints

Now we have that the user-defined constraints are the ones that can be written as a relation of the form R_Q over the root labels plus some conditional constraints defined by a condition function of the form χ_Q.

Definition 4.5.1 *Let Σ be a ranked alphabet. The alphabet of user-definable relations is the smallest set of relations, Γ^Σ, such that*

- *$\langle\!\langle R_Q, \chi_Q \rangle\!\rangle \in \Gamma_\Sigma$ with rank n, if $R_Q \subseteq \Sigma^n$ is finitely definable and*

  $$\chi_Q : R_Q \to \mathcal{P}((\Gamma^\Sigma \cup \{=\})[\Psi, \emptyset])$$

  is a uniform condition function, where $\mathrm{rank}(=) = 2$.

- *$\nu X.\langle\!\langle R_Q, \chi_Q \rangle\!\rangle \in \Gamma_\Sigma$ with rank n, if $R_Q \subseteq \Sigma^n$ is finitely definable and*

  $$\chi_Q : R \to \mathcal{P}((\Gamma^\Sigma \cup \{X, =\})[\Psi, \emptyset])$$

  is a uniform condition function, where $\mathrm{rank}(=) = 2$ *and* $\mathrm{rank}(X) = n$.

The two kinds of constraints defined here are the simple user-defined constraint and the recursive user-defined constraint. They are possibly conditional on previously defined constraints or equality constraints. That the constraints will have to be defined *previously* is ensured by the requirement that the constraint alphabet is the smallest possible such set. The 'ν' in the definition of the recursive constraints suggests that they are defined as the greatest fixpoint of a function. And indeed this is the case as the following definition shows.

Definition 4.5.2 *Let Σ be a ranked alphabet. The interpretation*

$$\mathcal{I}^\Sigma : \Gamma^\Sigma \cup \{=\} \to \bigcup_{n=0}^{\infty} \mathcal{P}({\mathsf{T}_\Sigma}^n)$$

is defined inductively as follows.

- $\mathcal{I}^{\Sigma}(=) = \{(t,t) | t \in \mathsf{T}_{\Sigma}\}$
- $\mathcal{I}^{\Sigma}(\langle\!\langle R_Q, \chi_Q \rangle\!\rangle) = \{\underline{t} \in {\mathsf{T}_{\Sigma}}^n | R_Q(\underline{t}(\epsilon)), \varphi_{\underline{t}} \models_{\mathcal{I}^{\Sigma}} \chi_Q(\underline{t}(\epsilon)\}$, *where* $\varphi_{\underline{t}}$ *is the assignment defined as* $\varphi_{\underline{t}}(\langle i/\alpha \rangle) = t_i/\alpha$ *and* $\underline{t}(\epsilon) = (t_1(\epsilon), \ldots, t_n(\epsilon))$.
- $\mathcal{I}^{\Sigma}(\nu X.\langle\!\langle R_Q, \chi_Q \rangle\!\rangle)$ *is the* largest *relation* $S \subseteq {\mathsf{T}_{\Sigma}}^n$ *such that* $S(t_1, \ldots, t_n)$, *iff the following hold.*
  - $R_Q(t_1(\epsilon), \ldots, t_n(\epsilon))$
  - $\varphi_{\underline{t}} \models_{\mathcal{I}^{\Sigma}[X \leftarrow S]} \chi_Q((t_1(\epsilon), \ldots, t_n(\epsilon))$ *where* $\varphi_{\underline{t}}$ *is the assignment defined as* $\varphi_{\underline{t}}(\langle i/\alpha \rangle) = t_i/\alpha$.

We say that the conditional constraints from $\chi_Q(t_1(\epsilon), \ldots, t_n(\epsilon))$ are *forced*.

As an example let Ac' be defined as in Section 4.2 and let χ_{Ac} be defined as

$$\chi_{Ac}(\sigma, \sigma') = \begin{cases} \{Tc(\langle 1/1 \rangle, \langle 2/1 \rangle)\} & \text{if } \sigma = \sigma' = \mathsf{Set} \\ \{\langle 1/1 \rangle = \langle 2/1 \rangle\} & \text{if } \sigma = \sigma' = \wedge \\ \emptyset & \text{otherwise} \end{cases}$$

Then $\mathcal{I}^{\Sigma^{\mathsf{TP}}}(\langle\!\langle Ac', \chi_{Ac} \rangle\!\rangle)$ is the relation Ac.

Similarly, we can define $\leqslant_{\mathsf{st}}$ using the relation $R_{\mathsf{st}} = R_Q$ defined in Section 4.3 and the function $\chi_{\mathsf{st}} = \chi_Q$ where

$$\chi(\sigma, \sigma') = \begin{cases} \{X(\langle 2/1 \rangle, \langle 1/1 \rangle), X(\langle 2/1 \rangle, \langle 2/2 \rangle)\} & \text{if } \sigma = \sigma' = \to \\ \{\forall j \in [1]\&[2] : X(\langle 1/j \rangle, \langle 2/j \rangle)\} & \text{if } \sigma = \sigma' = \mathsf{Prod} \\ \{\forall j \in [1]\&[2] : X(\langle 1/j \rangle, \langle 2/j \rangle)\} & \text{if } \sigma = \sigma' = \mathsf{Sum} \\ \{X(\langle 1/1 \rangle, \langle 2/1 \rangle)\} & \text{if } \sigma = \sigma' = \mathsf{List} \\ \emptyset & \text{otherwise} \end{cases}$$

We have that $\leqslant_{\mathsf{st}}$ is $\mathcal{I}^{\Sigma^{\mathsf{st}}}(\nu X.\langle\!\langle R_{\mathsf{st}}, \chi_{\mathsf{st}} \rangle\!\rangle)$ *restricted to* $\mathsf{Fin}_{\Sigma^{\mathsf{st}}}$. Actually, the relation $\leqslant_{\mathsf{st}}$ is not definable with these constraints. Only the extension to *regular* trees is definable. The way to remedy this situation when we have only $=$ (e.g. in [60]) is to analyse the structure of the constraint set and use the knowledge from Proposition 1.2.1 to determine in advance whether the constraint set has finite solutions. Here we can use the same technique to restrict our set of solutions to finite solutions. Hence we shall also allow the special case where the types are finite.

In contrast to the above relations the unifiability relation, $\approx$, is not definable in any form. The reason for this is its non-local properties. For example, we have that $\alpha \approx \mathsf{Int}$ and $\alpha \approx \mathsf{Real}$, but $\alpha \to \alpha \not\approx \mathsf{Int} \to \mathsf{Real}$.

In the next section we shall see that assignment compatibility is stable, whereas $\leqslant_{\mathsf{st}}$ is not.

4.6 Stability of user-defined constraints

The stability requirement on the constraint satisfaction problem assures that we can choose the least possible solution and thus avoid backtracking (see Section 6.1). The idea is that by choosing the smallest solution we cannot do too much — it is always possible to choose a greater solution if need be. Extended to the whole of Γ^{Σ} this requirement says that the conditionals should be stable under the choice of a greater solution. That is, we will force no conditional constraints which we will later regret. Additionally, we will need to know that there is an end to the forcing of constraints. That is, we need to be sure that the recursive constraints do not lead to infinite solutions (which is reserved for the equality constraints). In conclusion we have that the condition function should comply with the following two requirements:

1. The condition function must be *monotone*.
2. The condition function must be *strict*.

The main point of this section is to define the concepts of monotone and strict with respect to condition functions and use these to define stability, which is then used to define the OIH constraint domain.

As a starting point we need to look at the orderings that apply in the definition of stability. The monotonicity of condition functions was needed to ensure that no choice will be regretted. For the same reason we demand that the rank and $|\cdot|$ functions are monotone (in the usual sense). I addition to this the ordering should comply with the definitions in Section 4.3. Hence the following definition.

Definition 4.6.1 *Let $\Sigma = \Sigma'_{\mathcal{E}}$ be a ranked alphabet, equipped with a partial ordering $\leqslant = \sqsubseteq_{\preceq}$. The ordering $(\Sigma, \leqslant)$ is* applicable*, iff the following hold:*

- *$(\Sigma', \sqsubseteq)$ forms a lower semi-lattice and $(\mathcal{E}(\sigma), \preceq_\sigma)$ form lattices, where that all non-empty subsets of $\mathcal{E}(\sigma)$ have greatest lower bounds.*
- rank *and the $|\cdot|_\sigma$ functions are all monotone.*
- $\mathrm{rank}(\bigwedge \Sigma') = 0$ *and for each $\sigma \in \mathrm{dom}(\mathcal{E})$: $|\bot_{\mathcal{E}(\sigma)}| = 0$.*

In addition to this there is a requirement for the relabeling functions over the extensions of the alphabets. That is, the relabeling functions on the form $\rho_{e \preceq_\sigma e'}$. The requirements ensures that the &s in the quantified constraints actually represents $\cap$. The definition follows.

Definition 4.6.2 *Let $(\mathcal{E}(\sigma), \preceq_\sigma)$ be a lattice. We say that $\rho : (\mathcal{E}(\sigma))^2 \to (\omega \to \omega)$ is a* smooth *relabeling function, iff for each $e, e' \in \mathcal{E}(\sigma)$ we have that*

$$\mathrm{codom}(\rho_{e \curlywedge e' \preceq e \curlyvee e'}) = \mathrm{codom}(\rho_{e \preceq e \curlyvee e'}) \cap \mathrm{codom}(\rho_{e' \preceq e \curlyvee e'})$$

Since $\mathrm{codom}(\rho_{e \curlywedge e' \preceq e \curlyvee e'}) \subseteq \mathrm{codom}(\rho_{e \preceq e \curlyvee e'}) \cap \mathrm{codom}(\rho_{e' \preceq e \curlyvee e'})$ follows directly from the transitivity rule of Definition 4.3.4, all the requirement says is that the rank should not increase too quickly.

Let us now turn to the requirements for the conditional constraints. The first requirement was that conditional constraints for the smaller solutions are more general than the ones for greater solutions. In order to compare conditional constraints we will have to consider the relabelings. We introduce the following notation: Let $\underline{\sigma} = (\sigma_1, \ldots, \sigma_n)$ and $\underline{\sigma'} = (\sigma'_1, \ldots, \sigma'_n)$, and assume that $\underline{\sigma} \leqslant \underline{\sigma'}$. We write $\rho_{\underline{\sigma} \leqslant \underline{\sigma'}}$ for the tuple $(\rho_{\sigma_1 \leqslant \sigma'_1}, \ldots, \rho_{\sigma_n \leqslant \sigma'_n})$. If $\rho = (\rho_1, \ldots, \rho_n)$ is a tuple of relabeling functions, and $C \in \Gamma[\Psi^n_{\underline{k}}]$ we write $\rho(C)$ for the conditional constraint $C[\langle i/j \rangle \leftarrow \langle i/\rho_i(j) \rangle]$, provided the $\rho_i(j)$ are well-defined. If $\mathcal{C}$ is a set of conditional constraints, then

$$\rho(\mathcal{C}) = \{C | \exists C' \in \mathcal{C} : \rho(C') = C\}$$

With this in mind we can write the following definition.

Definition 4.6.3 *Let Σ be a ranked alphabet equipped with a lower semi-lattice, $\leqslant$, and a smooth relabeling function ρ. We say that a condition function $\chi : \Sigma^n \to \mathcal{P}(\Gamma[\Psi, \emptyset])$ is* monotone*, iff*

$$\forall \underline{\sigma}, \underline{\sigma'} \in \mathrm{dom}(\chi) : \underline{\sigma} \leqslant \underline{\sigma'} \Rightarrow \rho_{\underline{\sigma} \leqslant \underline{\sigma'}}(\chi(\underline{\sigma})) \subseteq \chi(\underline{\sigma'})$$

If χ is monotone, no conditions from smaller solutions can violate conditions from greater solutions.

The termination requirements says that there are limits on how we can force the conditional constraints. More precisely, it says that if we fix a number of variables in the constraint $R(v_1, \ldots, v_n)$ and let the rest be completely free we cannot force a conditional constraint (unless we fix *all* the variables).

Definition 4.6.4 *Let Σ be a ranked alphabet equipped with a lower semi-lattice, $\leqslant$, and let $R \subseteq \Sigma^n$ be a relation. We say that a condition function $\chi : R \to \Gamma[\Psi, \emptyset]$ is* strict*, iff*

$$\forall S \subsetneq 1..n,\ \sigma{:}S \to \Sigma : \chi(\textstyle\bigwedge(R \cap (\prod_{i \in S}\{\sigma(i)\} \times \prod_{i \notin S} \Sigma))) = \emptyset$$

In the definition above the expression $\prod_{i \in S}\{\sigma(i)\} \times \prod_{i \notin S} \Sigma$ should be read such that we rearrange the tuples so that the ith value gets to the ith place.

This is all the requirements we need to define the stability of a relation from Γ^Σ.

Definition 4.6.5 *Let $(\Sigma, \leqslant)$ equipped with the smooth relabeling function ρ be applicable, and let $\Sigma = \Sigma'_{\mathcal{E}}$ and $\leqslant = \sqsubseteq_{\preceq}$. The $\leqslant$-stable relations from Γ^Σ are defined recursively as follows.*

- $\langle\!\langle R_Q, \chi_Q \rangle\!\rangle$ *is* $\leqslant$*-stable, iff* R *is* $\sqsubseteq$*-stable, for every* $\underline{\sigma} \in R_Q$ *the relations in* $\chi_Q(\underline{\sigma})$ *— apart from* $=$ *— are* $\leqslant$*-stable, and* Q *and* χ_Q *are monotone.*
- $\nu X.\langle\!\langle R_Q, \chi_Q \rangle\!\rangle$ *is* $\leqslant$*-stable, iff* R *is* $\sqsubseteq$*-stable, for every* $\underline{\sigma} \in R$ *the relations in* $\chi_Q(\underline{\sigma})$ *— apart from* $=$ *and* X *— are* $\leqslant$*-stable,* Q *is monotone and* χ_Q *is monotone and strict.*

We shall say about a relation $R \subseteq \mathsf{Reg}_\Sigma{}^n$ that it is $\leqslant$-stable, iff there is a $\leqslant$-stable relation symbol $R' \in \Gamma^\Sigma$ such that $R = \mathcal{I}^\Sigma(R')$.

Let us return to the examples that motivated these definitions — the relations Ac and $\leqslant_{\mathsf{st}}$. In Section 4.5, we saw the following.

$$\leqslant_{\mathsf{st}} = \mathcal{I}^{\Sigma^{\mathsf{st}}}(\nu X.\langle\!\langle R_{\mathsf{st}}, \chi_{\mathsf{st}} \rangle\!\rangle) \qquad Ac = \mathcal{I}^{\Sigma^{\mathsf{TP}}}(\langle\!\langle Ac', \chi_{Ac} \rangle\!\rangle)$$

The question is whether these relations are stable.

Let us first look at the relation Ac. With respect to the ordering

$$\mathsf{Integer} \leqslant \mathsf{Real} \leqslant \mathsf{Set} \leqslant {}^\wedge$$

it is unstable since $\chi_{Ac}(\mathsf{Set}, \mathsf{Set}) \not\subseteq \chi_{Ac}({}^\wedge, {}^\wedge)$. Instead we can use the ordering

- $\mathsf{Real} \leqslant \mathsf{Integer}$
- $\mathsf{Real} \leqslant \mathsf{Set}$
- $\mathsf{Real} \leqslant {}^\wedge$

Now, the monotonicity is restored and the other requirements hold as well.

The situation is a bit more problematic for $\leqslant_{\mathsf{st}}$. It is not stable for any ordering if the field sets are ordered by set inclusion. The problem is with the strictness of χ_{st}. To see this observe the label $\mathsf{Prod}[a]$. The set $R_{\mathsf{st}} \cap (\{\mathsf{Prod}[a]\} \times \Sigma^{\mathsf{st}})$ is $\{(\mathsf{Prod}[a], \mathsf{Prod}[\alpha]) | a \in \alpha\}$. The least member of this set is $(\mathsf{Prod}[a], \mathsf{Prod}[a])$ and

$$\chi_{\mathsf{st}}(\mathsf{Prod}[a], \mathsf{Prod}[a]) = \{X(\langle 1/1 \rangle, \langle 1/2 \rangle)\} \neq \emptyset$$

There is no obvious way to remedy this situation, but in Section 8.3.1 we shall look at an extended type system which allow stable constraints.

Another relation with the same problem is $=$. The definition of $=$ would be $\nu X.\langle\!\langle R, \chi \rangle\!\rangle$, where

$$R = \{(\sigma, \sigma) | \sigma \in \Sigma\}$$
$$\chi(\sigma, \sigma) = \{X(\langle 1/i \rangle, \langle 2/i \rangle) | i \in 1..\mathrm{rank}(\sigma)\}$$

Since $R \cap (\{\sigma\} \times \Sigma) = \{(\sigma, \sigma)\}$ and $\chi(\sigma, \sigma) \neq \emptyset$ if $\mathrm{rank}(\sigma) > 0$, χ is not strict — regardless of the ordering — and $=$ is thus never a stable relation.

4.7 The OIH Constraint Domain

Now let us define the constraint domains. The relations they define are simply the stable subset of the user-defined relations plus $=$.

Definition 4.7.1 *Let $(\Sigma, \leqslant)$ be applicable. The* ordered constraint alphabet *over $(\Sigma, \leqslant)$ is*

$$\Gamma^{\Sigma}_{\leqslant} = \{R \in \Gamma^{\Sigma} | R \textit{ is } \leqslant\textit{-stable}\} \cup \{=\}$$

Now we can define the ordered finitary and infinitary Herbrand domains.

Definition 4.7.2 *Let $(\Sigma, \leqslant)$ be applicable and equipped with a smooth relabeling function.*

1. *The* Ordered Infinitary Herbrand Universe *over $(\Sigma, \leqslant)$ is the constraint domain*

   $$\mathsf{OIH}_{(\Sigma,\leqslant)} = (\mathsf{Reg}_{\Sigma}, \Gamma^{\Sigma}_{\leqslant}, \Sigma, \mathcal{I}^{\Sigma}, \mathcal{J}^{\mathsf{OIH}})$$

   where $\mathcal{J}^{\mathsf{OIH}}(\sigma) = \sigma^{\mathsf{T}_{\Sigma}}$

2. *The* Ordered Herbrand Universe *over $(\Sigma, \leqslant)$ is the constraint domain*

   $$\mathsf{OH}_{(\Sigma,\leqslant)} = (\mathsf{Fin}_{\Sigma}, \Gamma^{\Sigma}_{\leqslant}, \Sigma, \mathcal{I}^{\Sigma}_{\mathsf{Fin}}, \mathcal{J}^{\mathsf{OH}})$$

   where

   - $\mathcal{I}^{\Sigma}_{\mathsf{Fin}}(R) = \mathcal{I}^{\Sigma}(R) \cap \mathsf{Fin}_{\Sigma}{}^{n}$ *where* $R \in \Gamma^{\mathsf{OH}}_{n}$.
   - $\mathcal{J}^{\mathsf{OH}}(\sigma) = \sigma^{\mathsf{T}_{\Sigma}}$

Since the domains are *ordered* universes we would expect it to have an ordering. This ordering is defined below.

Definition 4.7.3 *Let $(\Sigma, \leqslant)$ be applicable. The monotone extension of $\leqslant$, $\leqslant_{\Sigma}$, is defined as the largest relation such that $\sigma(t_1, \ldots, t_n) \leqslant_{\Sigma} \sigma'(t'_1, \ldots, t'_m)$, iff one of the following hold*

- $\sigma < \sigma'$
- $\sigma = \sigma'$ *and for each* $i \in 1..\mathrm{rank}(\sigma)$, *we have that* $t_i \leqslant_{\Sigma} t'_i$.

There is an alternative definition of $\leqslant_{\Sigma}$.

Lemma 4.7.1 *Let $(\Sigma, \leqslant)$ be applicable. We have that*

$$\forall t, t' \in \mathsf{Reg}_{\Sigma} : t \leqslant_{\Sigma} t' \Leftrightarrow \mathrm{dom}(t) \subseteq \mathrm{dom}(t') \wedge \forall \alpha \in \mathrm{dom}(t) : t(\alpha) \leqslant t'(\alpha)$$

Proof: Follows from Definition 4.7.3. □

We can use this to show that $\leqslant_\Sigma$ has the following property.

Proposition 4.7.2 $(\mathsf{Reg}_\Sigma, \leqslant_\Sigma)$ *and* $(\mathsf{Fin}_\Sigma, \leqslant_\Sigma)$ *form lower semi-lattices.*

Proof: Let $t, t' \in \mathsf{Reg}_\Sigma$. We define $t'' \in \mathsf{Reg}_\Sigma$ by

$$\begin{array}{rcl} \mathrm{dom}(t'') & = & \{i_1 i_2 \ldots i_n \in \mathrm{dom}(t) \cap \mathrm{dom}(t') | \\ & & \quad \forall j \in 1..n-1 : i_j \leq \mathrm{rank}(t(i_1 i_2 \ldots i_{j-1}) \wedge t'(i_1 i_2 \ldots i_{j-1}))\} \\ t''(\alpha) & = & t(\alpha) \wedge t'(\alpha) \end{array}$$

That t'' is regular follows from the regularity of t and t', and that t'' is the greatest lower bound of t and t' follows from Lemma 4.7.1.

It is clear that t'' is finite whenever t and t' are finite, so the proof above extends to $(\mathsf{Fin}_\Sigma, \leqslant_\Sigma)$ as well. □

It doesn't follow from the above that $(\mathsf{Reg}_\Sigma, \leqslant_\Sigma)$ and $(\mathsf{Fin}_\Sigma, \leqslant_\Sigma)$ form *complete* lower semi-lattices. Hence, we cannot use the result to prove that all satisfiable constraint sets has a *least* solution. Instead we give a constraint solver for the domains in Chapter 6. This will constitute a constructive proof of the following theorem.

Theorem 4 *Any satisfiable set of* $\mathsf{OIH}_{(\Sigma,\leqslant)}$*-constraints has a* $\leqslant_\Sigma$*-least solution.*

Proof: See Chapter 6. □

It is important for this work that the OIH-domain is sound for constraint logic programming purposes. The first step in proving this is the following lemma.

Lemma 4.7.3 *Let* $\{e_1 = e'_1, \ldots, e_n = e'_n\}$ *be a finite set of term equations over* Σ *and* $\mathcal{V}$*. There is a constraint* $C \in \Gamma^\Sigma_{\leqslant}[\mathcal{V}, \Sigma]$*, such that* $\varphi \models C$ *iff* φ *is a solution to* $\{e_1 = e'_1, \ldots, e_n = e'_n\}$*.*

Proof: Let the relation $\langle\!\langle R, \chi \rangle\!\rangle \in \Gamma^\Sigma_{\leqslant}$ be defined by the following.

$$\begin{array}{l} R = \{(\sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_n) \in \Sigma^{2n} | \forall i \in 1..n : \sigma_i = \sigma'_i\} \\ \chi(\sigma_1, \ldots, \sigma_n, \sigma'_1, \ldots, \sigma'_n) = \{\langle i/\epsilon \rangle = \langle i+n/\epsilon \rangle | i \in 1..n\} \end{array}$$

The constraint we are looking for is $\langle\!\langle R, \chi \rangle\!\rangle(e_1, \ldots, e_n, e'_1, \ldots, e'_n)$. □

This leads to the main theorem of this chapter.

Theorem 5 *If* $(\Sigma, \leqslant)$ *is applicable,* $\mathsf{OIH}_{(\Sigma,\leqslant)}$ *and* $\mathsf{OH}_{(\Sigma,\leqslant)}$ *are solution-compact.*

Proof: It follows from Proposition 1.2.1 that every element in Reg_Σ as well as Fin_Σ is uniquely definable by a finite set of term equations.

Now, assume that $\varphi \not\models C$. From Proposition 1.2.1 we have a finite set of term equations, $E = \{e_1 = e'_1, \ldots, e_n = e'_n\}$, such that φ is the only solution to E. From Lemma 4.7.3 we get a constraint C', so that φ is the only satisfying assignment to C'. Hence $\varphi \models C$ and $\{C, C'\}$ is unsatisfiable. □

In fact, the solution-compactness requirement is one of the reasons that the universe of OIH is Reg_Σ rather than T_Σ, since in the latter case the needs would arise for non-uniform relations if the domain should remain solution-compact. The restriction has no practical consequences, since no irregular term is finitely definable.

Chapter 5

Stack persistent data structures

The need for backtracking puts an algorithmical challenge to the construction of the data structure. It is no longer sufficient to have an incremental algorithm because of the need to remove constraints from the constraint set. In this chapter we will show how to solve this problem by using *stack persistence*.

Having defined stack persistence in this chapter, it suffices to find an incremental constraint solver in Chapter 6.

5.1 partial, full and stack persistence

A persistent data structure provides access to the older versions of the data structure. Usually, two types of persistence are considered:

Partial persistence. A partially persistent data structure only allows retrieval of values from older versions of the data structure. That is, it is possible to *read* the past but not to *alter* the past.

Full persistence. In a fully persistent data structure it is allowed not only to *read* the past but also to *create* an *alternative history*. That is, it is possible to go back in time and pursue a different course of events. A fully persistent data structure provides access to all the created histories.

More precisely, let us assume that an ephemeral (ie. non-persistent) data structure, D, has two operations: $\mathsf{query}(D, x)$ and $\mathsf{update}(D, x)$. The partially persistent version of this data structure will have the following operations:

- $\mathsf{query}_{PP}(D_{PP}, x, v)$: Give the result of the call $\mathsf{query}(D_v, x)$ where D_v is version number v of the data structure.

– $\mathsf{update}_{PP}(D_{PP}, x)$: Let D be the current version of the data structure, and assume D is version number v. Make a new version, D', which is like D and has version number $v + 1$ and call $\mathsf{update}(D', x)$. Make D' current.

The fully persistent version has these operations:

– $\mathsf{query}_{FP}(D_{FP}, x, v)$: Give the result of the call $\mathsf{query}(D_v, x)$ where D_v is version number v of the data structure.

– $\mathsf{update}_{FP}(D_{FP}, x, v)$: Make a new version D' which is like D_v and call $\mathsf{update}(D', x)$. The version number of D' is the smallest unused version number.

Partial persistence is not sufficient for backtracking since it does not allow for alternative actions. On the other hand, full persistence does too much when it provides access to all the histories. There is no need to access a history that leads to failure.

Let us instead introduce a third possibility, stack persistence, which is perfectly suited for backtracking:

Stack persistence. In a stack persistent data structure it is allowed to restore the data structure to an older version. That is, we can access the past by returning to it. The part of the history that lies ahead of the new current time is lost.

Given the ephemeral data structure described above the stack persistent version provides these three operations:

– $\mathsf{query}_{SP}(x)$: Give the result of the call $\mathsf{query}(D, x)$ where D is the current version.

– $\mathsf{update}_{SP}(x)$: Make a new version, D', which is like the current version and call $\mathsf{update}(D', x)$. Make D' current. If D has version number v, D' has version number $v + 1$.

– $\mathsf{backtrack}_{SP}$: Assume that version number v is current. Discard it and make version $v - 1$ current.

Note that stack persistence is not a generalisation of partial persistence. There is no way to access older versions without destroying the current version and intermediate versions.

Obviously, we can use full persistence to implement stack persistence. The distinction between full and stack persistence only becomes important because of an efficient way of implementing stack persistence on a RAM.

5.2 Stack persistence on a RAM

In the following we shall assume that we are given an ephemeral data structure of size n, and that we want to make t operations on the persistent version of the data structure. For simplicity we shall assume $n \leq t$ — that is, all cells are being used.

In [18] it is shown how to obtain partially and fully persistent versions of a linked data structure. Both versions uses $\mathrm{O}(t)$ space. They run in time $O(t)$ if each node in the data structure has a constant, bounded in-degree — $\mathrm{O}(t \log t)$ otherwise.

On a RAM partial persistency can be obtained with the use of van Emde Boas trees [44]. The running time thus obtained is $\mathrm{O}(t \log \log t)$ and the space usage is $\mathrm{O}(t)$.

We will show how to obtain stack persistence on a RAM in time and space $\mathrm{O}(t)$. We do this by first considering a RAM as an ephemeral data structure and showing how a stack persistent version can be obtained. Later, we will show how to make a general data structure stack persistent in an efficient way.

A RAM of size n is a data structure with two operations:

- $\mathsf{query}_{\mathrm{RAM}}(i)$: If $i \in 1..n$ this returns the current value of cell number i.
- $\mathsf{update}_{\mathrm{RAM}}(i, x)$. If $i \in 1..n$ this sets the current value of cell number i to x.

We implement the stack persistent RAM by keeping all values from the different versions in stacks. A stack has the following three operations:

- $\mathsf{push}(S, x)$: Puts x on top of S.
- $\mathsf{pop}(S)$: Removes the top element of S.
- $\mathsf{top}(S)$: The top element of S.

Basically, what we want to do is to store for each cell a stack of the values it has taken together with the numbers of the versions in which it first took those values. Then all the backtrack operation needs to do is to count down the version number by 1. For efficiency reasons we use *lazy removal*. That is, we postpone the removal of obsolete values from the stack until we call query or update.

This tactic introduces a problem. Consider the following calls to the stack persistent data structure:

update(1,0)
update(1,3)
backtrack
update(2,2)

Now what should the value of lookup(1) be? Clearly, it should be 0. But — assuming that the first update was in version number 1 — the current version number is 2, and the call to update(1,3) also took place in version number 2. The naive solution would thus return the value 3 as lookup(1).

What the above example shows is that we can have different versions with the same version number. In order to remedy this situation we give each version a time-stamp as well as a number. Thus the versions in the example above would be (1,1), (2,2), and (2,3), respectively.

We now have a data structure with three arrays of stacks:

- *content*[1..n]: The values of the cells.
- *version*[1..n]: The version numbers of the values.
- *timestamp*[1..n]: The time-stamp of the values.

In addition we have an array, *latest*[1..t] which contains for each version number the time stamp of the latest version with that number, and integers *current* and *time* containing the current version number and time, respectively.

The implementation of the three stack persistent operations of the data structure is as follows.

```
update_{RAM,SP}(i, x):
    while top(version[i]) > current ∨
          top(timestamp[i]) ≠ latest[top(version[i])] do
        pop(content[i]);
        pop(version[i]);
        pop(timestamp[i]);
    end;
    current := current + 1;
    time := time + 1;
    latest[current] := time;
    push(version[i], current);
    push(timestamp[i], time);
    push(content[i], x);

lookup_{RAM,SP}(i):
    while top(version[i]) > current ∨
          top(timestamp[i]) ≠ latest[top(version[i])] do
        pop(content[i]);
        pop(version[i]);
        pop(timestamp[i]);
    end;
    return top(content[i]);
```

```
backup(k):
    current := current − k;
```

Note the change in backup where it is allowed to backup k steps at a time. That is, we go from version number v to version number $v - k$ rather than just $v - 1$. All the intermediate versions are discarded.

Clearly each stack will be no larger than the total number of calls to update on the corresponding cell. So the total number of stack heights is never more than $3t$ and hence the total space used is at most $4t + o(t)$. To show that the total number of stack operations is $O(t)$ we use the potential function technique of Tarjan [72].

Let $\Phi = \sum_{i=1}^{n} 3 * |content(i)|$. The number of stack operations in a call to $\mathsf{update}_{\mathrm{RAM},SP}$ is $3k+3$ where k is the number of obsolete values on the stack. The change in the potential function is $3-3k$ and thus the amortised number of stack operations is 6. Similarly, we get that the amortised number of stack operations in a call to $\mathsf{lookup}_{\mathrm{RAM},SP}$ is 1. It is clear that the number of stack operations dominate the total running time, and so we get the following proposition.

Proposition 5.2.1 *The amortised complexity of the procedures* $\mathsf{update}_{\mathrm{RAM},SP}$, $\mathsf{lookup}_{\mathrm{RAM},SP}$, *and* backtrack *is* $O(1)$.

While the amortised time is asymptotically optimal, it may still be the case that the on-line time can be quite high. It is a special problem that the above implementation makes a new version for each call to the update function. If we are to use the persistent RAM data structure in the implementation of a more complicated data structure we would want a better control of the version numbers. Hence the following version:

```
update_RAM,SP(i, x):
    while top(version[i]) > current ∨
          top(timestamp[i]) ≠ latest[top(version[i])] do
        pop(content[i]);
        pop(version[i]);
        pop(timestamp[i]);
    end;
    if top(version[i]) = current ∧ top(timestamp[i]) = latest[top(version[i])] then
        top(version[i]) := current;
        top(timestamp[i]) := time;
        top(content[i]) := x;
    else
        push(version[i], current);
        push(timestamp[i], time);
```

```
        push(content[i], x);
    endif

lookup_{RAM,SP}(i):
    while top(version[i]) > current ∨
          top(timestamp[i]) ≠ latest[top(version[i])] do
        pop(content[i]);
        pop(version[i]);
        pop(timestamp[i]);
    end;
    return top(content[i]);

backup(k):
    current := current − k;

advance:
    current := current + 1;
    time := time + 1;
    latest[current] := time;
```

In the version above the version number is no longer automatically incremented with each update. Instead we have introduced a new procedure, advance, which makes a new version with the version number $current + 1$ and makes the new version current.

Let us now turn to the general problem: Given an ephemeral data structure, how do we make a stack persistent version of this data structure? Assuming that the data structure implements the operations update and query the implementation is as follows:

update$_{SP}(D, x)$:
Call advance;
Call update(D, x) where all calls to update$_{\mathrm{RAM}}(i, v)$ are replaced with update$_{\mathrm{RAM},SP}(i, v)$ and all calls to lookup$_{\mathrm{RAM}}(i)$ are replaced with calls to lookup$_{\mathrm{RAM},SP}(i)$.

query$_{SP}(D, x)$:
Call query(D, x) where all calls to update$_{\mathrm{RAM}}(i, v)$ are replaced with update$_{\mathrm{RAM},SP}(i, v)$ and all calls to lookup$_{\mathrm{RAM}}(i)$ are replaced with calls to lookup$_{\mathrm{RAM},SP}(i)$.

backtrack$_{SP}(k)$:
Call backtrack(k).

The efficiency of the stack persistent RAM lends itself to the effieciency of the general stack persistent data structure, so from proposistion 5.2.1 we get are main result of this section.

Theorem 6 *Given an ephemeral data structure running in time t per operation and using space n on a RAM, there is a stack persistent version of this data structure running in amortised time $\mathrm{O}(t)$ per* update *and* lookup *operation and worst-case time $\mathrm{O}(1)$ per* backtrack *operation using space $4t + \mathrm{o}(t)$.*

There are two things to note here.

1. The algorithm turns the worst-case time of the data structure into an amortised time. It is not known how to construct a stack persistent data structure with efficient worst-case time.
2. If the ephemeral data structure has a good amortised running time we are unable to exploit that fact. We cannot know whether one of the expensive operations are repeated many times.

This means that where the Union-Find algorithm of Hopcroft and Ullman [30] (see Section 6.1) has a worst-case running time of $\mathrm{O}(\log n)$ and an amortised running time of $\mathrm{O}(\alpha(n))$, the stack persistent version of the Union-Find algorithm has an *amortised* running time of $\mathrm{O}(\log n)$ which is worse than the original algorithm.

Westbrook and Tarjan [76] have devised several algorithms for stack persistence union-find where each of the operations update, lookup, and backtrack(1) has amortised complexity $\mathrm{O}(\log n / \log\log n)$. Apostolico et al. [5] have improved this result by devising an algorithm where update and lookup has *worst-case* complexity $\mathrm{O}(\log n / \log\log n)$ and backtrack(k) has worst-case complexity $\mathrm{O}(1)$.

Chapter 6

The Constraint Solver

In the OIH constraint domain we had three kinds of relations (see Section 4.7):

- $=$
- $\langle\!\langle R_Q, \chi \rangle\!\rangle$
- $\nu X.\langle\!\langle R_Q, \chi \rangle\!\rangle$

The constraint solver must deal with the different components of this constraint domain. In this chapter we show how to deal with each of the components separately and then how to combine them into one constraint solver for the entire domain. The chapter is organised in seven sections. The first four are dealing with the individual components:

Section 6.1: Dealing with $=$.

Section 6.2: Dealing with R.

Section 6.3: Dealing with Q.

Section 6.4: Dealing with χ.

Section 6.5 provides two algorithms to be used as subroutines to other algorithms, Section 6.6 is about the combination of the elements into the full algorithm, and Section 6.7 deals with the satisfiability of the requirements of Chapter 4.

6.1 Unification on regular terms

Dealing with $=$ on terms is a much researched and well-understood problem [60, 58]. The algorithm for solving equality constraints (i.e. term equations) is the *unification algorithm*. A unification of two expressions e and e' is an assignment,

$\varphi : \mathcal{V} \to \mathsf{Expr}_\Sigma$, such that $\varphi \models e = e'$. If all other unifications, φ', can be written on the form $\varphi'' \circ \varphi$ then φ is the *most general unifier* (or m.g.u.) of e and e'.

The unification described above is on finite terms because we require that $\varphi : \mathcal{V} \to \mathsf{Expr}_\Sigma$. For this case Robinson [60] has an algorithm running in linear time. A more general form of unification would allow infinite solutions. That is, the assignment could be of the form $\varphi : \mathcal{V} \to \mathsf{Expr}_\Sigma^\infty$ (from Proposition 1.2.1(2) we know that the most general solution gives regular trees). In this case Paterson and Wegman [58] give several algorithms. One of these (Algorithm A) gives a solution (if one exists) in *pseudo-linear* time. (They also give an algorithm in linear time, but this algorithm is harder to make incremental).

The running time of an algorithm is pseudo-linear if it is $\mathrm{O}(n\alpha(n))$ where $\alpha(n)$ is the inverse to the Ackerman function. The $\alpha(n)$ comes from the use of Hopcroft and Ullman's union-find algorithm [30] which is shown to have amortised complexity $\mathrm{O}(\alpha(n))$ in [73].

The union-find algorithm is an incremental algorithm maintaining a set of equivalence classes of the universe $1..n$. The algorithm provides the following operations.

- $\mathsf{Init}(n)$: Create the equivalence classes $\{\{1\}, \{2\}, \ldots, \{n\}\}$.
- $\mathsf{Union}(n, n')$: Let S and S' is the equivalence classes containing n and n', respectively. Remove S and S' and insert $S \cup S'$ in their place.
- $\mathsf{Find}(n)$: Return the canonical element of the equivalence class containing n.

The algorithm provides no control over which elements should be the canonical elements of the equivalence classes, it only assures that they exists so that we can ask whether two numbers n and n' are equivalent by asking if $\mathsf{Find}(n)$ equals $\mathsf{Find}(n')$.

We will give an incremental version of Paterson and Wegman's algorithm here. Constructing the incremental version from the off-line version is quite straightforward. The Paterson and Wegman algorithm maintains a set of automaton nodes which are unified but whose sons may not be. In the incremental version all we need to do is to insert the nodes of the new equation into this set and proceed with the body of the original algorithm.

For the extension of our algorithm with the algorithms for the other elements, we shall represent the solution by means of a special kind of term automaton — the *base automaton*.

Definition 6.1.1 *Let Σ be a ranked alphabet, $\mathcal{V}$ a set of variables, and assume that $(\Sigma, \leqslant)$ forms a lower semi-lattice where $\mathrm{rank}(\bigwedge \Sigma) = 0$. A base automaton over $(\Sigma, \leqslant)$ is a quadruple $M = (Q, \mathcal{P}(\Sigma), L, \Delta)$ where*

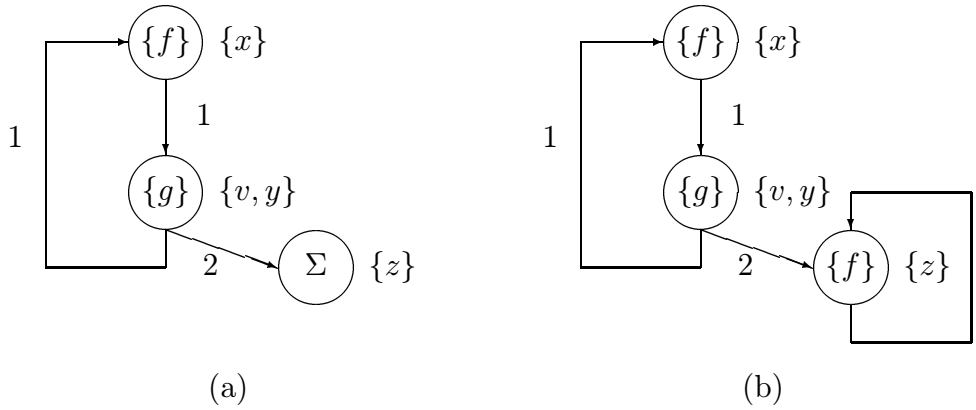


Figure 6.1: Two base automata.

- $Q \subseteq \mathcal{P}(\mathcal{V})$ *is a finite set of equivalence classes — the states.*
- $L : Q \to \mathcal{P}(\Sigma)$ *is the labelling function such that for all* $q \in Q$, $L(q) \neq \emptyset$.
- $\Delta : Q \times \omega \to Q$ *is the transition function — a partial function such that for each* $q \in Q$,

$$\{i \in \omega | (q, i) \in \mathrm{dom}(\Delta)\} = 1..\mathrm{rank}(\bigwedge L(q))$$

If $\mathrm{rank}(\bigwedge \Sigma) > 0$ we can introduce a new label, Ω, where $\mathrm{rank}(\Omega) = 0$, and Ω is smaller than any label from Σ.

The idea is that we introduce the base automaton as a way of representing the set of solutions to a set of equations. As an example consider the equation set

$$\begin{aligned} x &= f(y) \\ y &= g(x, z) \\ y &= v \end{aligned}$$

The base automaton representing the set of solutions to the above equation is the automaton in Figure 6.1(a).

In the base automaton we can find all the solutions for x by looking at the automaton with initial state $\{x\}$. That $L(\{z\}) = \Sigma$ corresponds to the fact that the equation set has no constraint on what z should be. We can look at another automaton which also represents a set of solutions to the equation set — the base automaton in Figure 6.1(b). The set of solutions that this base automaton represents is the singleton set with the unique solution for which $z = f(f(f(\ldots)))$.

We have then that the automaton in Figure 6.1(a) is more general than the one in Figure 6.1(b). When a base automaton, M, is more general than another, M', we shall write $M \Subset M'$. Let us move towards the formal definition of $\Subset$. Since

the rank of a set of labels is the rank of their greatest lower bound, we need to consider how ranks change when we move up in the ordering. For this we need to look at the relabeling function.

Let $v \in \bigcup Q$. We write Q_v for the unique equivalence class such that $v \in Q_v$ and $Q_v \in Q$. If $q \in Q$ we write $\lceil q \rceil$ for the canonical element of q. Now we can define a relabeling function over addresses as follows. Let $M = (Q, \mathcal{P}(\Sigma), L, \Delta)$ and $M' = (Q', \mathcal{P}(\Sigma), L', \Delta')$ be base automata, and let $v \in (\bigcup Q) \cap (\bigcup Q')$. We define the relabeling function $\rho^v_{M,M'} : \omega^* \to \omega^*$ inductively as follows:

- $\rho^v_{M,M'}(\epsilon) = \epsilon$
- Assume that $\rho^v_{M,M'}(\alpha)$, $\Delta(Q_v, \alpha)$, and $\Delta'(Q'_v, \rho^v_{M,M'}(\alpha))$ are all defined. Let $\sigma = \bigwedge L(\Delta(Q_v, \alpha))$ and $\sigma' = \bigwedge L'(\Delta'(Q'_v, \rho^v_{M,M'}(\alpha)))$. If $\sigma \leqslant \sigma'$ and $i \in 1..\mathrm{rank}(\sigma)$ we define $\rho^v_{M,M'}(\alpha i) = \rho^v_{M,M'}(\alpha)\rho_{\sigma \leqslant \sigma'}(i)$.

Basically, $\rho^v_{M,M'}$ is the inductive extension of ρ to addresses, but we need to take care so that the definition always makes sense.

Definition 6.1.2 *Let $M = (Q, \mathcal{P}(\Sigma), L, \Delta)$ and $M' = (Q', \mathcal{P}(\Sigma), L', \Delta')$ be base automata. We say that M' specialises M (or equivalently that M generalises M') and write $M \Subset M'$ iff the following hold.*

1. $\bigcup Q = \bigcup Q'$
2. *For all $q \in Q$ there is a $q' \in Q'$ such that $q \subseteq q'$.*
3. *For all $v \in \bigcup Q$ we have that if $\alpha \in \omega^*$ is accepted by $(Q, \mathcal{P}(\Sigma), Q_v, L, \Delta)$ then $\rho^v_{M,M'}(\alpha)$ is defined, $\rho^v_{M,M'}(\alpha)$ is accepted by $(Q', \mathcal{P}(\Sigma), Q'_v, L', \Delta')$, and $L'(\Delta'(Q'_v, \rho^v_{M,M'}(\alpha))) \subseteq L(\Delta(Q_v, \alpha))$.*

The second requirement in the definition says that if M' specialises M then the variables that are equivalent in M should also be equivalent in M', so that the equations remain valid in all specialisations. This means that if M is the base automaton in Figure 6.2(a) and M' is the base automaton in Figure 6.2(b) then $M \Subset M'$ but $M' \notin M$. We say that M' is obtained from M by the *unification* of y and z.

The automaton in Figure 6.1(b) is special because it corresponded to a unique solution. We say that the base automaton $M = (Q, \mathcal{P}(\Sigma), L, \Delta)$ is *definite* if for all $q \in Q$, we have that there is a $\sigma \in \Sigma$ so that $L(q) = \{\sigma\}$. If M is definite we define for each $v \in \bigcup Q$ the automaton M_v defined as

$$M_v = (Q, \Sigma, Q_v, l, \Delta)$$

where $l(q) = \sigma$ if $L(q) = \{\sigma\}$. In this way a definite automaton defines a solution $\mathcal{S}_M(v) = t^{M_v}$. Now we can define the *solution space* of a general base automaton,

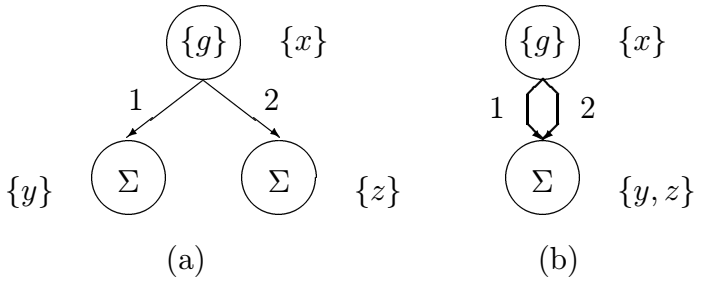


Figure 6.2: Unifying two variables.

M, as

$$\mathcal{L}(M) = \{\mathcal{S}_{M'} : \mathcal{V} \to \mathsf{Reg}_\Sigma | M' \text{ is definite} \wedge M \Subset M'\}$$

The central step in the algorithm is that of the unification of two states. For the automaton in Figure 6.2(a) this was quite easy, but in general we have to do slightly more work. The unification algorithm maintains a base automaton, $M = (Q, \mathcal{P}(\Sigma), L, \Delta)$, and uses a set of (equality) constraints, $\mathcal{C}$. The algorithm is shown in Figure 6.3.

The NewVariable procedure constructs a new state $\{v\}$ where v is a new variable and $L(\{v\}) = \Sigma$. It then returns v.

Since the UnifyVar procedure may introduce a new variable, it may extend the domain of the solutions. Hence, the solution spaces of the base automaton before and after the call to UnifyVar(v, v') could be incomparable. In order to be better able to reason about these solution spaces we shall identify two assignments that are equal when restricted to a given set of original variables. With this understanding we get the following lemma.

Lemma 6.1.1 *The set $\mathcal{L}(M) \cap \mathcal{L}(\mathcal{C})$ is invariant for the* **while***-loop in* UnifyVar*.*

Proof: Follows by straightforward induction. □

Corollary 6.1.2 *Let M be the base automaton before the call to* UnifyVar(v, v') *and M' be the base automaton after the call. Then we have*

$$\mathcal{L}(M') = \{\varphi \in \mathcal{L}(M) | \varphi(v) = \varphi(v')\}$$

Now we can use the UnifyVar procedure for solving equality constraints. First we need to deal with the expressions. Let e be an expression and $M = (Q, \mathcal{P}(\Sigma), L, \Delta)$ a base automaton, and assume that we have a variable $v_{e'}$ for every subexpression e' of e, such that $v_{e'} = v$ if $e' = v$ and $v_{e'}$ is a new variable if e' is on the form $\sigma(e''_1, \ldots, e''_k)$. We define the base automaton of M and e, base(M, e) recursively as follows.

Procedure UnifyVar(v, v');
  $\mathcal{C} := \{v = v'\}$;
  while $\mathcal{C} \neq \emptyset$ **do**
    Let $v_1 = v_2 \in \mathcal{C}$;
    $\mathcal{C} := \mathcal{C} \setminus \{v_1 = v_2\}$;
    if $Q_{v_1} \neq Q_{v_2}$ **then**
      if $L[Q_{v_1}] \cap L[Q_{v_2}] = \emptyset$ **then**
        Output "Unsatisfiable" and halt.
      else
        $Q := Q \setminus \{Q_{v_1}, Q_{v_2}\} \cup \{Q_{v_1} \cup Q_{v_2}\}$;
        $L[Q_{v_1} \cup Q_{v_2}] := L[Q_{v_1}] \cap L[Q_{v_2}]$;
        $\sigma_1, \sigma_2, \sigma_{12} := \bigwedge L[Q_{v_1}], \bigwedge L[Q_{v_2}], \bigwedge L[Q_{v_1} \cup Q_{v_2}]$;
        for $i \in 1..\mathrm{rank}(\sigma_{12})$ **do**
          if $i \in \mathrm{codom}(\rho_{\sigma_1 \leqslant \sigma_{12}}) \cap \mathrm{codom}(\rho_{\sigma_2 \leqslant \sigma_{12}})$ **then**
            $\Delta[Q_{v_1} \cup Q_{v_2}, i] := \Delta[Q_{v_1}, \rho_{\sigma_1 \leqslant \sigma_{12}}{}^{-1}(i)]$;
            $\mathcal{C} := \mathcal{C} \cup \{\lceil \Delta[Q_{v_1}, \rho_{\sigma_1 \leqslant \sigma_{12}}{}^{-1}(i)] \rceil = \lceil \Delta[Q_{v_2}, \rho_{\sigma_2 \leqslant \sigma_{12}}{}^{-1}(i)] \rceil\}$
          elseif $i \in \mathrm{codom}(\rho_{\sigma_1 \leqslant \sigma_{12}})$ **then**
            $\Delta[Q_{v_1} \cup Q_{v_2}, i] := \Delta[Q_{v_1}, \rho_{\sigma_1 \leqslant \sigma_{12}}{}^{-1}(i)]$
          elseif $i \in \mathrm{codom}(\rho_{\sigma_2 \leqslant \sigma_{12}})$ **then**
            $\Delta[Q_{v_1} \cup Q_{v_2}, i] := \Delta[Q_{v_2}, \rho_{\sigma_2 \leqslant \sigma_{12}}{}^{-1}(i)]$
          else
            $\Delta[Q_{v_1} \cup Q_{v_2}, i] := \{\mathsf{NewVariable}\}$
          end
        end
      end
    end
  end
end UnifyVar

Figure 6.3: The UnifyVar procedure.

- $\mathsf{base}(M, v) = M$
- Let $e = \sigma(e'_1, \ldots, e'_k)$ and for each j, $\mathsf{base}(M, e'_j) = (Q_j, \mathcal{P}(\Sigma), L_j, \Delta_j)$. Then we have that $\mathsf{base}(M, e) = (\{\{v_e\}\} \cup \bigcup_{j=1}^{k} Q_j, \mathcal{P}(\Sigma), L', \Delta')$, where

$$L'(q) = \begin{cases} L_j(q) & \text{if } q \in Q_j \\ \{\sigma\} & \text{if } q = \{v_e\} \end{cases}$$

$$\Delta'(q, i) = \begin{cases} \Delta_j(q, i) & \text{if } q \in Q_j \\ q' & \text{if } q = \{v_e\} \wedge v_{e_i} \in q' \end{cases}$$

The Unify procedure for solving equality constraints consists simply of two calls to base and a single call to UnifyVar:

Procedure Unify(e, e')
  $M :=$ base(base$(M, e), e')$;
  UnifyVar$(v_e, v_{e'})$
end Unify

That this is correct is follows from corollary 6.1.2. Hence we have the following.

Proposition 6.1.3 *Let M be the base automaton before the call to* Unify(e, e') *and M' be the base automaton after the call. Then we have*

$$\mathcal{L}(M') = \mathcal{L}(M) \cap \mathcal{L}(e = e')$$

It follows that if we can initialise the base automaton correctly, we can solve a set of equality constraints. This initialisation is quite straightforward. Let L_0 be defined as $L_0(\{v\}) = \Sigma$ and Δ_0 be everywhere undefined. Then the initialisation procedure is as follows.

Procedure Init$_M$
  $M := (\{\{v\} | v \in \mathcal{V}\}, \mathcal{P}(\Sigma), L_0, \Delta_0)$
end Init$_M$

Now we can formulate the correctness of unification.

Proposition 6.1.4 *Let $\mathcal{C} = \{e_1 = e'_1, \ldots, e_m = e'_m\}$ be a set of term equations, and the M be the base automaton obtained by the calls*

$$\text{Init}_M; \text{Unify}(e_1, e'_1); \ldots; \text{Unify}(e_m, e'_m)$$

Then $\mathcal{L}(M) = \mathcal{L}(\mathcal{C})$.

Proof: Follows by induction from Proposition 6.1.3. □

For the analysis of the procedure we note that we cannot unify two variables more that $|\mathcal{V}|$ times since the size of the state set decreases every time. Each time we unite two variables we use time to unite the states, calculate the intersection, and update Δ. If we use the Union-Find structure to implement the states and bit vectors to implement the labels, we get an amortised running time of $O(\alpha(n) + s + r)$ where $n = |\mathcal{V}|$, $s = |\Sigma|$, and r is the rank of the new label.

Since there is a variable for each subterm of the expressions in our equality constraints, we have that the size of $\mathcal{V}$ will be the sum of the sizes of the expressions. So let n be the sum of the sizes of all the expressions in the equality constraints, $s = |\Sigma|$, and r the highest rank that any label will get. Then we have the following.

Proposition 6.1.5 *The total running time of the calls*

$$\mathsf{Init}_M; \mathsf{Unify}(e_1, e_1'); \ldots; \mathsf{Unify}(e_m, e_m')$$

is $\mathrm{O}(n \cdot (\alpha(n) + s + r))$.

In the case where all constraints are on the form $e = e'$, all the labels in the base automaton are either a singleton set or Σ. We can use this fact to get a more efficient implementation of the intersection of the label. Furthermore, we can see that all labels stem from the input, and hence we can get a bound on the rank of the labels in terms on n. In this case then we can get a running time of $\mathrm{O}(n\alpha(n))$ which is what Paterson and Wegman does in [58].

In the general case where the constraints could be any constraints from Γ^{OIH} we cannot assume that the sets have this form, though, and we are stuck with the running time of Proposition 6.1.5 above.

We can use the base automaton to extract the information we want. For instance we could define a variable v_q for every $q \in Q$ such that $L(q) = \Sigma$. Then let the automaton $M_{\mathrm{mgu}} = (Q, \mathcal{P}(\Sigma \cup \mathcal{V}), L_{\mathrm{mgu}}, \Delta)$ be defined by

$$L_{\mathrm{mgu}}(q) = \begin{cases} \{v_q\} & \text{if } L(q) = \Sigma \\ L(q) & \text{otherwise} \end{cases}$$

Then $\mathcal{L}(M_{\mathrm{mgu}})$ has only one element — the most general unifier of the set of equations.

As an example, which is more relevant for this work, we can define $M_{\min} = (Q, \mathcal{P}(\Sigma), L_{\min}, \Delta)$ by $L_{\min}(q) = \{\bigwedge L(q)\}$. Then the only element of $\mathcal{L}(M_{\min})$ is the $\leqslant_\Sigma$-least solution to the set of equations.

6.2 Arc Consistency

The arc consistency problem is the problem of finding a conservative approximation to the set of solutions to a constraint satisfaction problem (see Section 4.2) by assigning to each variable the set of values it can assume. The problem is finding an approximation that is (arc) consistent. That is, no value should be present which cannot appear in a tuple where all other values are present. The approximation for a variable v is called the *domain* of v and is written $D(v)$, the function $D : \mathcal{V} \to \mathcal{P}(\Sigma)$ is called the *domain assignment*.

As an example of arc consistency, let $\Sigma = \{a, b\}$ and consider the constraint $R(v, v')$, where $R = \{(a, a), (b, a), (b, b)\}$. Now, assume that for some reason we know that v cannot take the value b, so we have $D(v) = \{a\}$. This situation is

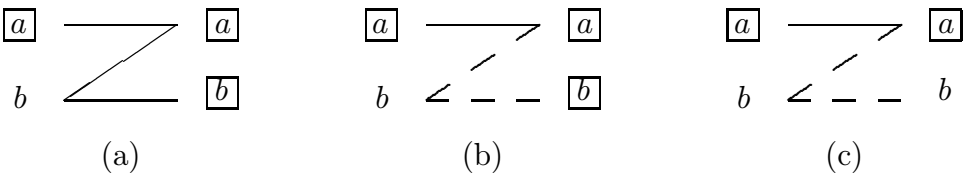


Figure 6.4: An example of arc consistency.

illustrated in Figure 6.4(a), where the column on the left is $D(v)$, the column on the right is $D(v')$, the boxed values are the ones belonging to the domains, and the lines illustrates the relation R.

We can see that two of the pairs in R will not be relevant since they assume that v is b, which we know it cannot be. These pairs are the dashed lines in Figure 6.4(b). Now, that we know that only the solid line represent a possible solution we have that since b in $D(v')$ is adjacent to no solid line from R, we can remove it from $D(v')$ without losing any possible solutions. This is the situation in Figure 6.4(c).

We say that $a \in D(v)$ is the *support* for $a \in D(v')$ and that $b \in D(v')$ is *unsupported*. A domain is *consistent* if there are no unsupported values. For simplicity, we assume that the constraint set is *permutation-closed* in the formal definition of consistency. We say that a set of constraints, $\mathcal{C}$, is permutation-closed, iff for each constraint $R(v_1, \ldots, v_n) \in \mathcal{C}$ and each permutation $\pi : 1..n \to 1..n$ we have that $R_\pi(v_{\pi(1)}, \ldots, v_{\pi(n)}) \in \mathcal{C}$.

Definition 6.2.1 *Let Σ be a finite set of values, $D : \mathcal{V} \to \mathcal{P}(\Sigma)$ a domain assignment, and $\mathcal{C}$ a permutation-closed set of constraints over CSP_Σ. We say that D is* consistent *with respect to $\mathcal{C}$, iff*

$$\forall v \in \mathcal{V}, R(v, v_1, \ldots, v_n) \in \mathcal{C}, d \in D(v) : R \cap (\{d\} \times \prod_{i=1}^{n} D(v_i)) \neq \emptyset$$

If C is not permutation-closed it is consistent iff its closure under permutation is consistent. In the definition above the set $R \cap (\{d\} \times \prod_{i=1}^{n} D(v_i))$ corresponds to the solid lines which are adjacent to d.

In the binary case we say that the domain assignment is *node consistent* if it is consistent with respect to the node constraint, and *arc consistent* if it is consistent with respect to the arc constraints. It is easy to see that a domain assignment is node consistent iff for all $v \in \mathcal{V}$ and $R(v) \in \mathcal{C}$ we have $D(v) \subseteq R$, and that it is arc consistent iff for each $v \in \mathcal{V}$ and $R(v, v') \in \mathcal{C}$ we have that for each $d \in D(v)$ there is a $d' \in D(v')$ such that $R(v, v')$. This is the definition of node and arc consistency usually given in the literature (e.g. in [50, 22]).

It follows directly from Definition 6.2.1 that the domain assignment $D(v) = \emptyset$ is consistent with respect to any set of constraints. This is of course a far cry from being conservative, so we will look for the proper consistent assignment. We need then to look at some of the properties of consistency.

Lemma 6.2.1 *Let Σ be a finite set of values and $\mathcal{C}$ a permutation-closed set of constraints over CSP_Σ. We have the following.*

1. *Let $\varphi : \mathcal{V} \to \Sigma$ and define $D : \mathcal{V} \to \mathcal{P}(\Sigma)$ as $D(v) = \{\varphi(v)\}$. Then D is consistent with respect to $\mathcal{C}$, iff φ is a solution to $\mathcal{C}$.*
2. *Let $D : \mathcal{V} \to \mathcal{P}(\Sigma)$ and $D' : \mathcal{V} \to \mathcal{P}(\Sigma)$ be consistent with respect to $\mathcal{C}$. Then $D'' : \mathcal{V} \to \mathcal{P}(\Sigma)$ defined as $D''(v) = D(v) \cup D'(v)$ is consistent with respect to $\mathcal{C}$.*

Proof: Both properties follow directly from Definition 6.2.1. □

Corollary 6.2.2 *Let Σ be a finite set of values and $\mathcal{C}$ a permutation-closed set of constraints over $\mathcal{CSP}_\Sigma$. There is a maximal, consistent domain assignment with respect to $\mathcal{C}$.*

Proof: Define $D : \mathcal{V} \to \mathcal{P}(\Sigma)$ as

$$D(v) = \bigcup\{D'(v) | D' \text{ is consistent}\}$$

It follows from Lemma 6.2.1(2) that D is consistent, and from the definition of D it follows that it is greater than all other consistent domain assignments. □

This maximal, consistent domain assignment is the conservative approximation we are looking for. The conservativity is shown below.

Proposition 6.2.3 *Let Σ be a finite set of values, $\mathcal{C}$ a permutation-closed set of constraints over CSP_Σ, and D the maximal, consistent domain assignment with respect to $\mathcal{C}$. We have the following:*

$$\forall\varphi\colon \mathcal{V} \to \Sigma.\ \varphi \models \mathcal{C} \Rightarrow \forall v \in \mathcal{V} : \varphi(v) \in D(v)$$

Proof: Assume to the contrary that $\varphi \models \mathcal{C}$ but $\varphi(v) \notin D(v)$. By Lemma 6.2.1 we have that D' defined as $D'(u) = D(u) \cup \{\varphi(u)\}$ is consistent. But then we have a consistent domain assignment D' such that $D(v) \subsetneq D'(v)$ in contradiction with the maximality of D. □

Mohr and Henderson[50] has devised an efficient algorithm for computing the maximal arc and node consistent domain assignment for a binary constraint satisfaction problem. Frigioni, Machetti-Spaccemela, and Nanni [22] has an incremental version of that algorithm. The running time of both algorithms is $O(s^2m)$,

where $s = |\Sigma|$ and $m = |C|$. If we assume that the relations are represented as bit matrices then s^2m is exactly the size of the input. The algorithms are thus optimal.

We will show a generalisation of the latter of these algorithms to the general case. The algorithms in [50, 22] assumes that the constraint set is permutation-closed. Since the rank of the relations is at most 2, this only gives a computational overhead of a factor of 2. In our case however the rank of the relations can be arbitrarily high. Assume that the rank of the relations is n. Then computing the permutation closure of the constraint set gives a computational overhead of a factor of $n!$ which is clearly unreasonable. Instead, we implement the relations in such a way that all the permuted relations are implicitly represented.

The algorithm uses the following data structures:

- The *domain assignment*, $D[v]$, implemented by an array of bit vectors.
- The *support sets*, $Support[C, v, d]$, implemented as a bit vector and an array of pointers to tuples. The support set $Support[R(v_1, \ldots, v_n), v, d]$, where $v = v_i$, is the set $R \cap (\prod_{j=1}^{i-1} D[v_j] \times \{d\} \times \prod_{j=i+1}^{n} D[v_j])$.
- The *adjacency sets*, $E[v]$, implemented as an array of linked lists. For each variable v, $E[v]$ is the set $\{(R(v_1, \ldots, v_n), i) \in \mathcal{C} \times \omega | v = v_i\}$.
- The *relations*, R, implemented as an array (or linked list) of *tuples*, which are implemented as an array of values.
- The *prune queue*, P, implemented by standard methods. The prune queue is a queue of pairs, (v, d), where we have removed d from $D[v]$ during the pruning process. We have access to P through the procedures RemoveFirst and InsertQueue.

The index in $E[v]$ defines which of the permutations of R is adjacent to v. This is sufficient for the algorithm.

The important step in the procedure is that of *pruning* ("implicit deletion" in [22]). Pruning is the repetition of the procedure illustrated in Figure 6.4 until no unsupported values exist. The procedure Prune can be seen in Figure 6.5.

In [22] the *increment* step is taken by removing pairs from the relation between two variables and calling Prune to restore arc consistency. Initially, the constraint $\Sigma^2(v, v')$ is present for all $v, v' \in \mathcal{V}$, and the insertion of a constraint, $R(v, v')$, is done by removing the pairs in $D[v] \times D[v'] \setminus R$. Here, we will have no constraints initially, but we will insert the relations when they appear and then simulate the removal of the pairs $\prod_{i=1}^{n} D[v_i] \setminus R$. The procedure Insert is shown in Figure 6.6.

```
Procedure Prune;
    while P ≠ ∅ do
        (v, d) := RemoveFirst(P);
        for (R(v_1, ..., v_n), i) ∈ E[v] do
            for (d_1, ..., d_n) ∈ Support[R(v̲), v, d], j ∈ 1..n \ {i} do
                Support[R(v̲), v_j, d_j] := Support[R(v̲), v_j, d_j] \ {(d_1, ..., d_n)};
                if Support[R(v̲), v_j, d_j] = ∅ ∧ d_j ∈ D[v_j] then
                    D[v_j] := D[v_j] \ {d_j};
                    InsertQueue(P, d, v_j)
                end
            end
        end
    end
end Prune
```

Figure 6.5: The Prune procedure.

```
Procedure Insert(R(v_1, ..., v_n))
    for i ∈ 1..n do
        E[v_i] := E[v_i] ∪ {(R(v_1, ..., v_n), i)};
        for d ∈ D[v_i] do Support[R, v_i, d] := ∅ end
    end;
    for (d_1, ..., d_n) ∈ R ∩ (∏_{i=1}^{n} D[v_i]), i ∈ 1..n do
        Support[R, v_i, d_i] := Support[R, v_i, d_i] ∪ {(d_1, ..., d_n)}
    end;
    for i ∈ 1..n, d ∈ D[v_i] do
        if Support[R, v_i, d] = ∅ then
            D[v_i] := D[v_i] \ {d};
            InsertQueue(P, v_i, d)
        end
    end;
    Prune
end Insert
```

Figure 6.6: The Insert procedure.

The initialisation of the data structures is quite straightforward:

Procedure Init_D
  for $v \in \mathcal{V}$ **do**
    $D[v] := \Sigma$;
    $E[v] := \emptyset$
  end
  $P := \emptyset$
end Init_D

The correctness of the procedure follows from the following fact about arc consistency.

Lemma 6.2.4 *Let Σ be a finite set of values and $\mathcal{C}$ a permutation-closed set of constraints over CSP_Σ. Define the function $\mathcal{F}_\mathcal{C} : (\mathcal{V} \to \mathcal{P}(\Sigma)) \to (\mathcal{V} \to \mathcal{P}(\Sigma))$ as*

$$\mathcal{F}_\mathcal{C}(D)(v) = \bigcap_{R(v,v_1,\ldots,v_n)\in\mathcal{C}} \{d \in D(v) | (\{d\} \times \prod_{i=1}^{n} D(v_i)) \cap R \neq \emptyset\}$$

Then $\mathcal{F}_\mathcal{C}(D) = D$, iff D is consistent with respect to $\mathcal{C}$.

Proof: Follows directly from Definition 6.2.1. □

It follows that the maximal, consistent domain assignment with respect to $\mathcal{C}$ is the largest fixpoint of $\mathcal{F}_\mathcal{C}$.

Proposition 6.2.5 *Let $\mathcal{C} = \{C_1, \ldots, C_m\} \subseteq \mathsf{CSP}_\Sigma[\mathcal{V}]$. The domain assignment obtained by calling*

$$\mathsf{Init}; \mathsf{Insert}(C_1); \ldots; \mathsf{Insert}(C_m)$$

is the maximal, consistent domain assignment with respect to $\mathcal{C}$.

Proof: Since the call to Prune is a fixpoint iteration for the largest fixpoint of $\mathcal{F}_\mathcal{C}$ the proposition follows by straightforward induction in m. □

The complexity of the general algorithm is optimal as shown in the following. For simplicity we assume that $|\Sigma| \leq |R|$ for the relations in the constraint set. Hardly a controversial assumption given that $|R|$ can be as high as $|\Sigma|^{\mathrm{rank}(R)}$.

Proposition 6.2.6 *The amortised complexity of the incremental consistency algorithm is $\mathrm{O}(|\mathcal{V}|)$ for a call to Init and $\mathrm{O}(n \cdot |R|)$ for a call to $\mathsf{Insert}(R(v_1, \ldots, v_n))$.*

Proof: We will analyse the complexity by the potential function method of Tarjan [72]. Let the potential function Φ be defined as

$$\Phi = c \cdot \sum_{R(v_1,\ldots,v_n)\in\mathcal{C}} n \cdot |R \cap \prod_{i=1}^{n} D[v_i]|$$

Where $c > 0$ is some constant. The complexity of Init follows directly from this since $\Phi = 0$ after the call to Init.

Let us look at a call to Insert$(R(v_1, \ldots, v_n))$. The running time of the call without the call to Prune is at most $O(n \cdot (|\Sigma| + |R|)) \leq O(n \cdot |R|)$ (by assumption). It is clear that Φ cannot increase by more than $n \cdot |R|$ what remains to be shown is that the time of the call to Prune has an amortised running time of $O(1)$.

Let us look at a single run through the outer **while**-loop. The loop begins with finding a pair (v, d) on the queue. That the pair is in fact on the queue means that d has been removed from $D[v]$ during this call to Insert. The removal of $D[v]$ has brought a decrease in Φ. The size of this decrease is

$$\begin{aligned} \Delta\Phi &= c \cdot \sum_{(R(v_1,\ldots,v_n),i)\in E[v]} n \cdot |R \cap (\prod_{j=1}^{i-1} D[v_j] \times \{d\} \times \prod_{j=i+1}^{n} D[v_j])| \\ &= c \cdot \sum_{(C,i)\in E[v]} n \cdot |\mathit{Support}[C, v, d]| \end{aligned}$$

The time for the inner **for**-loop is $O(n \cdot |\mathit{Support}[C, v, d]|)$ and hence the time of a single run through the **while**-loop is $O(\sum_{(C,i)\in E[v]} n \cdot |\mathit{Support}[C, v, d]|)$. It follows that we can choose c such that $\Delta\Phi$ is greater than the running time of Prune. The proposition follows. □

If we cannot assume that $|\Sigma| \leq |R|$ we would have to write the running time of Insert$(R(v_1, \ldots, v_n))$ as $O(n \cdot (|R| + |\Sigma|))$. This would be necessary in e.g. the degenerate case where all the relations are singleton sets.

Note that in the binary case we have $n \leq 2$ and $|R| \leq |\Sigma|^2$. So for binary CSP the running time of the algorithm reduces to amortised $O(|\Sigma|^2)$ for each insert — the same as in [22].

Now we have the results necessary to make good on our promise from Section 4.2 to show that the stable constraint satisfaction problem is solvable in linear time. First we need to know the following about the relation between consistency and lower semi-lattices.

Lemma 6.2.7 *Let $(\Sigma, \leqslant)$ be a lower semi-lattice, $\mathcal{C}$ a set of constraints over CSP_Σ, and D a consistent domain assignment with respect to $\mathcal{C}$. Assume that $R(v_1, \ldots, v_n) \in \mathcal{C}$. If for all $i \in 1..n$ we have $D(v_i) \neq \emptyset$, then we have the following.*

- $R \cap (\prod_{i=1}^{n} D(v_i)) \neq \emptyset$
- $\bigwedge(R \cap (\prod_{i=1}^{n} D(v_i))) = \bigwedge(\prod_{i=1}^{n} D(v_i))$

Proof: Follows directly from Definition 6.2.1 and the definition of $\bigwedge$. □

This leads to the central proposition about stability and consistency.

Proposition 6.2.8 *Let $(\Sigma, \leqslant)$ be a lower semi-lattice, $\mathcal{C}$ a set of $\leqslant$-stable constraints over CSP_Σ, and D the maximal, consistent domain assignment with respect to $\mathcal{C}$. Then we have:*

1. *If there is a variable $v \in \mathcal{V}$ such that $D(v) = \emptyset$ then $\mathcal{C}$ is unsatisfiable.*
2. *If $D(v)$ is non-empty for all variables $v \in \mathcal{V}$ then the assignment $\varphi : \mathcal{V} \to \Sigma$ defined as $\varphi(v) = \bigwedge D(v)$ is the $\leqslant$-least solution to $\mathcal{C}$.*

Proof:

1. Follows from Proposition 6.2.3.
2. Let $R(v_1, \ldots, v_n) \in \mathcal{C}$ be arbitrary. From Lemma 6.2.7 we have that $R \cap (\prod_{i=1}^n D(v_i)) \neq \emptyset$, and hence by the stability of R, $\bigwedge(R \cap (\prod_{i=1}^n D(v_i))) \in R$. Using Lemma 6.2.7 again we get that $(\varphi(v_1), \ldots, \varphi(v_n)) = \bigwedge(\prod_{i=1}^n D(v_i)) = \bigwedge(R \cap (\prod_{i=1}^n D(v_i)))$. Hence we have that $R(\varphi(v_1), \ldots, \varphi(v_n))$ as requested. The minimality of φ follows from the definition of φ and Proposition 6.2.3.

□

This leads to the main theorem of this section.

Theorem 7 *There is an optimal, incremental algorithm for the stable constraint satisfaction problem.*

Proof: Follows from Propositions 6.2.5, 6.2.6, and 6.2.8. □

It follows that there is a linear algorithm for the satisfiability of Horn formulae. This well-known fact is proven in e.g. [63], where an algorithm is given which finds the $\Rightarrow$-least solution just as we do here.

6.3 Infinite Alphabets and Consistency

If the alphabet is of the form $\Sigma_\mathcal{E}$ we need to be able to deal with relations on the form R_Q. The way to deal with them is to apply the definitions of stability (Definition 4.2.2) and consistency (Definition 6.2.1) to infinite alphabets as well and compute consistent, infinite domain assignments. As we saw in the definition of R_Q in Section 4.3 we only need to deal with cone sets and singleton sets of extensions.

Now, let us write $\widehat{e}$ for the cone set $\{e' \in \mathcal{E}_\sigma | e' \succeq_\sigma e\}$, and $\sigma[E] = \{\sigma[e] | e \in E\}$ if $E \subseteq \mathcal{E}_\sigma$, and let $R = \{(\sigma, \sigma) \in \Sigma\}$ and let $Q(\sigma, \sigma) = \{1 \preceq_\sigma 2\}$ if $\sigma \in \mathrm{dom}(\mathcal{E})$ and

$Q(\sigma,\sigma) = \emptyset$ otherwise. If $D(v) = \sigma[\widehat{e}]$ and $D(v') = \sigma[\widehat{e'}]$ we have that D is arc consistent with respect to the constraint $R_Q(v,v')$, iff $e \preceq_\sigma e'$.

The above example serves to illustrate that when the constraint satisfaction problem is resolved with respect to the finite part of the alphabet the problem reduces to a problem of solving $\preceq_\sigma$ constraints over the extensions. In fact, this is possible in general as we shall see in the following.

We operate with two domain assignments: $D : \mathcal{V} \to \mathcal{P}(\Sigma)$ and $D_\mathcal{E} : \mathcal{V} \to \mathcal{P}(\bigcup_{\sigma\in\mathrm{dom}(\mathcal{E})} \mathcal{E}(\sigma))$. The domain assignment D is the normal domain assignment from the consistency algorithm and $D_\mathcal{E}$ is the extension domain assignment which assigns the extensions to the variables. If $S \subseteq \Sigma$ and $S' \subseteq \bigcup_{\sigma\in\mathrm{dom}(\mathcal{E})} \mathcal{E}(\sigma)$ we define the set of extended labels from S and S' as

$$S[S'] = (S \setminus \mathrm{dom}(\mathcal{E})) \cup \bigcup_{\sigma\in\mathrm{dom}(\mathcal{E})} \{\sigma[e] | e \in S' \cap \mathcal{E}(\sigma)\}$$

Then the actual domain of v is $D(v)[D_\mathcal{E}(v)]$.

The extension domain $D_\mathcal{E}(v)$ can take on three different kinds of values.

1. A singleton set $D_\mathcal{E}(v) = \{e\}$.
2. A cone set $D_\mathcal{E}(v) = \widehat{e}$.
3. An unlimited set $D_\mathcal{E}(v) = \bigcup_{\sigma\in\mathrm{dom}(\mathcal{E})} \mathcal{E}(\sigma)$. We denote this set by the special symbol $\top$.

By using $\top$ we avoid the need to have sets of the form $\widehat{\bot_{\sigma_1}} \cup \ldots \cup \widehat{\bot_{\sigma_n}}$, since $\{\sigma_1,\ldots,\sigma_n\}[\widehat{\bot_{\sigma_1}} \cup \ldots \cup \widehat{\bot_{\sigma_n}}] = \{\sigma_1,\ldots,\sigma_n\}[\top]$.

The algorithm proceeds by performing arc consistency on the finite parts of the alphabet and relations only and considering a sufficient number of inequality constraints. The inequality constraints we need to consider are the inequality constraints from $Q(\bigwedge(R \cup \prod_i D(v_i)))$. Due to the monotonicity of Q any solution will have to satisfy these constraints.

Since we have that the constraints in $Q(\sigma_1,\ldots,\sigma_n)$ are of the form $i \preceq_\sigma j$, where $\sigma = \sigma_i = \sigma_j$, we can be assured that once there is a constraint of the form $v \preceq_\sigma v'$ we have that $D(v) = D(v') = \{\sigma\}$. Hence we have for each $\sigma \in \mathrm{dom}(\mathcal{E})$ a constraint set where the constraints are of the form $v \preceq_\sigma v'$. All these *inequality subsystems* are disjoint (i.e. no variable appears in more than one constraint set), so we can solve each of them separately without any consideration about the others.

Let D be the maximal, consistent domain assignment for the finite subproblem, and $D_\mathcal{E}$ the extension domain assignment which is the maximal, (arc) consistent domain assignment when restricted to each inequality subsystem and has $D_\mathcal{E}(v) =$

```
Procedure Insert⊆(i, j)
    for k ∈ Dε[i] \ Dε[j] do
        S := {j};
        Dε[j] := Dε[j] ∪ {k};
        while S ≠ ∅ do
            v := Pop(s);
            for w ∈ E[v] do
                if k ∉ Dε[w] then
                    Dε[w] := Dε[w] ∪ {k};
                    Push(S, w)
                end
            end
        end
    end;
    E[i] := E[i] ∪ {j}
end Insert⊆
```

Figure 6.7: The $\mathsf{Insert}_{\subseteq}$ procedure.

$\top$ for all variables not in any inequality subsystems. We have that the assignment φ defined as

$$\varphi(v) = \begin{cases} (\bigwedge D(v))[\lambda D_{\mathcal{E}}(v)] & \text{if } \bigwedge D(v) \in \mathrm{dom}(\mathcal{E}) \\ \bigwedge D(v) & \text{otherwise} \end{cases}$$

is the least solution to the constraint set. This follows directly from Proposition 6.2.8 and the monotonicity of Q.

The remaining problem is that of solving the arc consistency problem over the domains $\mathcal{E}(\sigma)$. The best way to do it is domain-specific. We will show an efficient algorithm for sets of field names. Then we will show how this can in principle be generalised to work for arbitrary domains. Only in the latter case there is no guarantee that this will be an efficient way of doing it.

The algorithm is inspired by the incremental transitive closure algorithm of La Poutré and van Leeuwen [43]. To see what a transitive closure has to do with set inclusion constraints consider the following reduction. Let $G = (V, E)$ be a directed graph. Now we construct a constraint set in the following way: For each node $v \in V$ we have a variable, $[\![v]\!]$. For each node we have the reflexivity constraint $\{v\} \subseteq [\![v]\!]$, and for each edge $(v, v') \in E$ we have the closure constraint $[\![v]\!] \supseteq [\![v']\!]$. Then the least solution to the constraint system assigns to each variable $[\![v]\!]$ the set of nodes that are reachable from v.

La Poutré and van Leeuwen's transitive closure algorithm works in the following way. When inserting an edge from i to j, then for each k such that i is reachable

from k we proceed by depth-first search through all the nodes k' reachable from j which was not reachable from k before the insertion of (i, j). For each such pair (k, k') we update the transitive closure so that k' is reachable from k. Our algorithm is based on the dual of La Poutré and van Leeuwen's algorithm. That is, the algorithm which for every node reachable from j but not from i goes through the predecessors of i by reverse depth-first search and makes the same updates as before. This means that we make the search in the direction of the $\subseteq$-constraints.

The algorithm is described by the procedure $\mathsf{Insert}_{\subseteq}$ in Figure 6.7. It maintains a least solution, $D_{\mathcal{E}}$, to a set of set inclusion constraints under the insertion of a constraint $i \subseteq j$. In order to do this it maintains the adjacency list $E[v] = \{w | v \subseteq w \ \in \mathcal{C}\}$. The algorithm uses the adjacency list to do a depth-first search for edges (i, j) (that is, constraints $i \subseteq j$) where $k \in D_{\mathcal{E}}[i]$ but $k \notin D_{\mathcal{E}}[j]$ it then pushes k along the edge (i, j) so that $k \in D_{\mathcal{E}}[j]$ afterwards.

It is a shortcoming of the insert procedure that it can deal with constraints of the form $i \subseteq j$ only. If all constraints are of this form the smallest solution will be $D_{\mathcal{E}}[v] = \emptyset$. We add constraints of the form $\alpha \subseteq j$ by making a new variable i with $D_{\mathcal{E}}[i] = \alpha$ and $E[i] = \emptyset$. We mark i and call $\mathsf{Insert}_{\subseteq}(i, j)$, where we have made the change that if we try to change the value of $D_{\mathcal{E}}[v]$ where v is marked, we stop and output "Unsatisfiable".

We can see that no value can be pushed along the same edge twice. So if n is the number of different values occurring in the sets and we implement the sets by bitvectors, then the total running time of m calls to the $\mathsf{Insert}_{\subseteq}$ procedure is $\mathrm{O}(mn)$ — the same complexity as in [43]. In other words the $\mathsf{Insert}_{\subseteq}$ procedure has an *amortised* running time of $\mathrm{O}(n)$.

In the general case where we do not know what the lattice is, we cannot just push single elements along the edges — we have to push the whole extension along the edges. Instead of the expression $D_{\mathcal{E}}[v] \cup \{k\}$ we must use the expression $D_{\mathcal{E}}[v] \curlyvee e$, where e is the extension pushed along the edge to v. This leads us to the general algorithm shown as the procedure $\mathsf{Insert}_{\preceq}$ in Figure 6.8.

The analysis from $\mathsf{Insert}_{\subseteq}$ above breaks down on two accounts:

1. There is no bound on the number of times we can perform the operation $D_{\mathcal{E}}[v] := D_{\mathcal{E}}[v] \curlyvee D_{\mathcal{E}}[w]$.

2. There is no way of knowing what the running time of computing $\curlyvee$ is.

Still, there is something we *can* say. Since we perform a depth-first search we can know that we make no more than $\mathrm{O}(m)$ computations of $\curlyvee$.

We thus arrive at our main result.

```
Procedure Insert_⪯(i, j)
    if D_E[i] ⋠ D_E[j] then
        S := {j};
        D_E[j] := D_E[j] ϒ D_E[i];
        while S ≠ ∅ do
            v := Pop(s);
            for w ∈ E[v] do
                if D_E[i] ⋠ D_E[w] then
                    D_E[w] := D_E[w] ϒ D_E[i];
                    Push(S, w)
                end
            end
        end
    end;
    E[i] := E[i] ∪ {j}
end Insert_⪯
```

Figure 6.8: The $\mathsf{Insert}_{\preceq}$ procedure.

Theorem 8

1. *There is an incremental constraint solver for the set inclusion problem on finite sets running in amortised time* $\mathrm{O}(n)$ *for each insert operation, where* n *is the number of different elements in the set.*
2. *There is an incremental constraint solver for the* $\preceq$ *problem running in worst-case time* $\mathrm{O}(m \cdot T[\![\curlyvee]\!])$*, where* m *is the number of constraints and* $T[\![\curlyvee]\!]$ *is the time for computing* $\curlyvee$*.*

If we use the second of these results we get that there is an incremental solver for the set inclusion problem on finite sets running in worst-case $\mathrm{O}(m \cdot T[\![\cup]\!]) = \mathrm{O}(mn)$ time. This is also the worst-case time of $\mathsf{Insert}_{\subseteq}$. The only difference is that it has a better amortised running time.

6.4 Computing the Conditionals

The strategy from the previous section was to consider only the $\preceq$ constraints that arose in connection with the smallest current solution. The same strategy applies to the conditionals as well. It works for the same reasons as in the previous section: The conditions function is monotone. This means that if the constraints that are forced in the smallest solution constitutes an unsatisfiable set, so does every set of forced constraints.

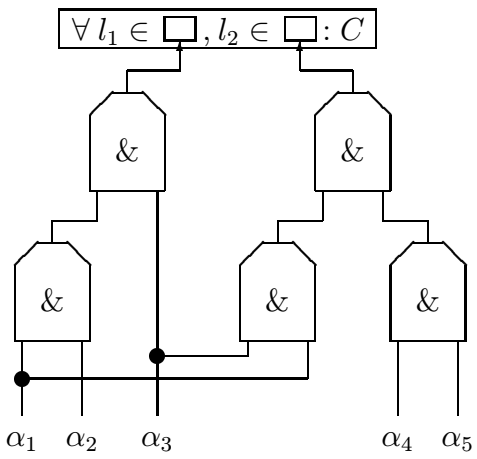


Figure 6.9: The &-network for $\forall l_1 \in \alpha_1 \,\&\, \alpha_2 \,\&\, \alpha_3, l_2 \in \alpha_1 \,\&\, \alpha_3 \,\&\, \alpha_4 \,\&\, \alpha_5 : C$.

It is quite easy to compute the constraint corresponding to a conditional constraint. The problem is computing χ_Q that is to compute $\mathrm{Def}(QC, \underline{\sigma})$ for a quantified constraint QC.

What we need to do is to compute the *new* constraints from $\mathrm{Def}(QC, \underline{\sigma})$. That is, we assume that we have computed the constraints from $\mathrm{Def}(QC, \underline{\sigma}')$ for some $\underline{\sigma}' \leqslant \underline{\sigma}$, and want to compute the remaining constraints $\mathrm{Def}(QC, \underline{\sigma}) \setminus \mathrm{Def}(QC, \underline{\sigma}')$.

Consider the quantified constraint

$$QC = \forall l_1 \in \alpha_1 \,\&\, \alpha_2 \,\&\, \alpha_3, l_2 \in \alpha_1 \,\&\, \alpha_2 \,\&\, \alpha_4 \,\&\, \alpha_5 : C$$

This quantified constraint represents a set consisting of one conditional constraint for each element of $1..|\alpha_1 \curlywedge \alpha_2 \curlywedge \alpha_3| \times 1..|\alpha_1 \curlywedge \alpha_3 \curlywedge \alpha_4 \curlywedge \alpha_5|$. Now, let $\alpha_1' \preceq \alpha_1$, $\alpha_2' \preceq \alpha_2$, etc. and assume that we have computed the constraints for the elements of $1..|\alpha_1' \curlywedge \alpha_2' \curlywedge \alpha_3'| \times 1..|\alpha_1' \curlywedge \alpha_3' \curlywedge \alpha_4' \curlywedge \alpha_5'|$. Now which are the new constraints we should compute? Clearly this depends on the relabeling function $\rho_{(\alpha_1 \curlywedge \alpha_2 \curlywedge \alpha_3) \preceq (\alpha_1' \curlywedge \alpha_2' \curlywedge \alpha_3)}$.

For simplicity we shall assume that there is an extension, $\top$, such that for all extensions, e, appearing in the solution, $e \preceq \top$. Now we define for each extension, e, the set $\mathrm{P}(e) = \{\rho_{e \leq \top}(i) | i \in 1..|e|\}$. Now, the transitivity rule of Definition 4.3.4 assures that whenever $e' \leq e'$ we have that $\rho_{e' \leq \top}(\rho_{e \leq e'}(i)) = \rho_{e \leq \top}(i)$ so that we can address the subterms by the numbers $\rho_{e \leq \top}(1)$, ..., $\rho_{e \leq \top}(|e|)$ instead of 1, ..., $|e|$, and be assured that these numbers never change throughout the running of the algorithm.

From the smoothness of ρ it follows that $\mathrm{P}(e \curlywedge e') = \mathrm{P}(e) \cap \mathrm{P}(e')$. So the problem of computing the conditional constraints reduces to the problem of computing

$$((\textstyle\bigcap_{i \in \{1,2,3\}} \mathrm{P}(\alpha_i)) \times (\bigcap_{i \in \{1,3,4,5\}} \mathrm{P}(\alpha_i))) \setminus ((\bigcap_{i \in \{1,2,3\}} \mathrm{P}(\alpha_i')) \times (\bigcap_{i \in \{1,3,4,5\}} \mathrm{P}(\alpha_i')))$$

To compute this efficiently we build an &-network consisting of an &-gate for each & in the quantified constraint. The &-network for QC is shown in Figure 6.9.

With this network in place we can compute the set above if we can perform two operations:

1. Given $\mathrm{P}(\alpha_1) \setminus \mathrm{P}(\alpha'_1)$, $\mathrm{P}(\alpha_2) \setminus \mathrm{P}(\alpha'_2)$, $\mathrm{P}(\alpha'_1)$, and $\mathrm{P}(\alpha'_2)$ we must compute $(\mathrm{P}(\alpha_1) \cap \mathrm{P}(\alpha_2)) \setminus (\mathrm{P}(\alpha'_1) \cap \mathrm{P}(\alpha'_2))$.

2. Given $\mathrm{P}(\alpha_1) \setminus \mathrm{P}(\alpha'_1)$, ..., $\mathrm{P}(\alpha_k) \setminus \mathrm{P}(\alpha'_k)$ and $\mathrm{P}(\alpha'_k)$, ..., $\mathrm{P}(\alpha'_k)$ we must compute $\prod_{i=1}^{k} \mathrm{P}(\alpha_i) \setminus \prod_{i=1}^{k} \mathrm{P}(\alpha'_i)$.

This is made easier by the realisation that we only need to look at sets in certain forms. In the $\mathsf{Insert}_{\preceq}$ procedure (Figure 6.8) all changes were of the form $D[w] := D[w] \curlyvee D[i]$. We get from the smoothness of ρ that $\mathrm{P}(e \curlyvee e') = \mathrm{P}(e) \cup \mathrm{P}(e')$. So we only have three cases for each input to the &-gates:

1. $\alpha_1 = \alpha'_1 \cup \beta$, $\alpha_2 = \alpha'_2 \cup \beta$

2. $\alpha_1 = \alpha'_1 \cup \beta$, $\alpha_2 = \alpha'_2$ (or vice versa)

3. $\alpha_1 = \alpha'_1$, $\alpha_2 = \alpha'_2$

The last case is trivial. The first two cases are dealt with by using the following identities:

1. $(\alpha'_1 \cup \beta) \cap (\alpha'_2 \cup \beta) \setminus (\alpha'_1 \cap \alpha'_2) = (\beta \setminus \alpha'_1) \cup (\beta \setminus \alpha'_2)$

2. $((\alpha'_1 \cup \beta) \cap \alpha'_2) \setminus (\alpha'_1 \cap \alpha'_2) = (\beta \setminus \alpha'_1) \cap \alpha'_2$

In this way we minimise the extra work needed to compute the new sets. In fact, if we implement the α'_i as bitvectors the computation time is linear in the size of the input. (Keep in mind that the input to the &-gate is $\beta \setminus \alpha'_1$ and $\beta \setminus \alpha'_2$.) That is, the time for performing the computation in a whole network is at most $|\beta|$ times the size of the network.

In a similar way we compute the product $\prod_{i=1}^{k} \mathrm{P}(\alpha_i) \setminus \prod_{i=1}^{k} \mathrm{P}(\alpha'_i)$. Define $S \subseteq 1..k$ such that $i \in S \Rightarrow \alpha_i = \alpha'_i \cup \beta$ and $i \notin S \Rightarrow \alpha_i = \alpha'_i$. The product is computed using the following identity:

$$\prod_{i=1}^{k} \mathrm{P}(\alpha_i) \setminus \prod_{i=1}^{k} \mathrm{P}(\alpha'_i) = \bigcup_{i \in S} \left(\prod_{j=1}^{i-1} \alpha_j \times (\beta \setminus \alpha'_i) \times \prod_{j=i+1}^{k} \alpha_j \right)$$

This can be computed in linear time in the size of $\prod_{i=1}^{k} \mathrm{P}(\alpha_i) \setminus \prod_{i=1}^{k} \mathrm{P}(\alpha'_i)$.

6.5 Graph algorithms

For use in the other algorithms we need two graph algorithms. One is for a heuristic dealing with inequalities in Section 7.4, the other is for detecting when the constraint set has no finite solutions.

6.5.1 Strongly Connected Components

Let $G = (V, E)$ be a directed graph. Two nodes $v, w \in V$ are *strongly connected* G iff there is a path from v to w and a path from w to v in G. It is easy to show that strong connectivity is an equivalence relation. The strongly connected components of G are the equivalence classes of the strong connectivity relation.

The algorithmic problem for this section is the maintenance of strongly connected components under the insertion of edges. We require the following operations.

$\mathsf{insert}(v, w)$ Insert the edge (v, w) into G. If the insertion creates a new connected component return a list of the old connected components which comprise the new connected component.

$\mathsf{connected}(\mathsf{v}, \mathsf{w})$ Returns true iff v and w are strongly connected.

We expect the list of connected components to be represented as a list of canonical elements from the connected components.

It is immediate that this algorithm is related to the transitive closure algorithm. Let $G^* = (V, E^*)$ be the reflexive, transitive closure of G and let $G^T = (V, E^T)$ be the transposed of G, that is the graph with all the edges reversed. Then we have the v and w is strongly connected iff $(v, w) \in E^*$ and $(w, v) \in E^*$. Since $(w, v) \in E^*$ iff $(v, w) \in (E^T)^*$, we have that strong connectivity is the relation defined by $E^* \cap (E^T)^*$. What we need to do then is to maintain the transitive closure of G and G^T together with a union-find data structure of the strongly connected components.

The insert procedure proceeds by inserting the edge (v, w) into G and G^T and performing transitive closure algorithm for both graphs. Then for all nodes i and j such that (i, j) has been inserted into E^* or $(E^T)^*$ and $(i, j) \in E^* \cap (E^T)^*$ after the insertion of (v, w) we do the following: If $\mathsf{Find}(i) \neq \mathsf{Find}(j)$, we add $\mathsf{Find}(i)$ and $\mathsf{Find}(j)$ to the set of canonical elements to be returned and call $\mathsf{Union}(i, j)$.

All this can be performed by two calls to the transitive closure algorithm plus a number of calls to the union-find algorithm which is linear in the work done by the transitive closure algorithm. We have then that the amortised complexity of the strongly connected components algorithm is $O(|V|)$ per insert and $O(1)$ for

each call to connected as for the transitive closure algorithm of La Poutré and van Leeuwen [43].

6.5.2 Maintaining L-acyclic graphs

Let $G = (V, E)$ be a directed labelled graph, where all edges are labelled with either a label from a set L or with the empty string $\epsilon \notin L$. Furthermore, we assume that for all edges $v \xrightarrow{\epsilon} w \in E$ there is also an edge $w \xrightarrow{\epsilon} v \in E$. Hence we shall write the ϵ-edges as $v \xleftrightarrow{\epsilon} w$.

Now, let us define the transitive closure of a labelled graph as follows.

- For all $v \in V$, $v \xrightarrow{\epsilon}{}^{*} v$.
- If $u \xrightarrow{\alpha}{}^{*} v$ and $v \xrightarrow{\epsilon} w$ then $u \xrightarrow{\alpha}{}^{*} w$.
- If $u \xrightarrow{\alpha}{}^{*} v$ and $v \xrightarrow{l} w$ then $u \xrightarrow{\alpha l}{}^{*} w$.

It follows that if $v \xrightarrow{\epsilon}{}^{*} w$ then $w \xrightarrow{\epsilon}{}^{*} v$. Hence we shall write $v \xleftrightarrow{\epsilon}{}^{*} w$.

An L-cycle in G is a path $v \xrightarrow{\alpha} v$ where $\alpha \neq \epsilon$. The algorithmic task is that of maintaining whether G has L-cycles under insertions of ϵ- and L-edges. Since the property of being L-cyclic is monotone all we need to do is to recognise if the insertion of an edge makes an otherwise L-acyclic graph L-cyclic. Let us consider the possibilities:

- Insert $v \xrightarrow{l} w$. If $w \xrightarrow{\alpha}{}^{*} v$ for some α (possibly ϵ) before the insertion of the edge, the graph is now L-cyclic. Otherwise it remains L-acyclic.
- Insert $v \xleftrightarrow{\epsilon} w$. If $v \xleftrightarrow{\epsilon}{}^{*} w$ before the insertion of the edge, the graph remains acyclic. Now, assume that it is not the case that $v \xleftrightarrow{\epsilon}{}^{*} w$. If $v \xrightarrow{\alpha}{}^{*} w$ or $w \xrightarrow{\alpha}{}^{*} v$, it must then be the case that $\alpha \neq \epsilon$ and hence the graph becomes L-cyclic. Otherwise it remain L-acyclic.

It follows from the above that all we need to do is to maintain the equivalence relation $\xleftrightarrow{\epsilon}{}^{*}$ and the transitive closure of G. Using the Union-Find algorithm and La Poutré and van Leeuwens transitive closure algorithm (since the labels turn out to be immaterial) we get an amortised running time of $O(|V|)$ for each insert.

6.6 The Constraint Satisfaction Algorithm

The constraint satisfaction algorithm is simply the combination of the elements from the previous sections into a single algorithm. Let us first show how to

implement the algorithm without equality and then how to combine the implementation with equality.

In the design of the algorithm we take as a starting point the consistency algorithm from Section 6.2. We reuse the data structures from the original algorithm. The domain assignments, support sets, and prune queue are as above. The adjacency lists contains elements of the form $(\langle\!\langle R, \chi \rangle\!\rangle(\underline{v}), i)$ instead of $(R(\underline{v}), i)$, and the χ is implemented as a pointer for each tuple in R to an array of quantified or conditional constraints.

In addition to the data structures from the consistency algorithm we also have the base automaton from the unification algorithm. In the equality-free case we have that each state in the automaton is a singleton, and we can maintain L simply by letting $L(\{v\}) = D(v)$. The final thing we need is the constraint set, $\mathcal{C}$. This constraint set no longer holds only equality constraints, it can hold any kind of constraint from Γ^{OIH}.

It is quite straightforward to deal with a constraint of the form $\langle\!\langle R, \chi \rangle\!\rangle(e_1, \ldots, e_n)$. We start by computing the new base automaton $\mathsf{base}(\mathsf{base}(\ldots \mathsf{base}(M, e_1), \ldots), e_n)$. Then we let $\mathcal{C}$ be $\{\langle\!\langle R, \chi \rangle\!\rangle(v_{e_1}, \ldots, v_{e_n})\}$ and proceed in the following way until $\mathcal{C} = \emptyset$:

1. Take a constraint $\langle\!\langle R, \chi \rangle\!\rangle(v_1, \ldots, v_n) \in \mathcal{C}$ and remove it from $\mathcal{C}$.

2. Insert $\langle\!\langle R, \chi \rangle\!\rangle(v_1, \ldots, v_n)$ in the adjacency lists and run the normal consistency algorithm with respect to the R-part only.

3. For each relation $\langle\!\langle R', \chi' \rangle\!\rangle(v'_1, \ldots, v'_m)$ for which at least one of the domains $D(v'_i)$ has changed, compute $\chi'(\bigwedge D(v'_1), \ldots, \bigwedge D(v'_m))$, and *release* the constraints in the following way. For each conditional constraint (that has not been released previously) C replace the pseudo-variables of the form $\langle i/\alpha \rangle$ with v'_i/α defined by $\delta^*(\{v'_i\}, \alpha) = \{v'_i/\alpha\}$. Insert the resulting constraint into $\mathcal{C}$.

The above procedure does not describe relations of the form $\langle\!\langle R_Q, \chi_Q \rangle\!\rangle$. To do this we need to use the algorithms described in Sections 6.3 and 6.4. The algorithm in Section 6.3 is $\mathsf{Insert}_{\preceq}$. We use it by releasing the constraints from $Q(\bigwedge D(v'_1), \ldots, \bigwedge D(v'_m))$ in exactly the same way as in step 3 above. The algorithm described in Section 6.4 is used to compute the conditional constraints which has not previously been released.

We deal with constraints $\nu X.\langle\!\langle R, \chi \rangle\!\rangle(e_1, \ldots, e_n)$ in much the same way as above, but there are two differences:

- Instead of inserting $\nu X.\langle\!\langle R, \chi \rangle\!\rangle(v_1, \ldots, v_n)$ into the adjacency lists we insert $\langle\!\langle R, \chi[X \leftarrow \nu X.\langle\!\langle R, \chi \rangle\!\rangle] \rangle\!\rangle$. That is we unfold the definition once.

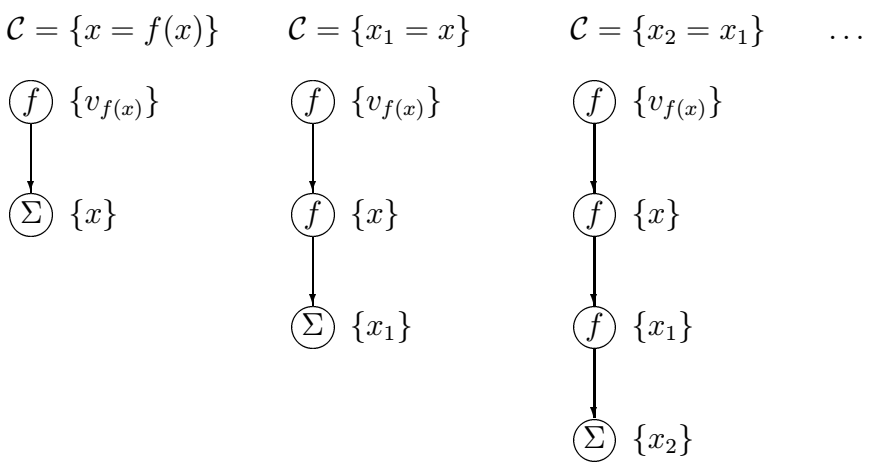


Figure 6.10: Successive approximations to $x = f(x)$.

- To stop the potentially infinite releasing of constraints we keep for each recursively defined constraint a search tree of the tuple of variables for which it has previously been released. We never release the same recursively defined constraint twice.

The size of the search trees are at most $|\mathcal{V}|^n$ hence the time for accessing the search trees is at most $\mathrm{O}(n \cdot \log(|\mathcal{V}|))$.

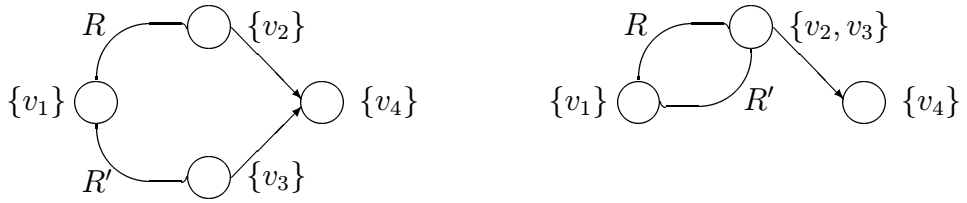
The process above shows how important it is that condition functions in recursive definitions are strict. Since there is a finite number of ways a recursive constraint can be released on existing variables, sooner or later it will be released in the form $\nu X.\langle\!\langle R, \chi \rangle\!\rangle(v_1, \ldots, v_n)$ where at least one v_i is new. Assume for now that this is the only remaining constraint. Since $D(v_i) = \Sigma$ we have that after the pruning $\bigwedge D(v_i)$ becomes the least σ such that $R(\bigwedge D(v_1), \ldots, \sigma, \ldots, \bigwedge D(v_n))$. Hence we have that

$$\chi(\bigwedge(\prod_j D(v_j))) = \chi(\bigwedge(\Sigma \times \prod_{j \neq i} \{\bigwedge D(v_j)\})) = \emptyset$$

by the strictness of χ and the procedure terminates. If there are more than one constraint the matter is more complicated as we shall see.

Consider by contrast the unstable relation $=$. The instability stems from the non-strictness of the condition function. This means that if we try to solve the equation $x = f(x)$ by the procedure described above we would continue to get new conditional constraints over new variables and get an infinite succession of approximations as shown in Figure 6.10.

This is the reason we need to unify the two states $\{v_{f(x)}\}$ and x. The result of this is a state of the form $\{x, v_{f(x)}\}$, and we have lost the property that all states are



(a) (b)

Figure 6.11: Unifying v_2 and v_3 with $D = L$. (a) Before unification. (b) After unification.

singleton sets. The problem now is how to combine the data structures in these circumstances. The idea from before that $L(\{v\}) = D(v)$ allowed us to identify L and D. If we continue with this idea we arrive at an algorithm in which we need to rearrange the adjacency sets every time we unify two states as shown in Figure 6.11.

The figure illustrates how the automaton change under the unification of v_2 and v_3. For the new node we must have $L[\{v_2, v_3\}] = L[\{v_2\}] \cap L[\{v_3\}]$ and its adjacency set must contain all the constraints that used to be in the old adjacency sets for $\{v_2\}$ and $\{v_3\}$. Hence we need to move the relations from $E[\{v_3\}]$ to $E[\{v_2, v_3\}]$. In the worst case we have that $|E[\{v_3\}]|$ is proportional to the number of inserted constraints. Since this can be done once for each variable we may move as many as $\Theta(|\mathcal{C}| \cdot |\mathcal{V}|)$ relations.

There is a more efficient way of doing this. If we abandon the idea that $L = D$ we get two different kinds of functions, $L : Q \to \Sigma$ and $D : \mathcal{V} \to \Sigma$. The relation between them can be expressed as $D(v) = L(Q_v)$ and $L(q) = D(\lceil q \rceil)$. It follows that for each $q \in Q$ we have that for all $v, v' \in q$, $D(v) = D(v')$. In order to assure this we introduce the *diagonal constraint*, $\Delta(v, v')$. The relation Δ is defined as $\langle\!\langle R_Q, \chi \rangle\!\rangle$, where $R = \{(\sigma, \sigma) | \sigma \in \Sigma\}$, $Q(\sigma, \sigma) = \{1 \preceq_\sigma 2, 2 \preceq_\sigma 1\}$ if $\sigma \in \mathrm{dom}(\mathcal{E})$ and $Q(\sigma, \sigma) = \emptyset$ otherwise, and $\chi(\sigma, \sigma) = \emptyset$. If D is arc consistent with respect to $\Delta(v, v')$ then $D(v) = D(v')$.

Using the diagonal constraint we can unify two variables much easier than above. We unify the states as in the unification procedure described in Section 6.1, but we keep the domain as it is. In order to rectify the domain we insert the constraint $\Delta(v_2, v_3)$ as illustrated in Figure 6.12. The circles in this figure are the domains, the double circles illustrate the canonical elements of the states. We have drawn the transitions as going between the canonical elements.

This method is more efficient than the first one. All the extra work we need to do is to add one constraint for each call to Unify which change the automaton. Since there can be at most $|\mathcal{V}| - 1$ calls to Unify which change the state there can

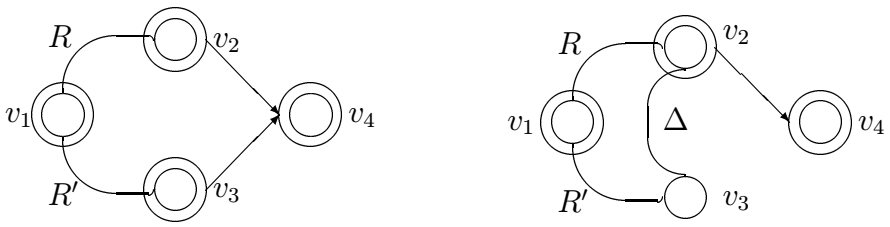


Figure 6.12: Unifying v_2 and v_3 with $D \neq L$. (a) Before unification. (b) After unification.

be at most $|\mathcal{V}| - 1$ extra constraints.

The correctness of the procedure follows by a proof similar to the proof of the correctness of Unify. All we need to do is note that the set of assignments which satisfy all the constraints in the adjacency sets and $\mathcal{C}$, and is in the solution-space of the base automaton is invariant. Then the result follows like above.

Thus we have the following proposition.

Proposition 6.6.1 *Let $\mathcal{C}$ be a set of constraints and let $M = (Q, \mathcal{P}(\Sigma), L, \Delta)$ be the base automaton resulting from running the algorithm described above on $\mathcal{C}$. We have the following.*

1. *If $L(q) = \emptyset$ for some $q \in Q$ then $\mathcal{C}$ is unsatisfiable.*
2. *Assume $L(q)$ is non-empty for all $q \in Q$, and let $L_{\min} : Q \to \Sigma$ be defined by $L_{\min}(q) = \{\bigwedge L(q)\}$. Then $\mathcal{S}_{(Q,\mathcal{P}(\Sigma),L_{\min},\Delta)}$ is the $\leqslant_\Sigma$-least solution to $\mathcal{C}$.*

If the algorithm terminates in a state where all the $L(q)$ are non-empty, we say that the solver *succeeds*. If it terminates in any other state it *fails*.

The above proposition shows that if the algorithm terminates, it correctly solves the problem for the constraint domain OIH. In order to solve the problem for OH we must make some definitions.

For all variables v and w we say that $v \overset{\epsilon}{\longleftrightarrow} w$ if $Q_v = Q_w$, and for states q and q' we say that $\lceil q \rceil \xrightarrow{i} \lceil q' \rceil$ if $\Delta(q, i) = q'$. The base automaton is cyclic iff the resulting graph is ω-cyclic (see Section 6.5.2). It follows that there is a finite solution to the constraint set iff the graph is ω-acyclic.

We do not need to insert all the ϵ-edges in the definition above. It is enough that the reflexive, transitive closure has the property described. We can accomplish this by inserting an ϵ-edge in UnifyVar each time $Q_{v_1} \neq Q_{v_2}$. In this way the total running time of the ω-cycle algorithm is kept at $\mathrm{O}(|\mathcal{V}|^2)$.

Using this extension we have turned the OIH constraint solver into a constraint solver for OH. All we need to do is to fail when a ω-cycle occurs, and proceed like above if no such cycle occurs.

This leaves the question of the termination of the algorithm. As we saw above the important moment arrives when we have only recursively defined constraints left and we generate a new variable. In the case discussed above we assumed that there was only one recursive constraint, but assume that we have several recursive constraints in the queue. In that case we have to look at the result of applying arc consistency to all the constraints, and in this case it is not enough that each of the constraints are strict, they have to be strict when seen as a whole.

Now let $\mathcal{C}$ be a set of recursively defined constraints and assume that the constraints are over the same variables . That is, each constraint belonging to $\mathcal{C}$ is of the form $\nu X.\langle\!\langle R_i, \chi_i \rangle\!\rangle(v_1, \ldots, v_n)$, where $1 \leq i \leq m$. When we look at arc consistency we can see that arc consistency for the constraint set corresponds to arc consistency for the single constraint $(\bigcap_{i=1}^{m} R_i)(v_1, \ldots, v_n)$. Hence, the strictness requirement must be extended to

$$\forall i \in 1..m, \forall S \subsetneq 1..n,\, \sigma{:}\, S \to \Sigma : \chi_i(\bigwedge((\bigcap_{j=1}^{m} R_j) \cap (\prod_{j \in S}\{\sigma(j)\} \times \prod_{j \notin S} \Sigma))) = \emptyset$$

in order to ensure the termination of the algorithm. When a constraint set is like the above we call it *collectively strict*.

It is easy to see that any constraint $\nu X.\langle\!\langle R, \chi \rangle\!\rangle(v_1, \ldots, v_n)$ is equivalent to the constraint $\nu X.\langle\!\langle R \times \Sigma, \chi' \rangle\!\rangle(v_1, \ldots, v_n, v)$, where $\chi'(\sigma_1, \ldots, \sigma_n, \sigma) = \chi(\sigma_1, \ldots, \sigma_n)$. Since we also have that our constraint domain is closed under permutation of the variables, we have that any set of constraints can be brought on the form described above. We say of an arbitrary set of recursively defined constraints that it is collectively strict, iff it is collectively strict when brought on the form above.

The idea of termination is to ensure that the constraint set is collectively strict. The following definition shows a way of doing just that.

Definition 6.6.1 *A set of recursively defined relations, Γ, is* strictness-closed*, iff all sets of constraints from $\mathcal{P}(\Gamma[\mathcal{V}, \emptyset])$ are collectively strict.*

Now, we can formulate the main theorem.

Theorem 9 *Let $(\Sigma, \leqslant)$ be applicable.*

1. *Let $\mathcal{C} \subseteq \mathsf{OIH}_{(\Sigma,\leqslant)}[\mathcal{V}]$ be such that the set of recursively defined relations in $\mathcal{C}$ is strictness-closed. Then the constraint solver for* OIH *succeeds if $\mathcal{C}$ is satisfiable in $\mathsf{OIH}_{(\Sigma,\leqslant)}$, and fails if $\mathcal{C}$ is unsatisfiable in $\mathsf{OIH}_{(\Sigma,\leqslant)}$*

2. *Let $\mathcal{C} \subseteq \mathsf{OH}_{(\Sigma,\leqslant)}[\mathcal{V}]$ be such that the set of recursively defined relations in $\mathcal{C}$ is strictness-closed. Then the constraint solver for OH succeeds if $\mathcal{C}$ is satisfiable in $\mathsf{OH}_{(\Sigma,\leqslant)}$, and fails if $\mathcal{C}$ is unsatisfiable in $\mathsf{OH}_{(\Sigma,\leqslant)}$*

The running time of the algorithm is very problem-dependent. The reason is that certain relations may lead to the release of a very large number of constraints. In order to get a bound on the running time we have to look at the total number of released constraints. So let $\mathcal{C}$ be the inserted constraints and $\mathcal{C}'$ released constraints. Let the size of a constraint be defined by the following.

- $\text{size}(R(e_1, \ldots, e_n)) = \text{size}(R) + \sum_{i=1}^{n} \text{size}(e_i)$
- $\text{size}(=) = 1$
- $\text{size}(\langle\!\langle R_Q, \chi_Q \rangle\!\rangle) = \text{size}(\nu X.\langle\!\langle R_Q, \chi_Q \rangle\!\rangle) =$
  $\text{rank}(R) \cdot |R| + \sum_{\underline{\sigma} \in R} |Q(\underline{\sigma})| + \sum_{\underline{\sigma} \in R} |\chi(\underline{\sigma})|$
- $\text{size}(v) = 1$
- $\text{size}(\sigma(e_1, \ldots, e_k)) = 1 + \sum_{i=1}^{k} size(e_i)$

Then we have the following.

Proposition 6.6.2 *Let m be the total number of released $\preceq$-constraints and n the size of the greatest extension in the solution.*

1. *The total running time of the OIH constraint solver is*

$$\text{O}(\sum_{C \in \mathcal{C} \cup \mathcal{C}'} \text{size}(C) + \sum_{\nu X.\langle R,\chi \rangle \in \mathcal{C}} \text{rank}(R) \cdot \log(|\mathcal{V}|) + m \cdot n)$$

2. *The total running time of the OH constraint solver is*

$$\text{O}(\sum_{C \in \mathcal{C} \cup \mathcal{C}'} \text{size}(C) + \sum_{\nu X.\langle R,\chi \rangle \in \mathcal{C}} \text{rank}(R) \cdot \log(|\mathcal{V}|) + m \cdot n + |\mathcal{V}|^2)$$

In the above we have assumed that all the extension domains are sets.

It is usually the case that the time from the search tree operations are negligible. Furthermore, it is often the case that the term is dominated by either the $\preceq$-constraints or the other constraints. So in many of the practical cases the time reduces to one of the following:

1. $\text{O}(\sum_{C \in \mathcal{C} \cup \mathcal{C}'} \text{size}(C))$
2. $\text{O}(m \cdot n)$

The precise analysis will have to be performed for each application.

6.7 Deciding Applicability and Stability

It makes no sense to define the constraint domain $\mathsf{OIH}_{(\Sigma,\leqslant)}$ unless $(\Sigma, \leqslant)$ is applicable. Similarly, many parts of the constraint satisfaction algorithm are quite meaningless if the relations are unstable. Fortunately, these properties are decidable as we shall see.

Lemma 6.7.1 *Let Σ be a finite set. Given a partial ordering, $\sqsubseteq$ over Σ it is decidable whether $\sqsubseteq$ forms a lower semi-lattice.*

Proof: Since Σ is finite we can make a topological sort of Σ. Assume that Σ is sorted as $\sigma_1 < \sigma_2 < \cdots < \sigma_n$. Then we can for each pair of values σ_i and σ_j compute the set

$$S = \{\sigma_k | \sigma_k \sqsubseteq \sigma_i \wedge \sigma_k \sqsubseteq \sigma_j\}$$

Now, let k be the largest number such that $\sigma_k \in S$. Now it should be the case for all $\sigma \in S$ that $\sigma \sqsubseteq \sigma_k$. This can be tested by simple enumeration. □

The following is an important corollary.

Corollary 6.7.2 *Let $(\Sigma, \sqsubseteq)$ be a finite lower semi-lattice. The greatest lower bound is computable.*

Proof: σ_k in the proof of Lemma 6.7.1 is the greatest lower bound of σ_i and σ_j. □

In fact, the implementation of the algorithm uses a topological sort of Σ to compute the greatest lower bound.

Lemma 6.7.3 *Let $(\Sigma, \sqsubseteq)$ be a finite lower semi-lattice, and let* rank *be a function* $\text{rank} : \Sigma \to \omega$. *The following are decidable.*

1. rank *is monotone.*
2. $\text{rank}(\bigwedge \Sigma) = 0$.

Proof:

1. Trivial by enumerating all possibilities.
2. Trivial using Corollary 6.7.2.

□

We now have the proposition we looked for.

Proposition 6.7.4 *Let Σ be a ranked alphabet, $\sqsubseteq$ an ordering over Σ, and $\mathcal{E}$ a partial function with* $\mathrm{dom}(\mathcal{E}) \subseteq \Sigma$. *Furthermore, assume that each $\mathcal{E}(\sigma)$ is a recursively enumerable set of extensions equipped with a complete lattice, $\preceq_\sigma$ and a monotone size function $|\cdot|_\sigma$, such that $|\bot_{\mathcal{E}(\sigma)}| = 0$. It is decidable whether the ordering $(\Sigma_\mathcal{E}, \sqsubseteq_\preceq)$ is applicable.*

Proof: Follows from Definition 4.6.1 and Lemmas 6.7.1 and 6.7.3. □

In the proposition above it was assumed that we had suitable sets of extensions. One such set is the power set of a finite set equipped with the subset ordering and a size function which is either $|a|_\sigma = 0$ or $|a|_\sigma = |a|$, where $|a|$ is the number of elements in a.

Lemma 6.7.5 *Let $(\Sigma, \sqsubseteq)$ be a finite lower semi-lattice, and let $R \subseteq \Sigma^n$. It is decidable whether R is stable with respect to $\sqsubseteq$.*

Proof: For each pair of tuples $(\sigma_1, \ldots, \sigma_n)$ and $(\sigma'_1, \ldots, \sigma'_n)$ from R, we can compute $(\sigma_1 \wedge \sigma'_1, \ldots, \sigma_n \wedge \sigma'_n)$ using Corollary 6.7.2. Now, it is straightforward to check whether this tuple is in R. □

The following lemma follows directly from Definition 4.4.5.

Lemma 6.7.6 *Let χ be an extended condition function. We have the following.*

1. *χ_Q is monotone if and only if χ is monotone.*
2. *χ_Q is strict if and only if χ is strict when restricted to unquantified constraints.*

Lemma 6.7.7 *Given a condition function it is decidable whether it is monotone.*

Proof: Let the condition function be χ_Q. By lemma 6.7.6(1) it is sufficient to check whether χ is monotone. But χ has finite domain and it is then easy to check monotonicity by enumerating the possibilities. □

Lemma 6.7.8 *Given a condition function it is decidable whether it is strict.*

Proof: Let the condition function be $\chi_Q : R_Q \to \Gamma[\Psi, \emptyset]$. By Lemma 6.7.6(2) it is sufficient to look at $\chi' : R \to \Gamma[\Psi, \emptyset]$ where χ' is χ restricted to unquantified constraints. That is, it is sufficient to decide whether

$$\forall S \subsetneq 1..n,\ \sigma{:}\, S \to \Sigma : \chi(\textstyle\bigwedge(R \cap (\prod_{i \in S}\{\sigma(i)\} \times \prod_{i \notin S} \Sigma))) = \emptyset \qquad (6.1)$$

Since there is a finite number of sets $S \subsetneq 1..n$ and a finite number of functions $\sigma{:}\, S \to \Sigma$, we can enumerate these, and since $\bigwedge$ is computable by Corollary 6.7.2 we can decide whether condition (6.1) is met. □

Proposition 6.7.9 *Given a relation of the form* $\langle\!\langle R_Q, \chi_Q \rangle\!\rangle$ *or* $\nu X.\langle\!\langle R_Q, \chi_Q \rangle\!\rangle$ *it is decidable whether the relation is stable.*

Proof: Follows from Lemmas 6.7.5, 6.7.7, and 6.7.8. □

Note that the above does not imply that it is decidable whether or not the constraint satisfaction algorithm terminates. It is not known if it is decidable whether a set of relations are strictness-closed.

Chapter 7

The CLP(OIH) Language

Theoretically, the CLP(OIH) language is defined by the general semantics of Section 3.2, but some aspects are still unresolved. This chapter is about the actual version of CLP(OIH). We discuss the precise definition of the domain, then we proceed with a language question which arises from the introduction of generic labels, and finally we present a manual for using the interpreter.

7.1 User-defined Domains

The first task in writing a program is defining the domain. The domain of course is either $\mathsf{OIH}_{(\Sigma,\leqslant)}$ or $\mathsf{OH}_{(\Sigma,\leqslant)}$ for some suitable Σ and $\leqslant$. Thus we need ways to distinguish OIH from OH and to define a ranked alphabet and an ordering. These tasks are handled by the keywords `Finite`, `Labels`, and `Ordering`. The definition of the domain follows the following syntax.

$$\begin{array}{rcl} D & ::= & \texttt{Labels}\ L\ \texttt{Ordering}\ O \mid \texttt{Finite Labels}\ L\ \texttt{Ordering}\ O \\ L & ::= & \mathit{labeldefinition}. \mid \mathit{labeldefinition}, L \\ O & ::= & \mathit{inequality}. \mid \mathit{inequality}, O \end{array}$$

Each label definition is either on the form σ/k or $\sigma[a]/s$. In the first case σ is a non-generic label of rank k in the second it is a generic label. The only extension allowed is sets of strings (e.g. a set of field names) and the size function is either $|e|_\sigma = 0$ or $|e|_\sigma = |e|$ where $|e|$ is the number of elements in the set. There are thus two possibilities for the form of the expression s in $\sigma[a]/s$. If $s \equiv |a| + k$ we have that $|e|_\sigma = |e|$, but if $s \equiv k$ we have that $|e|_\sigma = 0$. In both cases σ has rank k. We allow $|a|$ as a shorthand for $|a| + 0$.

The basic form of the orderings is $\sigma < \sigma'$. It is understood that $\sqsubseteq$ is the reflexive, transitive closure of the ordering thus defined. In this simple case all

relabeling functions are simply the inclusion function $\rho_{\sigma \sqsubseteq \sigma'}(i) = i$. If another relabeling function is required it can be explicitly defined by an ordering of the form $\sigma(v_{i_1}, \ldots, v_{i_k}) < \sigma'(v_1, \ldots, v_{k'})$. The relabeling function thus defined is $\rho_{\sigma \sqsubseteq \sigma'}(i_j) = j$. The ordering of the extensions is set inclusion and the relabeling function for the extensions is arbitrary.

As an example let us look at the definition of Σ^{st} with some ordering. We define it as follows

```
Finite

Labels

    int/0,
    bool/0,
    prod[a]/|a|,
    sum[a]/|a|,
    list/1,
    arrow/2.

Ordering

    bool < int,
    int < prod,
    int < sum,
    int < list.
```

If we want to infer types for $\mathsf{Reg}_{\Sigma^{\mathsf{st}}}$ rather than $\mathsf{Fin}_{\Sigma^{\mathsf{st}}}$ we just remove the `Finite` keyword from the beginning of the definition.

Given an alphabet as above we can write down terms in the usual way, e.g. `arrow(list(bool), int)`. Where generic labels are concerned there are four ways of writing down terms depending on the rank and the size function.

1. If σ is defined as $\sigma[a]/0$, the terms are on the form $\sigma[a_1, \ldots, a_n]$.
2. If σ is defined as $\sigma[a]/k$ where $k > 0$, the terms are on the form $\sigma[a_1, \ldots, a_n](t_1, \ldots, t_k)$.
3. If σ is defined as $\sigma[a]/|a|$, the terms are on the form $\sigma[a_1 : t_1, \ldots, a_n : t_k]$.
4. If σ is defined as $\sigma[a]/|a| + k$ where $k > 0$, the terms are on the form $\sigma[a_1 : t_1, \ldots, a_n : t_n](t'_1, \ldots, t'_k)$.

In the terms above, the a_i stand for strings and the t_i and t'_i stand for terms.

7.2 CLP and the semantics of ellipsis

The introduction of generic labels in the alphabet raises the issue of infinite CLP programs. What we need is a way to express a uniform behaviour of the program for all instances of the generic label. In everyday language this is expressed by the use of ellipsis. We shall use a similar approach here.

Consider the following type rule from [59]:

$$\frac{\Gamma \vdash e_1 : \omega_1 \quad \ldots \quad \Gamma \vdash e_n : \omega_n}{\Gamma \vdash \langle i_1 : e_1, \ldots, i_n : e_n \rangle : \mathsf{prod}(i_1 : \omega_1, \ldots, i_n : \omega_n)}$$

In CLP (or Prolog) we might define a predicate $\mathtt{Type}(\Gamma, e, \omega)$ standing for $\Gamma \vdash e : \omega$. The above rule would then be implemented as

$$\mathtt{Type}(\Gamma, \mathtt{bracket}[\mathtt{i}_1 : e_1, \ldots, \mathtt{i}_n : e_n], \mathtt{prod}[\mathtt{i}_1 : \omega_1, \ldots, \mathtt{i}_n : \omega_n]) \ \mathtt{:-} \ \mathtt{Type}(\Gamma, e_1, \omega_1), \ldots, \mathtt{Type}(\Gamma, e_n, \omega_n).$$

The above is not a CLP rule in the usual sense. Only when the ellipsis are concretised for a fixed n can the above be written as a rule. Furthermore, if we understand the $\mathtt{i}_\mathtt{j}$ as part of the label these will have to be fixed as well before we can write the above as a CLP rule. Since there are infinite possibilities for fixing n and the $\mathtt{i}_\mathtt{j}$ the "pseudo-rule" above corresponds to an infinite set of rules.

The goal of this section is to define a syntax and semantics of ellipsis so that the pseudo-rule above is expressible in CLP. The first step is to write this in a simpler form where the indexes are absent:

$$\mathtt{Type}(\Gamma, \mathtt{bracket}[\mathtt{..i} : e \mathtt{..}], \mathtt{prod}[\mathtt{..i} : \omega \mathtt{..}]) \ \mathtt{:-} \ \mathtt{Type}(\Gamma, e, \omega).$$

In the above expression the *bound* variables e and ω are related in the sense that there is a one-to-one correspondence between the es and the ωs. This correspondence is defined by their common *context*, the field variable $\mathtt{i}$. The context thus becomes important when interpreting the expression $\mathtt{Type}(\Gamma, e, \omega)$ where e and ω occur freely. The field variable ranges over all possible sets of field names. The programming language obtained by introducing $\mathtt{..i} : x \mathtt{..}$ is CLP^∞.

For each set of field names the expression takes on a concrete form. For instance we have that for the set of field names $\mathtt{i} = \{\mathtt{a}, \mathtt{b}\}$ the rule has the form:

$$\mathtt{Type}(\Gamma, \mathtt{bracket}[\mathtt{a} : e^\mathtt{a}, \mathtt{b} : e^\mathtt{b}], \mathtt{prod}[\mathtt{a} : \omega^\mathtt{a}, \mathtt{b} : \omega^\mathtt{b}]) \ \mathtt{:-} \ \mathtt{Type}(\Gamma, e^\mathtt{a}, \omega^\mathtt{a}), \mathtt{Type}(\Gamma, e^\mathtt{b}, \omega^\mathtt{b}).$$

We note the following:

1. The bound occurrences of e and ω are replaced by the expressions $\mathtt{a} : e^\mathtt{a}, \mathtt{b} : e^\mathtt{b}$ and $\mathtt{a} : \omega^\mathtt{a}, \mathtt{b} : \omega^\mathtt{b}$, respectively.

2. The free occurrences of e and ω are replaced with $e^{\mathtt{a}}$, $e^{\mathtt{b}}$, $\omega^{\mathtt{a}}$, or $\omega^{\mathtt{b}}$ according to their contexts.

3. The *non-bound* variable, Γ, remains as is in the instantiation.

To see how the context influences the interpretation of the antecedents, consider the following rule.

$$\mathtt{P}(\mathtt{f}[\,..\mathtt{i} : x\,..], \mathtt{f}[\,..\mathtt{j} : y\,..]) \;\mathtt{:-}\; \mathtt{P}(x, y).$$

Here x and y are unrelated (having different contexts) and it is no longer meaningful to assign one predicate to each member of the field set. Instead there will be a predicate for each *combination* of elements from the *two* field sets.

As an example let us consider the field set assignment which lets $\mathtt{i} = \mathtt{j} = \{\mathtt{a}, \mathtt{b}\}$. The instantiation of the rule is

$$\mathtt{P}(\mathtt{f}[\mathtt{a} : x^{\mathtt{a}}, \mathtt{b} : x^{\mathtt{b}}], \mathtt{f}[\mathtt{a} : y^{\mathtt{a}}, \mathtt{b} : \mathtt{y}^{\mathtt{b}}]) \;\mathtt{:-}\; \mathtt{P}(x^{\mathtt{a}}, y^{\mathtt{a}}), \mathtt{P}(x^{\mathtt{b}}, y^{\mathtt{a}}), \mathtt{P}(x^{\mathtt{a}}, y^{\mathtt{b}}), \mathtt{P}(x^{\mathtt{b}}, y^{\mathtt{b}}).$$

The bound occurrences are instantiated in the same way as above, but since x and y have different contexts there is no relation between $x^{\mathtt{a}}$ and $y^{\mathtt{a}}$. As a result there is an antecedent for each element of $\{\mathtt{a}, \mathtt{b}\} \times \{\mathtt{a}, \mathtt{b}\}$. We can say that the context of the antecedent $\mathtt{P}(x, y)$ is the set $\{\mathtt{i}, \mathtt{j}\}$, the set of contexts of the bound variables occurring freely in $\mathtt{P}(x, y)$.

In order to understand how instantiation works in general, consider the following generic rule:

$$\mathtt{P}(e_1, \ldots, e_k) \;\mathtt{:-}\; \mathtt{Q_1}(\epsilon^1_1, \ldots, \epsilon^1_{m_1}), \ldots, \mathtt{Q_n}(\epsilon^n_1, \ldots, \epsilon^n_{m_n}).$$

where the $\mathtt{Q_i}$ are either predicates or constraints, and the e_i and ϵ^j_i may contain $..\mathtt{i} : x..$ or $..\mathtt{i}...$For simplicity, we shall only consider the expressions of the former kind, expression of the latter kind is similar.

Each variable, x, appearing as $..\mathtt{i} : x..$ is a bound variable, and the context of x is $\kappa(x) = \{\mathtt{i}\}$. No bound variable is allowed to appear in different contexts. That is, the rule cannot contain $..\mathtt{i} : x..$ and $..\mathtt{j} : x..$ for $\mathtt{i} \neq \mathtt{j}$. Furthermore, no bound variable may occur freely in the conclusion. The context of a non-bound variable is $\emptyset$.

The context function κ is extended to expressions as the set of contexts of the free variables. Formally we have:

- $\kappa(..\mathtt{i} : x..) = \emptyset$
- $\kappa(\mathtt{i} : e) = \kappa(e)$
- $\kappa(f[e_1, \ldots, e_k](t_1, \ldots, t_n)) = \left(\bigcup_{i=1}^{k} \kappa(e_i)\right) \cup \left(\bigcup_{i=1}^{n} \kappa(t_i)\right)$
- $\kappa(\mathtt{Q}(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} \kappa(t_i)$

With this we can write the constraint on the conclusion as $\kappa(\mathtt{P}(e_1,\ldots,e_k)) = \emptyset$.

The other important concept in the instantiation is the actual field set assignment. If Φ is a field set assignment and φ is a function from field variables to field names, we write $\varphi \in \Phi$, iff $\mathrm{dom}(\varphi) = \mathrm{dom}(\Phi)$ and $\forall x \in \mathrm{dom}(\varphi) : \varphi(x) \in \Phi(x)$. We now have the terminology to define the instantiation of the right-hand side by means of the *expansion* of an antecedent.

Definition 7.2.1 *Let $\mathtt{Q}(\epsilon_1,\ldots,\epsilon_n)$ be an antecedent containing free occurrences of the bound variables $x_1,\ldots,x_l$. The expansion of $\mathtt{Q}(\epsilon_1,\ldots,\epsilon_n)$ relative to a context function, κ, and a field set assignment, Φ, is the set*

$$E(\mathtt{Q}(\epsilon_1,\ldots,\epsilon_n),\kappa,\Phi) \\ = \{\mathtt{Q}(\epsilon'_1,\ldots,\epsilon'_n) | \exists\varphi \in \Phi : \forall i \in 1..n : \epsilon'_i = \epsilon_i[x_1 \leftarrow x_1^{\varphi(\kappa(x_1))},\ldots,x_l \leftarrow x_l^{\varphi(\kappa(x_1))}]\}$$

where the x_j^a are new variables.

The instantiated right-hand side thus consists of the union of the expansions of the antecedents in the uninstantiated rule. And the only remaining task is that of instantiating the bound occurrences of variables.

For this task we introduce the following notation: Let $a = \{\mathtt{a_1},\ldots,\mathtt{a_n}\}$ be a field set. We write $a : x^a$ for the list $\mathtt{a_1} : x^{\mathtt{a_1}},\ldots,\mathtt{a_n} : x^{\mathtt{a_n}}$.

Definition 7.2.2 *Let*

$$\mathtt{P}(e_1,\ldots,e_k) \ \mathtt{:-}\ \mathtt{Q_1}(\epsilon_1^1,\ldots,\epsilon_{m_1}^1),\ldots,\mathtt{Q_n}(\epsilon_1^n,\ldots,\epsilon_{m_n}^n).$$

be a CLP *rule containing the bound variables $x_1,\ldots,x_l$, and κ the context function defined by the rule. For an arbitrary field set assignment, Φ, let sub be the substitution*

$$[\,..\kappa(x_1) : x_1\,.. \leftarrow \Phi(\kappa(x_1)) : x_1^{\Phi(\kappa(x_1))}]\ldots[\,..\kappa(x_l) : x_l\,.. \leftarrow \Phi(\kappa(x_l)) : x_l^{\Phi(\kappa(x_l))}]$$

The instantiation of the rule with respect to Φ is

$$\mathtt{P}(e_1,\ldots,e_k)sub \ \mathtt{:-}\ \bigcup_{i=0}^{n} E(\mathtt{Q_i}(\epsilon_1^i,\ldots,\epsilon_{m_i}^i)sub,\kappa,\Phi).$$

Let us turn again to the example

$$\mathtt{Type}(\Gamma,\mathtt{bracket}[\,..\mathtt{i} : e\,..],\mathtt{prod}[\,..\mathtt{i} : \omega\,..]) \ \mathtt{:-}\ \mathtt{Type}(\Gamma,e,\omega).$$

with the field set assignment $\Phi(\mathtt{i}) = \{\mathtt{a},\mathtt{b}\}$. There are only two functions, $\varphi_\mathtt{a}$ and $\varphi_\mathtt{b}$, such that $\varphi_\mathtt{a},\varphi_\mathtt{b} \in \Phi$. These functions are defined as $\varphi_\mathtt{a}(\mathtt{i}) = \mathtt{a}$ and $\varphi_\mathtt{b}(\mathtt{i}) = \mathtt{b}$. We have then that

$$E(\mathtt{Type}(\Gamma,\mathrm{e},\omega),\kappa,\Phi) = \{\mathtt{Type}(\Gamma,\mathrm{e}^\mathtt{a},\omega^\mathtt{a}),\mathtt{Type}(\Gamma,\mathrm{e}^\mathtt{b},\omega^\mathtt{b})\}$$

The substitution from definition 7.2.2 becomes

$$[..\mathtt{i} : e.. \leftarrow \{\mathtt{a}, \mathtt{b}\} : e^{\{\mathtt{a},\mathtt{b}\}}; ..\mathtt{i} : \omega.. \leftarrow \{\mathtt{a}, \mathtt{b}\} : \omega^{\{\mathtt{a},\mathtt{b}\}}]$$

We thus have that the instantiation of the rule is

$$\mathtt{Type}(\Gamma, \mathtt{bracket}[\mathtt{a} : e^{\mathtt{a}}, \mathtt{b} : e^{\mathtt{b}}], \mathtt{prod}[\mathtt{a} : \omega^{\mathtt{a}}, \mathtt{b} : \omega^{\mathtt{b}}]) \quad \mathtt{:-}$$
$$\mathtt{Type}(\Gamma, e^{\mathtt{a}}, \omega^{\mathtt{a}}), \mathtt{Type}(\Gamma, e^{\mathtt{b}}, \omega^{\mathtt{b}}).$$

as above.

Definition 7.2.3 *Let R be a CLP^{∞}-rule. The expansion of R, $E(R)$, is the set of all possible instantiations of R.*

We have that $E(R)$ is infinite, if and only if R contains expressions of the form $..\mathtt{i} : e..$ giving rise to an infinite number of field set assignments.

Definition 7.2.4 *Let π be a CLP^{∞}-program. The expansion of π is*

$$E(\pi) = \bigcup_{R \in \pi} E(R)$$

Note that E is the identity when restricted to finite CLP.

7.3 Interpreting CLP^{∞} Programs

In the previous section we saw that for any program, $\pi \in \mathsf{CLP}^{\infty} \setminus \mathsf{CLP}$, we have that $|E(\pi)| = \infty$. Because of this it is impossible to interpret π by using a CLP-interpreter on $E(\pi)$. It is the goal of this section to devise a method for interpreting CLP^{∞} programs.

Consider the example from the previous section:

$$\mathtt{Type}(\Gamma, \mathtt{bracket}[..\mathtt{i} : e..], \mathtt{prod}[..\mathtt{i} : \omega..]) \ \mathtt{:-}\ \mathtt{Type}(\Gamma, e, \omega).$$

together with the goal $(\mathtt{Type}(\Gamma, \mathtt{bracket}[\mathtt{a} : e^{\mathtt{a}}, \mathtt{b} : e^{\mathtt{b}}], t); \emptyset)$, where Γ is a type environment, $e^{\mathtt{a}}$ and $e^{\mathtt{b}}$ are terms, and t is a variable. The rule to use here is simply

$$\mathtt{Type}(\Gamma, \mathtt{bracket}[\mathtt{a} : e^{\mathtt{a}}, \mathtt{b} : e^{\mathtt{b}}], \mathtt{prod}[\mathtt{a} : \omega^{\mathtt{a}}, \mathtt{b} : \omega^{\mathtt{b}}]) \quad \mathtt{:-}$$
$$\mathtt{Type}(\Gamma, e^{\mathtt{a}}, \omega^{\mathtt{a}}), \mathtt{Type}(\Gamma, e^{\mathtt{b}}, \omega^{\mathtt{b}}).$$

The above example is easily resolved because the clause allowed only one possible instantiation that could work. But consider instead the goal

$$(\mathtt{Type}(\Gamma, e, t); \{t = \mathtt{prod}[\mathtt{a} : \omega^{\mathtt{a}}, \mathtt{b} : \omega^{\mathtt{b}}]\})$$

where Γ is a type environment, and e and t are variables. Here we have to look at the constraint set because the clause matches all possible instantiations of the rule. Still, the constraint set allows only the instantiation given above.

Even more difficult is the goal $(\mathtt{Type}(\Gamma, e, t); \{t \geq \mathtt{prod}[\mathtt{a} : \omega^{\mathtt{a}}, \mathtt{b} : \omega^{\mathtt{b}}]\})$, where the constraint set allows an infinite number of instantiations. But here we nevertheless is a unique *least* instantiation, where $\Phi \leq \Phi'$ iff $\forall \varphi \in \Phi : \varphi \in \Phi'$. This unique least instantiation is the one above.

To help us in finding such an instantiation we define the relation $t \leq t'$ such that $\sigma(t_1, \ldots, t_n) \leq \sigma'(u_1, \ldots, u_m)$ if the following hold:

- $\sigma \sqsubseteq \sigma'$
- $\forall i \in 1..\mathrm{rank}(\sigma) : t_i = u_{\rho(i)}$, where ρ is the relabelings from σ to σ'.

As a special case we have that $\mathtt{f}[\mathtt{a_1} : t_1, \ldots, \mathtt{a_n} : t_n] \leq \mathtt{f}[\mathtt{b_1} : u_1, \ldots, \mathtt{b_m} : u_m]$ iff $\{\mathtt{a_i} | \mathtt{i} \in 1..n\} \subseteq \{\mathtt{b_i} | \mathtt{i} \in 1..m\}$ and $\forall i \in 1..n, j \in 1..m : a_i = b_j \Rightarrow t_i = u_j$.

While it is true that there is a unique least instantiation it is by no means certain that the use of this least instantiation will lead to success. Consider the following program:

```
P(f[..i..]) :- Q(f[..i..]).
Q(f[a]).
```

and the goal $(\mathtt{P}(x); \emptyset)$. The least rule allowed by the constraint set is

```
P(f[]) :- Q(f[]).
```

But this rule leads to failure. Obviously, the right rule is

```
P(f[a]) :- Q(f[a]).
```

The rule we are looking for is the least instantiation that leads to success.

In order to do this effectively we make the following assumptions about the constraint solver:

1. Actual field names are insignificant.
2. The constraint $t \leq t'$ is among the ones accepted by the constraint solver.
3. There is a $\leq$-least solution to any satisfiable constraint set, and the constraint solver provides access to this solution.

By the first assumption we mean that the constraint solver does not manipulate field names (only sets of fixed field names). Formally, we can say that the relations are invariant under substitution of field names. That is, for any relation, R, terms, $t_1, \ldots, t_{\text{arity}(R)}$, and one-to-one substitutions of field names, sub, we have that $R(t_1, \ldots, t_{\text{arity}(R)}) \Leftrightarrow R(t_1 sub, \ldots, t_{\text{arity}(R)} sub)$.

Since the constraint solver cannot manipulate field names, there cannot occur any field names that are not present either in the program or in the original query. Thus the class of field sets actually occurring during the run of the program is a complete lattice of finite height. Hence, the possible instantiation also form a complete lattice of finite height. We exploit this fact by using fixed point iteration to look for an instantiation.

In the example above we can use the rule

```
P(f[]) :- Q(f[]).
```

as a first *approximation* to the correct instantiation. We get this approximation by observing that $x = y = \texttt{f}[]$ is the least solution to the constraint set $\{x = y, y \geq \texttt{f}[]\}$. Here we use the new variable y to stand for the instantiation of the term `f[..i..]`. By the same strategy we introduce a variable y' to stand for the instantiation of `f[..i..]` in `Q(f[])`. This may seem redundant but it is actually necessary in more involved cases.

With the above we can see what the computation looks like. The new goal becomes $(\texttt{Q}(y'); \{x = y, y \geq \texttt{f}[], y' \geq \texttt{f}[]\})$. In this way the door is left open to the possibility that the actual instantiation may be greater than the current one.

From the above goal we can use the second rule of the program to obtain

$$(\emptyset, \{x = y, y \geq \texttt{f}[], y' \geq \texttt{f}[], y' = \texttt{f}[\texttt{a}]\})$$

This set of constraints is satisfiable and the least solution is $x = y = \texttt{f}[]$ and $y' = \texttt{f}[\texttt{a}]$ (here we see the necessity of using the constraint $y' \geq \texttt{f}[]$ rather than $y' = \texttt{f}[]$). But this solution is inconsistent with the use of the rule

```
P(f[]) :- Q(f[]).
```

We can use the solution to find the next approximation to the rule. Since y' is at least `f[a]` we must have that $\Phi(\texttt{i})$ is at least $\{\texttt{a}\}$. Thus, the next approximation becomes

```
P(f[a]) :- Q(f[a]).
```

At this point we *add* the constraints $y \geq \texttt{f}[\texttt{a}]$ and $y' \geq \texttt{f}[\texttt{a}]$. Now we have made the constraint set to comply with the new choice of instantiation, and since the least solution, $x = y = y' = \texttt{f}[\texttt{a}]$ is consistent with this choice we have completed our search for a computation.

As a more involved example let us look at the program

$\mathtt{P}(\mathtt{f}[..\mathtt{i}..])\ \mathtt{:-}\ \mathtt{Q}(\mathtt{g}[..\mathtt{i}:z..]), R(z).$
$\mathtt{Q}(\mathtt{f}[\mathtt{a}:z])\ \mathtt{:-}\ R'(z).$

where $R(z)$ and $R'(z)$ are constraints. As above we begin by using the smallest possible rule:

$\mathtt{P}(\mathtt{f}[])\ \mathtt{:-}\ \mathtt{Q}(\mathtt{g}[]).$

Resulting in the goal $(Q[y'], \{x = y, y \geq \mathtt{f}[], y' \geq \mathtt{g}[]\})$.

In the following step we get

$$(\emptyset, \{x = y, y \geq \mathtt{f}[], y' \geq \mathtt{g}[], y' = \mathtt{g}[\mathtt{a}:z'], R'(z')\})$$

Again, we have a conflict with the first choice of the rule and we have to choose a larger one:

$\mathtt{P}(\mathtt{f}[\mathtt{a}])\ \mathtt{:-}\ \mathtt{Q}(\mathtt{g}[\mathtt{a}:z]), R(z).$

Unlike above, this introduces an extra constraint $R(z')$, since $y' = \mathtt{g}[\mathtt{a}:z']$. We can assure this by introducing the extra constraints $y \geq \mathtt{f}[a]$, $y' \geq \mathtt{g}[\mathtt{a}:z]$, and $R(z)$. Note how the use of $\geq$ provides "access" to the subterms of y' by the requirement that equal field names should correspond to equal subterms.

Let us now turn to the generalisation of this technique for a program, π. The first step is to define the transition described above. Let us call this transition $\overline{\rhd}_\pi$. The starting point is a goal, $(\mathtt{P}(u_1, \ldots, u_n), \mathcal{B}; \mathcal{C})$, and a CLP$^\infty$-rule from π:

$\mathtt{P}(t_1, \ldots, t_n)\ \mathtt{:-}\ B, C.$

where B is a set of clauses and C is a set of constraints. As usual, we assume that the variables are properly renamed so that none of them occur in $\mathcal{B}$ or $\mathcal{C}$.

For simplicity we shall assume that all generic labels appear in the form $\mathtt{f}[..\mathtt{i}:x..]$ and that the subterms of such a form are e_1, e_2, ..., e_k. We introduce the new variables y_{e_1}, ..., y_{e_k} and define ψ and l as the functions that for each variable, y_e, gives the field set variable and generic label contained in e, respectively. That is, for $e = \mathtt{f}[..\mathtt{i}:x..]$ we have $\psi(y_e) = \mathtt{i}$ and $l(y_e) = \mathtt{f}$.

We translate the rule by using the y_e variables instead of the corresponding CLP$^\infty$-terms. So, let t'_i be the term t_i with $y_{e_1}, \ldots, y_{e_k}$ substituted for $e_1, \ldots, e_k$, respectively, and define B' and C' in a similar manner.

The next step is to find the first approximation to the instantiation. First we look at the constraint set $\mathcal{C} \cup \{t'_i = u_i | 1 \leq i \leq n\}$. If it is satisfiable it has a least solution, φ_{min}. To a variable $y_{\mathtt{f}[..\mathtt{i}:x..]}$, φ_{min} will assign a solution of the form $\mathtt{f}[\mathtt{a}_1 : t_1, \ldots, \mathtt{a}_l : t_l]$. We shall denote the set $\{\mathtt{a}_1, \ldots, \mathtt{a}_l\}$ by $\alpha(y_{\mathtt{f}[..\mathtt{i}:x..]})$.

Now, the first approximation is found by using the field set assignment $\Phi(\mathtt{i}) = \bigcup_{\psi(y)=\mathtt{i}} \alpha(y)$. The approximation is found as the expansion of the rule relative to κ and Φ, where κ is the context function of the original rule. So, if we let $B'' = \bigcup_{b\in B'} E(b,\kappa,\Phi)$ and $C'' = \bigcup_{c\in C'} E(c,\kappa,\Phi)$ we get that

$$(\mathtt{P}(u_1,\ldots,u_n),\mathcal{B};\mathcal{C})\overline{\rhd}_\pi(\mathcal{B}\cup B'';\mathcal{C}')$$

where

$$\mathcal{C}' = \mathcal{C}\cup C''\cup\{t'_i = u_i | i\in 1..n\}\cup\{y_{e_i}\geq l(y_{e_i})[\Phi(\psi(y_e)) : x^{\Phi(\psi(y_e))}] | i\in 1..k\}$$

As with $\rhd$ we omit the π when no ambiguities can arise.

By repeating the process described above we eventually reach a stage where there are no more unresolved clauses (unless the computation fails or diverges in the usual CLP or Prolog fashion). The goal now has the form $(\emptyset,\mathcal{C})$. On the way to this goal we have made several approximations as described by the field set assignment Φ above. All these field set assignments have disjoint domains (since we rename variables) so we can assume that we have accumulated them in a large field set assignment. Let us call this assignment Φ. In a similar way let l and ψ be the accumulated versions of the local ls and ψs.

We find our next approximation by looking at the least solution as in the single step above and defining α in the same way. Now, let Φ' be the field set assignment defined by $\Phi'(\mathtt{i}) = \bigcup_{\psi(y)=\mathtt{i}} \alpha(y)$. If $\Phi' = \Phi$ we say that Φ is *consistent* with $\mathcal{C}$. In this case we are done, but if $\Phi(\mathtt{i}) \subsetneq \Phi'(\mathtt{i})$ for some field set variable, $\mathtt{i}$, we have an *inconsistency* between the constraint set and our choice of approximation.

In order to resolve this inconsistency we add new constraints and clauses in the following way. Assume that the variable $\mathtt{i}$ originated in the (properly renamed) rule

$$\mathtt{P}(t_1,\ldots,t_n)\ \mathtt{:-}\ B,C.$$

For each y such that $\psi(y) = \mathtt{i}$ we add the constraint $y \geq l(y)[\Phi'(\mathtt{i}) : x^{\Phi'(\mathtt{i})}]$. Furthermore, we add the constraints $E(C,\kappa,\Phi')\setminus E(C,\kappa,\Phi)$ and the clauses $E(B,\kappa,\Phi')\setminus E(B,\kappa,\Phi)$.

If $\mathcal{B}'$ and $\mathcal{C}'$ are the sets of clauses and constraint added by repeating the above for every $\mathtt{i}$ such that $\Phi(\mathtt{i}) \subsetneq \Phi'(\mathtt{i})$, we get the new goal $(\mathcal{B}';\mathcal{C}\cup\mathcal{C}')$ and proceed from there. This process is repeated until $\Phi = \Phi'$. If $(\mathcal{B};\mathcal{C})$ is the initial goal and the process ends in $(\emptyset;\mathcal{C}')$ we shall write $(\mathcal{B};\mathcal{C}) \blacktriangleright_\pi \mathcal{C}'$.

The process thus described constitutes a fixed point iteration in search of the least fixed point, Φ. If such a fixed point exists, termination will be guaranteed because the lattice has finite height. If no fixed point exists the process will either fail due to unsatisfiability of the constraint set or it will diverge just as the interpretation of any CLP-program might.

To see that this process coincides with the usual operational semantics of the expansion of a program, we add some more information to the relations $\rhd$ and $\overline{\rhd}$. The extra information that we need is which rule was used to make the transition. This is necessary since the semantics is non-deterministic and we need to make sure that the two computations took equivalent paths.

Assume that $(\mathcal{B};\mathcal{C}) \rhd (\mathcal{B}';\mathcal{C}')$ because of a rule $R \in E(P)$, where P is a CLP$^\infty$-program. Then we shall write $(\mathcal{B};\mathcal{C}) \rhd_R (\mathcal{B}';\mathcal{C}')$. If

$$(\mathcal{B};\mathcal{C}) \rhd_{R_1} (\mathcal{B}_1;\mathcal{C}_1) \rhd_{R_2} \cdots \rhd_{R_n} (\mathcal{B}_n;\mathcal{C}_n)$$

we shall write $(\mathcal{B};\mathcal{C}) \rhd^*_{(R_1,\ldots,R_n)} (\mathcal{B}_n;\mathcal{C}_n)$.

For the similar extension of $\overline{\rhd}$ we need the following definition: Let R and R' be rules. We say that $R \leq R'$ if and only if R and R' are instantiations of the same rules with respect to the field assignments Φ and Φ', respectively and $\Phi \leq \Phi'$.

Now assume that $(\mathcal{B};\mathcal{C})\overline{\rhd}(\mathcal{B}';\mathcal{C}')$ because of a rule $R \in P$ and let Φ be the field set assignment used in the approximation. If we write R_Φ for the instantiation of R with respect to Φ we shall write $(\mathcal{B};\mathcal{C})\overline{\rhd}_{\{R\in E(R)|R\geq R_\Phi\}}(\mathcal{B}';\mathcal{C}')$. This corresponds to the intuition that there is an infinite set of possible rules that could be used instead of the approximation.

If $(\mathcal{B};\mathcal{C})\overline{\rhd}_M(\mathcal{B}';\mathcal{C}')$ there is a unique field set assignment used in the approximation. We call this field set assignment Φ_M. For reasoning purposes we generalise so that whenever $\Phi' \supseteq \Phi$ we have $(\mathcal{B};\mathcal{C})\overline{\rhd}_{\{R\in E(R)|R\geq R_{\Phi'}\}}(\mathcal{B}'';\mathcal{C}'')$, where we arrive at $\mathcal{B}''$ and $\mathcal{C}''$ by the usual method but using Φ' instead of Φ. If $(\mathcal{B};\mathcal{C})\overline{\rhd}_M(\mathcal{B}';\mathcal{C}')$ and it is impossible to make any transition for a more general set of rules, we say that M is *minimal* (actually M is the greatest possible set, but Φ_M is the least possible field set assignment — hence 'minimal'). Note that the notion of minimality is meaningful only in the context of a specific transition.

We define $\overline{\rhd}^*_{(M_1,\ldots,M_n)}$ analogously with the definition of $\rhd^*_{(R_1,\ldots,R_n)}$ above. For sequences of sets we write $(M_1,\ldots,M_n) \subseteq (M'_1,\ldots,M'_n)$, iff $\forall i \in 1..n : M_i \subseteq M'_i$. Similarly, we write $(R_1,\ldots,R_n) \in (M_1,\ldots,M_n)$, iff $\forall i \in 1..n : R_i \in M_i$. If $\underline{M} = (M_1,\ldots,M_n)$ we define $\Phi_{\underline{M}}$ as $\Phi_{\underline{M}}(\mathtt{i}) = \Phi_{M_k}(\mathtt{i})$, iff $\mathtt{i} \in \mathrm{dom}(\Phi_{M_k})$. We say that $(M_1,\ldots,M_n)$ is minimal if each M_i is minimal.

The following lemma follows from the definition of $\overline{\rhd}^*_{\underline{M}}$.

Lemma 7.3.1

1. *If $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}}(\mathcal{B}';\mathcal{C}')$ and $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}}(\mathcal{B}'';\mathcal{C}'')$ then $\mathcal{B}' = \mathcal{B}''$ and $\mathcal{C}' = \mathcal{C}''$.*
2. *If $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}}(\mathcal{B}';\mathcal{C}')$ then there is a minimal set of rules, $\underline{M}'$, a set of formulae, $\mathcal{B}''$, and a constraint set, $\mathcal{C}''$, such that $\underline{M}' \supseteq \underline{M}$ and $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}'}(\mathcal{B}'';\mathcal{C}'')$.*

3. *If* $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}}(\mathcal{B}';\mathcal{C}')$, $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}'}(\mathcal{B}'';\mathcal{C}'')$ *and* $\underline{M} \supseteq \underline{M}'$ *then* $\mathcal{B}' \subseteq \mathcal{B}''$ *and* $\mathcal{M} \models \mathcal{C}'' \rightarrow \mathcal{C}'$, *where* $\mathcal{M}$ *is the least model of the program.*

The approximation step (getting from $(\emptyset,\mathcal{C})$ to $(\mathcal{B}',\mathcal{C}\cup\mathcal{C}')$) above is dependent on how the goal $(\emptyset,\mathcal{C})$ was reached — that is, it depended on the $\underline{M}$ in $(\mathcal{B},\emptyset)\overline{\rhd}^*_{\underline{M}}(\emptyset,\mathcal{C})$. Again we generalise so that Φ' can be any field set assignment such that for all $\mathtt{i}$ and y, $\psi(y) = \mathtt{i} \Rightarrow \alpha(y) \subseteq \Phi'(\mathtt{i})$. Now we can write this formally by the notation $(\emptyset;\mathcal{C}) \rightsquigarrow^{\Phi'}_{\underline{M}} (\mathcal{B}';\mathcal{C}\cup\mathcal{C}')$.

The interpretation described in this section introduces a number of redundant constraints in the constraint set. In the second of the examples we had the constraint set

$$\{x = y, y \geq \mathtt{f}[], y' \geq \mathtt{g}[], y' = \mathtt{g}[\mathtt{a} : z'], R'(z'), y \geq \mathtt{f}[a], y' \geq \mathtt{g}[\mathtt{a} : z], R(z)\}$$

Here the constraints $y \geq \mathtt{f}[]$ and $y' \geq \mathtt{g}[]$ are redundant because they are implied by $y \geq \mathtt{f}[a]$ and $y' \geq \mathtt{g}[\mathtt{a} : z]$, respectively. We have also two different variables z and z' which are really the same subterms of y'. Thus, the following constraint set is equivalent to the one above:

$$\{x = y, y' = \mathtt{g}[\mathtt{a} : z], y \geq \mathtt{f}[a], y' \geq \mathtt{g}[\mathtt{a} : z], R(z)\}$$

For simplicity, we shall identify constraint sets that are equivalent in this way.

Using the identity above we get the following lemma.

Lemma 7.3.2 *Let* $\mathcal{B}$ *be a set of atomic formulae,* $\mathcal{C}$ *and* $\mathcal{C}'$ *satisfiable sets of constraints, and assume that* $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}}(\emptyset;\mathcal{C}')$. *Furthermore, assume that* $\Phi_{\underline{M}}$ *and* $\mathcal{C}'$ *are inconsistent and that* $(\emptyset;\mathcal{C}') \rightsquigarrow^{\Phi}_{\underline{M}} (\mathcal{B}';\mathcal{C}'\cup\mathcal{C}'')$. *Then there is an* $\underline{M}' \subseteq \underline{M}$, *such that* $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}'}(\mathcal{B}';\mathcal{C}'\cup\mathcal{C}'')$.

Proof idea: We obtain an $\underline{M}'$ that works by replacing $\{R \in E(R) | R \geq R_{\Phi_{\underline{M}}}\}$ with $\{R \in E(R) | R \geq R_\Phi\}$ everywhere in $\underline{M}$. □

The lemma expresses that the approximation step is sound. That is, the approximation step yields a result that corresponds to an actual computation using $\overline{\rhd}$. Hence the following corollary.

Corollary 7.3.3 *Let* $\mathcal{B}$ *be a set of atomic formulae, and let* $\mathcal{C}$ *and* $\mathcal{C}'$ *be constraint sets. We have that* $(\mathcal{B};\mathcal{C}) \blacktriangleright \mathcal{C}'$, *iff there is a minimal sequence of sets of rules,* $\underline{M}$, *such that* $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}}(\emptyset;\mathcal{C}')$.

Proof: From the definition of $\blacktriangleright$ by induction on the length of the derivation using lemma 7.3.2 and lemma 7.3.1(2). □

Now that $\blacktriangleright$ is reduced to $\overline{\rhd}$, we must look at the relation between $\overline{\rhd}$ and $\rhd$. They are not exactly the same since $\overline{\rhd}$ leaves open the possibility of choosing greater instantiations along the computation path, whereas $\rhd$ chooses a single instantiation and demands that any solution should comply with this choice of instantiation. For this reason the main lemma concerning the equivalence of $\overline{\rhd}$ and $\rhd$ deals with the *solutions* that could possibly come out of a goal.

Lemma 7.3.4 *Let $\underline{M} = (M_1, \ldots, M_n)$ and $\underline{R} = (R_1, \ldots, R_k)$. We have the following:*

1. *If $(\mathcal{B};\mathcal{C}) \rhd^*_{\underline{R}} (\mathcal{B}';\mathcal{C}')$, there is a constraint set, $\mathcal{C}''$, such that $\mathcal{M} \models \mathcal{C}' \Rightarrow \mathcal{C}''$ and $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}'}(\mathcal{B}';\mathcal{C}'')$, where*

$$\underline{M}' = (\{R' | R' \geq R_1\}, \ldots, \{R' | R' \geq R_k\})$$

2. *If $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}}(\mathcal{B}';\mathcal{C}')$ and φ satisfies $\mathcal{C}'$ then there is a constraint set, $\mathcal{C}''$, and a sequence of rules, $\underline{R}'$, such that φ satisfies $\mathcal{C}''$, $(\mathcal{B};\mathcal{C}) \rhd^*_{\underline{R}'} (\mathcal{B}';\mathcal{C}'')$, and $\underline{R}' \in \underline{M}$.*

Proof: Follows directly from the definitions of $\rhd^*_{\underline{R}}$ and $\overline{\rhd}^*_{\underline{M}}$ by induction in k and n respectively. □

From corollary 7.3.3 and lemma 7.3.4 we get the main theorem of this section. It says that with respect to satisfying assignments the interpretation is sound and complete.

Theorem 10 *Let π be a CLP$^\infty$-program, and let $\mathcal{M}$ be the least model for $E(\pi)$. We have the following.*

1. (Soundness) *If $(\mathcal{B};\mathcal{C}) \blacktriangleright_\pi \mathcal{C}'$ and φ satisfies $\mathcal{C}'$ then there is a constraint set, $\mathcal{C}''$ such that $(\mathcal{B};\mathcal{C}) \rhd^*_{E(\pi)} (\emptyset;\mathcal{C}'')$ and φ satisfies $\mathcal{C}''$.*
2. (Completeness) *If $(\mathcal{B};\mathcal{C}) \rhd^*_{E(\pi)} (\emptyset;\mathcal{C}')$ then there is a constraint set, $\mathcal{C}''$ such that $(\mathcal{B};\mathcal{C}) \blacktriangleright_\pi \mathcal{C}''$ and $\mathcal{M} \models \mathcal{C}' \Rightarrow \mathcal{C}''$.*

Proof:

1. Follows directly from lemma 7.3.4(2) and corollary 7.3.3.
2. Assume that $(\mathcal{B};\mathcal{C}) \rhd^* (\emptyset;\mathcal{C}')$. Then by definition there is an $\underline{R}$ such that $(\mathcal{B};\mathcal{C}) \rhd^*_{\underline{R}} (\emptyset;\mathcal{C}')$. From lemma 7.3.4(1) we get an $\underline{M}$ and a $\mathcal{C}'''$ such that $\mathcal{M} \models \mathcal{C}' \Rightarrow \mathcal{C}'''$ and $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}}(\emptyset;\mathcal{C}''')$. Lemma 7.3.1(2) gives us a minimal $\underline{M}'$, such that $(\mathcal{B};\mathcal{C})\overline{\rhd}^*_{\underline{M}'}(\mathcal{B}';\mathcal{C}''')$, where $\underline{M}' \supseteq \underline{M}$, and thus (from lemma 7.3.1(3)) $\mathcal{M} \models \mathcal{C}''' \Rightarrow \mathcal{C}''$ and $\mathcal{B}' \subseteq \emptyset$. The latter gives us $\mathcal{B}' = \emptyset$, so by using corollary 7.3.3 and the transitivity of $\Rightarrow$ we are done.

□

The remaining question is that of the efficiency of the interpretation described above. No doubt a program can be written where Φ grows very slowly. The interpretation of such a program could go through the process of finding a new approximation very many times. In the worst case as many as $\sum_{\mathtt{i}} |\Phi(\mathtt{i})|$.

In practice, however, the number of runs will probably be quite small. Most of the field names will come from the input and they will propagate quite rapidly throughout the constraint set. The field sets in the input would also determine a unique possible instantiation of many of the rules.

This would indeed be the case in the type inference example that motivated the introduction of ellipsis into the CLP program. In this important special case the first approximation will also be the only possible and the interpreter would find the answer on the first attempt. The fixed point iteration strategy is thus perfectly suited for the implementation of type inference.

7.4 Using CLP(OIH)

When the domain is in place there are two things which are needed for a CLP(OIH) program: The user-defined constraints and the rules.

7.4.1 Defining the Relations

The user-defined constraints are as shown in Chapter 4. That is, they follow the syntax defined by the category *reldef* below.

reldef	::=	*name*(*name*, ..., *name*) `<->` *tuplelist*.
	|	`rec` *name*(*name*, ..., *name*) `<->` *tuplelist*.
tuplelist	::=	*tuple* | *tuple*; *tuplelist*
tuple	::=	(*value*, ..., *value*)
	|	(*value*, ..., *value*) `if` *conditions*
value	::=	*name* | *name*[*name*]
conditions	::=	*condition* | *condition*, *conditions*
condition	::=	*name* `<=` *name*
	|	*conditional constraint*
	|	*quantified constraint*
quantified constraint	::=	@*quantifiers* : *conditional constraint*
quantifiers	::=	*quantifier* | *quantifier*, *quantifiers*

$$
\begin{array}{rcl}
\textit{quantifier} & ::= & \textit{name}\ \texttt{in}\ \textit{name}\ \&\ \cdots\ \&\ \textit{name} \\
\textit{conditional constraint} & ::= & \textit{pseudovar} = \textit{pseudovar} \\
& | & \textit{name}(\textit{pseudovar}, \ldots, \textit{pseudovar}) \\
\textit{pseudovar} & ::= & \textit{name} \mid \textit{name}/\textit{number}
\end{array}
$$

Where the category *number* defines the positive numbers, and the category *name* defines the set of strings consisting of letters, digits, underscore (_), and prime ($'$), which begins with a letter. That is, the string `tau_1'` is a *name*.

The list of names in the heading of the *reldef* defines names for the arguments so that we can write the pseudovariables as e.g. `tau/1` rather than $\langle 1/1 \rangle$. The pseudovariable $\langle 1/\epsilon \rangle$ is written as `tau`. Similarly, we can name the generic labels by the construct *name*[*name*]. In this way we can write `a <= b` for $1 \preceq_\sigma 2$ and `@l in a & b` for $\forall l \in [1]\&[2]$.

The relation $\leqslant_{\mathsf{st}}$ is defined in CLP(OIH) as follows.

```
rec leq_st(tau, tau') <->
    (int, int);
    (bool, bool);
    (prod[a], prod[b])
        if b <= a, @i in a&b:  leq_st(tau/i, tau'/i);
    (sum[a], sum[b])
        if a <= b, @i in a&b:  leq_st(tau/i, tau'/i);
    (list, list) if leq_st(tau/1, tau'/1);
    (arrow, arrow)
        if leq_st(tau'/1, tau/1), leq_st(tau/2, tau'/2).
```

The keyword `rec` must be used whenever a relation on the form $\nu X.\langle\!\langle R, \chi \rangle\!\rangle$ is defined.

When we look at the definition of strictness-closed (Definition 6.6.1) we can see that there is a special class of relations that are problematic — the recursively defined inequalities. Due to the anti-symmetry of inequalities we have that the constraint set $\{x \leqslant y, y \leqslant x\}$ is equivalent with the single constraint $x = y$. Since we assume that $\leqslant$ is recursively defined, this corresponds to having a recursively defined $=$ constraint, which in turn leads to infinite successions of approximations as in Figure 6.10. In fact, it is the case that if we look at the constraint set as a directed graph where there is an edge from x to y, iff there is a constraint $x \leqslant y$ in the constraint set, we get that all variables which are strongly connected in this graph should be equal.

We can exploit the fact about the strongly connected variables and make a heuristic for dealing with the inequalities. If we maintain the graph described above, we can use the strong connectivity algorithm of Section 6.5.1 to make sure that all strongly connected components are unified in a single state.

To recognise these cases we introduce the keyword `ineq` which semantically is like `rec`, but in addition demands that strongly connected components should be unified. Use of this keyword allows the algorithm to terminate in many cases where it would not have terminated otherwise. Unfortunately, the inequalities that are not strictness-closed, are still not strictness-closed even when we disallow the possibilities of cycles in the definition. It follows that `ineq` remains a pragmatic tool rather than a strengthening of the expressibility of the domain.

7.4.2 The CLP(OIH) Programming Language

A CLP(OIH) program has the same syntax as any other CLP programming language (see Section 3.2). The unique characteristics are those of the form of the expressions and the constraints. An expression is one of the following

- A variable.
- One of the following term expressions
  - $\sigma(e_1, \ldots, e_k)$
  - $\sigma[a_1, \ldots, a_n]$
  - $\sigma[a_1, \ldots, a_n](e_1, \ldots, e_k)$
  - $\sigma[a_1 : e_1, \ldots, a_n : e_n]$
  - $\sigma[a_1 : e_1, \ldots, a_n : e_n](e'_1, \ldots, e'_k)$

  where the e_i and e'_i are expressions.
- One of the following ellipsis expressions
  - $\sigma[..a..]$
  - $\sigma[..a..](e_1, \ldots, e_k)$
  - $\sigma[..a : v..]$
  - $\sigma[..a : v..](e_1, \ldots, e_k)$

  where v is a variable and the e_i are expressions.

We demand that $\kappa(e) = \emptyset$ if e is an ellipsis expression.

The constraints are even simpler. They are on the following forms.

- $R(e_1, \ldots, e_n)$
- $e_1 = e_2$
- e_1 `<=` e_2

where R is (the name of) a user-defined relation and the e_i are expressions. The last constraint uses the relation <=. This is the relation $\leq$ defined in Section 7.2. It turns out that this relation is important enough to warrant making it a predefined relation.

7.4.3 Using the CLP(OIH) Interpreter

The CLP(OIH) interpreter uses two files: the definitions file and the program file. The definitions file contains the definitions of the domain as described in Section 7.1 and the relation definitions. The program file contains the CLP rules.

The program can be started in three different ways:

`clp`: This is the basic way to start the program. The user will be prompted for the definitions file and the program file.

`clp <file>`: If `<file>` does not contain a '.' it is assumed that the definitions and program files are `<file>.def` and`<file>.clp`, respectively. Otherwise it is assumed that the definitions file is `<file>` and the user will be prompted for the program file.

`clp <file1> <file2>`: In this case `<file1>` is the definitions file and `<file2>` is the program file.

If the given files are not syntactically correct, the program terminates with an error message. Another thing that will create an error is if the reflexive, transitive closure of the user-defined ordering is not anti-symmetric. Ideally, the program should also perform the tests from Section 6.7, but this has not been implemented.

If the files are correct the user is prompted for a goal. The prompt looks like

```
?-
```

A goal is entered a constraint or predicate at a time — one for each line. That is, to enter the goal ($\mathtt{Type}(\Gamma, \mathtt{e}, \mathtt{t}); \{\mathtt{t} \geq \mathtt{prod}[\mathtt{a} : \omega^{\mathtt{a}}, \mathtt{b} : \omega^{\mathtt{b}}]\}$) we type the following

```
?- Type(Gamma, e, t)
?- prod[a: omega_a, b: omega_b]
?- #run
```

The command `#run` starts the interpreter on the goal. If the goal fails the program will output

```
** No **
```

If it succeeds it will output

```
** Yes **
```

Followed by a system of equations which defines the $\leqslant_\Sigma$-least solution. The program will then prompt for the next goal.

If the goal is large it becomes tedious to type it in online. The command `#read` takes a file as an argument and reads the goal from that file. For example if the file `foo.goal` contains

```
Type(Gamma, e, t)
prod[a: omega_a, b: omega_b]
```

then we could have the dialogue

```
?- #read foo.goal
Type(Gamma, e, t)
prod[a: omega_a, b: omega_b]
?-
```

Note, that the program echoes the lines it reads from the goal file. The goal file can itself contain lines of the form `#read <file>` in which case the program recursively reads the goal from the file `<file>`.

The command `#quit` or `#q` end the session.

7.4.4 Troubleshooting

There are several ways in which we can get unexpected results. Here are some of the most frequent.

The program gives a run-time error. This is always due to some of the requirements not being met. Most fatal are the following.

- The ordering does not form a lower semi-lattice. This will invariably result in a segmentation fault.
- The rank is not monotone.
- Some conclusion(s) contain(s) free occurrences of bound variables.
- Some ellipsis expression(s) contain(s) free occurrences of bound variables.

The program gives unexpected error messages. Provided the files are in fact syntactically correct this should rarely happen. One possibility is that the last line in one of the files does not end with an end-of-line (only an end-of-file). Another is that a goal file contains empty lines or leading whitespace on one of the lines.

The program terminates with an unexpected result. Barring the obvious possibility that our expectations are wrong and that the result is a perfectly valid solution to the program we intended to write there are still a couple of possibilities.

- There is a misspelled identifier in one of the rules, thus creating a free variable. This will make the interpreter come up with solutions that are in fact not solutions to the problem.
- There is a misspelled predicate name. If the predicate name is in the conclusion of the rule, then this will effectively remove that rule from the program. If the predicate name is in the antecedents then it will be a predicate without a rule. That is, it will always fail. In the latter case the interpreter will issue a warning. In both cases the interpretation might fail when there is in fact solutions to the problem.
- The goal contains an unsatisfiable constraint set. If this is the case the interpreter will issue a warning. The results are completely unpredictable if the constraint set is unsatisfiable.

Chapter 8

Applications of CLP(OIH)

CLP(OIH) is a strict extension of PROLOG and PROLOG-II. As such it contains all the application of these languages. In this chapter we concentrate on the class of applications CLP(OIH) is designed for — type inference algorithms. We open our discussion with a tutorial on implementing type inference algorithms in CLP(OIH) using the simply typed λ-calculus as an example.

8.1 Type Inference in CLP(OIH)

In this section we describe the general techniques for implementing type inference algorithms in CLP(OIH). We use the simply typed λ-calculus to demonstrate these techniques (see [6] for more about the λ-calculus). The expressions in the simply typed λ-calculus follow the syntax

$$e ::= n \mid x \mid \lambda x.\, e \mid (e\ e')$$

where n is an integer and x is a variable. The types are of the form

$$\tau ::= i \mid \tau \to \tau'$$

The two kinds of types are *integer* and *function* types respectively. An expression $\lambda x.\, e$ defines the function whose description is $f(x) = e$. Hence the type of $\lambda x.\, e$ has a function type $\tau \to \tau'$ where τ is the type of x and τ' is the type of e given that x has type τ.

For instance we have that $\lambda x.\, x$ has among others the type $i \to i$ and the type $(i \to i) \to (i \to i)$. We write this as the *type judgements* $\vdash \lambda x.\, x : i \to i$ and $\vdash \lambda x.\, x : (i \to i) \to (i \to i)$, respectively. We arrive at the former of these judgements by noting that if x has type i then x has type i. We write this as the judgement $x : i \vdash x : i$.

The reasoning that leads us to the judgement $\vdash \lambda x.\, x : i \to i$ is written in the form of a *type deduction*:

$$\frac{x : i \vdash x : i}{\vdash \lambda x.\, x : i \to i}$$

The type system is defined by *typing rules* that define which deductions are valid. The typing rules for the simply typed λ-calculus are the following.

$$\Gamma \vdash n : i \qquad\qquad \Gamma,\, x : \tau \vdash x : \tau$$

$$\frac{\Gamma,\, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\ e') : \tau}$$

There is no mention above about what should be done in the case where the same variable appears in different λ-expressions (e.g. $\lambda x.\, \lambda x.\, x$). For simplicity, we shall assume that all variables are distinct.

As a more involved example we have the deduction of $\vdash (\lambda x.\, x)(\lambda y.\, y) : i \to i$:

$$\frac{\dfrac{x : i \to i \vdash x : i \to i}{\vdash \lambda x.\, x : (i \to i) \to (i \to i)} \qquad \dfrac{y : i \vdash y : i}{\vdash \lambda y.\, y : i \to i}}{\vdash (\lambda x.\, x)(\lambda y.\, y) : i \to i}$$

In the case of type inference we do not know the target type for the expression. Neither do we know the types of the variables in advance. This necessitates the use of constraints. While we cannot know what the type of $\lambda x.\, x$ is we can know that if the type of x is τ the type of $\lambda x.\, x$ will be $\tau \to \tau$. We write this as the constraint $[\![\lambda x.\, x]\!] = [\![x]\!] \to [\![x]\!]$, where expressions of the form $[\![e]\!]$ are type variables standing for the type of e.

For the simply typed λ-calculus we generate the following constraints:

- For each integer constant, n, the constraint $[\![n]\!] = i$.
- For each variable, x, the constraint $[\![x]\!] = [\![x]\!]$.
- For each λ-abstraction $\lambda x.e$, the expression $[\![\lambda x.e]\!] = [\![x]\!] \to [\![e]\!]$.
- For each function application $(e\ e')$ the constraint $[\![e]\!] = [\![e']\!] \to [\![(e\ e')]\!]$.

The redundant constraint $[\![x]\!] = [\![x]\!]$ is added to ensure that we generate constraints for every subexpression.

The encoding of the types in CLP(OIH) is quite straightforward. There are two labels, and the types are simply the set of all terms that can be generated by these labels. We implement this in CLP(OIH) by the following lines.

```
int/0,
arrow/2
```

But encoding the types is not enough. In order to generate the right constraints the program must know of the syntax of the program. In addition to this we have the type environment, Γ, from the expression $\Gamma \vdash e : t$.

The encoding of environments in CLP-languages usually poses a problem. In [39] Kozen uses predicates of the form

$$\texttt{ma}(\texttt{x}_1, \ldots, \texttt{x}_n, \text{"}\texttt{p}\text{"}, \texttt{y}_1, \ldots, \texttt{y}_n)$$

where the $\texttt{x}_i$ are the values of the program variables before the analysis, the $\texttt{y}_i$ are the values after the analysis, and `"p"` is some encoding of the program. The drawback of this is that there is a new predicate for each program, which is clearly impractical.

Here, we exploit that we can use extensions. We introduce a label `env` defined in the program as

```
env[dom]/|dom|
```

Using this label we can describe the environment mapping x to τ_x and y to τ_y as $\texttt{env}[\texttt{x} : \tau_\texttt{x}, \texttt{y} : \tau_\texttt{y}]$.

This encoding of environments dictates the encoding of variable references. The variable reference x is encoded as `var[x]`, where `var` is the label and `[x]` is the extension. This has a curious consequence. Since we have no restriction on the size of extensions and the only constraints we allow on extensions is $\subseteq$, both `var[]` and `var[x,y,z]` are valid expressions. We will assume, however, that expressions of these forms do not occur.

Just as with variable references, variables as λ-abstractions are handled using extensions, so the expression $\lambda x.\, e$ is encoded `lambda[x](e)`. From a typing point of view there is no difference between integer constants, so they are all handled by the single label `intconst`. The label `app` handles function application and hence the set of labels for the encoding of expressions is as follows.

```
intconst/0,
var[x]/0,
lambda[x]/1,
app/2
```

With the above extensions Reg_Σ no longer contains only types. This raises a two questions.

1. What is the ordering between non-type labels?
2. What do we do with expressions such as `lambda[x](arrow(env[], int))`?

The answer to the first question is that we first order each kind of label with some lower semi-lattice. In the case of the types we pick the one that the algorithm

dictates. For the other labels the ordering is arbitrary. For instance we could have the following ordering between the expression labels.

```
intconst < var,
intconst < lambda,
intconst < app,
```

Then we add inequalities to the effect that the smallest type is also the smallest label:

```
int < intconst,
int < env
```

The ordering thus obtained is the one used for the program.

In order to ensure that meaningless expressions do not occur, we define three constraints `IsType`(τ), `IsEnvironment`(Γ), and `IsExpression`(e). These constraints ensure that their arguments are of the right kind. As an example the constraint `IsType` is defined by

```
rec IsType(tau) ↔
    (int);
    (arrow) if IsType(tau/1), IsType(tau/2).
```

The implementation of the type inference algorithm consists of simply following the type deduction rules. We define a predicate `Type`/3 such that for all environments Γ, expressions e, and types τ we have that `Type`$(\Gamma, \mathtt{e}, \tau) \Leftrightarrow \Gamma \vdash e : \tau$. The easiest case is that of integer constants.

```
Type(gamma, intconst, int).
```

In this case we do not explicitly generate a constraint. The constraint $[\![e]\!] = i$ is implicitly present in the semantics of CLP(OIH). In the case of variable references we need to access Γ.

```
Type(gamma, var[..x..], t) :-
    env[..x:t'..]  <= gamma, t = t'.
```

Note that it is necessary to write `var[..x..]` rather than just `var[x]` because the latter is just a term containing no variables, so it would apply only to a variable named x. It follows from the rule above that `var[]` has any type, and that `var[x,y,z]` has a type only if all of x, y, and z have that type. The rule for λ-abstractions is similar.

```
Type(gamma, lambda[..x..](e), t) :-
    gamma <= gamma', env[..x:t'..]  <= gamma',
    Type(gamma', e, t''), t = arrow(t', t'').
```

Note the use of two <=-constraints to model $\cup$. The rule for function application is straightforward.

```
Type(gamma, app(e, e'), t) :-
    Type(gamma, e, arrow(t',t)), Type(gamma, e', t').
```

Again, the two constraints are implicit.

The rules above implement the type inference algorithm apart from the problem that e.g. `Type(env[x : intconst], var[x], intconst)` succeeds. In order to disallow this we define the predicate `Typing/3` as follows.

```
Typing(gamma, e, t) :-
    IsEnvironment(gamma), IsType(t), IsExpression(e),
    Type(gamma, e, t).
```

The algorithm always finds the smallest Γ, such that $\Gamma \vdash e : \tau$. Hence, it will infer $\vdash (\lambda x.\, x)(\lambda y.\, y) : i \to i$, but it will not give back the type of x or y. One way to get these types is to change `gamma` <= `gamma'` to `gamma` = `gamma'` in the rule for λ-abstraction. A way to do it without changing the program is to start the program on the goal

```
?- Typing(gamma, e, t), env[x : t_x, y : t_y] <= gamma
```

This completes the implementation of the type inference algorithm. The complete program can be found in Appendix A.1.1.

8.2 Type Inference with Overloading

A polymorphic function can take arguments of several different types. There is in general two different kinds of polymorphism (see [9]):

1. *Parametric polymorphism*, where the same code is used to implement the function for different argument types.
2. *Ad-hoc polymorphism* or *overloading*, where different code is used for different types.

As an example of parametric polymorphism we have concatenation of lists, where only the structure of the list matters and the elements are moved around using pointers. A typical example of overloaded functions is the + functions which can add integers as well as reals.

In the context of type inference, overloading is different from parametric polymorphism because theoretically there is no restrictions on the types that an overloaded function can have whereas the types of a polymorphic function can be

described as a type expression with free variables, e.g. $\mathsf{list}(\alpha) \to \mathsf{list}(\alpha) \to \mathsf{list}(\alpha)$ — the type of the concatenation function.

8.2.1 Overloading in the Lambda Calculus

Type inference in the presence of overload functions has been studied by several authors (e.g. [67] and [75]). The difficulty immediately arises that the problem is so general that type inference becomes impractical if not impossible.

Smith [67] examines an extension of the λ-calculus with overloading. He proves that the general problem is undecidable. Instead he focuses on special cases where the overloading is a sort of restricted parametric polymorphism, where the variables can be restricted to types built of certain constructors. Consider the overloaded equality, $e = e'$. The expression $e = e'$ is legal whenever e and e' is of the same type which contains no function types (since equality of functions is undecidable). That is, = has the restricted polymorphic type

$$\forall \alpha_{\mathsf{int},\mathsf{real},\mathsf{bool},\mathsf{list}}.\alpha \to \alpha \to \mathsf{bool}$$

This kind of restriction is easily modelled in CLP(OIH). The key is the relation `EqType` defined as

```
rec EqType(tau) ↔
    (real);
    (int);
    (bool);
    (list) if EqType(tau/1).
```

A similar relation can be build for each $\Sigma' \subseteq \Sigma$.

This restricted version of overloading lacks expressibility, though. It is not possible to express the type of an overloaded + function with the types

$$\begin{array}{l} + : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \\ + : \mathsf{int} \to \mathsf{real} \to \mathsf{real} \\ + : \mathsf{real} \to \mathsf{int} \to \mathsf{real} \\ + : \mathsf{real} \to \mathsf{real} \to \mathsf{real} \end{array}$$

In fact, the rules for + can only be expressed by a trinary constraint, but if we allow the typing

$$+ : \mathsf{int} \to \mathsf{int} \to \mathsf{real}$$

as well we can express the types as $+ : \tau \to \tau' \to \tau''$, iff there is a type $\tau_0 \in \{\mathsf{int}, \mathsf{real}\}$ such that $\tau \leqslant \tau_0$, $\tau' \leqslant \tau_0$, and $\tau_0 \leqslant \tau''$, where $\mathsf{int} < \mathsf{real}$.

Using inequalities over the base types is known as *coercion*. Basically, in a type system built on coercion we have that any expression of type int can be coerced into being an expression of type Real. Wand and O'Keefe [75] have examined type inference with coercion and have shown that it is in general *NP*-complete. However, they prove that if the ordering forms a tree, the type inference can be solved in polynomial time. In fact, this requirement is too strict. The only thing that is needed for their proof to work is that any set with an upper bound has a least upper bound.

This result should not surprise us, since all sets of types with an upper bound have a least upper bound if and only if the ordering is a lower semi-lattice. And since it follows from the definition of stability that any ordering, $\leqslant$, is $\leqslant$-stable, our work may be viewed as a generalisation of the result in [75].

To demonstrate the above we show how to implement a type inference algorithm for a lambda calculus with coercion. First, we look at how coercion extends to function types:

$$\tau_1 \to \tau_1' \leqslant \tau_2 \to \tau_2' \Leftrightarrow \tau_1' \leq \tau_1 \wedge \tau_2 \leq \tau_2'$$

If we think of coercion as set inclusion the rule above becomes obvious.

Wand and O'Keefe uses the observation of Mitchell [49] that all comparable types have the same structure. Hence the algorithm proceeds by first inferring the structure of the types and then computing a set of constraints over the base types only. We use a similar approach. The structure of a type can be written as

$$\tau_s ::= \mathsf{basetype} \mid \tau_s \to \tau_s$$

For the purpose of our example, let us use the following types:

$$\tau ::= \mathsf{PosInt} \mid \mathsf{Int} \mid \mathsf{PosReal} \mid \mathsf{Real} \mid \tau \to \tau$$

with the obvious ordering. Then we can define a relation between the structure of the types and the types themselves. If τ_s is the structure of the type τ we say that τ is an *instance* of τ_s defined in CLP(OIH) as

```
rec Instance(tau, tau') ↔
    (basetype, pos_int);
    (basetype, int);
    (basetype, pos_real);
    (basetype, real);
    (arrow, arrow)
        if Instance(tau/1, tau'/1), Instance(tau/2, tau'/2).
```

Now it is easy to define the predicate `StructuralType/3` which infers the structural type in a way similar to the type inference in Section 8.1. Having a type

structure τ_s we produce the constraint `Instance`(τ_s, τ) so that τ gets the right structure. This only works if τ_s is finite which we can assure by using the keyword `Finite`.

Now that we have inferred the structure of the types of the program and all the variables, we call the predicate `Type/3` which infers the types in the same way as in Section 8.3.1. Note that it is important that this is done in the right order. We accomplish this by using the fact that the interpreter deals with the antecedents in order from right to left. The result is the predicate `Typing/3` defined as

```
Typing(gamma, e, t) :-
    IsEnvironment(gamma), IsType(t), IsExpression(e),
    Type(gamma, e, t),
    Instance(t_s, t), Instance_gamma(gamma_s, gamma),
    StructuralEnvironment(gamma_s), StructuralType(t_s),
    Type_struct(gamma_s, e, t_s).
```

The whole program is found in Appendix A.1.2.

8.2.2 Turbo Pascal

Type inference for Turbo Pascal has been studied by Hougaard, Schwartzbach, and Askari in [32] and Hougaard and Askari in [31]. The basic method is that of generating constraints for the overloaded operators, $*$, $+$, etc.

The authors found that there is a single constraint which can be used in expressing the typeability of most of the operators. This constraint uses the relation Op which is defined such that $Op(\tau, \tau', \tau'')$, iff τ'' is the result type of an operation with argument types τ and τ'. In addition to this some of the operators have limited scope so that the constraints for e.g. $e * e'$ is $Op([\![e]\!], [\![e']\!], [\![e * e']\!])$ and $[\![e]\!], [\![e']\!] \in \mathcal{M}_*$ where $\mathcal{M}_*$ is the set of types that $*$ can operate on.

Other important relations in Turbo Pascal are type compatibility, Tc, and assignment compatibility, Ac. Basically, type compatibility tells us when the expression $e = e'$ is allowed, and assignment compatibility tells us when the assignment $e := e'$ is allowed.

These three central relations are defined in CLP(OIH) by simply enumerating the possibilities allowed in Turbo Pascal. For example the definition of `Ac` is as follows.

```
Ac(t, t') ↔
    (boolean, boolean);
    (integer, integer);
    (real, integer);
```

```
(real, real);
(char, char);
(pointer, pointer);
(pointer, pointer_to);
(string, string);
(string, char);
(set, set) if Tc(t/1, t'/1);
(pointer_to, pointer);
(pointer_to, pointer_to) if t = t', Notfile(t/1);
(array, array) if t = t', Notfile(t/2);
(record[a], record[b]) if t = t', Notfile(t).
```

where `Notfile`(τ), iff τ contains no file types.

Note the ad-hoc character of the definition above. In most well-designed programming languages the relation corresponding to assignment compability is a partial ordering or at least a preorder. Assignment compability is neither. In fact assignment compability is neither reflexive, transitive, symmetric, nor antisymmetric. Nevertheless, it is still stable with the proper ordering of the types.

The implementation of Turbo Pascal is interesting in several aspects:

- The sheer size of the problem. Turbo Pascal is a real life example and as such it contains many details that need attending. Just the amount of syntax is enough to make the program large — almost 700 lines of CLP(OIH) code.
- The ad-hoc features. Turbo Pascal contains many features that makes life easier for the programmer but more difficult for the type inference algorithm. For example the special type string can be used as an array type in some contexts but is quite different in other contexts. This means that the expression $v[e]$ can have two different typings depending on whether v is a string or an array. In CLP(OIH) we deal with this by having two rules for $v[e]$.
- The procedures and functions. Procedures and function are not just variables of a certain type as in the functional languages. They are seperate entities and need their own environments.
- Name equivalence. In Turbo Pascal two types are not equal just because they have the same structure — they need to have the same name. So if two types are defined by the equations

$$\begin{aligned} \tau &= \mathsf{record}[a : \mathsf{integer}] \\ \tau' &= \mathsf{record}[a : \mathsf{integer}] \end{aligned}$$

  they are different, but if they are defined as

$$\tau = \mathsf{record}[a : \mathsf{integer}]$$

$$\tau' = \tau$$

they are equal. In the implementation this is dealt with by making explicit equality constraints whenever necessary. The union-find algorithm inherent in the unification algorithm takes care of the necessary work so that the resulting equations defines the proper name equivalence relation.

The full implementation of the type inference algorithm for Turbo Pascal is in appendix A.2.

For the analysis of the algorithm we note that each constraint leads to a release of a constant number of constraints only. Furthermore, for all practical programs the size of the field sets are negligible. Hence, we arrive at a running time of $O(n \log n)$. In practice, though, the running time is dominated by the interpretation of the CLP(OIH) program due to the large set of labels and number of rules for all the syntax.

8.3 Type Inference with Subtyping

The subtype relation is introduced in type system to express that expressions of one type can be used also as expressions of another type. The canonical example is the *product* (or *record*) type. The expression $e.l$ is valid whenever the type of e is a product which contains the field label l. Furthermore, the construction $e.l$ is the only one that provides access to product values. What this means is that if e' is another expression with a product type with more fields that the type of e, the program would still be valid with e' replaced for e throughout. In other words, it is *sound* to consider e' to be of the same type as e. If e has type τ and e' has type τ' we say that $\tau' \leqslant \tau$.

The above leads us to the *subsumption rule*. This typing rule is common for all type systems build on subtyping.

$$\frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau} \quad \tau' \leqslant \tau$$

The main difference between the type systems then is the choice of the subtype relation.

8.3.1 Reynolds' Style Subtyping

The subtype ordering used by Reynolds in [59] has already been defined in Section 4.1. Let us recapitulate the main features:

- Smaller products give larger types.
- Larger sums give larger types.
- Prod, Sum, and List are monotone or *covariant*.
- $\to$ is monotone in the second argument but anti-monotone or *contravariant* in the first argument.

The contravariance of $\to$ stems from the fact that if $\vdash \lambda x.\, e : \tau \to \tau'$, then for all $\sigma \leqslant \tau$ and $\sigma' \geqslant \tau'$ we have $\vdash \lambda x.\, e : \sigma \to \sigma'$.

The combination of contravariance of $\to$ and covariance gives us the *field update problem*. A field update is an expression of the form $e[l := e']$. If the value of e is a product type with a field label l the value of $e[l := e']$ is a product type with the same fields as e but with the l field updated with the value of e'.

The problem is that we cannot give a reasonable type to the function $f \equiv \lambda x.x[l := x.l + 1]$. The obvious type is $\mathsf{Prod}(l : \mathsf{int}) \to \mathsf{Prod}(l : \mathsf{int})$, but if f is applied to $\langle l : 7, l' : 0\rangle$ we get

$$f : \mathsf{Prod}(l : \mathsf{int}) \to \mathsf{Prod}(l : \mathsf{int}) \vdash f\ \langle l : 7, l' : 0\rangle : \mathsf{Prod}(l : \mathsf{int})$$

and we have lost information. If we try $f : \mathsf{Prod}(l, l' : \mathsf{int}) \to \mathsf{Prod}(l, l' : \mathsf{int})$ we cannot apply f to $\langle l : 7\rangle$.

This update problem has haunted type theory for more than a decade. Cardelli and Wegner [9] has suggested that the type of f should be

$$\forall\alpha \leq \mathsf{Prod}(l : int) : \alpha \to \alpha$$

but this is unsound unless we give up on the covariance of Prod. Schwartzbach [64] gives a type system for an imperative language without contravariant constructors, in which there is no update problem. In general, the problem remains unsolved — especially for object-oriented languages which have implicit contravariance.

In Reynolds [59] there is no field update and hence no field update problem. Still, the problem can arise in other contexts. We will not discuss these issues further but instead look at the type inference problem.

The problem with the ordering is that it does not comply with the requirements of Theorem 9. In fact, the relation $\leqslant_{\mathsf{st}}$ is itself non-strict since

$$\bigwedge((\Sigma_{\mathsf{st}} \times \{\to\}) \cap R_{\mathsf{st}}) = (\to, \to)$$

and $\chi_{\mathsf{st}}(\to, \to) \neq \emptyset$. This means that an attempt to solve the inequality $x \leqslant_{\mathsf{st}} x \to x$ will result in an infinite series of approximations as in Figure 6.10.

We can improve on this situation by making an extension suggested by Reynolds himself in [59]: introducing the *universal type* and the *nonsense type*. The universal type, univ, is a subtype of every type, which means that an expression of the universal type can be used everywhere. The nonsense type, ns, is a supertype of every type, which means that every expression has type ns.

With this extension the constraint $x \leqslant_{\mathsf{st}} x \to x$ has the finite solution $x = \mathsf{univ}$. If we make the ordering between the labels so that univ is the smallest, all constant types are between univ and ns, and all other labels are greater that univ we get that $\leqslant_{\mathsf{st}}$ is a stable, strict relation.

Unfortunately, this is insufficient to satisfy the termination requirement of Theorem 9 because certain constraint sets may still be collectively non-strict, and indeed they are. First, we note that the constraint set $x \leqslant_{\mathsf{st}} x \to x, x \geqslant_{\mathsf{st}} x \to x$ can be dealt with by defining $\leqslant_{\mathsf{st}}$ explicitly as an inequality. The definition then is as follows:

```
ineq leq(t, t') ↔
    (univ, univ);
    (univ, int);
    (univ, bool);
    (univ, ns);
    (univ, prod);
    (univ, sum);
    (univ, list);
    (univ, arrow);
    (int, int);
    (int, ns);
    (bool, bool);
    (bool, ns);
    (prod[a], prod[b])
        if b <= a, ∀ i in a&b:  leq(t/i, t'/i);
    (prod, ns);
    (sum[a], sum[b])
        if a <= b, ∀ i in a&b:  leq(t/i, t'/i);
    (sum, ns);
    (list, list) if leq(t/1, t'/1);
    (list, ns);
    (arrow, arrow) if leq(t'/1, t/1), leq(t/2, t'/2);
    (arrow, ns);
    (ns, ns).
```

This definition works in many practical cases but there are still problems as we shall see at the end of this section.

For the implementation of type inference for type systems with subtyping we must consider how to implement the subsumption rule. The first idea that comes to mind is to implement it directly as the following.

```
Type(Gamma, e, t') :- Type(Gamma, e, t), leq(t, t').
```

This is not a very good idea, though, because we do not make any progress. Since the expression is the same on both sides, the rule can be used an arbitrary number of times and we cannot guarantee termination. A way of helping this situation is to use the determinism in the implementation. If we place this rule as the last one it will only be tried when all other avenues fail. This is however not very helpful. If the program is not typeable the interpretation will still not terminate, and even when the program is typeable this strategy will result in too much backtracking.

The right idea is to scatter a good number of $\leqslant_{\mathsf{st}}$ constraints wherever they can be useful. Due to the reflexivity of $\leqslant_{\mathsf{st}}$ there is no problem in using too many subsumption rules. On the other hand, having two subsumptions in a row is of no use, since by the transitivity of $\leqslant_{\mathsf{st}}$ they can be replaced by a single one. Hence, our strategy becomes one of inserting a subsumption at the end of every other rule. Using this strategy the rule for variable references (identifiers in [59]) looks as follows.

```
Type(Gamma, ident[..x..], t) :-
    env[..x:t_x..]  <= Gamma, leq(t_x, t).
```

In the same way the expression $e.l$ (encoded as `select`[`l`](`e`)) is handled by the following rule.

```
Type(Gamma, select[..l..](e), t) :-
    Type(Gamma, e, prod[..l:t_l..]), leq(t_l, t).
```

Note that we must infer exactly the type $\mathsf{Prod}(l : t_l)$ for e. This is always possible because of the use of subsumption in the recursive call. The entire program can be found in Appendix A.3.

Let us now return to the question of the termination of the algorithm. We can immediately see that the termination condition is not fulfilled due the the constraint set $x \leqslant_{\mathsf{st}} y \leqslant_{\mathsf{st}} z$. Fixing x and z as $\rightarrow$-types forces y to be an $\rightarrow$-type as well, and this in turn forces new constraints violating the strictness criterion. The same problem applies to the other constructors. For simplicity, we shall use List in the following examples but the problems apply equally to the other constructors.

The problem appears with the constraint set

$$\{x = \mathsf{List}(x), z = \mathsf{List}(z), x \leqslant_{\mathsf{st}} y \leqslant_{\mathsf{st}} z\}$$

The arc consistency will fix y as having the label List and hence it will create a free variable which is a subterm of y. Let us call this variable y'. Now, the forced constraints from the $\leqslant_{\mathsf{st}}$ constraints are $x \leqslant_{\mathsf{st}} y' \leqslant_{\mathsf{st}} z$ since x and z are their own subterms. This puts us in a situation that is isomorphic to the original constraint set, and hence the algorithm will continue forever.

As a solution we could restrict our solution space to the finite types using the keyword `Finite`. This, however, does not save us as the following constraint set shows:

$$\mathsf{List}(x) \leqslant_{\mathsf{st}} x \leqslant_{\mathsf{st}} y \leqslant_{\mathsf{st}} \mathsf{List}(y)$$

Again the forced constraints form a constraint set isomorphic to the original one, but this time no explicit cycle is formed.

Experiments suggest that the algorithm terminates for most examples. There is, however counterexamples to this. The algorithm does not terminate when run on the expression $(\lambda x.xxx)(\lambda y.y)$. We must conclude that Reynolds style subtyping is beyond implementation in CLP(OIH). This is perhaps not that surprising. As far as this author knows, there is no known algorithm for type inference with Reynolds style subtyping.

Palsberg, Wand, and O'Keefe [57, 55] have studied the type inference problem for a simpler type system which has only $\rightarrow$, univ, and ns (which they call $\rightarrow$, $\bot$, and $\top$, respectively). They find that the typeability problem is decidable in polynomial time for regular as well as finite types, and that the result holds even if base types are added to the system. It is still open, though, whether it is possible to infer minimum-size types in the presence of base types. The algorithm used for these systems are highly specialised, and it is no surprise that these simplifications is insufficient to make the system fit in into CLP(OIH).

It is worth noting that it is decidable whether a constraint set is isomorphic to another. Hence, it is possible to detect when these kinds of infinite loops occur, and therefore possible to decide Reynolds style subtyping in the finite case. This avenue does not look promising for practical purposes, though.

8.3.2 The Object Calculus

The object calculus (or ς-calculus) has been devised by Abadi and Cardelli in [1]. It serves as a tool for investigating aspects of object-oriented programming languages and type systems in a controlled environment. The key feature that will interest us here is the use of the subsumption rule to implement subclassing.

The calculus implements three operations: Object definition, method send, and method override. The syntax is as follows:

$$o ::= x \mid [l_1 = \varsigma(x_1)o_1, \ldots, l_n = \varsigma(x_n)o_n] \mid o.l \mid o.l \Leftarrow \varsigma(x)o'$$

In the definition above, x and the x_i are variables, $[l_1 = \varsigma(x_1)o_1, \ldots, l_n = \varsigma(x_n)o_n]$ is an object definition with method names $l_1, \ldots, l_n$ and methods $\varsigma(x_1)o_1$, $\ldots$, $\varsigma(x_n)o_n$, $o.l$ is a method send, and $o.l \Leftarrow \varsigma(x)o'$ is a method override.

We shall write $o = [l_i = \varsigma(x_i)o_i{}^{i\in 1..n}]$ for the object $o = [l_1 = \varsigma(x_1)o_1, \ldots, l_n = \varsigma(x_n)o_n]$. If o is like above we have that the method send $o.l_j$ reduces to $o_j[x_j \leftarrow o]$ and the method override $o.l_j \Leftarrow \varsigma(y)o'$ reduces to

$$[l_1 = \varsigma(x_1)o_1, \ldots, l_j = \varsigma(y)o', \ldots, l_n = \varsigma(x_n)o_n]$$

It follows that $o.l$ and $o.l \Leftarrow \varsigma(x)o'$ yield errors if o does not implement a method named l. One of the goals of typing rules is to catch these errors statically.

The type of an object $[l_i = \varsigma(x_i)o_i{}^{i\in 1..n}]$ is $[l_i : \tau_i{}^{i\in 1..n}]$ where τ_i is the type of $o_i[x_i \leftarrow o]$ — that is, the return type of the method. For example, we have that the object $[l = \varsigma(x).x]$ has the infinite type τ defined by $\tau = [l : \tau]$.

The set of all types is defined in CLP(OIH) by the label `object` defined by

```
object[l]/|l|
```

The typing rules are based on this key observation: Whenever $o.l$ is legal and o' is like o but with added methods $o'.l$ is legal as well. The same can be said for method override. Hence, we have that $[l_i : \tau_i^{i\in 1..n}] \leqslant_{\mathsf{oc}} [l_i : \tau_i^{i\in 1..m}]$ if $m \leq n$. In order to escape the problem with updating (see section 8.3.1) there is no congruence rule in the definition of $\leqslant_{\mathsf{oc}}$; that is, object types are not covariant.

With this we can define `Leq_oc` *non-recursively* in CLP(OIH) as follows:

```
Leq_oc(t, t') ↔
    (object[l], object[m])
        if m <= n, ∀ i ∈ l & m:  t/i = t'/i.
```

Since this is clearly stable, it follows that the problem is in OIH. In fact, `Leq_oc` is the transposed of the built-in <=-relation restricted to object types. That is, for all object types, τ and τ', we have that `Leq_oc`(τ, τ'), iff $\tau' <= \tau$.

In [52], Palsberg gives a reduction from a type inference problem to a constraint satisfaction problem using $\leqslant_{\mathsf{oc}}$-constraints. We can implement this reduction directly in CLP(OIH).

The constraints for the expression $o.l$ are given in [52] as $[\![o]\!] \leqslant_{\mathsf{oc}} [l : \langle o.l \rangle]$ and $\langle o.l \rangle \leqslant_{\mathsf{oc}} [\![o.l]\!]$, where $[\![e]\!]$ and $\langle e \rangle$ are type variables. In CLP(OIH) this becomes

```
Type(Γ, message[..l..](a), t) :-
    Type(Γ, a, t_a), object[..l:t'..]  <= t_a, t <= t'.
```

The object definition $[l = \varsigma(x)x]$ is encoded as `object_def[l:sigma[x](var[x])]`. Hence we have two rules for the typing of object definitions:

```
Type(Γ, object_def[..l:m..],t) :-
    t <= t', t' = object[..l:t_b..], Type(Γ, m, t', t_b).

Type(Γ, sigma[..x..](b), t_x, t) :-
    Type(Γ', b, t), Γ <= Γ', env[..x:t'..]  <= Γ', t' = t_x.
```

This corresponds to the constraints $[l_i : [\![o_i]\!]^{i\in 1..n}] \leqslant_{\mathsf{oc}} [\![[l_i = \varsigma(x_i)o_i^{i\in 1..n}]]\!]$ and $x_i = [l_i : [\![o_i]\!]^{i\in 1..n}]$. In the latter constraints, x_i are type variables standing for $\Gamma(x_i)$.

The full program can be seen in Appendix A.4.1. It is possible to extend the type inference program to a language with added atomic types. An example of this is in Appendix A.4.2.

Palsberg [52] gives an algorithm for the solution of systems of $\leqslant_{\mathsf{oc}}$ constraints. The algorithm maintains a directed graph — an *AC-graph*. An AC-graph has two kinds of nodes — N-nodes and S-nodes — and two kinds of edges — L-edges and $\leqslant$-edges. The L-edges are labelled with method names, and they can go from N-nodes only.

The idea of the AC-graph is that the S-nodes represent type variables and the N-nodes expressions of the form $[l_i : {\tau_i}^{i\in 1..n}]$. In the latter case there would be n L-edges from the corresponding N-node. The ith edge would be labelled l_i and go to the (S-)node representing τ_i. The $\leqslant$-edges represent the $\leqslant_{\mathsf{oc}}$ constraints.

Given an AC-graph representing constraint system, Palsberg computes the closure of the graph under the following properties:

1. The graph consisting of the $\leqslant$-edges is reflexive and transitive.
2. For all nodes u, v, v', w, and w', whenever there are edges $u \xrightarrow{\leqslant} v \xrightarrow{l} w$ and $u \xrightarrow{\leqslant} v' \xrightarrow{l} w'$ there should be edges $w \xrightarrow{\leqslant} w'$ and $w' \xrightarrow{\leqslant} w$.

If the resulting graph is well-formed the constraint set is satisfiable. An AC-graph is well-formed, iff for all N-nodes u and v with $u \xrightarrow{\leqslant} v$ we have that if v has an outgoing edge labelled l, so has u.

It is interesting to note the similarities and differences between Palsbergs algorithm and the one given in this work (see Chapter 6). The L-edges of Palsberg corresponds to the transitions of the base automaton; the $\leqslant$-edges corresponds to the constraints in E; the reflexivity, transitivity, and well-formedness of the AC-graph corresponds to the consistency requirement; and the second closure property corresponds to the releasing of forced constraints.

The main difference is that Palsberg distinguishes S-nodes from N-nodes. There are two major consequences of this:

1. When Palsberg releases the forced constraints all he needs to do is to make two edges between S-nodes. This will not result in any further releasing of forced constraints, since S-nodes have no outgoing L-edges.

   In contrast, our released constraints may be between nodes with outgoing transitions. For this reason we release $=$-constraints rather than two `Leq_oc`-constraints. As a matter of fact, if we had defined `Leq_oc` by

   ```
   ineq Leq_oc(t, t') ↔
       (object[l], object[m])
           if m <= n,
              ∀ i ∈ l & m:  Leq_oc(t/i,t'/i),
              ∀ i ∈ l & m:  Leq_oc(t'/i,t/i).
   ```

   the program might not terminate. The result is that less work is required in Palsbergs algorithm than in the algorithm presented here.

2. Palsberg does not obtain the result explicitly. In our case we have for each type variable the set of method names in the root of its type as well as direct access to all the subtrees. This is not possible with the prohibition of outgoing L-edges. Instead, Palsberg outputs the result as a *non-deterministic* term automaton. It is not clear how to efficiently obtain a deterministic term automaton or a set of equations from a non-deterministic term automaton.

For the analysis of the algorithm, we must take a closer look at what is happening. Assume that the constraint system has m constraints using p variables and k distinct labels. There can be at most k transitions out of each variable, this gives us at most pk new variables. We release at most mk $=$-constraints over these new variables, and each of these $=$-constraints give rise to 2 $\preceq$-constraints. There will be transitions out of the new variables only if the released $=$-constraints result in unification with original variables. Hence, there will not be generated any variables beyond the pk new variables.

We now have the figures to plug into the formula of Proposition 6.6.2. The sum of the sizes of the $\leqslant_{\mathsf{oc}}$-constraints is m times the size of $\leqslant_{\mathsf{oc}}$ plus the sum of the sizes of the expressions. The latter of these is $\mathrm{O}(p+k)$ and the size$(\leqslant_{\mathsf{oc}}) = 5$, so the time contributed by the $\leqslant_{\mathsf{oc}}$-constraints is $\mathrm{O}(m+p+k)$. Similarly, the time contributed by the $=$-constraints is $\mathrm{O}(mk)$. The $m + 2mk$ $\preceq$ *−constraints* contribute by a time of $\mathrm{O}(mk^2)$. In all we have that the total time is

$$\mathrm{O}(m+p+k+mk+mk^2) = \mathrm{O}(p+mk^2)$$

Now, assume that we are given a program of size n. By inspection of the constraints we can see that there can be no more than 4 constraints and 2 type variables for each node in the syntax tree, and that all the labels in the constraint set

come from the program. Hence we have that p, m, and k are all $O(n)$, and that the total running time of the type inference program is $O(n + n \cdot n^2) = O(n^3)$ — the same as in [52].

We conclude that the algorithm presented here is to be preferred to the one in [52] since it gives the types explicitly, yet has the same running time.

The lack of covariance of the $\leqslant_{\mathsf{oc}}$-relation raises certain problems. Consider the following object definitions:

$$\begin{aligned} o &= [l = \varsigma(x)x] \\ o' &= [l = \varsigma(x)x, l' = \varsigma(y)y.l'] \end{aligned}$$

Note that o' is just o with an added method. We have that o has the type τ_o defined by $\tau_o = [l : \tau_o]$ and similarly that o' has type τ'_o where $\tau'_o = [l : \tau'_o, l' : \tau'_o]$. Unfortunately, $\tau'_o \not\leqslant_{\mathsf{oc}} \tau_o$, so the natural types of these objects does not have the proper relation.

In the above example we can find types for o and o' that do in fact have the proper ordering between them (as both have type []), but it becomes a problem when o and o' becomes parts of larger programs. Palsberg and Jim [54] gives the following example of a program that should be typeable, but is not:

$$\begin{aligned} \mathit{Point} &= [\mathit{move} = \varsigma(x)x] \\ \mathit{ColourPoint} &= [\mathit{move} = \varsigma(y)y, \mathit{setcolour} = \varsigma(z)z] \\ \mathit{Circle} &= [\mathit{centre} = \varsigma(d)\mathit{Point}] \\ \mathit{ColourCircle} &= \mathit{Circle.centre} \Leftarrow \varsigma(e)(\mathit{ColourPoint.move.setcolour}) \\ \mathit{Main} &= \mathit{ColourCircle.centre.move} \end{aligned}$$

The example is the core of a quite realistic program involving coloured and uncoloured geometrical figures. The problem is like in the example above that the natural types of *Point* and *ColourPoint* are unrelated, and that the body of *ColourCircle*'s *centre* method makes it impossible to find a type for *ColourPoint* which is a subtype of the type of *Point*.

The whole problem arises in the use of recursiveness to capture the type of self. A more promising approach is to introduce an explicit selftype (similar to like Current in Eiffel). With the explicit selftype the object o would be typed as $[l : \mathsf{selftype}]$ and o' as $[l, l' : \mathsf{selftype}]$. Now we have that $[l, l' : \mathsf{selftype}] \leqslant_{\mathsf{oc}} [l : \mathsf{selftype}]$, so the problem above has been overcome.

The type expression selftype always refers to the enclosing object type, so we cannot allow type judgements of the form $\Gamma \vdash e : \mathsf{selftype}$. This means that when we have an expression e of type $\tau \leq_{\mathsf{oc}} [l : \tau']$, the expression $e.l$ will have type τ

if $\tau' =$ selftype (rather that having type selftype). Of course, if $\tau' \neq$ selftype, $e.l$ will have type τ'. This can be expressed using the following notation:

$$\tau'\{\tau\} = \begin{cases} \tau & \text{if } \tau' = \text{selftype} \\ \tau' & \text{otherwise} \end{cases}$$

Using this notation we can write the type of $e.l$ as $\tau'\{\tau\}$.

From a type inference point of view the problem is that expressions of the form $\tau'\{\tau\}$ enter the constraints. We can try to solve this problem by introducing a constraint $Alt(\tau, \tau', \tau'')$ which is true, iff $\tau'\{\tau\} = \tau''$. The relation *Alt* is defined as follows

```
Alt(tau, tau', tau'') ↔
    (selftype, selftype, selftype);
    (object[l], selftype, object[l']) if tau = tau'';
    (selftype, object[l], object[l']) if tau' = tau'';
    (object[l], object[l'], object[l'']) if tau' = tau''.
```

Unfortunately, this relation is unstable regardless of the ordering. If we use the ordering `selftype` $<$ `object`, we get the problem that

$$\begin{aligned} &(\texttt{object}, \texttt{selftype}, \texttt{object}) \wedge (\texttt{selftype}, \texttt{object}, \texttt{object}) \\ &= (\texttt{selftype}, \texttt{selftype}, \texttt{object}) \end{aligned}$$

and the latter tuple is not in the relation. If we let `object` $<$ `selftype`, we get the problem that χ is non-monotone.

There is no use trying to find a way to express *Alt* using stable relations. Palsberg and Jim [54] has proven that the type inference problem for the object calculus with selftype is *NP*-complete. Instead we implement *Alt* as a predicate in CLP(OIH) as follows:

```
Alt(selftype, tau, tau).
Alt(tau', tau, tau') :- NotSelftype(tau').
```

This of course results in a potentially exponential search for a successful computation (see Section 3.3). But since the problem is *NP*-complete this is the best we can hope for unless $P = NP$.

With this addition to the program, we can type the ColourCircle example. The full program is in Appendix A.4.3.

In [53] Palsberg gives a non-deterministic polynomial-time algorithm for solving the type inference problem. The algorithm guesses which of the type variables, v, in expressions of the form $v\{e\}$ ends up as selftype and proceeds with the ordinary algorithm (from [52]). This corresponds to guessing which of the lines in the definition of `Alt` should be used.

8.4 Control Flow Analysis

In functional languages we do not have the same control flow information at compile time as we have for imperative languages. As an example look at the expression (adapted from [66])

$$(\lambda f.\lambda g.\lambda x.if(x > 0)(fx)(gx))(\lambda y.y)(\lambda z.z)$$

How does the control flow look at the *if*-expression? As in any programming language it is undecidable whether the control passes to f or g but unlike most languages it is equally undecidable which functions (or λ expressions) f and g represents — it will be decided at run-time only.

The control flow problem is the problem of finding at each of the call sites which functions could be called at that call site. In the current example we have that the set of functions f could be instantiated to is $\{(\lambda y.y)\}$.

We can also talk of the *effect* of evaluation expressions. The effect of evaluating an expression is the set of functions that are called during the evaluation of that expression. Returning to the example, the effect of evaluating (fx) is $\{(\lambda y.y)\}$ while the effect of evaluating the entire expression is $\{(\lambda f.\ldots), (\lambda g.\ldots)\}$. Note that the effect does not contain $(\lambda y.y)$. This is because the function will not be called before the expression is applied to an argument. We say that the expression has a *latent* effect, and write this with the types as below.

$$\vdash (\lambda f.\lambda g.\lambda x.if(x > 0)(fx)(gx))(\lambda y.y)(\lambda z.z) : i \xrightarrow{\{(\lambda y.y),(\lambda z.z),(\lambda x....)\}} i$$

For simplicity we name the λ-abstractions and omit the brackets, so that we can write the type above as $i \overset{n_y,n_z,n_x}{\longrightarrow} i$.

In her Ph.D.-dissertation [69], Yan-Mei Tang examined the use of effect systems for control flow analysis. We shall see how these results are implemented in CLP(OIH).

The basic elements of using effect systems are captured in the example language

$$e ::= x \mid (\lambda_n\, x.\, e) \mid \mathsf{letrec}_n\, f(x) = e \mid (e\ e')_l$$

Note that we have named the λ-abstractions as well as the letrec definitions. In addition we have a label on each call site (function application) in order to abstract the flow information.

The control flow information is a function from call sites to effects. For each call site the function gives the set of (names of) λ-abstractions that are called in the execution of the function call.

The problem is — as noted above — that finding the *exact* set of λ-abstraction called is undecidable. Hence we must restrict ourselves to finding a set containing *at least* every λ-expression called.

A trivial solution to the above is the set of all λ-abstractions in the program, but this is clearly uninformative. The goal is to find as small a set as possible. Yan-Mei Tang examines two strategies for obtaining this: *subeffecting* [70] and *subtyping* [71].

8.4.1 Effect Systems using Subeffecting

The control flow analysis is derived by a static semantics of the language. The static semantics is basically a type systems where types can contain latent effects and effects are derived along with the types. The types are defined as below.

$$\tau ::= i \mid \tau \xrightarrow{c} \tau'$$

where c is an effect — that is, a set of λ-abstraction names. Note that there is a type i without a matching syntactical construct. We deal with integers by predefined variables of type i.

The typing judgements are of the form $\Gamma \vdash e : \tau, c$, where Γ is the usual type environment, τ is a type (with latent effects), and c is an effect. For instance, the typing rule for λ-abstractions is as follows.

$$\frac{\Gamma[x \mapsto t_x] \vdash e : \ t, c}{\Gamma \vdash \lambda_n x.\, e : \ t_x \xrightarrow{\{n\} \cup c} t, \emptyset}$$

Note that the effect of the λ-abstraction is $\emptyset$. All effects from the body of the λ as well as $\{n\}$ itself are latent. The effects appear when we apply the abstraction to something as the following rule shows.

$$\frac{\Gamma \vdash e : t' \xrightarrow{c''} t, c \quad \Gamma \vdash e' : t', c'}{\Gamma \vdash (e\ e')_l : t, c \cup c' \cup c''}$$

Now, consider the expression $\lambda_{n_a} a.\, a$. If we assume that a is an integer we get the type judgement $a : i \vdash a : i, \emptyset$. From the rule for λ-abstractions above we get $\vdash \lambda_{n_a} a.\, a : i \xrightarrow{n_a} i, \emptyset$. Similarly, we get $\vdash \lambda_{n_b} b.\, b : i \xrightarrow{n_b} i, \emptyset$.

If we observe the rule for function above we can see that the type of the argument must be the same for every application. Now, since the types of $\lambda_{n_a} a.\, a$ and $\lambda_{n_b} b.\, b$ are different it follows that there is *no* type, τ, so that we have for some τ_a, τ_b, c_a, and c_b that $f : \tau \vdash f(\lambda_{n_a} a.\, a) : \tau_a, c_a$ and $f : \tau \vdash f(\lambda_{n_b} b.\, b) : \tau_b, c_b$.

This is a very unfortunate situation. It means that there are programs which are typeable in the usual type system but cannot be analysed because they are untypeable in the effect system. Tang [69] gives the following example:

$$((\lambda_{n_f} f.\,(+\ (f(\lambda_{n_a} a.\,a))_{l_a}\ (f(\lambda_{n_b} b.\,b))_{l_b}))(\lambda_{n_g} g.\,(g\ 1)_{l_g}))_{l_f}$$

Here we cannot find a suitable type for f.

This example shows that we have to allow some slack in our type judgement. The simplest way to do this is to introduce the slack in the effects by the following rule.

$$\frac{\Gamma \vdash t, c'}{\Gamma \vdash t, c} \quad c' \subseteq c$$

Using this rule we can infer $a : i \vdash a : i, \{n_b\}$ and hence $\vdash \lambda_{n_a} a.\,a : i \xrightarrow{n_a, n_b} i, \emptyset$. The same type can be inferred for $\lambda_{n_b} b.\,b$. Now, the example is typeable with the type $(i \xrightarrow{n_a, n_b} i) \xrightarrow{n_a, n_b, n_g} i$ for f. In fact, every program which is typeable in the usual type system is now typeable in the effect system.

The implementation in CLP(OIH) is straightforward. We use two labels `effect` and `flow` defined as

```
effect[c]/0,
flow[l]/|l|
```

We encode an effect $c = \{n_1, \ldots, n_k\}$ as the term $\texttt{effect}[\texttt{n}_1, \ldots, \texttt{n}_\texttt{k}]$, the flow is a mapping from call sites to effects, encoded in the obvious fashion. To allow latent effects the label for function types are defined as

```
arrow[c]/2
```

The names of the λ-abstractions and letrec-expressions and the labels of the call sites are encoded using the following labels.

```
lambda[name]/2,
letrec[name]/3,
app[site]/2,
```

We encode $\lambda_n\, x.\, e$ as `lambda[n](var[x], e)` and $(e\ e')_l$ as `app[l](e, e')`.

The program defines a predicate `Analyse/5`, such that `Analyse(`Γ`,e,t,eff,f)` iff $\Gamma \vdash e : t, \mathit{eff}$, and f records the flow information. The `Analyse` predicate makes use of the predicate `Type/5`. As usual `Type` is the predicate that encodes the type deduction system.

The implementation of `Type` differs little from the corresponding predicate in the simply typed λ-calculus. See for example the rule for variables

```
Type(Gamma, var[..x..], t, eff, f) :-
    env[..x:t'..]  <= Gamma, t = t', effect[] <= eff.
```

According to the typing rule for variables the effect is $\emptyset$. The inequality is there to allow for possible subsequent subeffecting.

The rule for applications is similar:

```
Type(Gamma, app[..site..](e, e'), t, eff, f) :-
    Type(Gamma, e, arrow[..c..](t',t), eff', f),
    Type(Gamma, e', t', eff'', f),
    eff' <= eff, eff'' <= eff, effect[..c..]  <= eff,
    flow[..site:  eff'''..]  <= f, eff''' = eff.
```

Note how we use three inequalities $c \subseteq \mathit{eff}$, $\mathit{eff}' \subseteq \mathit{eff}$, and $\mathit{eff}'' \subseteq \mathit{eff}$ instead of the single inequality $c \cup \mathit{eff}' \cup \mathit{eff}'' \subseteq \mathit{eff}$. They are clearly equivalent but the latter is impossible to express in CLP(OIH) because of the use of $\cup$. This means that the restrictive version without the use of subeffecting cannot be implemented in CLP(OIH).

The algorithm given above is similar to the $\mathcal{R}$ algorithm of Yan-Mei Tang [69]. It can be seen in Appendix A.5.1.

8.4.2 Effect Systems using Subtyping

In the previous section we used subeffecting to be able to apply the same function variable to both $\lambda_{n_a} a.\, a$ and $\lambda_{n_b} b.\, b$. The result was that both were given the type $i \xrightarrow{n_a, n_b} i$. This results in a loss of information because the actual latent effects are smaller than the analysed effect.

Another way to go is to allow the application of the same function variable to arguments of different types. We can achieve this by the use of *subtyping*. Basically, a function type, $\tau \xrightarrow{c} \tau'$, is a subtype of another, $\tau \xrightarrow{c'} \tau'$, if it has a smaller latent effect. That is, if $c \subseteq c'$. When we extend this definition with the usual contravariant rule we have that $\leqslant_{\mathsf{es}}$ is the smallest relation such that:

- $i \leqslant_{\mathsf{es}} i$
- $\tau_1 \xrightarrow{c} \tau_2 \leqslant_{\mathsf{es}} \tau_1' \xrightarrow{c'} \tau_2'$, iff $c \subseteq c'$, $\tau_1' \leqslant_{\mathsf{es}} \tau_1'$, and $\tau_2 \leqslant_{\mathsf{es}} \tau_2'$.

As usual the ordering is introduced into the system by the subsumption rule:

$$\frac{\Gamma \vdash t', c}{\Gamma \vdash t, c} \quad t' \leqslant_{\mathsf{es}} t$$

Using this rule we can infer

$$f : (i \xrightarrow{n_a,n_b} i) \xrightarrow{n_a,n_b,n_c} i \vdash f : (i \xrightarrow{n_a} i) \xrightarrow{n_a,n_b,n_c} i, \emptyset$$
$$f : (i \xrightarrow{n_a,n_b} i) \xrightarrow{n_a,n_b,n_c} i \vdash f : (i \xrightarrow{n_b} i) \xrightarrow{n_a,n_b,n_c} i, \emptyset$$

And so we have that f is applicable to both $\lambda_{n_a} a.\, a$ and $\lambda_{n_b} b.\, b$ without having to change the types of these expressions.

In CLP(OIH), the ordering is defined by the following.

```
ineq Subtype(t, t') ↔
    (int, int);
    (arrow[c0], arrow[c1])
        if c0 <= c1,
           Subtype(t'/1, t/1),
           Subtype(t/2, t'/2).
```

Unfortunately, this relation does not guarantee termination. Assume that `Subtype` is $\nu X.\langle\!\langle R, \chi \rangle\!\rangle$. Then we have that $\wedge(R \cap (\{\rightarrow\} \times \Sigma)) = (\rightarrow, \rightarrow)$, but $\chi(\rightarrow, \rightarrow) \neq \emptyset$. This means that an attempt to solve the inequality $x \leqslant_{\mathsf{es}} x \xrightarrow{\emptyset} x$ would result in an infinite loop.

In order to remedy this situation we use the observation of Tang [69, 71] that the *structure* of the types is the same as for the classical types. Similarly to Tang's algorithm $\mathcal{S}$ and the algorithm from Section 8.2.1 we start by finding the classical types of the program and then — if they are finite — proceed with the analysis.

In Tang [69, 71], a classical type is a type generated by the grammar

$$\tau_c ::= i | \tau_c \rightarrow \tau_c'$$

Here, we encode classical types as non-classical types with no constraints over the latent effects. In this way we implement a predicate `Classical_type/3` such that `Classical_type`$(\Gamma, \mathtt{e}, \mathtt{t})$, iff $\Gamma \vdash_c e : \mathit{Erase}(t)$, where $\vdash_c$ is the classical type judgement and *Erase* is a function that removes the latent effects from a type.

Now, it suffices to run the classical type inference program on the expression (remembering the types in `gamma`), and — if the types are finite — call the inference algorithm with subtyping.

Two things are important:

1. The types should be finite.
2. The call to `Classical_type/3` should occur *before* the call to `Type/5`.

The first goal is obtained by using the keyword `Finite`. How to obtain the second is implementation dependent. Many implementations of Prolog and other CLP-languages have constructs that allow you to determine the order in which the

predicates should be picked. In CLP(OIH) no such construct is implemented, but we can use the knowledge that the predicates are read from right to left. Using this knowledge the predicate `Analyse/5` is implemented as

```
Analyse(Gamma, e, t, eff, f) :-
    Type(Gamma, e, t, eff, f), Classical_ type(Gamma, e, t),
    Environment(Gamma), IsType(t), IsEffect(eff), IsFlow(f).
```

The program can be seen in its entirety in Appendix A.5.2.

Conclusion

We have studied the constraints used in different type inference problems and found that they belonged to a general constraint domain: The ordered infinitary Herbrand universe or OIH. We have studied the complexity properties of this universe and found that there was an important quality that the constraints needed for allowing efficient solving — the quality of stability. When relations are stable we know that we can maintain a conservative approximation to the set of solutions and then find the minimal solution by local minimisation in the conservative approximation.

We provided an efficient constraint solver for the OIH constraint domain and showed that the constraints have polynomial solutions if they are stable. In addition we found that different degrees of strictness came in to play in the termination conditions for the constraint solver.

In order to design and implement the CLP(OIH) language we studied the issue of backtracking and came up with the stack persistency algorithm, which works on all programs that use a RAM for storage. In addition, we looked at a way of formally defining the ellipsis that are so often used in typing rules and other formalisms. We found that there was a definition which both expressed the way ellipsis are used in these cases and fitted into the CLP scheme. We also devised an interpreter for a CLP language with this extension.

We used this insight to implement an interpreter for CLP(OIH). We have thus provided a programming language well-suited for type inference, which we demonstrated by implementing a number of type inference problems in the language.

Our work left a number of open problems and things to consider.

- The stack persistency algorithm turned a worst-case complexity into an amortised complexity. An interesting open problem is whether there is a stack persistency algorithm with linear worst-case complexity.
- In the implementation of the type inference algorithm the syntax of the languages entered into the constraint satisfaction problem. It will probably be useful to redesign the language using a two-sorted algebra so that the

syntax and types are kept separate.

- In some of the applications the order of the rules became important. It may be useful to include features which allow the programmer to control the order of interpretation.
- In connection with Reynolds' style subtyping we saw a problem for which we could not guarantee termination. The example clearly showed that there are varying degrees of non-termination from the completely non-strict recursive relation to the strict (but not strictness-closed) inequalities over the finitary domain. It would be interesting to pursue this difference in degrees further and see whether there is an efficient way to implement Reynolds' style subtyping and several other similar problems.

Bibliography

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Proc. TACS'94, Theoretical Aspects of Computing Software*, pages 296–320. LNCS 789, Springer-Verlag, 1994.

[2] Alexander Aiken, Dexter Kozen, Moshe Vardi, and Ed Wimmers. The complexity of set constraints. In *Proc. 7th Int'l Workshop on Computer Science Logic*, pages 1–17. LNCS 832, Springer Verlag, 1993.

[3] Alexander Aiken, Dexter Kozen, and Ed Wimmers. Decidability of systems of set constraints with negative constraints. *Information and Computation*, 122(1):30–44, October 1995.

[4] Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *Proc. 18th Symp. Principles of Programming Languages*, pages 279–290. ACM, 1991.

[5] Alberto Apostolica, Giuseppe F. Italiano, Giorgio Gambosi, and Maurizio Talamo. The set union problem with unlimited backtracking. *SIAM Journal on Computing*, 23(1):50–70, 1994.

[6] Henk Barendregt and Kees Hemerik. Types in lambda calculi and programming languages. In *Proc. ESOP'90, European Symposium on Programming*, pages 1–35. LNCS 432, Springer-Verlag, 1990.

[7] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.

[8] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint hierarchies and logic programming. In *Proc. 6th Int'l Logic Programming Conference*, pages 149–164, June 1989.

[9] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–520, December 1985.

[10] Alain Colmerauer. Prolog and infinite trees. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, New York, New York, 1982.

[11] Alain Colmerauer. *PROLOG II — Reference Manual and Theoretical Model*. Université Aix-Marseille, 1982.

[12] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–91, July 1990.

[13] Bruno Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory*, 13:131–180, 1979.

[14] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(1):95–169, 1983.

[15] Luis Damas and Robin Milner. Principal type schemes for functional programming. In *9th ACM conf. on Principels Of Programming Languages*, 1982.

[16] R. Detcher and J. Pearl. Network based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34, 1988.

[17] M. Dincbas et al. The constraint logic programming language CHIP. In *5th Generation Computer Systems*, volume 2, pages 693–702. Springer-Verlag, November 1988.

[18] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[19] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal of Computing*, 5(4):691–703, December 1976.

[20] Bjorn N. Freeman-Benson and Alan Borning. The design and implementation of kaleidoscope'90, a constraint imperative programming language. In *International Conference on Computer Languages*, pages 174–180. IEEE, 1992.

[21] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 21(11), 1978.

[22] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Dynamization of backtrack-free search for the constraint satisfaction problem. In M. Bonucelli, P. Crescenzi, and R. Petreschi, editors, *Algorithms and Complexity*, pages 136–151. LNCS 778, Springer-Verlag, February 1994.

[23] Früwirth et al. Constraint logic programming — an informal introduction. ECRC 92-6i, ECRC, 1992.

[24] Rémi Gilleron, Sophie Tison, and Marc Tommasi. Solving systems of set constraints with negated subset relationships. In *Proc. 34th Symp. Foundations of Coomput. Sci.*, pages 372–280. IEEE, 1993.

[25] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[26] Nevin Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, 1993.

[27] Nevin Heintze and Joxan Jaffar. A finite presentation theorem for approximating logic programs. In *Proc. 17th Symp. Principles of Programming Languages*, pages 111–119. ACM, 1990.

[28] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[29] C. J. Hogger. *Introduction to Logic Programming*. Academic Press, London, 1984.

[30] J. Hopcroft and J. D. Ullman. Set-merging algorithms. *SIAM Journal of Computing*, 2:294 – 303, 1973.

[31] Ole I. Hougaard and Hosein Askari. Implicit typing for turbo pascal. Master's thesis, Dep. of Computer Science, University of Aarhus, 1994. In Danish.

[32] Ole I. Hougaard, Michael I. Schwartzbach, and Hosein Askari. Type inference for Turbo Pascal. *Software — Concepts and Tools*, 16:160–169, 1995.

[33] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proc. 14th Principles of Prog. Lang.* ACM, January 1987.

[34] Joxan Jaffar, Michael Maher, Peter Stuckey, and Roland Yap. Output in CLP($\mathcal{R}$). In *Proceedings of the 1992 Conference on Fifth Generation Computer Systems, Tokyo*, 1992.

[35] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[36] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type recursion in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.

[37] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41:368–398, 1994.

[38] Dexter Kozen. Logical aspects of set constraints. TR 94-1421, Department of Computer Science, Cornell University, Ithaca, New York, May 1994.

[39] Dexter Kozen. Set constraints and logic programming. TR 94-1467, Computer Science Department, Cornell University, November 1994.

[40] Dexter Kozen. Set constraints and logic programming (abstract). In *Proc. 1st Conf. Constraint in Computational Logics*, pages 302–303. LNCS 845, Springer-Verlag, 1994.

[41] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proc. 20th Symp. Princip. Programming Lang.*, pages 419–428. ACM, January 1992.

[42] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(3):289–308, 1987.

[43] J. A. La Poutré and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Lecture Notes in Computer Science 314, Springer-Verlag, Berlin, 1988.

[44] H.-P. Lenhof and M. Smid. Using persistent data structures for adding range restrictions to searching problems. Technical Report A 22/90, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1990.

[45] Harry G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In *Seventeenth Symposium on Principles of Programming Languages*, pages 382–401. ACM Press, January 1990.

[46] John Maloney. *Using Constraints for User Interface Construction*. PhD thesis, University of Washington, 1991. Department of Computer Science, Technical Report 91-08-12.

[47] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348 – 375, 1978.

[48] Prateek Mishra. Towards a theory of types in Prolog. In *Proc. 1st Symp. Logic Programming*, pages 289–298. IEEE, 1984.

[49] John C. Mitchell. Coercion and type inference. In *11th ACM Symp. on Principles of Programming Languages*, pages 175–184, 1984.

[50] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[51] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making Type Inference Practical. In *Proceedings of the ECOOP '92 European Conference on Object-oriented Programming*, pages 329–349. LNCS 615, Springer-Verlag, 1992.

[52] Jens Palsberg. Efficient inference of object types. In *9th Logic in Computer Science*, pages 186–195. IEEE Computer Society Press, July 1994.

[53] Jens Palsberg. Type inference with selftype. Technical Report RS-95-34, BRICS, 1995.

[54] Jens Palsberg and Trevor Jim. Type inference with simple selftypes is NP-complete. *Nordic Journal of Computing*, 4(3):259–286, 1997.

[55] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Proc. POPL '95, 22nd Symposium on Principles of Programming Languages*, pages 376–378, 1995.

[56] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *6th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161. ACM SIGPLAN, 1991.

[57] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computer Science*, 9:49–67, 1997.

[58] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[59] John Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development.* LNCS 185, Springer Verlag, 1985.

[60] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[61] Michael Sannella. The skyblue constraint solver and its applications. In *Workshop on Principles and Practice of Constraint Programming*, pages 385–406. MIT Press, 1993.

[62] Michael Sannella, John Maloney, Bjorn N. Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software — Practice and Experience*, 23(5):529–566, May 1993.

[63] Uwe Schöning. *Logic for Computer Scientists.* Birkhäuser, 1989.

[64] Michael I. Schwartzbach. Static correctness of hierarchical procedures. In *ICALP'90.* LNCS 443, Springer-Verlag, 1990.

[65] Michael I. Schwartzbach. Type inference with inequalities. In *Proceedings of TAPSOFT'91.* LNCS 493, Springer-Verlag, 1991.

[66] Olin Shivers. Control-flow analysis in Scheme. *ACM SIGPLAN Notices*, 23(7):164–174, 1988. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation.

[67] G. S. Smith. Polymorphic type inference with overloading and subtyping. In *TAPSOFT '93: Theory and Practice of Software Development*, pages 671–685. LNCS 668, Springer-Verlag, 1993.

[68] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):171–215, 1977.

[69] Yan-Mei Tang. *Systèmes d'Effet et Interprétation Abstraite pour l'Analyse de Flot de Contrôle*. Rapport a/258/cri, Ecole de Mines de Paris, 1994.

[70] Yan-Mei Tang and Pierre Jouvelot. Control-flow effects for closure analysis. In *Proceedings og the 2nd Workshop on Semantics Analysis*, pages 313–321, 1992.

[71] Yan Mei Tang and Pierre Jouvelot. Effect systems with subtyping. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 45–53, 1995.

[72] R. E. Tarjan. Amortized computational complexity. *SIAM J. Algebraic Discrete Methods*, 6:306–318, 1985.

[73] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215 – 225, 1975.

[74] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115 – 122, 1987.

[75] Mitchell Wand and Patrick M. O'Keefe. On the complexity of type inference with coercion. In *Conf. on Functional Programming Languages and Computer Architecture*, pages 293–298, 1989.

[76] Jeffrey Westbrook and Robert E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *SIAM Journal on Computing*, 18(1):1–11, 1989.

Appendix A

Implementation of the Applications

A.1 The Lambda Calculus

A.1.1 The Simply Typed Lambda Calculus

lambda.def:

```
Labels

   int/0,
   arrow/2,
   intconst/0,
   var[x]/0,
   lambda[x]/1,
   app/2,
   env[e]/|e|.

Ordering

   int < arrow,
   intconst < var,
   intconst < lambda,
   intconst < app,
   int < intconst,
   int < env.

rec IsType(tau) <->
   (int);
   (arrow) if IsType(tau/1), IsType(tau/2).

IsEnvironment(gamma) <->
```

```
   (env[dom]) if @ x in dom: IsType(gamma/x).

rec IsExpression(e) <->
   (intconst);
   (var[x]);
   (lambda[x]) if IsExpression(e/1);
   (app) if IsExpression(e/1), IsExpression(e/2).
```

lambda.clp:

```
Type(gamma, intconst, int).
Type(gamma, var[..x..], t) :- env[..x:t'..] <= gamma, t = t'.
Type(gamma, lambda[..x..](e), t) :-
   gamma <= gamma', env[..x:t'..] <= gamma',
   Type(gamma', e, t''), t = arrow(t', t'').
Type(gamma, app(e, e'), t) :-
   Type(gamma, e, arrow(t',t)), Type(gamma, e', t').

environment(env[..x:t..]) :- IsType(t).

Typing(gamma, e, t) :-
   IsEnvironment(gamma), IsType(t), IsExpression(e),
   Type(gamma, e, t).
```

A.1.2 The Overloaded Lambda Calculus

overload.def:

```
Finite

Labels

   basetype/0,
   pos_int/0,
   int/0,
   pos_real/0,
   real/0,
   arrow/2,
   var[x]/0,
   lambda[x]/1,
   app/2,
   env[e]/|e|.

Ordering

   basetype < pos_int,
   pos_int < int,
   pos_int < pos_real,
   int < real,
```

```
   pos_real < real,
   real < arrow,
   var < lambda,
   var < app,
   int < var,
   int < env.

rec IsType(tau) <->
   (pos_int);
   (int);
   (pos_real);
   (real);
   (arrow) if IsType(tau/1), IsType(tau/2).

rec StructuralType(tau) <->
   (basetype);
   (arrow) if StructuralType(tau/1), StructuralType(tau/2).

IsEnvironment(gamma) <->
   (env[dom]) if @ x in dom: IsType(gamma/x).

StructuralEnvironment(gamma) <->
   (env[dom]) if @x in dom: StructuralType(gamma/x).

rec IsExpression(e) <->
   (var[x]);
   (lambda[x]) if IsExpression(e/1);
   (app) if IsExpression(e/1), IsExpression(e/2).

ineq Leq(tau, tau') <->
   (pos_int, pos_int);
   (pos_int, int);
   (pos_int, pos_real);
   (pos_int, real);
   (int, int);
   (int, real);
   (pos_real, pos_real);
   (pos_real, real);
   (real, real);
   (arrow, arrow) if Leq(tau'/1, tau/1), Leq(tau/2, tau'/2).

rec Instance(tau, tau') <->
   (basetype, pos_int);
   (basetype, int);
   (basetype, pos_real);
   (basetype, real);
   (arrow, arrow) if Instance(tau/1, tau'/1), Instance(tau/2, tau'/2).

Instance_gamma(gamma, gamma') <->
   (env[dom], env[dom'])
      if dom <= dom', dom' <= dom,
```

```
@x in dom & dom': Instance(gamma/x, gamma'/x).
```

overload.clp:

```
Type_struct(gamma, var[..x..], t) :- env[..x:t'..] <= gamma, t = t'.
Type_struct(gamma, lambda[..x..](e), t) :-
   gamma <= gamma', env[..x:t'..] <= gamma',
   Type_struct(gamma', e, t''), t = arrow(t', t'').
Type_struct(gamma, app(e, e'), t) :-
   Type_struct(gamma, e, arrow(t',t)), Type_struct(gamma, e', t').

Type(gamma, var[..x..], t) :- env[..x:t'..] <= gamma, leq(t', t).

Type(gamma, lambda[..x..](e), t) :-
        gamma <= gamma', env[..x:t'..] <= gamma',
        Type(gamma', e, t''), leq(arrow(t', t''), t).

Type(gamma, app(e, e'), t'') :-
        Type(gamma, e, arrow(t',t)), Type(gamma, e', t'), leq(t,t'').

Typing(gamma, e, t) :-
   IsEnvironment(gamma), IsType(t), IsExpression(e), Type(gamma, e, t),
   Instance(t_s, t), Instance_gamma(gamma_s, gamma),
   StructuralEnvironment(gamma_s), StructuralType(t_s),
   Type_struct(gamma_s, e, t_s).
```

A.2 Turbo Pascal

tp.def:

```
Labels

   boolean/0,
   integer/0,
   real/0,
   char/0,
   pointer/0,
   string/0,
   file/0,
   text/0,
   set/1,
   pointer_to/1,
   file_of/1,
   array/2,
   record[n]/|n|,

   semicolon/2,
   skip/0,
```

```
if_then_else/3,
while/2,
repeat_until/2,
for/4,
case[name]/|name| + 1,
variant/2,
assign/2,
proc_call[P]/1,

eq/2,
neq/2,
geq/2,
leq/2,
grt/2,
lst/2,
in_set/2,
empty_set/0,
set_of[name]/|name|,
uni_plus/1,
uni_minus/1,
plus/2,
minus/2,
times/2,
divide/2,
div/2,
mod/2,
shl/2,
shr/2,
and_/2,
or/2,
xor/2,
not/1,
address/1,
func_call[F]/1,

true/0,
false/0,
intconst/0,
realconst/0,
charconst/0,
stringconst/0,
nil/0,

ident[x]/0,
dot[a]/1,
index/2,
point/1,

defs/3,
var_defs[x]/0,
proc_defs[p]/|p|,
```

```
    func_defs[f]/|f|,
    proc_def/2,
    func_def/2,
    args[param]/|param|,
    var/0,
    val/0,
    var_t/1,
    val_t/1,

    program/2,

    env[dom]/|dom|,
    p_env[dom]/|dom|,
    f_env[dom]/|dom|,
    p_type[param]/|param|,
    f_type[param]/|param|+1.

Ordering

         boolean < program,

    boolean < real,
    real < integer,
    real < string,
    string < char,
    boolean < pointer,
    pointer < pointer_to,
    boolean < text,
    text < file_of,
    file_of < file,
    real < set,
    boolean < array,
    boolean < record,

    program < semicolon,
    semicolon < skip,
    semicolon < if_then_else,
    semicolon < while,
    semicolon < repeat_until,
    semicolon < for,
    semicolon < case,
    semicolon < variant,
    semicolon < assign,
    semicolon < proc_call,

    program < eq,
    eq < neq,
    eq < geq,
    eq < leq,
    eq < grt,
    eq < lst,
```

```
eq < in_set,
eq < empty_set,
eq < set_of,
eq < uni_plus,
eq < uni_minus,
eq < plus,
eq < minus,
eq < times,
eq < divide,
eq < div,
eq < mod,
eq < shl,
eq < shr,
eq < and_,
eq < or,
eq < xor,
eq < not,
eq < address,
eq < func_call,

nil < true,
nil < false,
nil < intconst,
nil < realconst,
nil < charconst,
nil < stringconst,
program < nil,

program < ident,
ident < dot,
ident < index,
ident < point,

program < defs,
program < var_defs,
var_defs < proc_defs,
proc_defs < func_defs,
program < proc_def,
proc_def < func_def,
program < args,
program < val,
val < var,
program < val_t,
val_t < var_t,

program < env,
program < p_env,
program < f_env,
program < p_type,
program < f_type.
```

```
Ordinal(t) <->
   (boolean);
   (integer);
   (char).

rec Notfile(t) <->
   (boolean);
   (integer);
   (real);
   (char);
   (pointer);
   (string);
   (set) if Notfile(t/1);
   (pointer_to) if Notfile(t/1);
   (array) if Notfile(t/1), Notfile(t/2);
   (record[a]) if @n in a: Notfile(t/n).

rec IsType(t) <->
   (boolean);
   (integer);
   (real);
   (char);
   (pointer);
   (string);
   (file);
   (text);
   (set) if Ordinal(t/1), IsType(t/1);
   (pointer_to) if IsType(t/1);
   (file_of) if Notfile(t/1), IsType(t/1);
   (array) if Ordinal(t/1), IsType(t/1), IsType(t/2);
   (record[a]) if @n in a: IsType(t/n).

IsEnvironment(Gamma) <->
   (env[dom]) if @x in dom: IsType(Gamma/x).

IsPar(p) <->
   (val_t) if IsType(p/1);
   (var_t) if IsType(p/1).

IsPtype(pt) <->
   (p_type[param]) if @par in param: IsPar(pt/par).

IsFtype(ft) <->
   (f_type[param]) if @par in param: IsPar(ft/par).

IsPenv(pe) <->
   (p_env[dom]) if @p in dom: IsPtype(pe/p).

IsFenv(fe) <->
   (f_env[dom]) if @f in dom: IsFtype(fe/f).
```

```
Eq_type(t) <->
   (boolean);
   (integer);
   (char);
   (real);
   (string);
   (pointer);
   (pointer_to);
   (set).

Geq_type(t) <->
   (boolean);
   (integer);
   (char);
   (real);
   (string);
   (set).

Grt_type(t) <->
   (boolean);
   (integer);
   (char);
   (real);
   (string).

Sign_type(t) <->
   (integer);
   (real).

Plus_type(t) <->
   (integer);
   (real);
   (set);
   (string);
   (char).

Minus_type(t) <->
   (integer);
   (real);
   (set).

And_type(t) <->
   (boolean);
   (integer).

Is_set(t) <->
   (set).

Is_file(t) <->
   (text);
   (file);
```

```
   (file_of).

Is_string(t) <->
   (char);
   (string).

Is_pointer(t) <->
   (pointer);
   (pointer_to).

Tc_ord(t, t') <->
   (boolean, boolean);
   (integer, integer);
   (char, char).

Tc(t, t') <->
   (boolean, boolean);
   (integer, integer);
   (integer, real);
   (real, integer);
   (real, real);
   (char, char);
   (char, string);
   (string, char);
   (string, string);
   (pointer, pointer);
   (pointer, pointer_to);
   (pointer_to, pointer);
   (pointer_to, pointer_to) if t = t';
   (set, set) if Tc_ord(t/1, t'/1);
   (file_of, file_of) if t = t';
   (text, text);
   (file, file);
   (array, array) if t = t';
   (record[a], record[b]) if t = t'.

Ac(t, t') <->
   (boolean, boolean);
   (integer, integer);
   (real, integer);
   (real, real);
   (char, char);
   (pointer, pointer);
   (pointer, pointer_to);
   (string, string);
   (string, char);
   (set, set) if Tc(t/1, t'/1);
   (pointer_to, pointer);
   (pointer_to, pointer_to) if t = t', Notfile(t/1);
   (array, array) if t = t', Notfile(t/2);
   (record[a], record[b]) if t = t', Notfile(t).
```

```
Op(t, t', t'') <->
   (boolean, boolean, boolean);
   (integer, integer, integer);
   (real, integer, real);
   (integer, real, real);
   (real, real, real);
   (char, char, string);
   (char, string, string);
   (string, char, string);
   (string, string, string);
   (set, set, set) if Tc(t/1, t'/1), Tc(t/1, t''/1).

Io(t, t') <->
   (text, boolean);
   (text, integer);
   (text, real);
   (text, char);
   (text, string);
   (file_of, boolean)   if t/1 = t';
   (file_of, integer)   if t/1 = t';
   (file_of, real)      if t/1 = t';
   (file_of, char)      if t/1 = t';
   (file_of, pointer)   if t/1 = t';
   (file_of, string) if t/1 = t';
   (file_of, set)    if t/1 = t';
   (file_of, pointer_to)   if t/1 = t', Notfile(t');
   (file_of, array)  if t/1 = t', Notfile(t');
   (file_of, record[a]) if t/1 = t', Notfile(t').
```

tp.clp:

```
IsProgram(program(definitions, body)) :-
   IsDefs(definitions), IsStatement(body).

IsDefs(defs(var_defs[..x..], proc_defs[..p:p_d..], func_defs[..f:f_d..])) :-
   IsPdef(p_d), IsFdef(f_d).

IsPdef(proc_def(a, body)) :- IsFormalParams(a), IsProgram(body).
IsFdef(func_def(a, body)) :- IsFormalParams(a), IsProgram(body).

IsFormalParams(args[..v:l..]) :- IsLabel(l).

IsLabel(var).
IsLabel(val).

IsStatement(assign(v, e)) :- IsVarExp(v), IsExp(e).
IsStatement(if_then_else(b, S1, S2)) :-
   IsExp(b), IsStatement(S1), IsStatement(S2).
IsStatement(while(b, S)) :- IsExp(b), IsStatement(S).
IsStatement(repeat_until(S, b)) :- IsStatement(S), IsExp(b).
```

```
IsStatement(for(v, e1, e2, S)) :-
   IsVarExp(v), IsExp(e1), IsExp(e2), IsStatement(S).
IsStatement(case[..name: c..](e)) :- IsVariant(c), IsExp(e).
IsStatement(proc_call[..P..](a)) :- IsActualParams(a).
IsStatement(semicolon(S1, S2)) :- IsStatement(S1), IsStatement(S2).

IsVariant(variant(c, S)) :- IsConstant(c), IsStatement(S).

IsConstant(true).
IsConstant(false).
IsConstant(intconst).
IsConstant(nil).
IsConstant(realconst).
IsConstant(charconst).
IsConstant(stringconst).

IsVarExp(ident[..x..]).
IsVarExp(dot[..a..](v)) :- IsVarExp(v).
IsVarExp(index(v, e)) :- IsVarExp(v), IsExp(e).
IsVarExp(point(func_call[..F..](a))) :- IsActualParams(a).
IsVarExp(point(v)) :- IsVarExp(v).

IsExp(eq(e,e')) :- IsExp(e), IsExp(e').
IsExp(neq(e,e')) :- IsExp(e), IsExp(e').
IsExp(geq(e,e')) :- IsExp(e), IsExp(e').
IsExp(leq(e,e')) :- IsExp(e), IsExp(e').
IsExp(grt(e,e')) :- IsExp(e), IsExp(e').
IsExp(lst(e,e')) :- IsExp(e), IsExp(e').
IsExp(in_set(e,e')) :- IsExp(e), IsExp(e').
IsExp(empty_set).
IsExp(set_of[..label:e..]) :- IsExp(e).
IsExp(uni_plus(e)) :- IsExp(e).
IsExp(uni_minus(e)) :- IsExp(e).
IsExp(plus(e,e')) :- IsExp(e), IsExp(e').
IsExp(minus(e,e')) :- IsExp(e), IsExp(e').
IsExp(times(e,e')) :- IsExp(e), IsExp(e').
IsExp(divide(e,e')) :- IsExp(e), IsExp(e').
IsExp(div(e,e')) :- IsExp(e), IsExp(e').
IsExp(mod(e,e')) :- IsExp(e), IsExp(e').
IsExp(shl(e,e')) :- IsExp(e), IsExp(e').
IsExp(shr(e,e')) :- IsExp(e), IsExp(e').
IsExp(and_(e,e')) :- IsExp(e), IsExp(e').
IsExp(or(e,e')) :- IsExp(e), IsExp(e').
IsExp(xor(e,e')) :- IsExp(e), IsExp(e').
IsExp(not(e)) :- IsExp(e).
IsExp(address(e)) :- IsExp(e).
IsExp(func_call[..F..](a)) :- IsActualParams(a).
IsExp(e) :- IsVarExp(e).
IsExp(e) :- IsConstant(e).

IsActualParams(args[..v:e..]) :- IsParam(e).
```

```
IsParam(var_t(e)) :- IsExp(e).
IsParam(val_t(e)) :- IsExp(e).


Typing(Gamma, Gamma_p, Gamma_f, p) :-
   IsEnvironment(Gamma), IsPenv(Gamma_p), IsFenv(Gamma_f), IsProgram(p),
   Type_program(Gamma, Gamma_p, Gamma_f, p).

Type_program(Gamma, Gamma_p, Gamma_f, program(definitions, body)) :-
   Type_defs(Gamma, Gamma_p, Gamma_f, definitions),
   Type_Statement(Gamma, Gamma_p, Gamma_f, body).

Type_defs(Gamma, Gamma_p, Gamma_f, defs(vd, pd, fd)) :-
   Type_var_defs(Gamma, vd), Type_proc_defs(Gamma, Gamma_p, Gamma_f, pd),
   Type_func_defs(Gamma, Gamma_p, Gamma_f, fd).

Type_var_defs(Gamma, var_defs[..x..]) :- env[..x:t..] <= Gamma.

Type_proc_defs(Gamma, Gamma_p, Gamma_f, proc_defs[..p: p_d..]) :-
   p_env[..p:p_t..] <= Gamma_p,
   Type_proc(Gamma, Gamma_p, Gamma_f, p_d, p_t).

Type_func_defs(Gamma, Gamma_p, Gamma_f, func_defs[..f: f_d..]) :-
   f_env[..f:f_t..] <= Gamma_f, env[..f: r_t..] <= Gamma,
   Type_func(Gamma, Gamma_p, Gamma_f, f_d, f_t, r_t).

Type_proc(Gamma, Gamma_p, Gamma_f, proc_def(a, body), p_t) :-
   Type_formal_params_p(Gamma, Gamma_p, Gamma_f, a, p_t),
   Type_program(Gamma, Gamma_p, Gamma_f, body).

Type_func(Gamma, Gamma_p, Gamma_f, func_def(a, body), f_t, r_t) :-
   Type_formal_params_f(Gamma, Gamma_p, Gamma_f, a, f_t, r_t),
   Type_program(Gamma, Gamma_p, Gamma_f, body).

Type_formal_params_p(Gamma, Gamma_p, Gamma_f, args[..v:l..], p_type[..v:e..]) :-
   env[..v:t..] <= Gamma, Label_match(l, e, t).

Type_formal_params_f
   (Gamma, Gamma_p, Gamma_f, args[..v:l..], f_type[..v:e..](r_t), r_t) :-
      env[..v:t..] <= Gamma, Label_match(l, e, t).

Label_match(var, var_t(t), t).
Label_match(val, val_t(t), t).

Type_Statement(Gamma, Gamma_p, Gamma_f, skip).

Type_Statement(Gamma, Gamma_p, Gamma_f,   semicolon(S, S')) :-
   Type_Statement(Gamma, Gamma_p, Gamma_f, S),
   Type_Statement(Gamma, Gamma_p, Gamma_f, S').
```

```
Type_Statement(Gamma, Gamma_p, Gamma_f, assign(v, e)) :-
   Type_Var_Exp(Gamma, Gamma_f, v, t_v),
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Ac(t_v, t_e).

Type_Statement(Gamma, Gamma_p, Gamma_f, if_then_else(b, S, S')) :-
   Type_Exp(Gamma, Gamma_f, b, boolean),
   Type_Statement(Gamma, Gamma_p, Gamma_f, S),
   Type_Statement(Gamma, Gamma_p, Gamma_f, S').

Type_Statement(Gamma, Gamma_p, Gamma_f, while(b, S)) :-
   Type_Exp(Gamma, Gamma_f, b, boolean),
   Type_Statement(Gamma, Gamma_p, Gamma_f, S).

Type_Statement(Gamma, Gamma_p, Gamma_f, repeat_until(b, S)) :-
   Type_Exp(Gamma, Gamma_f, b, boolean),
   Type_Statement(Gamma, Gamma_p, Gamma_f, S).

Type_Statement(Gamma, Gamma_p, Gamma_f, for(v, e, e', S)) :-
   Type_Var_Exp(Gamma, Gamma_f, v, t_v),
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Type_Exp(Gamma, Gamma_f, e', t_e'),
   Type_Statement(Gamma, Gamma_p, Gamma_f, S),
   Ac(t_v, t_e),
   Ac(t_v, t_e'),
   Ordinal(t_v).

Type_Statement(Gamma, Gamma_p, Gamma_f, case[..name:c..](e)) :-
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Type_Variant(Gamma, Gamma_p, Gamma_f, c, t_e),
   Ordinal(t_e).

Type_Variant(Gamma, Gamma_p, Gamma_f, variant(c, S), t) :-
   Type_Statement(Gamma, Gamma_p, Gamma_f, S),
   Type_Exp(Gamma, Gamma_f, c, t_c),
   Tc(t_c, t).

Type_Statement
   (Gamma, Gamma_p, Gamma_f,
   proc_call[read](args[f: val_t(e), v: var_t(e')])) :-
   Type_Exp(Gamma, Gamma_f, e, t), Type_exp(Gamma, Gamma_f, e', t'),
   Io(t, t').

Type_Statement
   (Gamma, Gamma_p, Gamma_f,
    proc_call[write](args[f: val_t(e), v: var_t(e')])) :-
      Type_Exp(Gamma, Gamma_f, e, t),
      Type_exp(Gamma, Gamma_f, e', t'),
      Io(t, t').

Type_Statement(Gamma, Gamma_p, Gamma_f, proc_call[new](args[v:var_t(e)])) :-
```

```
   Type_Var_exp(Gamma, Gamma_f, e, t),
   Is_Pointer(t).

Type_Statement(Gamma, Gamma_p, Gamma_f, proc_call[dispose](args[v:var_t(e)])) :-
   Type_Var_exp(Gamma, Gamma_f, e, t),
   Is_Pointer(t).

Type_Statement(Gamma, Gamma_p, Gamma_f, proc_call[..P..](a)) :-
   p_env[..P:pt..] <= Gamma_p,
   Type_Args_p(Gamma, Gamma_f, a, pt).

Type_Args_p(Gamma, Gamma_f, args[..v:e..], p_type[..v:t_v..]) :-
   Type_Arg(Gamma, Gamma_f, e, t_v).

Type_Constant(nil, Pointer).
Type_Constant(true, boolean).
Type_Constant(false, boolean).
Type_Constant(intconst, integer).
Type_Constant(realconst, real).
Type_Constant(charconst, char).
Type_Constant(stringconst, string).

Type_Var_Exp(Gamma, Gamma_f, ident[..x..], t) :-
   env[..x:t_x..] <= Gamma, t = t_x.

Type_Var_Exp(Gamma, Gamma_f, dot[..a..](v), t) :-
   Type_Var_Exp(Gamma, Gamma_f, v, t_v),
   record[..a:t_a..] <= t_v, t = t_a.

Type_Var_Exp(Gamma, Gamma_f, index(v, e), t) :-
   Type_Var_Exp(Gamma, Gamma_f, v, array(t', t)),
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Ac(t', t_e).

Type_Var_Exp(Gamma, Gamma_f, index(v, e), char) :-
   Type_Var_Exp(Gamma, Gamma_f, v, string),
   Type_Exp(Gamma, Gamma_f, e, integer).

Type_Var_Exp(Gamma, Gamma_f, point(func_call[..F..](a)), t) :-
   Type_Exp(Gamma, Gamma_f, func_call[..F..](a), pointer_to(t)).

Type_Var_Exp(Gamma, Gamma_f, point(v), t) :-
   Type_Var_Exp(Gamma, Gamma_f, v, pointer_to(t)).

Type_Exp(Gamma, Gamma_f, eq(e, e'), boolean) :-
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Type_Exp(Gamma, Gamma_f, e', t_e'),
   Tc(t_e, t_e'),
   Eq_Type(t_e), Eq_Type(t_e').

Type_Exp(Gamma, Gamma_f, neq(e, e'), boolean) :-
```

```
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Type_Exp(Gamma, Gamma_f, e', t_e'),
   Tc(t_e, t_e'),
   Eq_Type(t_e), Eq_Type(t_e').

Type_Exp(Gamma, Gamma_f, geq(e, e'), boolean) :-
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Type_Exp(Gamma, Gamma_f, e', t_e'),
   Tc(t_e, t_e'),
   Geq_Type(t_e), Geq_Type(t_e').

Type_Exp(Gamma, Gamma_f, leq(e, e'), boolean) :-
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Type_Exp(Gamma, Gamma_f, e', t_e'),
   Tc(t_e, t_e'),
   Geq_Type(t_e), Geq_Type(t_e').

Type_Exp(Gamma, Gamma_f, grt(e, e'), boolean) :-
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Type_Exp(Gamma, Gamma_f, e', t_e'),
   Tc(t_e, t_e'),
   Grt_Type(t_e), Grt_Type(t_e').

Type_Exp(Gamma, Gamma_f, lst(e, e'), boolean) :-
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Type_Exp(Gamma, Gamma_f, e', t_e'),
   Tc(t_e, t_e'),
   Grt_Type(t_e), Grt_Type(t_e').

Type_Exp(Gamma, Gamma_f, in_set(e, e'), boolean) :-
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Type_Exp(Gamma, Gamma_f, e', set(t')),
   Ordinal(t'),
   Tc(t', t_e).

Type_Exp(Gamma, Gamma_f, empty_set, set(t)).

Type_Exp(Gamma, Gamma_f, set_of[..label:e..], t) :-
   IsEnvironment(env[..label:t_e..]),
   Type_Exp(Gamma, Gamma_f, e, t_e),
   Ordinal(t_e), Tc(t, set(t_e)).

Type_Exp(Gamma, Gamma_f, uni_plus(e), t) :-
   Type_Exp(Gamma, Gamma_f, e, t), Sign_type(t).

Type_Exp(Gamma, Gamma_f, uni_minus(e), t) :-
   Type_Exp(Gamma, Gamma_f, e, t), Sign_type(t).

Type_Exp(Gamma, Gamma_f, plus(e, e'), t) :-
   Type_Exp(Gamma, Gamma_f, e, t_e), Type_Exp(Gamma, Gamma_f, e', t_e'),
   Op(t_e, t_e', t),
```

```
   Plus_type(t_e), Plus_type(t_e').

Type_Exp(Gamma, Gamma_f, minus(e, e'), t) :-
   Type_Exp(Gamma, Gamma_f, e, t_e), Type_Exp(Gamma, Gamma_f, e', t_e'),
   Op(t_e, t_e', t),
   Minus_type(t_e), Minus_type(t_e').

Type_Exp(Gamma, Gamma_f, times(e, e'), t) :-
   Type_Exp(Gamma, Gamma_f, e, t_e), Type_Exp(Gamma, Gamma_f, e', t_e'),
   Op(t_e, t_e', t),
   Minus_type(t_e), Minus_type(t_e').

Type_Exp(Gamma, Gamma_f, divide(e, e'), real) :-
   Type_Exp(Gamma, Gamma_f, e, t_e), Type_Exp(Gamma, Gamma_f, e', t_e'),
   Sign_type(t_e), Sign_type(t_e').

Type_Exp(Gamma, Gamma_f, div(e, e'), integer) :-
   Type_Exp(Gamma, Gamma_f, e, integer),
   Type_Exp(Gamma, Gamma_f, e', integer).

Type_Exp(Gamma, Gamma_f, mod(e, e'), integer) :-
   Type_Exp(Gamma, Gamma_f, e, integer),
   Type_Exp(Gamma, Gamma_f, e', integer).

Type_Exp(Gamma, Gamma_f, shl(e, e'), integer) :-
   Type_Exp(Gamma, Gamma_f, e, integer),
   Type_Exp(Gamma, Gamma_f, e', integer).

Type_Exp(Gamma, Gamma_f, shr(e, e'), integer) :-
   Type_Exp(Gamma, Gamma_f, e, integer),
   Type_Exp(Gamma, Gamma_f, e', integer).

Type_Exp(Gamma, Gamma_f, and_(e, e'), t) :-
   Type_Exp(Gamma, Gamma_f, e, t_e), Type_Exp(Gamma, Gamma_f, e', t_e'),
   Op(t_e, t_e', t),
   And_type(t_e), And_type(t_e').

Type_Exp(Gamma, Gamma_f, or(e, e'), t) :-
   Type_Exp(Gamma, Gamma_f, e, t_e), Type_Exp(Gamma, Gamma_f, e', t_e'),
   Op(t_e, t_e', t),
   And_type(t_e), And_type(t_e').

Type_Exp(Gamma, Gamma_f, xor(e, e'), t) :-
   Type_Exp(Gamma, Gamma_f, e, t_e), Type_Exp(Gamma, Gamma_f, e', t_e'),
   Op(t_e, t_e', t),
   And_type(t_e), And_type(t_e').

Type_Exp(Gamma, Gamma_f, not(e), t) :-
   Type_Exp(Gamma, Gamma_f, e, t), And_type(t).

Type_Exp(Gamma, Gamma_f, address(v), Pointer) :-
```

```
   Type_Var_Exp(Gamma, Gamma_f, v, t').

Type_Exp(Gamma, Gamma_f, func_call[..F..](a), t) :-
   f_env[..F:ft..] <= Gamma_f,
   Type_Args_f(Gamma, Gamma_f, a, ft).

Type_Exp(Gamma, Gamma_f, v, t) :-
   Type_Var_Exp(Gamma, Gamma_f, v, t), IsVarExp(v).

Type_Exp(Gamma, Gamma_f, c, t) :- Type_Constant(c, t), IsConstant(c).

Type_Args_f(Gamma, Gamma_f, args[..v:e..], f_type[..v:t_v..](t)) :-
   Type_Arg(Gamma, Gamma_f, e, t_v).

Type_Arg(Gamma, Gamma_f, var_t(e), var_t(t)) :-
   Type_Exp(Gamma, Gamma_f, e, t).

Type_Arg(Gamma, Gamma_f, val_t(e), val_t(t)) :-
   Type_Exp(Gamma, Gamma_f, e, t'),
   Ac(t, t').
```

A.3 Reynolds Style Subtyping

reynolds.def:

```
Finite

Labels

   univ/0,
   int/0,
   bool/0,
   ns/0,
   prod[a]/|a|,
   sum[a]/|a|,
   list/1,
   arrow/2,
   ident[x]/0,
   intconst/0,
   true/0,
   false/0,
   add/0,
   equals/0,
   lambda[x]/1,
   app/2,
   bracket[a]/|a|,
   select[a]/1,
   inject[a]/1,
   choose[a]/|a|,
```

```
   nil/0,
   cons/2,
   lchoose/2,
   cond/3,
   fix/1,
   env[dom]/|dom|.

Ordering

   univ < int,
   univ <bool,
   univ < prod,
   univ < sum,
   int < ns,
   bool < ns,
   prod < ns,
   sum < ns,
   ns < list,
   ns < arrow,
   univ < ident,
   univ < intconst,
   univ < true,
   univ < false,
   univ < add,
   univ < equals,
   univ < lambda,
   univ < app,
   univ < bracket,
   univ < select,
   univ < inject,
   univ < choose,
   univ < nil,
   univ < cons,
   univ < lchoose,
   univ < cond,
   univ < fix,
   univ < env.

rec IsType(t) <->
   (univ);
   (int);
   (bool);
   (ns);
   (prod[a]) if @i in a: IsType(t/i);
   (sum[a]) if @i in a: IsType(t/i);
   (list) if IsType(t/1);
   (arrow) if IsType(t/1), IsType(t/2).

ineq leq(t, t') <->
   (univ, univ);
   (univ, int);
```

```
      (univ, bool);
      (univ, ns);
      (univ, prod);
      (univ, sum);
      (univ, list);
      (univ, arrow);
      (int, int);
      (int, ns);
      (bool, bool);
      (bool, ns);
      (prod[a], prod[b]) if b <= a, @i in a&b: leq(t/i, t'/i);
      (prod, ns);
      (sum[a], sum[b]) if a <= b, @i in a&b: leq(t/i, t'/i);
      (sum, ns);
      (list, list) if leq(t/1, t'/1);
      (list, ns);
      (arrow, arrow) if leq(t'/1, t/1), leq(t/2, t'/2);
      (arrow, ns);
      (ns, ns).

   rec IsExpr(e) <->
      (ident);
      (intconst);
      (true);
      (false);
      (add);
      (equals);
      (lambda[x]) if IsExpr(e/1);
      (app) if IsExpr(e/1), IsExpr(e/2);
      (bracket[a]) if @i in a: IsExpr(e/i);
      (select[a]) if IsExpr(e/1);
      (inject[a]) if IsExpr(e/1);
      (choose[a]) if @i in a: IsExpr(e/i);
      (nil);
      (cons) if IsExpr(e/1), IsExpr(e/2);
      (lchoose) if IsExpr(e/1), IsExpr(e/2);
      (cond) if IsExpr(e/1), IsExpr(e/2), IsExpr(e/3);
      (fix) if IsExpr(e/1).

   IsEnvironment(E) <->
      (env[dom]) if @x in dom: IsType(E/x).
```

reynolds.clp:

```
   Type(Gamma, ident[..x..], t) :- env[..x:t_x..] <= Gamma, leq(t_x, t).

   Type(Gamma, intconst, t) :- leq(int,t).

   Type(Gamma, true, t) :- leq(bool,t).
```

```
Type(Gamma, false, t) :- leq(bool,t).

Type(Gamma, add, t) :- leq(arrow(int, arrow(int,int)), t).

Type(Gamma, equals, t) :- leq(arrow(int, arrow(int,bool)), t).

Type(Gamma, lambda[..x..](e), t) :-
   Gamma = Gamma', env[..x:t_x..] <= Gamma',
   Type(Gamma', e, t_e), leq(arrow(t_x,t_e), t).

Type(Gamma, app(f, e), t) :-
   Type(Gamma, f, arrow(t_e, t')), Type(Gamma, e, t_e), leq(t', t).

Type(Gamma, bracket[..a:e..], t) :-
   Type(Gamma, e, t_e), leq(prod[..a:t_e..],t).

Type(Gamma, select[..l..](e), t) :-
   Type(Gamma, e, prod[..l:t_l..]), leq(t_a,t).

Type(Gamma, inject[..a..](e), t) :-
   Type(Gamma, e, t_e), sum[..a:t_e..] <= t', leq(t',t).

Type(Gamma, choose[..a:e..], t) :-
   Type(Gamma, e, arrow(t_a, t')), leq(arrow(sum[..a:t_a..], t'), t).

Type(Gamma, nil, t) :- leq(list(t'),t).

Type(Gamma, cons(e, e'), t) :-
   Type(Gamma, e, t'), Type(Gamma, e', list(t')), leq(list(t'), t).

Type(Gamma, lchoose(e, e'), t) :-
   Type(Gamma, e, t''), Type(Gamma, e', arrow(t', arrow(list(t'), t''))),
   leq(arrow(list(t'), t''), t).

Type(Gamma, cond(b, e, e'), t) :-
   Type(Gamma, b, bool), Type(Gamma, e, t'), Type(Gamma, e', t'),
   leq(t', t).

Type(Gamma, fix(f), t) :-
   Type(Gamma, f, arrow(t',t')), leq(t', t).

Typing(Gamma, e, t) :-
   IsEnvironment(Gamma), IsExpr(e), IsType(t), Type(Gamma, e, t).
```

A.4 The Object Calculus

A.4.1 The Core Object Calculus

oc.def:

```
Labels

   object[a]/|a|,
   var[x]/0,
   object_def[a]/|a|,
   message[a]/1,
   override[a]/|a|+1,
   sigma[x]/1,
   env[x]/|x|.

Ordering

   object < var,
   object < object_def,
   object < message,
   object < override,
   object < sigma,
   object < env.

rec IsType(t) <->
   (object[a]) if @i in a: IsType(t/i).

Environment(e) <->
   (env[d]) if @x in d: IsType(e/x).
```

oc.clp:

```
Program(var[..x..]).
Program(object_def[..l:m..]) :- Method(m).
Program(message[..l..](a)) :- Program(a).
Program(override[..l:m..](a)) :- Program(a), Method(m).

Method(sigma[..x..](a)) :- Program(a).

Type(Gamma, var[..x..], t) :- env[..x:t'..] <= Gamma, t <= t'.
Type(Gamma, object_def[..l:m..],t) :-
   t <= t', t' = object[..l:t_b..], Type(Gamma, m, t', t_b).
Type(Gamma, message[..l..](a), t) :-
   Type(Gamma, a, t_a), object[..l:t'..] <= t_a, t <= t'.
Type(Gamma, override[..l:m..](a), t) :-
   Type(Gamma, a, t_a), t <= t_a, Type(Gamma, m, t_a, t_b),
   object[..l:t_b..] <= t_a.
```

```
Type(Gamma, sigma[..x..](b), t_x, t) :-
   Type(Gamma', b, t), Gamma <= Gamma', env[..x:t'..] <= Gamma',
   t' = t_x.

Typing(Gamma, e, t) :-
   Environment(Gamma), Program(e), IsType(t), Type(Gamma, e, t).
```

A.4.2 The Object Calculus with Bool

ocplus.def:

```
Labels

   bool/0,
   object[a]/|a|,
   false/0,
   true/0,
   var[x]/0,
   object_def[a]/|a|,
   message[a]/1,
   override[a]/|a|+1,
   sigma[x]/1,
   env[x]/|x|.

Ordering

   object < bool,
   object < var,
   object < false,
   object < true,
   object < object_def,
   object < message,
   object < override,
   object < sigma,
   object < env.

rec IsType(t) <->
   (bool);
   (object[a]) if @i in a: IsType(t/i).

Environment(e) <->
   (env[d]) if @x in d: IsType(e/x).
```

ocplus.clp:

```
Program(false).
Program(true).
Program(var[..x..]).
```

```
Program(object_def[..l:m..]) :- Method(m).
Program(message[..l..](a)) :- Program(a).
Program(override[..l:m..](a)) :- Program(a), Method(m).

Method(sigma[..x..](a)) :- Program(a).

Type(Gamma, false, bool).
Type(Gamma, true, bool).
Type(Gamma, var[..x..], t) :- env[..x:t'..] <= Gamma, t <= t'.
Type(Gamma, object_def[..l:m..],t) :-
   t <= t', t' = object[..l:t_b..], Type(Gamma, m, t', t_b).
Type(Gamma, message[..l..](a), t) :-
   Type(Gamma, a, t_a), object[..l:t'..] <= t_a, t <= t'.
Type(Gamma, override[..l:m..](a), t) :-
   Type(Gamma, a, t_a), t <= t_a, Type(Gamma, m, t_a, t_b),
   object[..l:t_b..] <= t_a.


Type(Gamma, sigma[..x..](b), t_x, t) :-
   Type(Gamma', b, t), Gamma <= Gamma', env[..x:t'..] <= Gamma',
   t' = t_x.

Typing(Gamma, e, t) :-
   Environment(Gamma), Program(e), IsType(t), Type(Gamma, e, t).
```

A.4.3 The Object Calculus with Selftype

selftype.def:

```
Labels

   selftype/0,
   object[a]/|a|,
   var[x]/0,
   object_def[a]/|a|,
   message[a]/1,
   override[a]/|a|+1,
   sigma[x]/1,
   env[x]/|x|.

Ordering

   object < selftype,
   object < var,
   object < object_def,
   object < message,
   object < override,
   object < sigma,
   object < env.
```

```
rec IsType(t) <->
   (selftype);
   (object[l]) if @i in l: IsType(t/i).

Environment(e) <->
   (env[d]) if @x in d: IsType(e/x).

NotSelftype(t) <->
   (object[l]).
```

selftype.clp:

```
Program(var[..x..]).
Program(object_def[..l:m..]) :- Method(m).
Program(message[..l..](a)) :- Program(a).
Program(override[..l:m..](a)) :- Program(a), Method(m).


Method(sigma[..x..](a)) :- Program(a).


Alt(selftype, A, A).
Alt(B, A, B) :- NotSelftype(B).


Type(Gamma, var[..x..], t) :- env[..x:t'..] <= Gamma, t <= t'.

Type(Gamma, object_def[..l:m..],t) :-
   t <= t', t' = object[..l:t_b'..], Type(Gamma, m, t', t_b),
   Alt(t_b', t', t_b), Dummy(object[..l:t_b..]).

Type(Gamma, message[..l..](a), t) :-
   Type(Gamma, a, t_a), object[..l:t'..] <= t_a, t <= t'',
        Alt(t', t_a, t'').

Type(Gamma, override[..l:m..](a), t) :-
   Type(Gamma, a, t_a), t <= t_a, Type(Gamma, m, t_a, t_b),
   object[..l:t_b..] <= t_a.

Type(Gamma, sigma[..x..](b), t_x, t) :-
   Type(Gamma', b, t), Gamma <= Gamma', env[..x:t'..] <= Gamma',
   t' = t_x.


Typing(Gamma, e, t) :-
   Environment(Gamma), Program(e), IsType(t), Type(Gamma, e, t).

Dummy(x).
```

A.5 Control Flow Analysis

A.5.1 Effect Systems using Subeffecting

subeffect.def:

```
Labels

   int/0,
   arrow[c]/2,
   env[dom]/|dom|,
   intconst/0,
   var[x]/0,
   lambda[name]/2,
   letrec[name]/3,
   app[site]/2,
   effect[c]/0,
   flow[site]/|site|.

Ordering

   int < arrow,
   int < env,
   int < intconst,
   intconst < var,
   intconst < lambda,
   intconst < letrec,
   intconst < app,
   int < effect,
   int < flow.

rec IsType(t) <->
   (int);
   (arrow[c]) if IsType(t/1), IsType(t/2).

Environment(e) <->
   (env[dom]) if @x in dom: IsType(e/x).

Variable(v) <->
   (var[x]).

rec Program(p) <->
   (intconst);
   (var[x]);
   (lambda[name]) if Variable(p/1), Program(p/2);
   (letrec[name]) if Variable(p/1), Variable(p/2), Program(p/3);
   (app[site]) if Program(p/1), Program(p/2).

Subeffect(eff, eff') <->
   (effect[c], effect[c']) if c <= c'.
```

```
IsEffect(eff) <->
   (effect[c]).

IsFlow(fl) <->
   (flow[f]).
```

subeffect.clp:

```
Type(Gamma, intconst, int, eff, f) :- effect[] <= eff.

Type(Gamma, var[..x..], t, eff, f) :-
   env[..x:t'..] <= Gamma, t = t', effect[] <= eff.

Type(Gamma, lambda[..name..](v, e), arrow[..c..](t, t'), eff, f) :-
   v = var[..x..],
   Gamma <= Gamma', env[..x:t_x..] <= Gamma', t = t_x,
   Type(Gamma', e, t', eff, f),
   eff <= effect[..c..],
   effect[..name..] <= effect[..c..],
   effect[] <= eff.

Type(Gamma, letrec[..name..](g, v, e), t, eff, f) :-
   g = var[..f..],
   Type(Gamma', lambda[..name..](v, e), t, effect[], f),
   Gamma <= Gamma', env[..f: t_f..] <= Gamma', t = t_f,
   effect[] <= eff.

Type(Gamma, app[..site..](e, e'), t, eff, f) :-
   Type(Gamma, e, arrow[..c..](t',t), eff', f),
   Type(Gamma, e', t', eff'', f),
   eff' <= eff, eff'' <= eff, effect[..c..] <= eff,
   flow[..site: eff'''..] <= f, eff''' = eff.

Analyse(Gamma, e, t, eff, f) :-
   Type(Gamma, e, t, eff, f),
   Environment(Gamma), IsType(t), IsEffect(eff), IsFlow(f).
```

A.5.2 Effect Systems using Subtyping

subtype.def:

```
Finite
Labels

   int/0,
   arrow[a]/2,
   env[dom]/|dom|,
   intconst/0,
```

```
   var[x]/0,
   lambda[name]/2,
        letrec[name]/3,
   app[site]/2,
   effect[a]/0,
   flow[site]/|site|.

Ordering

   int < arrow,
   int < env,
   int < intconst,
   intconst < var,
   intconst < lambda,
        intconst < letrec,
   intconst < app,
   int < effect,
   int < flow.

rec IsType(t) <->
   (int);
   (arrow[a]) if IsType(t/1), IsType(t/2).

Environment(e) <->
   (env[dom]) if @x in dom: IsType(e/x).

Variable(v) <->
   (var[x]).

rec Program(p) <->
   (intconst);
   (var[x]);
   (lambda[name]) if Variable(p/1), Program(p/2);
        (letrec[name]) if Variable(p/1), Variable(p/2), Program(p/3);
   (app[site]) if Program(p/1), Program(p/2).

Subeffect(eff, eff') <->
   (effect[e], effect[e']) if e <= e'.

ineq Subtype(t, t') <->
   (int, int);
   (arrow[c0], arrow[c1])
      if c0 <= c1, Subtype(t'/1, t/1),  Subtype(t/2, t'/2).

IsEffect(e) <->
   (effect[eff]).

IsFlow(f) <->
   (flow[sites]) if @l in sites: IsEffect(f/l).
```

subtype.clp:

```
Type(Gamma, intconst, t, effect[], f) :- Subtype(int, t).

Type(Gamma, var[..x..], t, effect[], f) :-
   env[..x:t'..] <= Gamma, Subtype(t',t).

Type(Gamma, lambda[..name..](v, e), t'', effect[], f) :-
   Subtype(arrow[..c..](t, t'), t''),
   v = var[..x..],
   Gamma <= Gamma', env[..x:t_x..] <= Gamma', t = t_x,
   Type(Gamma', e, t', eff, f),
   eff <= effect[..c..],
   effect[..name..] <= effect[..c..].

Type(Gamma, letrec[..name..](g, v, e), t, effect[], f) :-
   Subtype(t', t),
        g = var[..f..],
        Type(Gamma', lambda[..name..](v, e), t', effect[], f),
        Gamma <= Gamma', env[..f: t_f..] <= Gamma', t' = t_f.

Type(Gamma, app[..site..](e, e'), t'', eff, f) :-
   Subtype(t, t''),
   Type(Gamma, e, arrow[..c..](t',t), eff', f),
   Type(Gamma, e', t', eff'', f),
   eff' <= eff, eff'' <= eff, effect[..c..] <= eff,
   flow[..site: eff'''..] <= f, eff''' = eff.

Classical_type(Gamma, intconst, int).
Classical_type(Gamma, var[..x..], t) :- env[..x:t'..] <= Gamma, t = t'.
Classical_type(Gamma, lambda[..name..](v, e), arrow[..name'..](t,t')) :-
   v = var[..x..],
   Gamma = Gamma', env[..x:t_x..] <= Gamma', t = t_x,
   Classical_Type(Gamma', e, t').
Classical_type(Gamma, app[..site..](e, e'), t) :-
   Classical_type(Gamma, e, arrow[..name..](t',t)),
   Classical_type(Gamma, e', t').

Analyse(Gamma, e, t, eff, f) :-
   Type(Gamma, e, t, eff, f), Classical_type(Gamma, e, t),
   Environment(Gamma), IsType(t), IsEffect(eff), IsFlow(f).
```

Recent BRICS Dissertation Series Publications

DS-98-1 **Ole I. Hougaard.** ***The CLP(OIH) Language.*** **February 1998. PhD thesis. xii+187 pp.**

DS-97-3 **Thore Husfeldt.** ***Dynamic Computation.*** **December 1997. PhD thesis. 90 pp.**

DS-97-2 **Peter Ørbæk.** ***Trust and Dependence Analysis.*** **July 1997. PhD thesis. x+175 pp.**

DS-97-1 **Gerth Stølting Brodal.** ***Worst Case Efficient Data Structures.*** **January 1997. PhD thesis. x+121 pp.**

DS-96-4 **Torben Braüner.** ***An Axiomatic Approach to Adequacy.*** **November 1996. Ph.D. thesis. 168 pp.**

DS-96-3 **Lars Arge.** ***Efficient External-Memory Data Structures and Applications.*** **August 1996. Ph.D. thesis. xii+169 pp.**

DS-96-2 **Allan Cheng.** ***Reasoning About Concurrent Computational Systems.*** **August 1996. Ph.D. thesis. xiv+229 pp.**

DS-96-1 **Urban Engberg.** ***Reasoning in the Temporal Logic of Actions — The design and implementation of an interactive computer system.*** **August 1996. Ph.D. thesis. xvi+222 pp.**