Ninth International Conference on Principles and Practice of Constraint Programming – CP'03

WORKSHOP PROCEEDINGS

MultiCPL'03: Second International Workshop on Multiparadigm Constraint Programming Languages

and

RCoRP'03: Fifth International Workshop on Rule-Based Constraint Reasoning and Programming

Editors:

Michael Hanus
Petra Hofstedt
Armin Wolf

Slim Abdennadher
Thom Frühwirth
Arnaud Lallouet

September 29, 2003
Kinsale, County Cork, Ireland

Contents

Preface

The Workshops on Multiparadigm Constraint Programming Languages (MultiCPL'03) and on Rule-Based Constraint Reasoning and Programming (RCoRP'03) are held at the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03) on September 29, 2003 in Kinsale, County Cork, Ireland.

The aim of these workshops is to bring together people interested in multiparadigm constraint programming, language design and implementation and in using rule-based formalisms in constraint reasoning and programming to communicate and discuss recent developments, work in progress, and new research directions.

We would like to thank all the people who took part in the workshop as well as those who contributed to its organization. In particular we thank the additional referees Carsten Gips, Armin Kühnemann, Martin Leucker, André Metzner, and Thomas Nitsche as well as Stephan Frank for his extensive LaTeX support.

August 2003 The Workshop Organizers

MultiCPL'03: Second International Workshop on Multiparadigm Constraint Programming Languages

September 29, 2003
Kinsale, County Cork, Ireland
at the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03)

Multiparadigm programming languages combine different programming paradigms, such as functional, logic, imperative, constraint or concurrent ones.

The idea of a multiparadigm language is to increase expressiveness and problem-solving power such that the programmer can use a wide range of styles and language features from different paradigms. While the integration of constraints into general-purpose programming languages has been widely investigated for the case of logic programming, interesting solutions have been obtained as well by merging constraints and languages not based on a purely logic paradigm. The integration of constraints with other programming paradigms, even if it is not that exhaustively examined, is as well promising and a topic of current research.

This workshop addresses all aspects of multiparadigm constraint programming including:

- Combining constraints with imperative, object-oriented, concurrent, functional or functional logic languages,
- Language concepts,
- Implementation,
- Theory and semantics, and
- Applications.

Organization

Workshop organizers:

Petra Hofstedt (University of Technology Berlin)
Michael Hanus (University of Kiel)
Armin Wolf (Fraunhofer FIRST Berlin)

Program Committee:

Slim Abdennadher (University of Munich)
Thom Frühwirth (University of Ulm)
Martin Grabmüller (University of Technology Berlin)
Michael Hanus (University of Kiel)
Petra Hofstedt (University of Technology Berlin)
Georg Ringwelski (Cork Constraint Computation Center)
Peter Stuckey (University of Melbourne)
Armin Wolf (Fraunhofer FIRST Berlin)

Declarative Laziness in a Concurrent Constraint Language

Alfred Spiessens, Raphaël Collet, and Peter Van Roy

Université Catholique de Louvain,
Place Sainte-Barbe, 2, B-1348 Louvain-la-Neuve, Belgium
{fsp,raph,pvr}@info.ucl.ac.be

Abstract. This paper explains how to design and implement an extension for by-need synchronization for a confluent (subset of a) multi-paradigm concurrent constraint language, while keeping the extended language confluent. It reveals the subtleties and pitfalls that can easily lead to the loss of confluence, especially in languages with a powerful unification operator. The authors report on their own experiences, and provide guidelines for similar projects, based on considerations regarding the monotonicity of the constraint store. This paper also explores the boundaries of confluent extensibility for such languages.

1 Introduction

Confluence, or deterministic concurrency, has many advantages for application design and analysis, for security, but also for pedagogical purposes. As Oz is a multi-paradigm language, used for concept-based teaching of programming skills [8], and at the same time as an instrument for research, it was conceived to be very important to have a well-defined deterministic concurrent subset of the language that can guarantee the confluence of all programs written in it. As an initial construct for by-need synchronization turned out to be not confluent, an investigation was done to find the reasons for the loss of confluence and, if possible, also find the cure. We succeeded in both goals, and we report on our most important experiences and conclusions from this investigation in this paper.

The paper is organized as follows. Section 2 defines a concurrent constraint language that is confluent. Sections 3 and 4 give two different language extensions for by-need synchronization. The first is shown non-confluent, while the second respects confluence. Section 5 compares our results with another language, namely Curry. Section 6 then proposes a way to implement our ideas.

2 A Concurrent Language with Unification

This section defines a small concurrent language $\mathcal{L}$, with logic variables and unification on rational trees. A program in $\mathcal{L}$ consists of a set of *threads* that modify a shared *constraint store*. The computation model of $\mathcal{L}$ is very close to Saraswat's concurrent constraints [6]. The language $\mathcal{L}$ is a declarative subset of the multiparadigm programming language Oz [4, 7].

$$
\begin{array}{rl}
\text{store} & \sigma ::= \phi_1 \wedge \cdots \wedge \phi_n \\
\text{constraint} & \phi ::= x{=}y \mid x{=}v \mid \xi : \mathbf{proc}\ \{\$\ X_1 \cdots X_n\}\ S\ \mathbf{end} \\
\text{partial value} & v ::= l(x_1 \cdots x_n) \mid \xi
\end{array}
$$

Fig. 1. Abstract syntax of constraint stores

2.1 The Constraint Store

The syntax of constraint stores and partial values is given in Fig. 1. A constraint store σ consists of a conjunction of elementary constraints ϕ_i over store entities. The empty store, i.e., without constraint, is written $\top$. The main constraint in our system is equality between logic variables x, y, z, or between a variable and a partial value v. A value is either a *name* ξ (see below), or a *record* $l(x_1 \cdots x_n)$ with $n \geq 0$, where l denotes a literal. A record is also called a partial value because its contained variables x_i may be not constrained. The third kind of constraint is an association between a *name* ξ and a *closure* $\mathbf{proc}\ \{\$\ X_1 \cdots X_n\}\ S\ \mathbf{end}$. Such associations are always unique. A name is an internal store value without a representation in a program, while a closure represents an abstraction of a statement S.

A store σ *entails* a given constraint ϕ if ϕ is logically implied by the store constraints. We write this as $\sigma \models \phi$. Two stores σ and σ' that entail each other are said to be *equivalent*, which is written $\sigma \equiv \sigma'$. A variable x bound by equality to a value v is said to be *determined*. We write this as $\sigma \models \mathrm{det}(x)$.

A store is *consistent* if there exists a valuation of the variables that satisfies all its constraints, otherwise it is *inconsistent*. The constraints ϕ are chosen so that the consistency of a store is a decidable property. An *inconsistent* store is written $\bot$.

Two operations exist on constraint stores: *ask* and *tell*. Asking a constraint ϕ is waiting until the store entails or disentails ϕ, i.e., $\sigma \models \phi$ or $\sigma \models \neg\phi$. Telling a constraint ϕ to a store σ is updating the store to $\sigma \wedge \phi$. A program *fails* when it tells a constraint that makes the store inconsistent. In that situation, the whole program stops and the store becomes $\bot$. A practical language such as Oz actually does not tell constraints that makes the store inconsistent, but rather uses an exception mechanism. We did not include such a mechanism in our language, because in the presence of concurrency it leads to nondeterminism.

2.2 The Language $\mathcal{L}$

The syntax of statements S is given in Fig. 2. The letters X, Y, P denote identifiers, and t denotes a term. The lexical scope of an identifier X is restricted by

- a variable declaration ($\mathbf{local}\ X\ \mathbf{in} \ldots \mathbf{end}$),
- a case statement ($\mathbf{case} \ldots \mathbf{of}\ l(\ldots X \ldots)\ \mathbf{then} \ldots \mathbf{end}$),
- a procedure definition ($\mathbf{proc}\ \{P \ldots X \ldots\} \ldots \mathbf{end}$).

statement	$S ::=$	**skip**	(empty statement)
	$\mid$	$S_1\ S_2$	(sequence)
	$\mid$	**thread** S **end**	(thread creation)
	$\mid$	**local** X **in** S **end**	(variable declaration)
	$\mid$	$X{=}t$	(unification)
	$\mid$	**case** X **of** $l(Y_1 \cdots Y_n)$ **then** S_1 **else** S_2 **end**	(pattern matching)
	$\mid$	**proc** $\{P\ X_1 \cdots X_n\}\ S$ **end**	(procedure creation)
	$\mid$	$\{P\ X_1 \cdots X_n\}$	(procedure application)
term	$t ::=$	$Y \mid l(Y_1 \cdots Y_n)$	

Fig. 2. Syntax of the language $\mathcal{L}$

A small-step operational semantics of $\mathcal{L}$ is given in Fig. 3. It defines transition rules between *configurations*, which are applicable when a certain condition is satisfied. A configuration $\mathcal{T}/\sigma$ consists of a multiset $\mathcal{T}$ of threads $(T_i)_{1\leq i\leq n}$[1], together with a constraint store σ. A *thread* is a stack of *semantic statements*. A semantic statement is a statement S where every *free* identifier has been replaced by a store variable. An abstract syntax for threads is

$$T \ ::= \ \langle\rangle \mid \langle S\ T\rangle \ ,$$

where S denotes a semantic statement. The statement in front of a thread is the next statement to execute. A transition between configurations $\mathcal{T}/\sigma$ and $\mathcal{T}'/\sigma'$ is written

$$\frac{\mathcal{T} \parallel \mathcal{T}'}{\sigma \parallel \sigma'}\ \textit{condition}\ .$$

Rules (1) state that threads execute with an interleaving semantics. A thread with no statement is terminated, it eventually disappears. When the store becomes inconsistent, all threads may disappear. Rules (2) define the semantics for the empty statement, the sequence, and thread creation. In (3), a new variable x is introduced by replacing all the occurrences of the declared identifier X in its lexical scope by x. The substitution function is noted $\{X{\mapsto}x\}$. With (4), a unification simply tells a constraint in the constraint store; u is either a variable or a partial value. The store might become inconsistent. Rules (5) and (6) make the **case** statement block until the variable x becomes determined. The statement then reduces to the first statement if the value of x matches the pattern $l(Y_1 \cdots Y_n)$, or the second statement otherwise. In (7), the statement **proc** creates a closure in the store, with a fresh name ξ, then reduces to an unification. In (8), applying a procedure consists in taking the closure associated to p in the store, and substituting the arguments in the statement S.

[1] For the sake of simplicity, we denote $\mathcal{T}$ as a sequence $T_1 \ldots T_n$, the order being irrelevant in this context.

$$\frac{\mathcal{T}\ \mathcal{U} \,\|\, \mathcal{T}'\ \mathcal{U}}{\sigma \,\|\, \sigma'} \text{ if } \frac{\mathcal{T} \,\|\, \mathcal{T}'}{\sigma \,\|\, \sigma'} \qquad \frac{\langle\rangle \,\|}{\sigma \,\|\, \sigma} \qquad \frac{\mathcal{T} \,\|}{\bot \,\|\, \bot} \tag{1}$$

$$\frac{\langle \mathtt{skip}\ T\rangle \,\|\, T}{\sigma \,\|\, \sigma} \qquad \frac{\langle S_1\ S_2\ T\rangle \,\|\, \langle S_1\ \langle S_2\ T\rangle\rangle}{\sigma \,\|\, \sigma} \qquad \frac{\langle \mathtt{thread}\ S\ \mathtt{end}\ T\rangle \,\|\, T\ \langle S\ \langle\rangle\rangle}{\sigma \,\|\, \sigma} \tag{2}$$

$$\frac{\langle \mathtt{local}\ X\ \mathtt{in}\ S\ \mathtt{end}\ T\rangle \,\|\, \langle S\{X\mapsto x\}\ T\rangle}{\sigma \,\|\, \sigma}\ x \text{ fresh variable} \tag{3}$$

$$\frac{\langle x{=}u\ T\rangle \,\|\, T}{\sigma \,\|\, \sigma \wedge x{=}u} \text{ if } \sigma \wedge x{=}u \text{ is consistent} \qquad \frac{\langle x{=}u\ T\rangle \,\|}{\sigma \,\|\, \bot} \text{ otherwise} \tag{4}$$

$$\frac{\left\langle \begin{array}{l}\mathtt{case}\ x\ \mathtt{of}\ l(Y_1 \cdots Y_n)\\ \quad \mathtt{then}\ S_1\ \mathtt{else}\ S_2\ \mathtt{end}\ T\end{array}\right\rangle \,\|\, \langle S_1\{Y_1{\mapsto}y_1, \ldots, Y_n{\mapsto}y_n\}\ T\rangle}{\sigma \,\|\, \sigma} \text{ if } \sigma \models x{=}l(y_1 \cdots y_n) \tag{5}$$

$$\frac{\left\langle \begin{array}{l}\mathtt{case}\ x\ \mathtt{of}\ l(Y_1 \cdots Y_n)\\ \quad \mathtt{then}\ S_1\ \mathtt{else}\ S_2\ \mathtt{end}\ T\end{array}\right\rangle \,\|\, \langle S_2\ T\rangle}{\sigma \,\|\, \sigma} \text{ otherwise} \tag{6}$$

$$\frac{\langle \mathtt{proc}\ \{\, p\ X_1 \cdots X_n\,\}\ S\ \mathtt{end}\ T\rangle \,\|\, \langle p{=}\xi\ T\rangle}{\sigma \,\|\, \sigma \wedge \xi : \mathtt{proc}\ \{\,\$\ X_1 \cdots X_n\,\}\ S\,\mathtt{end}}\ \xi \text{ fresh name} \tag{7}$$

$$\frac{\langle \{\, p\ x_1 \cdots x_n\,\}\ T\rangle \,\|\, \langle S\{X_1{\mapsto}x_1, \ldots, X_n{\mapsto}x_n\}\ T\rangle}{\sigma \,\|\, \sigma} \text{ if } \sigma \models p{=}\xi \wedge \xi : \mathtt{proc}\ \{\,\$\ X_1 \cdots X_n\,\}\ S\,\mathtt{end} \tag{8}$$

Fig. 3. Small-step semantics of the language $\mathcal{L}$

We assume that the execution is *fair*, i.e., a thread cannot be kept runnable without being executed. The reader can easily check from the rules that a runnable thread stays runnable until execution.

2.3 The Confluence of $\mathcal{L}$

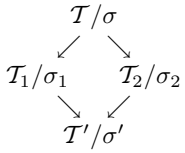
The concurrent nature of $\mathcal{L}$ is such that the language is *confluent*, which means that the "result" of a program (i.e., its final configuration) is always the same, whatever the order of thread reduction. In other words, every program is deterministic.

Some transition rules introduce fresh symbols, namely variables in (3) and names in (7). The confluence should not depend on the choice of those symbols. We thus define an *equivalence between configurations* as follows. Let $\mathcal{T}/\sigma$ and $\mathcal{T}'/\sigma'$ be configurations, and $\mathcal{V}$ be a set of variables. The configurations are said to be *equivalent modulo* $\mathcal{V}$, which we write $\mathcal{T}/\sigma \equiv \mathcal{T}'/\sigma'\ (\mathcal{V})$, if there exists a bijection r such that

– r maps variables to variables, and names to names;
– r is the identity function on $\mathcal{V}$;
– $r(\mathcal{T}) = \mathcal{T}'$ and $r(\sigma) \equiv \sigma'$, where r is used as a replacement on statements and stores.

We define a transition relation that relates configurations up to equivalence. We write $\mathcal{T}/\sigma \longrightarrow \mathcal{T}'/\sigma'$ $(\mathcal{V})$ if there exists a finite execution with the transition rules of $\mathcal{L}$, with $\mathcal{T}/\sigma$ as initial configuration, and whose final configuration is equivalent to $\mathcal{T}'/\sigma'$ modulo $\mathcal{V}$. The confluence property is then defined as follows.

Theorem 1 (Confluence). *Let $\mathcal{T}/\sigma$, $\mathcal{T}_1/\sigma_1$ and $\mathcal{T}_2/\sigma_2$ denote configurations, and $\mathcal{V}$ be a set of variables.*
If $\mathcal{T}_1/\sigma_1 \longleftarrow \mathcal{T}/\sigma \longrightarrow \mathcal{T}_2/\sigma_2(\mathcal{V})$, then there exists a configuration $\mathcal{T}'/\sigma'$ such that $\mathcal{T}_1/\sigma_1 \longrightarrow \mathcal{T}'/\sigma' \longleftarrow \mathcal{T}_2/\sigma_2(\mathcal{V})$.
The diagram on the right depicts this property.



From this property, we can easily characterize the executions of a program. For instance, if a program fails in one execution, it always fails. In that case, every execution will reach the final configuration $\{\}/\bot$. Note that this configuration is clearly not equivalent to a program that terminates with some threads still blocked (not runnable). Such a program can be qualified as *partially terminated*. This means that if an "external agent" tells some constraints in the program's store, the program might execute further, and possibly reach a new partial termination, which is unique by confluence.

2.4 Functional Programming in $\mathcal{L}$

Our language is expressive enough to reproduce the behavior of functional programs. The idea is simply to translate a functional program into $\mathcal{L}$, functions becoming procedures, and expressions being expressed in elementary operations. The following simple example gives an idea of this translation. The "bar" operator `X|Xr` is a simple notation for a record like `cons(X Xr)`.

```
fun {Append Xs Ys}
   case Xs of X|Xr then
      X|{Append Xr Ys}
   else Ys end
end
```

```
proc {Append Xs Ys Zs}
   case Xs of X|Xr then
      local Zr in
         Zs=X|Zr
         {Append Xr Ys Zr}
      end
   else Zs=Ys end
end
```

3 A Flawed Definition of By-need Synchronization

In this section we present the definition of by-need synchronization described in [3]. It is implemented in Mozart [4] at least until the current version (1.2.5). We will show that it does not respect the confluence of the language, and investigate why.

statement	$S ::= \ldots$	(syntax rules defined in Fig. 2)
	| `{ByNeed` P X`}`	(execution of P when X is needed)

Fig. 4. Syntax extension of the language $\mathcal{L}$ with `ByNeed`

3.1 Naive Definition and Semantics

The `ByNeed` construct allows a computation to be associated to a logic variable, which represents the result of the computation. The computation will be performed as soon as its result becomes needed. Figure 4 shows the added statement.

The operational semantics are set up in a way as to assure the following rules in the computation. We use x and p as the variable and its associated calculation, respectively.

- `{`p x`}` will be calculated as soon as a statement needs the variable x for its reduction.
- A statement needs x if it cannot reduce without x being determined.
- The unification of x with a determined variable needs x. This will protect x from being unified with a value before p itself has ended. Only the proper invocation of `{`p x`}` will be allowed to bind x.
- The unification of x with another by-need variable needs x. This rule ensures that, if `{`p`}` would itself return a by-need variable, the latter would immediately be calculated before being assigned to x.
- `{`p x`}` will be calculated at most once for every application of `{ByNeed` p x`}`.

The reader will notice that this definition of `ByNeed` is indeed inspired by functional programming. It is modeled after a typical "let" construct [1, 2, 5] allowing for the declaration of a value with a predefined expression, and in the mean time protecting the variable from being overwritten by another value. This was translated into our programming language, which provides functions as syntactic sugar for procedures with at least one parameter (see Sect. 2.4). However, most constructs in our language stem from concurrent constraint programming. It is one of these differences with other multi-paradigm languages [1, 2] that would turn out to be important in unexpected ways.

3.2 Counterexample for Confluence

We found a counterexample that proved `ByNeed` to introduce non-confluence in the language, when we examined the following procedure and its applications.

```
proc {ReadOnly X Y}  % make Y a read-only version of X
   Y={ByNeed proc {$ Z} {Wait X} Z=X end}
end
```

Since every attempt to unify `Y` to a value (or another read-only variable) will start the computation, and synchronize on `X` becoming bound, this was an obvious application for the `ByNeed` function. The following application defies confluence.

```
local X Y Z in
   thread X=Y end
   thread Y=2 end
   thread X=1 end
   thread X={ReadOnly Z} end
end
```

Depending on the order of execution, this example will fail or succeed. Let us look at it in detail:

1. Suppose these concurrent statements are executed in the order of their definition. First both free variables `X` and `Y` are unified. This statement just adds the equality constraint to the store. Then `Y=2` unifies `Y` with the constant `2`, resulting in both `X` and `Y` having the value `2`. The next statement `X=1` will then fail because it would introduce inconsistency into the store. The last statement will not be executed due to the failure.
2. When the concurrent statements are executed in the reverse order, something different happens. First `X` becomes a by-need variable, to be bound to the eventual value of `Z`. Next `X=1` triggers the computation of the value of `X`, causing an indefinite waiting for `Z` to become determined. The statement `Y=2` binds `Y` to `2`. At last, `X=Y` waits indefinitely for `X` to become determined via `Z`. Since `Z` is local within **`local ... end`**, no constraint can be added to the store afterwards, that would bind `Z` to a value. This means that both executions are not confluent.

The fact that `ByNeed` can be used to make unification block was quickly generalized to the following observation: *Any language construct that can make our unification operator block, will introduce non-confluence in the language.* This is shown in the following generalized example in pseudo-code.

```
local X Y Z in
   thread setup X and Z to make X=Z block end
   thread X=Y tell C1 end
   thread Y=Z tell C2 end
end
```

The execution of the first thread prevents the unification of `X` and `Z` to reduce. Therefore the second thread can still tell its constraint *C1*, but the third thread blocks. Changing the order of execution of the threads results in either *C1*, or *C2*, or both to be told to the store. If *C1* and *C2* are chosen to be incompatible, the computation can also fail due to inconsistency. In Sect. 5 we explain why this observation cannot be done in the subset of Curry, described in [2].

Our unification operator is constraint-oriented. That means that the unification of free variables x and y adds the constraint $x{=}y$ to the constraint store, and does not have to wait for x or y to become determined. It is also a very rich and powerful monotonic unification operator, that unifies partial values into the union of the information they carry. It causes a failure if the partial values x and y are incompatible.

3.3 The Reason for the Loss of Confluence

The deep reason for the loss of confluence is the loss of monotonicity. This is best understood in the context of the *ask* and *tell* operators of CCP (see [6] and Sect. 2.1). Before the introduction of `ByNeed`, unification was a simple *tell* operation, adding the equality constraint to the store and never blocking. `ByNeed` seems to have somehow turned unification into an *ask* operation, since it now can block. The unification of a by-need variable x with a needed variable y (or a partial value) transfers the need to x and triggers the evaluation of the expression associated with x. But in a monotonic setting, an operation should not block with a constraint store with more information available than in another store it does not block with.

To ensure confluence, monotonicity is to be kept in all the state transitions from a free to a determined variable. Since unification (now an *ask* operation) blocks for by-need variables, it should also block for free variables.

4 A Good Definition of By-need Synchronization

In this section we give the revised definition and semantics of by-need synchronization, and give its interpretation in terms of constraints, to show that confluence is indeed respected this time.

4.1 Revised Definition and Semantics

We use a new constraint, need(x), to express that the determinacy of x is needed by the program, together with a primitive statement `{WaitNeed x}` to ask that constraint. Figure 5 gives the syntax extension for this new primitive, and its operational semantics. The relation $\text{need}_\sigma(S, x)$ defines when a statement S needs a variable x in the store σ. We assume that this relation is *stable* as defined in Def. 1 below. With this primitive we build a confluent by-need construct, that we call `OnDemand` to avoid confusion with the previous one.

```
proc {OnDemand P X}
   thread {WaitNeed X} {P X} end
end
```

A by-need computation is now simply a thread that waits for a variable to be needed. In order to ensure monotonicity, determined variables are always needed, i.e., $\text{det}(x) \Rightarrow \text{need}(x)$. The need constraint allows to define three possible states for a variable, namely *free*, *needed* and *determined*. Those three states are presented in Fig. 6.

4.2 The Revised Definition Respects the Confluence of the Language

Let us analyze the new rules for by-need synchronization, in terms of constraints.

statement	$S ::= \ldots$	(syntax rules defined in Fig. 2)
	$\mid$ `{WaitNeed` X `}`	(wait for X to be needed)
constraint	$\phi ::= \ldots$	(syntax rule defined in Fig. 1)
	$\mid$ need(x)	(variable x is needed)

$$\frac{\langle \texttt{\{WaitNeed } x \texttt{\} } T \rangle \,\|\, T}{\sigma \,\|\, \sigma} \text{ if } \sigma \models \text{need}(x) \tag{9}$$

$$\frac{S \,\|\, S}{\sigma \,\|\, \sigma \wedge \text{need}(x)} \text{ if need}_\sigma(S, x) \text{ and } \sigma \not\models \text{need}(x) \tag{10}$$

Fig. 5. Syntax and operational semantics of `WaitNeed` and need()

1. `{WaitNeed` x `}` is a simple *ask* operation, just waiting for the constraint need(x) to be entailed by the store.
2. A statement S needing x to become determined for its reduction, will simply tell need(x). (Remember that once x is needed, it stays needed forever.)

Here the rules of the operational semantics no longer imply that a statement should block in order to trigger the on-demand computation associated with a variable. This is indeed the crucial difference with the previous version, and it is enabled by the introduction of a monotonic "needed" state for variables, visualized in Fig. 6. The unification of a needed variable with a free variable can now make the free variable needed, (performing its "monotonic" duty) without having to block for this *transfer of need* to be assured. The `ReadOnly` function from the example in Sect. 3.2 can no longer make unification block.

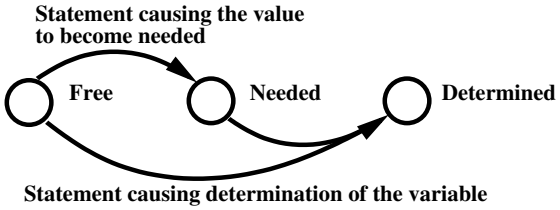


Fig. 6. State transitions of a variable, with the need constraint

The confluence of the language is ensured if the relation need$_\sigma(S, x)$ is stable.

Definition 1 (Stable need). *We say that the need relation* need$_\sigma(S, x)$ *is* stable *if, for every two stores* σ*,* σ' *such that* $\sigma' \models \sigma$*,*

$$\text{need}_\sigma(S, x) \textit{ and } \sigma \not\models \text{need}(x) \textit{ implies } \neg\text{reduce}_\sigma(S) \tag{11}$$

$$\text{need}_\sigma(S, x) \textit{ implies } \text{need}_{\sigma'}(S, x) \textit{ or } \sigma \models \text{need}(x) \tag{12}$$

This property of the relation guarantee that a statement that needs a variable x cannot reduce before need(x) is in store (11), even when the store is evolving monotonically (12). A simple case-analysis of all semantic rules, by checking the conditions for reduction, now reveals that every statement in the language respects confluence.

The following need relation, defined as in [8], is an example of a stable need relation, but others are possible.

$$\begin{aligned} \text{need}_\sigma(S, x) \text{ iff} \quad & \neg\text{reduce}_\sigma(S) \\ \text{and} \quad & \exists\sigma' : \sigma' \models \sigma \text{ and } \text{reduce}_{\sigma'}(S) \\ \text{and} \quad & \forall\sigma' : \sigma' \models \sigma \text{ and } \text{reduce}_{\sigma'}(S) \text{ implies } \sigma \models \det x \end{aligned}$$

4.3 A Few More Remarks

Need-Triggered Execution. In contrast with "function-oriented" multi-paradigm languages, by-need synchronization unifies the variable to its eventual value *inside* the procedure it was associated with by the application of `OnDemand`. But of course the decision to unify its parameter to a value is up to the procedure. This provides a more general mechanism for any kind of computation synchronizing on the need of a variable.

Efficient Implementation. The fact that `OnDemand` itself does not *tell* any information to the store before *asking* for the need(x) condition, indicates that there is no need for an "on-demand" state in this new definition. No difference is detectable in the store before and after the application of `OnDemand`. This observation drastically simplifies the implementation of `OnDemand`, as described in Sect. 6.

The Definitive Loss of a Confluent Read-Only. The operation `OnDemand` can no longer be used to build read-only variables that cause unification to block. A "read-only" variable build as in Sect. 3.2 will be forced to become determined upon unification with a value. In fact it becomes clear that our language will never be able to protect variables by causing their unification to block, while respecting confluence.

5 Related work

The language Curry [1, 2] is a good example to compare to. Curry is an integrated functional logic language. It combines features from functional programming (nested expressions, higher-order functions, lazy evaluation), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of expressions with synchronization on logical variables). Curry is confluent.

The definitions of laziness and unification in Curry are a bit different from our language. In order to show those differences, we can compare the Curry constraint on the left with the Oz constraint on the right:

```
                        local X Y in
let x = <expr1>            X={OnDemand proc {$ Z} Z=<expr1> end}
    y = <expr2>            Y={OnDemand proc {$ Z} Z=<expr2> end}
in x =:= y                 X=Y
                        end
```

Though they look similar, they behave differently. The Curry constraint `x =:= y` forces both expressions to be evaluated, then it checks for the satisfiability of the equality. The Oz constraint `X=Y` does not force the by-need computations, since both `X` and `Y` are simply free. Both will be forced only when `X` (or `Y`) becomes needed.

What makes Curry confluent is the way logic variables relate to "normal" variables. A variable declared in a `let` construct is associated to an expression that is evaluated lazily. So a variable is always declared *together* with an expression. A logic variable is a special case, where the associated expression does not give a value, but rather a *black hole*. Logic variables are typically declared as in

```
let x = x in x =:= 42
```

In Curry a logic variable is a variable whose *complete* evaluation does not lead to a value. By the way the language is defined, it is not possible to associate a lazy computation to a logic variable. This is why unification forces its both arguments to be evaluated.

6 Implementation

We now describe how to extend an existing implementation of Oz, which our language $\mathcal{L}$ was a subset of. The constraint store of Oz is implemented as a graph, where nodes are variables and partial values. Each equivalence class of variables has a union-find structure, i.e., each variable node (except one) has an outgoing edge to another variable node in the same equivalence class, and those nodes form a tree where the edges are directed to the root node. The latter is the class representative. It may have an outgoing edge to a value node, meaning that the variable is bound to the given value.

When a thread blocks on a variable (asking for a constraint on that variable), a reference to the thread is put in a *suspension list* associated to the class representative of the variable. When a constraint is put on the variable, all the threads in the suspension list are woken up, and given to the thread scheduler. The suspension list allows to synchronize threads on constraints.

We simply extend this suspension list so that it also handles the constraint need(x). Each class representative now has a *needed* state that tells whether the variable is needed or not. When a thread blocks on need, and the variable is not needed yet, the thread is simply put it the suspension list. As soon as the variable becomes needed, the threads in the suspension list are woken up.

If a thread blocks on determinacy, we set the variable in the *needed* state and schedule the threads in the suspension list. When a variable x is bound to a value, we simply schedule its suspension list.

The implementation is in progress and will be part of a future release of Mozart [4].

7 Conclusion

We have tried and succeeded in extending a deterministic (subset of a) multi-paradigm CCP language with by-need synchronization, while respecting confluence. For constraint-oriented multi-paradigm languages, such an extension is not straightforward, and should not be based naively on its counterpart in function-oriented multi-paradigm languages. We showed that the semantics of the unification operator makes a subtle difference that can have an important influence on the confluence of the extended language. Finally, we gave an example of how reasoning from constraints and monotonicity should guide the design for confluence-preserving extensions of multi-paradigm CCP languages.

Acknowledgements

This research was partly funded by the MILOS project of the Walloon Region of Belgium (convention 114856).

References

1. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.
2. M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.
3. Michael Mehl, Christian Schulte, and Gert Smolka. Futures and by-need synchronization. Technical report, Programming Systems Lab, DFKI and Universität des Saarlandes, May 1998. DRAFT.
4. Mozart Consortium (DFKI, SICS, UCL, UdS). The Mozart programming system (Oz 3), January 1999. Available at `http://www.mozart-oz.org`.
5. Simon Peyton Jones, editor. *Haskell 98 language and libraries: The revised report.* Cambridge University Press, 2003. Also published as the January 2003 Special Issue of the Journal of Functional Programming.
6. Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.
7. Gert Smolka. The Oz programming model. In *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.
8. Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* 2002. Work in progress. Expected publishing date 2003.

Implementing a Distributed Shortest Path Propagator with Message Passing

Luis Quesada, Stefano Gualandi, and Peter Van Roy

Université Catholique de Louvain
Place Sainte Barbe, 2, B-1348 Louvain-la-Neuve, Belgium
{luque, stegua, pvr}@info.ucl.ac.be

Abstract. In this article, we present an implementation of a distributed shortest path propagator. Given a graph and a goal node, this propagator maintains a finite domain variable for every node. The variable's lower bound is the minimal cost of reaching the goal from that node. The graph on which the propagator is based can be modified by removing edges, increasing the cost of edges, or by replacing edges by graphs. The propagator has been implemented using a message passing approach on top of the multi-paradigm programming language Oz [2]. One of the advantages of using a message passing approach is that distributing the propagator comes for free. Different shortest path propagators running on different machines may be working together on the same graph.

1 Introduction

In this article, we present the implementation of a distributed shortest path propagator. Given a graph and a goal node, this propagator offers the following services:

- It maintains, for every node, a Finite Domain (FD) variable. The lower bound of the variable is the minimal cost of reaching the goal from that node. The propagator maintains these lower bounds while the graph is dynamically modified.

- It allows the incremental definition of the graph on which the propagator is based. The user may start by providing an abstract graph (i.e., a graph whose edges are virtual) and then proceed by replacing each edge by its corresponding graph. The user can execute the propagator in a distributed way, since he can choose to launch the propagator associated with a virtual edge on a different machine.

The graph on which the propagator is based can be modified either by increasing the cost of an edge or by deleting an edge[1]. It is important to emphasize that we only consider monotonic changes in the graph (i.e., changes that lead to further

[1] The reader may also think of deleting an edge as increasing its cost to ∞. However, mostly for optimization reasons, we prefer to keep the two operations separate.

constrain the FD variables involved).
In order to maintain the minimal costs (and thus the FD variables) we follow the asynchronous dynamic algorithm described by [9], which is based on the principle of optimality.
Let us represent the shortest distance from node i to the goal node as $h^*(i)$. The shortest distance via a neighboring node j is given by $f_i^*(j) = k(i,j) + h^*(j)$, where $k(i,j)$ is the cost of the link between i and j. If node i is not the goal node, the path to the goal node must visit one of the neighboring nodes. Therefore, $h^*(i) = min_j f_i^*(j)$ holds.
If h^* is given for each node, the optimal path can be obtained by repeating the following procedure. For each neighboring node j of the current node i, compute $f_i^*(j) = k(i,j) + h^*(j)$. Then move to the j that gives $min_j f_i^*(j)$.
Asynchronous dynamic programming computes h^* by repeating the local computations at each node. Let us assume the following situation:

1. For each node i, there exists a process corresponding to i.
2. Each process records $h(i)$, which is the estimated value of $h^*(i)$. We initialize $h(i)$ to ∞.
3. For the goal node g, $h(g)$ is 0.
4. Each process can refer to the h values of neighboring nodes.

Each process updates $h(i)$ by the following procedure. For each neighboring node j, compute $f_i(j) = k(i,j) + h(j)$, where $h(j)$ is the current estimated distance from j to the goal node, and $k(i,j)$ is the cost of the link from i to j. Then, update $h(i)$ as follows: $h(i) \leftarrow min_j f_i(j)$. The execution order of the processes is arbitrary.
There are several ways of implementing the algorithm sketched above. We implement it by considering the set of processes as a multi-agent system where agents interchange synchronous and asynchronous messages and their transition state functions rely on data flow and constraint programming primitives.
The organization of the paper is as follows. In Section 2, we introduce the Oz language and explain how message-passing concurrency can be modeled in it. In Section 3, we show how to use the propagator and explain the implementation of it by showing its message passing diagram and its corresponding state transition system. We conclude in Section 4.

2 Message-passing concurrency in Oz[2]

2.1 The Oz language and its Execution Model (Declarative Subset)

The declarative part of the Oz execution model consists of a store and a set of dataflow threads that reference logic variables in the store (see Figure 1). Threads contain statement sequences S_i and communicate through shared references. A thread is a *dataflow* thread if it only executes its next statement when all the values the statement needs are available. Data availability is implemented using

[2] This section is a summary of section 2 of [7], and Chap. 5 of [6]

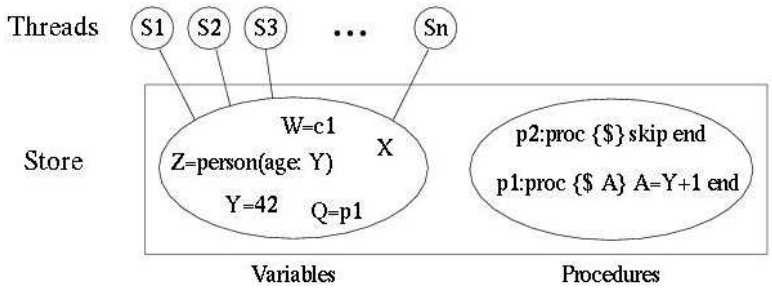


Fig. 1. The Oz execution model (Declarative Subset).

logic variables. If the statement needs a value that is not yet available, then the thread automatically blocks until the value is available. There is also a fairness condition: if all values are available then the thread will eventually execute its next statement.

The shared store is not physical memory; rather it is an abstract store that only allows legal operations for the entities involved, i.e., there is no direct way to inspect their internal representations. The store consists of two compartments, namely logic variables (with optional bindings) and procedures (named lexically scoped closures). Variables can reference the names of procedures. The external references of threads and procedures are variables. When a variable is bound, it disappears, i.e., all threads that reference it will automatically reference the binding instead. Variables can be bound to any entity, including other variables. The variable and procedure stores are monotonic, i.e., information can only be added to them, not changed or removed. Because of monotonicity, a thread that is not blocked is guaranteed to stay not blocked until it executes its next statement.

All Oz execution can be defined in terms of a kernel language whose semantics are outlined in [1], [8] and [6]. We will just describe the declarative part of it.

$S ::= S\ S$	Sequence
$\mid X = f(l_1 : Y_1 \ldots l_n : Y_n) \mid$	Value
$\mid X =$<number> $\mid X =$<atom>	
$\mid$ **local** $X_1 \ldots X_n$ **in** S **end** $\mid X = Y$	Variable
$\mid$ **proc** {X $Y_1 \ldots Y_n$}S **end** $\mid$ {X $Y_1 \ldots Y_n$}	Procedure
$\mid$ **if** X **then** S **else** S **end**	Conditional
$\mid$ **thread** S **end**	Thread

Table 1. The Oz declarative kernel language.

Table 1 defines the abstract syntax of a statement S in the (declarative part of the) Oz kernel language. Statement sequences are reduced sequentially inside a thread. All variables are logic variables, declared in an explicit scope defined by the local statement. Values (records, numbers, etc.) are introduced explicitly and can be equated to variables. Procedures are defined at run-time with the **proc** statement and referred to by a variable. Procedure applications block until the first argument references a procedure name. The **if** statement defines a conditional that blocks until its condition is **true** or **false** in the variable store. Threads are created explicitly with the **thread** statement.

In the following section, we are going to be using a bit of syntactic sugar to make programs easier to read. We will do so by considering that:

- **proc** `{P V1 V2 ... Vn} <Decl>` **in** `<Stmt>` **end** is equivalent to **proc** `{P V1 V2 ... Vn}` **local** `<Decl>` **in** `<Stmt>` **end end**, where `<Decl>` is a declaration (i.e., a statement declaring a variable) and `<Stmt>` is any statement.
- **fun** `{F V1 V2 ... Vn} <Stmt> <Exp>` **end** is equivalent to **proc** `{F V1 V2 ... Vn O} <Decl>` **in** `<Stmt> O=<Exp>` **end**, where `<Exp>` is an expression representing a value.
- **fun** `{F V1 V2 ... Vn} <Decl>` **in** `<Exp>` **end** is equivalent to **fun** `{F V1 V2 ... Vn}` **local** `<Decl>` **in** `<Exp>` **end end**.

Procedures are values in Oz. This means that a variable may be bound to a procedure. In particular, we have that **proc** `{X V1...Vn}...` **end** is equivalent to `X=`**proc** `{$ V1...Vn}...` **end**.

2.2 The message-passing concurrent model

The message-passing concurrent model extends the declarative concurrent model by adding ports. Ports are a kind of communication channel. Ports are no longer declarative since they allow observable nondeterminism: many threads can send a message to a port and their order is not determined. However, the part of the computation that does not use ports is still declarative.

Ports. A port is an Abstract Data Type (ADT) that has two operations:

`{NewPort S P}`: create a new port `P` associated with stream `S`.
`{Send P X}`: append `X` to the stream corresponding to the entry point `P`. Successive sends from the same thread appear on the stream in the same order in which they were executed. This property implies that a port is an asynchronous FIFO communication channel.

For example:

```
local S P in
   {NewPort S P}
```

```
   {Send P a}
   {Send P b}
   {Browse S}
end
```

This displays the stream `a|b|_`. Doing more sends will extend the stream. By asynchronous we mean that a thread that sends a message does not wait for reply; it immediately continues.

Port objects. A port object is a thread reading messages from port streams. This allows two things. First, many-to-one communication is possible: many threads can reference a given port object and send to it independently. Second, port objects can be embedded inside data structures (including messages).
Here is an example of a port object with one port that displays all the messages it receives:

```
local S P in
   {NewPort S P}
   thread {ForAll S proc {$ M} {Browse M} end} end
end
```

In this example, `ForAll` is a procedure that, given a list `L` and a procedure `P`, applies `P` to each element of `L`. Doing `{Send P hello}` will eventually display `hello`.

The NewPortObject abstraction. We can define an abstraction to make it easier to program with port objects. Let us define an abstraction for the case that the port object has just one port. To define the port object, we give the initial state `Init` and the state transition function `Fun`, which is of type $State \times Msg \rightarrow State$.

```
proc {NewPortObject Fun Init ?P}
   proc {MsgLoop S1 State}
      case S1 of Msg|S2 then
         {MsgLoop S2 {Fun Msg State}}
      [] nil then skip end
   end
   Sin
in
   thread {MsgLoop Sin Init} end
   {NewPort Sin P}
end
```

3 The Shortest Path Propagator

3.1 Interface of the Propagator

As shown in Figure 2, we need to specify the graph on which the propagator is based and the node in the graph that is the goal. The representation of the

graph is an adjacency list. The function `Create_Propagator` returns a record representing the interface of the propagator.
The propagator has the following interface:

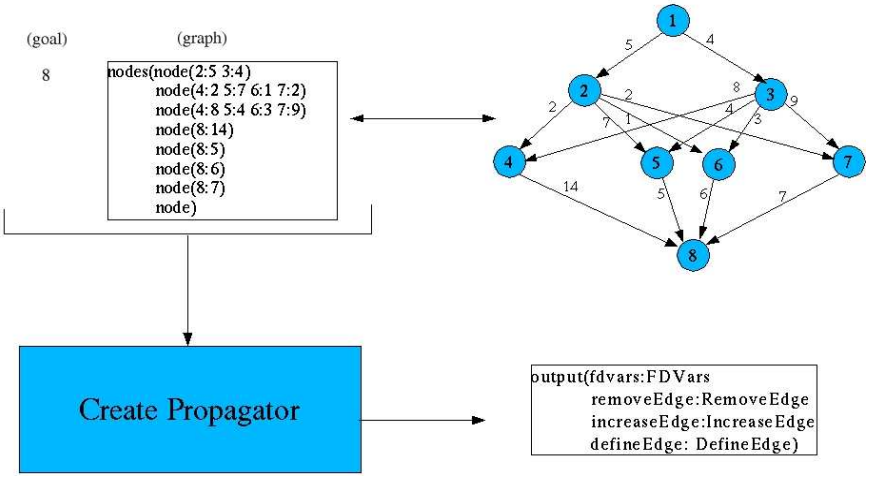


Fig. 2. Description of the propagator.

`fdVars` exports a tuple `T` of FD variables. `T.i` is the FD variable associated with node `i`. This FD variable corresponds to the cost of going from that node to the goal.

`removeEdge` exports a 1-argument procedure:
`proc` `{$ Edge} ...` **`end`**. The parameter `Edge` is the edge to be removed.

`increaseEdge` exports a 2-argument procedure:
`proc` `{$ Edge NewCost} ...` **`end`**. The parameter `Edge` is the edge whose cost is to be increased to `NewCost`[3].

`defineEdge` exports a 5-argument function:
`fun` `{$ Edge Graph Source Destination Host} ...` **`end`**. `Edge` is the edge to be replaced. `Graph` is the graph by which the edge is to be replaced. `Source` is the node in `Graph` with which the origin of `Edge` is to be associated. `Destination` is the node in `Graph` with which the destination of `Edge` is to be associated. `Host` is the url address of the host on which the propagator of `Graph` is going to be executed. If `Host` is nil, the propagator will be executed on the same machine.

In Oz, we use the following notation to represent FD variables: VariableName-{LowerBound#UpperBound}. In the implementation, the upper bounds of the

[3] The implementation assumes that the new cost is always greater than the current cost.

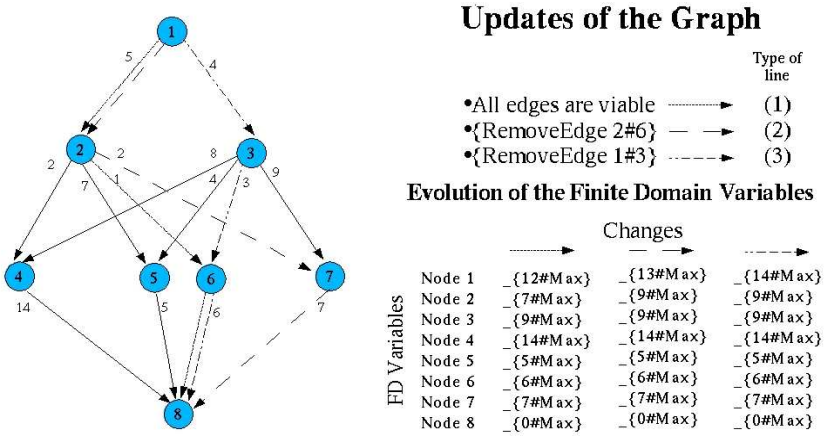


Fig. 3. Updating the FD variables of each node.

propagator's FD variables are set to a constant `Max`. This constant could be the sum of the edges' costs. As shown in Figure 3, after the creation of the propagator, the lower bound of the FD variable of each node is set to the minimal cost of reaching the destination. Internally, the propagator always keeps the shortest path to the destination for every node. In Figure 3, we show how the shortest path from node 1 to the goal node (node 8) is updated after removing edges `2#6` and `1#3`. In this Figure, you can also observe how the FD variables are updated with respect to the changes in the graph[4].

3.2 Implementing the propagator

The propagator is implemented as a set of port objects interchanging asynchronous and synchronous messages. The implementation makes use of the NewPortObject abstraction described in the previous section. In this section, we will describe the messages that our objects interchange, the attributes that these port objects have and their corresponding state transition functions. Even though Figure 4 shows three types of concurrent processes, only nodes are modeled as port objects since neither the environment nor the monitors receive messages. In the following, we will focus on the implementation of the nodes.

Attributes of the Nodes. As shown in Figure 4 each node has the following attributes:

`OutNodes`: the tuple of outgoing nodes[5].

[4] An edge is represented as a tuple of two elements. The edge `Ind1#Ind2` has `Ind1` as origin and `Ind2` as destination.

[5] `Y` is an outgoing/incoming node of `X` if `Y` is the destination/origin of one of `X`'s outgoing/incoming edges.

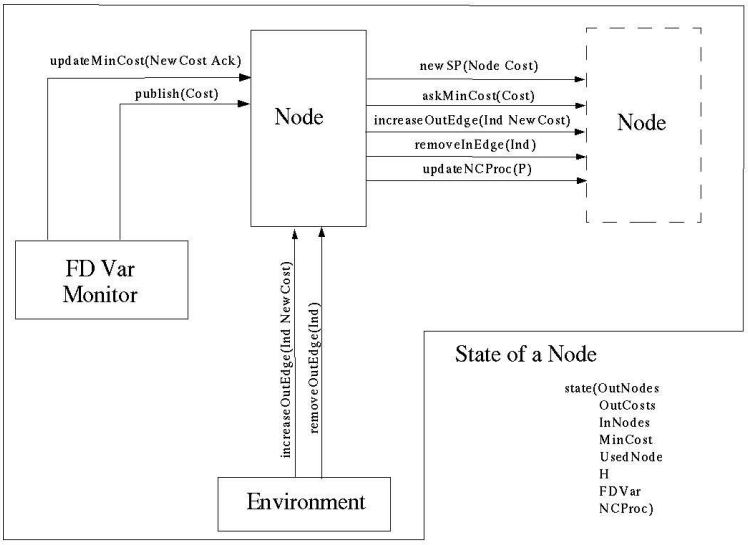


Fig. 4. Message Diagram and State of a Node.

`OutCost`: the tuple of costs of reaching the outgoing nodes.
`InNodes`: the tuple of incoming nodes.
`MinCost`: the minimum cost of reaching the destination.
`UsedNode`: the outgoing node that is being used to reach the destination.
`H`: the priority queue that keeps the outgoing nodes that are not being used. Each one of these nodes is associated with a key that represents the cost of going to the destination through that node.The priority queue is implemented with a heap.
`FDVar`: the FD variable maintains the cost of reaching the destination. One of the invariants of our system is that the lower bound of the FD variable is equal to the minimal cost of reaching the destination.
`NCProc`: the procedure to be executed whenever `MinCost` is updated. As we will see in Section 3.4, the implementation of virtual edges makes use of this attribute.

Message Diagram. Figure 4 shows the Message Diagram of the Shortest Path propagator. There are three kinds of entities: (*i*) nodes, (*ii*) FD variable monitors and (*iii*) environment. A node may receive messages from another node, from its FD variable monitor and from the environment. By environment, we mean all those entities independent to the propagator that may interact with it.

The messages that the environment may send to a particular node are:

`increaseOutEdge(Ind NewCost)`. It increases the cost of edge **`self`**`#Ind` to `NewCost` and updates the state of the node accordinly.

`removeOutEdge(Ind)`. It removes the edge **`self`**`#Ind` and updates the state of the node accordinly.

A set of propagators in a Constraint Satisfaction Problem communicate with each other through shared FD variables. The FD variables associated with each node may be updated by other propagators working concurrently with the Shortest Path Propagator. So, as the changes in the FD variables depend not only on the Shortest Path Propagator, a concurrent process (namely the FD Variable Monitor) is responsible for detecting those changes and updating the minimal cost of the corresponding node. The messages that a FD variable monitor may send to its node are:

`updateMinCost(NewCost Ack)`. It updates the minimal cost of the node and binds `Ack` once the cost has been updated. The `Ack` parameter is for the sender to wait until the execution of the message has finished. As we are using port objects, messages are asynchronous by default, so this mechanism is a way of modeling synchronous messages.

`publish(Cost)`. It communicates the new cost to the corresponding incoming nodes.

The messages that a node may send to another node are:

`newSP(Node Cost)`. It updates the state of the node according to the fact that node `Node` has a new minimal cost `Cost`.

`askMinCost(Cost)`. It binds `Cost` to the current minimal cost of the node.

`increaseOutEdge(Ind NewCost)`. It increases the cost of edge **`self`**`#Ind` to `NewCost` and updates the state of the node accordingly.

`removeInEdge(Ind)`. It removes edge **`self`**`#Ind` and updates the state of the node accordingly.

`updateNCProc(P)`. It updates `NCProc`.

3.3 State Transition

As we are using port objects, our algorithm is reduced to specifying how the state of our port objects evolve when receiving a particular message. We will consider two cases:

`removeOutEdge(Ind)`. As shown in Figure 5, there are two possibilities for the node when receiving this message depending on whether the edge to be removed is the one being used. If it is the used edge, another outgoing node is chosen from the priority queue, the FD variable and the minimum cost are updated and the procedure `NCProc` is executed. If it is not the edge used, the information of the edge is simply removed.

`newSP(Node Cost)`. As shown in Figure 6, the state transition for this message involves more cases. If `Node` is not the used node, the heap is updated with the new key for `Node`. If `Node` is the one used, there are two cases. One case is when **`self`** only has one outgoing node. If this is the case, there is no option

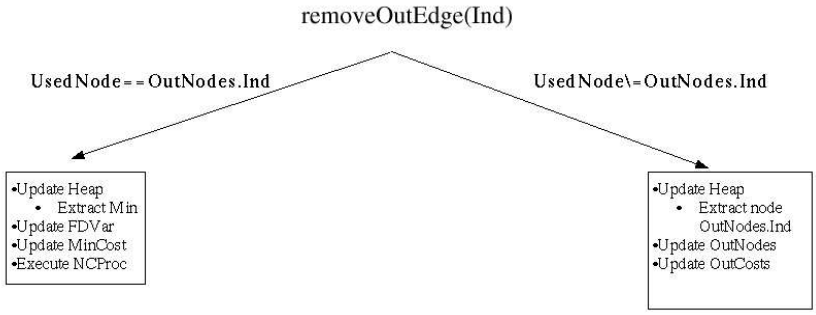


Fig. 5. State Transition for the message removeOutEdge.

but to keep using the same node. So, the FD variable and the minimal cost are updated according to `Cost` and the `NCProc` is executed. The other case is when **`self`** has more than one outgoing node. In this case there are two sub-cases. One sub-case is when the best option (to go to the destination) offered by the nodes in the heap is worse than the new option offered by the used node. In this sub-case, we simply update the FD variable and the minimal cost according to `Cost` and execute `NCProc`. Otherwise, we have to update the heap by extracting the node offering the best cost and inserting the current used node. We also have to update the FD variable, the minimal cost, the used node, and execute `NCProc`.

3.4 Dealing with virtual edges

The shortest path propagator allows the user to define the graph on which the propagator is based incrementally. It does so by letting the user associate graphs to edges. Thanks to the approach chosen to implement the propagator, the implementation of this facility is straightforward :

```
fun {DefineEdge Edge Graph Source Destination Host}
   if Host == nil then
      Ind1#Ind2=Edge
      SPP={Create_Propagator Graph Destination}
      proc {NCProc}
         thread {IncreaseEdge Ind1#Ind2 {SPP.askMinCost Source}} end
      end
   in
      {SPP.updateNCProc Source NCProc}
      SPP
   else ... end
end
```

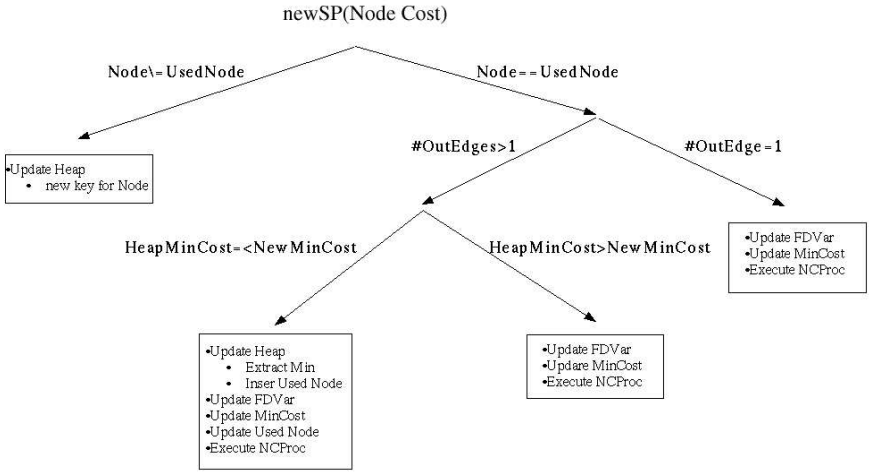


Fig. 6. State Transition for the message newSP.

An independent shortest path propagator is created for `Graph`. The port object associated with `Source` is set so that it sends an `increaseEdge` message to the origin of `Edge` whenever the corresponding `MinCost` of `Source` is updated. Here, we only show how to implement the case where all the concurrent processes are created on the same machine. Readers interested in seeing how this approach could be extended to manage the case where the processes are created on different machines may read Chap. 11 of [6].

4 Conclusion

We have presented the Shortest Path Propagator. Given a graph and a destination node, this propagator maintains, for every node, a finite domain variable whose lower bound is the minimal cost of reaching the destination from that node. Even though this is a problem already investigated (see for instance [5][6] and [9]), the value of our work lies in the fact that the algorithm is implemented using a message-passing approach on top of data flow and constraint programming primitives. The use of this sophisticated approach allows, for instance, the

[6] The propagator can also be used as an incremental algorithm for the single source shortest path problem. However, our incremental algorithm (besides only supporting monotonic changes in the graph) is not as efficient as the algorithm presented in [5]. We can only update MinCost monotonically (i.e., the value to which MinCost is updated must always be greater than the current one) because this variable is associated with the lower bound of a FD variable. [5] can perform better by allowing MinCost to be updated non monotonically (e.g., by temporarily setting affected nodes' cost to be too high) even though the changes to the graph are monotonic.

easy extension of the propagator to deal with the incremental definition of the graph on which the propagator is based. Another extension that comes for free is the promotion of the propagator to a distributed status. Different shortest path propagators running on different machines may be working together on the same graph.
We identify, at least, two scenarios where the presented propagator may be of great utility. One is for solving TSP derived problems using a hierarchical approach. [4], for instance, considers cases where the graph on which the problem is based is explored by demand. The other scenario has to do with the implementation of propagators for the same kind of problem which duty is to prune non-viable edges. If the cost of the nodes represent time, those nodes may be associated with time windows (i.e., the node can only be visited within some time periods). [3] suggests, for instance, the use of shortest path propagators for inferring nodes that have to be visited before others (due to the presence of time windows), thus avoiding failures during the search phase.

5 Acknowledgments

Special thanks are due to Kevin Glynn, who helped us to improve the quality of our paper with his valuable comments. We would also like to thank the Mozart Research group (at UCL) for their comments on the design of the shortest path propagator.

References

1. S. Haridi and N. Franzen. *Tutorial of Oz*. December 1999. Available at `http://www.mozart-oz.org/`.
2. Mozart Consortium. *The Mozart Programming System Version 1.2.5*. December 2002. Available at `http://www.mozart-oz.org/`.
3. G. Pesant, M. Gendreau, J. Potvin, and J. Rousseau. An exact constraint logic programming algorithm for the travelling salesman with time windows. *Transportation Science*, 32:12–29, 1998.
4. L. Quesada and P. Van Roy. A concurrent constraint programming approach for trajectory determination of autonomous vehicles. In *CP 2002 Proceedings*, 2002.
5. G. Ramalingam and T.W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
6. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. 2003. To be published by MIT Press. Expected publishing date 2004.
7. P. Van Roy, S. Haridi, P. Brand, M. Mehl, R. Scheidhauerand, and G. Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.
8. P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in distributed oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.
9. G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, 1999.

Game-based CSP

James Little, Eugene Freuder & Paidi Creed

Cork Constraint Computation Centre,*
Department of Computer Science,
University College Cork, Ireland
{j.little,e.freuder,p.creed}@4c.ucc.ie

Abstract. The search for a solution to a multi-criteria constraint optimisation problem can be shown to be analogous to game playing. By configuring agents to carry out game playing strategies within a constraint based search, gives a novel way of reaching solutions. In this paper, we describe how a constraint optimisation problem can be viewed as a game. For each formulation of a constraint problem as a game, the quality of solution depends on the gaming strategies employed by each player. We show that even when criteria are difficult to measure consistently, good balanced solutions can still be obtained using a heuristic approach.

1 Introduction

Constraint Optimisation Problems (COP) frequently require that a set of objectives are solved over a set of constraints. While each objective on its own is quantifiable (at least in terms of measuring how good a particular solution is), it is not apparent how they may be expressed as a single optimisation function or even how these different objectives interact. The focus of this paper is on finding balanced solutions, satisfying to some extent all the criteria as much as possible. The approach being considered here is to develop a multi-criteria heuristic based on AI game playing which will take us to a good solution. We will construct a set of games around a multi-criteria constraint problem and solve the problem by configuring players to play the game. In other words, we have a game master who configures a game and controls all the players' behaviour in such a way to realise the game objective of finding good balanced solutions. The belief is that if players are made to play in a selfish manner, making their best moves at each turn, then the solution will be close to the game objective.

As an experimental evaluation of the approach, we start by finding all the non-dominated solutions or "Pareto" optimal solutions to the problem. These are the solutions for which no other solution is better across all criteria. These solutions form a frontier on which all non-dominated solutions lie. By determining this frontier, we can see which solution is most balanced. For each criterion there is a solution which maximises that aspect. For every other solution, it will

* This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075.

satisfy this criterion to some extent. We define a "balanced" Pareto solution to be one which satisfies each criterion equally. Our approach is to develop a heuristic, based on well-known game playing strategies, which will get us close to the most balanced non-dominated solution. The evaluation is therefore how close we get.

A variety of ways of handling multi-criteria optimisation within COP's have already been proposed. Gavanelli[3] takes the approach of trying to find the complete non-dominant frontier through an efficient implementation based on Point Quad trees. He reports good performance with respect to the other algorithms in this area and shows results on problems with up to 4 different criteria. Hude et al[4] recognise that each optimisation criterion should be handled independently and within the search, through criterion-specific strategies and a 'criterion choice heuristic'. A mono-strategy is chosen and used through to the first solution. An alternative strategy is then chosen to continue searching for improved solutions. Focacci and Godard[1] propose introducing bounding constraints, relating to each criterion, on finding a solution. However, guaranteeing completeness is difficult as other non-dominated solutions can be ignored. Their iterative approach performed well against other standard approaches on a set of job shop problems. O'Sullivan[8] has also considered Pareto optimal solutions when looking at different design schemes represented as CSP's.

Many of the above approaches to multi-criteria optimisation rely on the user being able to rank the criteria or provide a relative way of measuring each criterion. Our approach does not assume such degree of precision by the user. Instead our initial aim is to provide a good, "balanced" solution for the user.

In the application of game playing to COP's Oon[7] has proposed using it to solve resource scheduling problems. These problems exhibit "dissimilar objective functions". Here the approach is based on a complex system of bidding, arbitration and negotiation using techniques from multi-agent systems, collaborative game theory and expert systems. Players can also be thought of as agents and in the paper by Freuder and Eaton[2] they configure teams of agents with individual interests to search through a set of constrained variables making assignments. They focus on techniques of compromise between conflicting interests, but do not incorporate any game-playing techniques. Although there are aspects of each of these approaches similar and even complementary to our own, this is a new approach using game playing strategies for multi-criteria COP's.

Game playing has been incorporated into constraint programming (CP) in other ways. Ricci[9] suggests that some problems are better solved by agents, strategies and co-operation. To this end, every CP variable can be thought of as a player, whose strategy is determined by its domain. However, a game may end in a non-feasible CSP solution. Solutions of the CSP problem are proven to correspond to Nash[6] Equilibrium points, since they cannot be improved upon (in terms of number of satisfied constraints). Elsewhere, Kolaitis and Vardi[5] show how underlying constraint consistency techniques behave like certain games.

In Section 2 of this paper, we describe the basics of AI game playing and show in Section 3 how playing a particular type of modified game is analogous

to searching for a solution to a COP, based on constraint programming. Further, we show how the different types of playing strategies correspond to different approaches to solving a multi-objective COP. In Section 4, we present a selection of games based on graph colouring, but with multi-criteria objectives added. We carry out different game playing strategies across these games and compare the resulting payoffs against exhaustive searches, in an attempt to assess the quality of solutions found. Section 5 provides some initial conclusions to this approach and speculates on future research direction suggested by the findings.

2 Game Playing

A game is made up of an initial position, a set of operators which define the legal moves between positions, and a payoff function which gives a numeric value to a player at the outcome of the game (terminal position). Players take turns to make moves to advance the position/state of a game to some conclusion, where no further moves are possible. At that point, each player has an outcome or payoff. The game finishes in a finite number of moves no matter how it is played. Within AI game playing, a game can be described as kind of search (Russell and Norvig[10]) through a space of possible game positions. Each player plays the game with some objective in mind, so when the game achieves a terminal state, they can calculate their payoff in terms of that objective. Each player has an associated behaviour to decide, on their turn, which move to make within the framework of legal moves. The player makes this decision, trying to direct the game towards a favourable outcome for themselves i.e. improving their own objectives. This decision may take into account their own objectives, the objectives of the other players and the state of the game at various possible positions ahead.

For small games, it is possible to evaluate all possible outcomes and from that to determine logically the best moves for each player to make. The analysis of such type of games is at the heart of Game Theory [11]. For larger games such as chess it is not always possible to do so. AI game playing seeks to estimate how to play a game without full information; depending instead on knowledge of the game, to develop good search strategies. This knowledge of the game involves evaluating the expected payoff for a player in reaching any given position, looking a few moves ahead. Obviously, this calculation is imprecise, but extremely important for the performance of the player. Once these estimates have been made, the player can use them within their gaming strategy to decide the best move to make.

One popular strategy which has found an amount of success in zero-sum games is the 'minimax' strategy[10] based on Von Neuman and Morgenstern's[11] theorem. Zero-sum games are those in which any improvement in one player's payoff is directly reflected in the same amount of deterioration in the total payoffs of the other players. Therefore, since each player starts the game with zero payoff, the sum of payoffs at the end of the game will also equate to zero. General sum games on the other hand imply the opposite; in them, one player's gain does

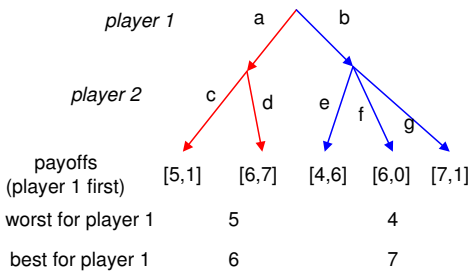


Fig. 1. *Game Tree Showing Different Payoff Possibilities*

not have an equal and opposite reflection in the others. Players can therefore be seen as co-operating, since it is possible to improve both their payoffs in one move. The 'minimax' strategy tries to minimise the worst possible outcome for a player. In general sum games, which are considered here, other strategies derived from the 'minimax' are also considered, see section 3.3.

In Figure 1, a partial game tree is shown. Player 1 can either make move a or b, followed by player 2 making move c,d,e,f or g. Several positions are then generated based on different combinations. At these positions, evaluation functions are run to determine the payoffs for each player. A higher score indicates that this is a preferable position for the player. What is a good outcome for player 1 is not necessarily good for player 2 and vice versa. In this example, if player 1 chooses move a, they will at worst get to a position with value 5, irrespective of what player 2 does. However, if player 1 had chosen move b, then there was a chance of being in a stronger position with a payoff of 7. However, there was also the possibility of being in a position of value 4. Using a maximin strategy, would result in player 1 choosing the route which leads to the best of worst outcomes i.e. the move corresponding to the left-hand branch.

3 Solving a Constraint Optimisation Problem as a Game

3.1 Structure of the Game

The game starts with a set of variables and for each member, a domain of possible values. Players take turns to choose a variable(s) and assign a value to it. All the variables must be assigned values, consistent with the constraints, for the game to terminate. The rules of the game are represented by the constraints on the variables, which prevent certain combinations of variable/values being chosen as the game proceeds. Each player has their own objective, reflected in strategies for assigning values to variables. A position in the game corresponds to some of the variables assigned values consistent with the constraints. To advance the game one position requires some of the unassigned variables to be given values. Due to constraint propagation taking place the set of possible moves becomes restricted during the game. This move and propagation creates the familiar constraint

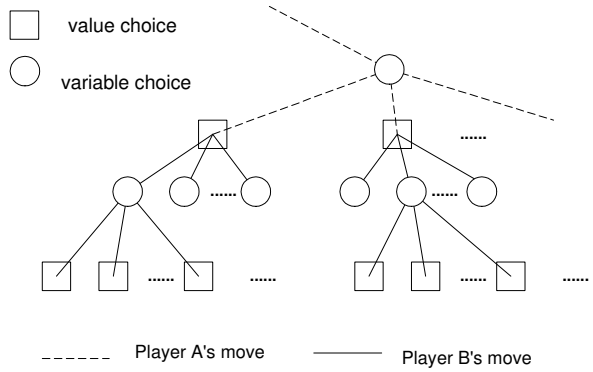


Fig. 2. *CP Search Tree as a Game*

based search. As a game, we could expect it to terminate without necessarily assigning all variables with values and still be able to calculate a payoff for each player. This cannot be the case here, as we require that all the variables have values satisfying the constraints. Therefore we need to modify the game to include backtracking and allow players to take a turn again, but this time choosing another move. The circumstance necessitating backtracking happens when a game state contains variables with no possible values. At a terminal position, each player calculates their payoff and this indicates how well they have satisfied their objectives. How each player plays the game is likely to have a bearing on these eventual payoffs.

At this point, it is apparent that many possible types of game can be designed and played, based on assigning values to variables. However, the game proposed here is a two-person, general sum game with each player choosing one variable and value in turn. Each player has an objective which matches one criterion of a 2-criteria optimisation problem. Each is also configured to play the game in a manner consistent with trying to meet their objective. The playing behaviours are under the control of the game master and each player knows what the others players' objectives are. In Figure 2, we illustrate a search tree as a game tree, where the development of the search is equivalent to the moves in a game.

3.2 The Player's Behaviour

Once a player has been given an objective there are many ways of configuring the player to take part in the game. A player's behaviour is characterised by two aspects; their evaluation function and their strategy around applying this function. When a player is deciding the next move to make, they need to look ahead and evaluate positions they could get to. Obviously they cannot look all the way to the end of the game, as this would in effect enumerating all possible outcomes, which is not the approach we are proposing. However, by advancing a certain number of moves, the player can assess how good a position would be

in terms of likely payoff. For this, players use an evaluation function to give a value to every considered position. To reach a position of depth greater than one, will require moves to be made by the other player. An evaluation of a position allows us to approximate the likely payoff in getting to that position without doing a complete search of the game tree. From this, the player would then assemble the results of these evaluations of each position considered, and decide on the next move to make, taking them hopefully nearer to one of those positions. However the player cannot expect to necessarily get to that position, as it is dependent on the moves of the other player. Choosing which positions to evaluate and subsequently how to process the evaluation results determines the player's strategy which in turn describes a style of playing.

3.3 Playing the Game

In this game, each player in turn chooses a variable and assigns a value to it. For these experiments we limit the number of moves ahead which are evaluated to a depth of two. After each move, arc-consistency is applied to ensure that the move is legal, before it is evaluated. In other words, player 1 will consider first all their possible moves and extend this to all those following on by player 2. At these positions the evaluation functions for player 1 and optionally player 2 are applied. All the results are passed back to player 1 and according to their strategy a move is made and variable is assigned a value. It is now player 2's turn and in the same way, player 2 will look a further two moves ahead and evaluate the valid positions. Then according to player 2's strategy a choice of move will be made. The game finishes when all the variables have been given values. At any time it may not be possible to make any of the positions consistent for a player. In this case, we will go back to the most recent move made by the other player and choose the next best move.

The order in which the variables and values are considered in searching the game tree is important. Currently we examine all combinations of variables and permissable values in the order in which they are input. However, when ties occur in the evaluations, it is the path of the variable and the value earliest in the input order which will be chosen.

In any game, players can be given whatever game-playing strategies the game master chooses, to play the game. Some common strategies used in the experiments are described in Table 1 along with an interpretation on the style of playing. In these set of initial experiments we shall consider both players adopting the same characteristics.

The minimax, maximin and maximax strategies are all derived from the traditional 'minimax' strategy used in AI game playing. This approach is suited to pure adversarial games where one player's gain is the other player's loss, since it splits the game into MAX and MIN levels, where the evaluation is maximised at one and minimised at the other [10]. Since our's is general sum game where the objectives (and from them the evaluation functions) are not necessarily opposed, we consider three different combinations of the MAX MIN levels. All these strategies reduce to the same when used in a zero-sum game.

Strategy	Description of Strategy	Playing Style
Depth 2 + maximin	Maximise the worst outcome from each of your payoffs	"conservative/ secure"
Depth 2 + minimax	Minimise the best outcome of the other player from each of your decisions	"opposing"
Depth 2 + maximax	Maximise your and the other player's expected outcome	"expectant"
Depth 2 + max weighted sum	Maximise the weighted sum of outcomes in a probabilistic way	"realistic"

Table 1. *CP Strategy Description*

Maximin assumes that the other player's move might minimise our score, so minimise on levels controlled by the opponent and maximises on levels under the current player's control. This will in effect maximise the minimum score. The game is split into MIN and MAX levels where we return the maximum score for the current player at the levels under their control and the minimum score for the current player at the levels under the opponent's control.

Minimax is a style to minimise the opponent's score no matter what the situation. It operates under the assumption that since the objectives are different, a move that is good for my opponent is not good for me. The game is split into MAX and MIN levels again, but in this case the opponent's score is maximised at the levels under their control and minimised at levels under the current player's control. This approach is fine for adversarial zero-sum games, although the games described here are less so.

Maximax takes both players' objectives into account. The game is split into MAX1 and MAX2 levels where we return the maximum score for player 1 on level MAX1 and player2 on MAX2. Since the strategies are not completely opposed, this strategy can lead to situations where both players get a better score that other strategies would have missed.

A fourth strategy called Max Weighted Sum is also considered. In every game tree there is a degree of uncertainty since the other player will always look further ahead. We have designed a strategy that instead of focusing on a single move at each level, takes all the moves into account. At levels under the current player's control we return the maximum score but at levels under the opponent's control we calculate the payoffs to both players and return the sum of each of our payoffs weighted by the probability that the opponent will make that move. The probability for each move is calculated by dividing the opponents payoff for that move by the sum of their payoffs for all moves at that level. In Figure 3 we give examples of how each of the strategies behaves with a given set of evaluation values. With these particular payoffs, a variety of different moves are recommended, indicated by the circled choice.

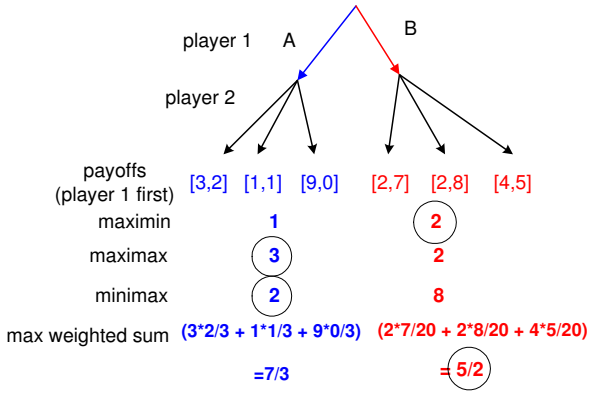


Fig. 3. *Examples of Different Strategies on the Choice of Move*

4 Experiments and Results

4.1 The Games

The experiments were carried out across a set of graph colouring problems randomly generated and modified to have a multi-criteria objective. A graph colouring problem consists of a set of nodes some of which are connected by links. Each node has to be coloured by one of four colours (red, blue, green and yellow) and those nodes which are linked cannot have the same colour. The problems considered, range in size from 4-node up to 15-node. Given these constraints and the overall objective of colouring all the nodes, three games have been designed around this problem. These are described below.

Game 1: Player 1 tries to maximise the number of nodes coloured red, while player 2 tries to maximise the number of nodes coloured blue.
Observations: this is a medium competitive game. Maximising on one colour does not directly mean that the other colour is minimised. Yet both players may compete on certain nodes to place their colour.

Game 2: Player 1 tries to maximise the number of nodes coloured red. For player 2 there is a benefit value associated with every node / colour combination. Player 2 tries to maximise the sum of these values.
Observations: This game is not directly competitive. There are situations where red provides greatest benefit to player 2 also. However there are times when red gives the smallest benefit. The numerical scale of the two objectives are different; each red gets a score of 1, while the benefit on each node ranges randomly from 0 to 3.

Game 3: Player 1 tries to maximise the number of nodes coloured red. Player 2 tries to maximise the number of nodes coloured red AND blue.
Observations: This game is the least competitive one. The objectives are different, but the players share one common goal.

4.2 Evaluation Functions

We use a different evaluation function for each objective. When the objective is counting the numbers of red or/and blue, then the evaluation function at a position is as follows. A score of 2 is given for every variable assigned to that colour or through propagation during the incremental moves made to that position. A score of 1 is given to any unassigned variable which has that colour still in its domain. The evaluation function will sum all the scores at that position. A 'good' position is therefore one in which the colour has been assigned to some variables in getting to that position and there is strong indication that there is plenty of opportunity for more to be assigned in the remainder of the game.

When the objective is to maximise the benefit, the evaluation function behaves as follows. The benefit scores are summed for the assigned variables (and those assigned through propagation)in making the two moves. This is then added to the mean of the maximum benefit values of the unassigned variables.

4.3 Experiments

For each game type, we consider different combinations of 2-player strategies. In all these experiments,we consider both players applying the same strategy with appropriate evaluation function. The reason for this is initially to try to have each player playing in a 'balanced' manner. We are also interested in recording the order in which the players make the first move to see whether this influences the outcome. On the non-dominated frontier there is a solution which is most balanced. The definition of the most balanced solution on the non-dominated frontier for a two criteria problem, is the one for which the difference between the percentage satisfaction of each criterion is smallest. As an example of determining the most balanced solution, consider that for criterion 1, the maximum achievable payoff is 100 and for criterion 2, it is 200. A solution on the frontier with payoffs of 100/100 would satisfy the criteria 100% and 50% respectively. Consequently, the measure of how balanced this solution is 100 - 50 = 50. However, a solution of 75/150 would satisfy each criterion by 75% and result in a balance factor of 75 - 75 = 0. In cases where two solutions give the same balance factor e.g. a solution of 50/100, then the sum of their percentage satisfactions is taken to determine the best. The non-dominated frontier is obtained for these experiments by exhaustive search. To evaluate the results we compare them against the non-dominated frontier and the most balanced solution on that frontier.

4.4 The Results

The results of Games 1, 2 and 3 are presented in Tables 2, 3 and 4 respectively.

No of Nodes	Maximin reds vs blues	Minimax reds vs blues	Maximax reds vs blues	W'ghted sum reds vs blues	No of reds / No of blues
4	(2,1)	(2,1)	*(3,1)**	(2,1)	3/1
5	(2,2)	(1,2)	*(3,2)**	(2,2)	3/2
6	*(2,2)**	(1,2)	*(2,2)**	*(2,2)**	3/1, 2/2
7	*(3,2)**	(1,1)	*(3,2)**	*(3,2)**	3/2
8	(3,2)	(2,2)	*(4,2)**	(3,2)	4/2
9	(2,3)	(2,2)	*(3,3)*	*(3,3)*	3/4
10	(3,3)	(2,2)	*(4,3)**	(3,3)	3/4
11	(5,2)	(2,1)	*(3,5)+*	*(5,3)+*	3/5, 4/4
12	*(4,5)**	(2,3)	(3,3)	*(5,4)**	4/5
13	*(6,4)*	(3,1)	*(6,4)*	(5,4)	7/3, 6/4, 5/5
14	*(4,4)*	(4,2)	*(4,4)*	*(4,4)*	5/4
15	*(5,5)**	(4,3)	(3,5)	*(5,5)**	5/5

Table 2. *Results for Game 1*

* represents the most balanced solution on the non-dominated frontier
\+ represents a solution on the non-dominated frontier
italics represents a non-dominated solution among those found

No of Nodes	Maximin reds vs prefs	Maximin prefs vs reds	Minimax reds vs prefs	Minimax prefs vs reds	Maximax reds vs prefs	Maximax prefs vs reds	W'ghted sum reds vs prefs	W'ghted sum prefs vs reds	No of reds/ sum of preferences
4	*(3,7)**	*(9,2)+*	(1,4)	(3,1)	(3,6)	*(9,2)+*	*(3,7)**	(8,2)	2/9, 3/7
5	*(3,10)**	*(12,2)+*	(1,4)	(5,2)	(2,8)	(10,2)	(2,11)	*(12,2)+*	1/13, 2/12, 3/10
6	*(3,10)**	*(10,3)**	(2,6)	(5,2)	*(3,10)**	(9,2)	*(3,10)**	*(10,3)**	2/12, 3/10
7	(3,11)	*(14,3)*	(1,11)	(7,1)	*(3,14)*	(12,3)	(3,13)	(12,3)	2/16, 3/15
8	(3,17)	(15,3)	(2,12)	(14,2)	*(4,16)**	(17,2)	(3,17)	(18,2)	2/20, 3/19, 4/16
9	(3,18)	(17,2)	(2,13)	(15,2)	(3,21)	*(23,3)**	(3,14)	(21,3)	3/23, 4/13
10	(4,20)	(23,3)	(2,11)	(12,2)	(4,19)	(19,3)	*(4,25)**	(20,4)	3/26, 4/25
11	*(4,25)*	(24,4)	(2,14)	(14,2)	*(5,22)*	(20,3)	(4,23)	*(25,4)*	2/28, 3/27, 4/25, 5/23
12	(4,18)	(20,4)	(2,15)	(18,2)	(4,21)	(19,4)	(4,22)	*(23,4)**	4/23, 5/15
13	(6,23)	(27,4)	(1,22)	(17,2)	(6,21)	*(28,5)*	*(7,27)**	(25,6)	2/31, 3/30, 6/28, 7/27
14	(4,31)	(30,5)	(2,23)	(17,3)	(4,24)	*(33,5)*	(4,31)	(28,3)	3/37, 4/36, 5/35
15	(5,27)	*(28,5)**	(4,9)	(21,3)	(4,17)	(26,5)	(4,26)	(22,4)	5/28

Table 3. *Results for Game 2*

No of Nodes	Maximin reds vs (reds/ blues)	Maximin (reds/ blues) vs reds	Minimax reds vs (reds/ blues)	Minimax (reds/ blues) vs reds	Maximax reds vs (reds/ blues)	Maximax (reds/ blues) vs reds	W'ghted sum reds vs (reds/ blues)	W'ghted sum (reds/ blues) vs reds	No of reds/No of reds + No of blues
4	*(3,4)**	*(4,3)**	(1,2)	(3,1)	*(3,4)**	*(4,3)**	*(3,4)**	*(4,3)**	3/4
5	(2,4)	(4,2)	(1,3)	(2,1)	*(3,5)**	*(5,3)**	*(3,5)**	*(5,3)**	3/5
6	*(3,4)**	*(4,3)**	(1,3)	(2,1)	(2,4)	*(4,3)**	*(3,4)**	(4,2)	3/4
7	*(3,5)**	*(5,3)**	(2,3)	(2,1)	*(3,5)**	(4,3)	*(3,5)**	(5,2)	3/5
8	*(4,6)**	*(6,4)**	(2,4)	(4,2)	(4,5)	(5,4)	*(4,6)**	(5,3)	4/6
9	(2,5)	*(7,4)**	(1,4)	(4,2)	*(4,7)**	(6,4)	(4,6)	(6,3)	4/7
10	(3,6)	(6,3)	(2,4)	(4,2)	(4,6)	*(7,4)**	*(4,7)**	(6,3)	4/7
11	(5,7)	*(8,5)**	(2,4)	(3,1)	(5,7)	(7,5)	*(5,8)**	(8,4)	5/8
12	(4,7)	(7,4)	(2,6)	(6,3)	*(5,9)**	(7,4)	(5,8)	(7,4)	5/9
13	*(7,10)**	(9,7)	(1,5)	(4,1)	(7,9)	*(10,7)**	*(7,10)**	(10,5)	7/10
14	*(5,9)**	*(9,5)**	(2,6)	(6,2)	(4,9)	(8,5)	(5,8)	(8,4)	5/9
15	(4,9)	(8,4)	(3,7)	(7,3)	*(5,10)**	*(10,5)**	*(5,10)**	(9,4)	5/10

Table 4. *Results of Game 3*

Across all the games, the strategies of maximin, maximax and weighted sum all record instances of finding the most balanced non-dominated solution. The minimax strategy however is not successful in finding good balanced solutions.

In game 1, the most balanced non-dominated solution is found by one or more of the strategies in 9 out of the 12 cases. On the remaining 3 cases the best solutions found were at most two units away from the best solution (some were also on the non-dominated frontier). The combined maximax strategies solved the majority of cases to find the most balanced solution(6) or to the non-dominated frontier(2). All these solutions are symmetrical that by swapping red for blue we get an equivalent solution. In game 2, the same pattern emerged as in game 1, with three strategies finding a majority of the most balanced non-dominated solutions(9) while the other best solutions, to the remaining problems were no more than 2 units away. Game 2 also seems to have fewer instances where the best solution is found. This game differs from the others in how the multi-criteria are measured on a different scale. Game 3 provided the only case where all the most balanced solution on the non-dominated frontier were found, by one of the gaming strategies. The weighted sum strategy provided most instances of finding the most balanced solution.

The order in which the game is played does result in different outcomes, but it is not apparent whether it is better to move first or second to get the best payoff for these types of games.

5 Conclusions and Future Work

We have shown that configuring simple games of two players with limited look-ahead, simple evaluation functions and strict input order of variables and values,

we are able to get good balanced solutions using a heuristic based on game playing over a set of small competitive games. The incorporation of game playing strategies into a CP search does provide an alternative way of looking at the issue of multi-criteria optimisation and of achieving good, balanced results. We found the most balanced solution on the non-dominated frontier in the majority of cases. In fact for game 3, all the best solutions were found, although not necessarily by the same game strategies. Where the best solution found, was not that of the most balanced on the non-dominated frontier, the difference was only at most 2 units. Only the minimax strategy did not return good results. This strategy focusses on the 'opponent' player and in these general-sum games it does not give a good indication of the outcome for the original player. In addition, it is not in the interest of the overall game to try to reduce one player's score. At this stage it is not apparent which of the three combinations is superior.

The advantage of this heuristic approach is that it is not necessary to search along the non-dominated frontier for the best solution. However, the heuristic does require an amount of processing. On larger problems, if more positions are evaluated, then time is likely to become an issue. There are techniques within game playing such as alpha-beta pruning [10] which help to reduce the search.

Acknowledgements
We would like to thank Dr Ken Brown for the useful discussions and suggestions he has provided us with.

References

1. Focacci, F. and D. Godard, A Practical Approach to Multi-Criteria Optimization Problems in Constraint Programming, *Proceedings of CPAIOR'02.*
2. Freuder, E. and P. S. Eaton, Compromise Strategies for Constraint Agents, Constraints and Agents,*Papers from AAAI Workshop 1997*, Technical Report WS-97-05.
3. Gavanelli, M., An Algorithm for Multi-Criteri Optimisation in CSPs, *Proceeding of ECAI 2002*, F. van Harmelen (ed.) IOS Press, pp136-140, 2002.
4. Le Hud, F., M. Grabisch, C. Labreuche and P. Savant, Multicriteria Search in Constraint Programming, *Proceedings CPAIOR'03.*
5. Kolaitis, P.G. and M.Y. Vardi, A Game-Theoretic Approach to Constraint Satisfaction, *Proceedings of AAAI 2000.*
6. Nash, J, Non-cooperative Games,*Annals of Mathematics*, 54:286-295, 1951.
7. Oon, W and A. Lim, Multi-Player Game Approach to Scheduling Problems, *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN02).*
8. O'Sullivan, B, Constraint-Aided Conceptual Design, PhD Thesis, Department of Computer Science, University College Cork, July, 1999.
9. Ricci, F, Equilibrium Theory and Constraint Networks, *Proceedings of Constraint Directed Reasoning Workshop*, AAAI-90, Boston 1990, and *Proceedings of the International Conference on Game Theory*, Florence, 1991.
10. Russell, S. J. and P. Norvig, *Artificial Intelligence A Modern Approach*, Prentice Hall, 1995.
11. Von Neumann, J. and O. Morgenstern, *Theory of Games and Economic Behavior*, Princeton University Press, 1944.

Implementing Constraint Imperative Languages with Higher-order Functions

Martin Grabmüller

magr@cs.tu-berlin.de
Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Franklinstr. 28/29, 10587 Berlin, Germany

Abstract. Constraint imperative programming languages combine declarative constraints and imperative language features into an integrated programming language. The language Turtle supports these programming paradigms and additionally integrates functional programming with higher-order functions and algebraic data types. This paper describes the implementation of Turtle, consisting of a compiler, a run-time system including constraint solvers and an extensive library of supporting modules.

1 Introduction

Declarative programming languages let the programmer concentrate on *what* the solution to a problem is, by specifying the properties the solution should have and letting the programming system find it. Imperative programming, on the other hand, emphasizes *how* to calculate the solution. An imperative program contains a step-wise description of the solution algorithm which finally leads to the desired result. Each of these programming paradigms has its respective advantages. Declarative programming builds on a strong mathematical foundation which simplifies program transformations (e.g. optimization) and verification. Imperative programs can often model real-world activities more naturally, because of their state-changing semantics. The available implementations of imperative languages also yield more efficient (in time and space) programs, despite the developments in compiler technology over the last decades.

Several research activities have tried to integrate declarative and imperative programming languages in order to combine the advantages of both. Constraint-imperative programming [1] is one instance of this combination. It integrates declarative constraints and imperative language constructs such as mutable data structures and assignment. Besides novel language design issues, constraint imperative languages require new techniques for an efficient implementation.

This paper reports on the implementation of the constraint imperative programming language Turtle [2], which integrates constraints, imperative constructs and features mainly known from functional programming languages, such as higher-order functions and algebraic data types.

This paper is organized as follows. Section 2 briefly describes the language Turtle. The implementation of Turtle is presented in Sect. 3. It consists of a description of the compiler, the run-time system and the Turtle library. Section 4 relates this paper to other work and finally concludes.

2 The Programming Language Turtle

This section gives a short survey on the programming language Turtle [2]. The language was designed by starting with an imperative base language with higher-order functions and a rich type system. Then, several language extensions for constraint programming were added. We will first describe the imperative and functional language constructs and then discuss the constraint programming features.

Imperative programming. Turtle provides all control structures known from traditional imperative languages: conditionals, loops, functions (and procedures) and assignment statements. Variables and data structures can be modified by assignment, and input/output is performed using side-effecting functions. Turtle supports a rich set of data types, including integers, reals, booleans, strings, characters, arrays, lists and tuples. Turtle also has a module system for encapsulating the declaration of functions, variables and data types using explicit import/export relations between modules. Modules can be parametrized by data types and functions can be defined in terms of these parameters, resulting in polymorphic functions. The Turtle implementation comes with a set of library modules which make extensive use of this feature, e.g. for providing functions to handle lists of arbitrary element types.

Functional programming. Higher-order functions, which can receive functions as parameters and can return functions as their value, are provided for functional programming. The supported data types also include algebraic data types as known from functional programming languages. In addition to imperative loops, iteration can also be expressed by recursion, as is normally done in functional languages. Turtle uses eager evaluation semantics to avoid conflicts with side-effects introduced by imperative programming.

Constraint programming. Four extensions were added to the imperative and functional base language: constrainable variables, constraint statements, user-defined constraints and constraint solvers.

Constrainable variables are special variables, introduced by data type annotations, e.g. a constrainable integer variable is declared with type **!** *int*. The values of normal variables are given by assignments, whereas the values of constrainable variables are determined by placing constraints on them. Since constrainable variables not only hold values but also need to store additional information for use with the constraint solvers, they are actually represented by *variable objects*,

which are explicitly created and must be dereferenced to obtain the variables' values.

Constraint statements are block-structured statements which consist of (1) a constraint conjunction and (2) of a sequence of statements, called the body. The following example shows a constraint statement constraining a variable x to a value greater than zero as long as the body (which prints the value of x) is running.

require $x > 0$ **in** io.put ($!x$); **end**;

When a constraint statement starts to execute, the constraints in the constraint conjunction are added to the constraint store and the built-in constraint solver tries to satisfy the constraints by assigning suitable values to the constrainable variables appearing in the constraints. Since a constrainable variable can only hold a single value, an arbitrary value which satisfies the constraints is chosen. When the solving process is successful, the statements in the body are executed. The variables remain bound to their values during the execution of the body, and the constraints are removed from the constraint store when the statement is left. If the constraint solver detects that the constraints are not satisfiable, an exception is raised which must either be handled by the program or otherwise terminates execution. A second variant of the constraint statement without a body is also provided. Constraints specified with such a statement remain valid as long as the variables in the constraint do exist.

User-defined constraints abstract over constraints similar to functions, which abstract over individual expressions or statements. User-defined constraints can contain arbitrary statements, but their main purpose is to place constraints on one or more of their parameters. When a user-defined constraint invocation appears in a constraint conjunction, its body is executed.

Constraint solvers are built into the run-time system of Turtle and are responsible for maintaining their associated constraint stores. Whenever constraints are added to the store, the solvers must satisfy their stores by calculating assignments for the constrainable variables. When the constraints in the stores are not satisfied, the solvers are responsible for raising an exception. Constraint statements in Turtle allow to specify the importance of individual constraints in a conjunction by so-called *strength annotations*. The constraint solvers will try to satisfy the most important constraints, even if that means that less important ones will be violated. This treatment of preferential constraints is called *constraint hierarchies* [3].

Currently, only linear constraints are supported by Turtle. Lifting this restriction requires modifications both to the solvers and the compiler, because constraints are analyzed at compile time.

The code fragment in Fig. 1 illustrates the constraint extensions of Turtle. The user-defined constraint *all_different* receives a list of constrainable variables and places inequality constraints on each pair of list elements. Line 11 declares three constrainable variables a, b and c and initializes the variables with variable objects holding different values. Line 12 invokes the user-defined constraint in

```
1   constraint all_different (l: list of !int)
2     while (tl l <> null) do
3       var ll: list of !int := tl l;
4       while (ll <> null) do
5         require hd l <> hd ll;
6         ll := tl ll;
7       end;
8       l := tl l;
9     end;
10  end;
    ...
11  var a: !int := var 0, b: !int := var 1, c: !int := var 2;
12  require all_different ([a, b, c]) in ... end;
```

Fig. 1. Constraint imperative example

a constraint statement and thereby ensures that the variables' values remain pairwise different while the body executes.

3 Implementation

The implementation of the Turtle system consists of a compiler, a run-time system and of a collection of library modules. The run-time system contains two experimental constraint solvers and a garbage collector and the library provides useful utility functions and abstract data structures. This section describes each of the parts of the Turtle programming system.

3.1 Compiler

The compiler translates Turtle source code into object code. We will briefly describe the general structure of the compiler and then present the handling of constraints and functional programming features in more detail. The compilation of the imperative base language into machine code will not be shown, because it is rather standard.

Compiler structure. Turtle source programs are first parsed and converted to a syntax tree, annotated with type information and fully resolved identifiers. This intermediate program representation is called high-level intermediate language (HIL). The HIL representation is converted to a low-level intermediate language (LIL) suitable for machine code generation. LIL is the language of a stack machine, designed for easy code generation and target language independence. The code emitter finally converts LIL to the target language, which in the current implementation is ANSI C. The compilation into machine code and the linking of the program modules with the run-time library is handled by a standard C compiler and object-code linker.

Compiling constraints. The handling of constraints requires the compilation of user-defined constraints and constraint statements. Constrainable variables only affect the type-checking and the compilation of the creation of variable objects and accesses to their contents. The creation of variable objects is implemented like the creation of user-defined data types and accesses are translated to simple fetch instructions.

User-defined constraints are compiled in the same way as normal functions, except that constraint statements contained in their bodies do not need to invoke the constraint solvers to check their stores for satisfiability. This is because user-defined constraints can only be invoked from constraint statements, which will do this as soon as their statement bodies are entered.

Constraints are specified in constraint statements in TURTLE. The architecture of the system is flexible enough to integrate new constraint solvers, so it is not in advance known how constraint solvers have to handle constraints in order to efficiently solve them. Therefore, a solver-independent constraint representation is built at run-time, whenever a constraint statement requires that a constraint is added to the constraint store.

The compiler distinguishes two kinds of constraints in constraint statements: first, we have trivial constraints which do not contain any references to constrainable variables. Second, there are non-trivial constraints. Trivial constraints are translated like normal boolean expressions, followed by a test whether the result was true or false. If the result was false and the constraint was required, an exception is raised. Such constraints may appear when constraint statements are used for stating program invariants instead of using them for calculating assignments for constrainable variables. For non-trivial constraints, compilation is more complicated. Since constraints are first-class objects (they have to remain accessible to the constraint solver until the scope of the constraints is left), a representation of the constraints is built. This representation must contain references to the constrainable variables so that the constraint solver can fetch the values of these variables and can store new values into them. References to normal variables, function calls and constant subexpressions are treated as constants, so that only the results of evaluating them have to be stored in the constraint representation. This is done by evaluating these expressions as soon as the containing constraint statement is entered, and by remembering the result for inclusion in the symbolic representation. After building the representation, the constraint is tagged with its strength (0 for required constraints, and values greater than 0 for preferential (non-required) constraints) and added to the constraint store. Adding the constraint will cause the constraint solver to re-solve the store. If any required constraints in the store cannot be satisfied after adding the new constraint, the new constraint will be removed and an exception will be raised. If any preferential constraints are not satisfied, the solver tries to satisfy as many preferential constraints as possible, but without raising an exception.

The actual compilation of constraint statements consists of two phases. First, the expressions of all constraints in the constraint conjunction are partitioned into constant terms on the one hand and the constrainable variables and their

```
1   var y: int ← 4;
2   var x: ! int ← var 0;
3   require 10 * x + 10 > 3 * y − 1;
```

Fig. 2. Constraint compilation example

coefficients on the other hand. In the second phase, the code for evaluating the constant terms (including function calls) is emitted as well as the code for creating the symbolic representation.

For illustration of the translation of constraint statements, we will translate the constraint statement in Fig. 2 step by step.

Line 1 declares the integer variable y. The variable x is declared in line 2 as a constrainable integer variable, so the compiler will need to translate the **require** statement as a non-trivial constraint statement. The translation of line 3 proceeds by first partitioning the terms of the constraint into constants (that includes non-constrainable variables and function calls, which are evaluated before the constraint is created) and constrainable variables, together with their coefficients. For our example, we have the constant expression

$$-10 + 3 * y - 1$$

and the constrainable variable with coefficient:

$$10 * x$$

Figure 3 shows the generated code for the example. The translation of the **require** statement consists of first pushing the constraint's strength onto the evaluation stack. For our example, since no strength was specified, 0 (the strongest strength) is assumed (line 1). After that, an indicator for the kind of constraint (the inequality '>') is pushed, followed by the number of constrainable variables, 1 in the example (lines 2–3). Then the value of the constant term (which must be evaluated before the constraint is created) is pushed onto the stack (lines 4–10), followed by all constrainable variables with their corresponding coefficients (lines 11–12). Finally, the constraint is added to the constraint store (line 13). The *add-constraint* instruction is the only instruction which interfaces to the constraint solvers (except for a *remove-constraint* instruction used at the end of constraint statement bodies) and causes the solver responsible for the constraint to build a representation from (1) the constraint strength, (2) the constraint kind, (3) the constant and (4) from the constrainable variables and their coefficients found on the stack. Whenever the solver needs to check the satisfiability of its store (because new constraints are added), it uses the internal constraint represenation it has built.

Note the difference between the variables x and y. The value of y (an integer) is used for calculating the constant term of the constraint, whereas the variable object stored in x is pushed onto the stack so that the constraint solver responsible for the constraint can access the variable.

```
 1   push-constant  0        // constraint strength
 2   push-constant  3        // constraint kind '>'
 3   push-constant  1        // number of constrainable variables
 4   push-variable  y        // calculate the constant term...
 5   load-constant  3
 6   mul
 7   push
 8   load-constant  -11
 9   add
10   push
11   push-variable  x        // load the constrainable variable object
12   push-constant  10       // load the coefficient
13   add-constraint          // add the constraint to the store
```

Fig. 3. Generated code for the constraint example

User-defined constraints, which may appear as elements of the constraint conjunction, are compiled into calls to the subroutines which are created when each user-defined constraint is compiled. Since the code for adding constraints to the store is contained in the user-defined-constraint, the call simply replaces the code emitted for primitive constraints.

Before translating the constraints into code, the compiler checks whether the constraint is representable in the symbolic representation. This check is purely syntactic. Non-linear constraints or constraints with relations not supported by the constraint solvers are rejected by the compiler.

In the current implementation the constraint solvers which are responsible for individual constraints are determined at compile time, using type information: constraints on integers are handled by the finite-domain solver and constraints on reals are passed to the Indigo solver (see Sect. 3.2). A modification to choose the solver at run-time, based on the constraint kind passed with the *add-constraint* instruction, is planned as future work.

The separation of normal and constrainable variables in TURTLE has two advantages for the implementation. First, it is very easy for the compiler to analyze constraints and to generate code for creating constraint representations. Second, because constrainable variables can only be determined by constraints, and constraints are monotonically added to the constraint store in nested constraint statements, semantic problems of assignments invalidating the constraint store are avoided. This makes reasoning about constraint imperative programs much easier.

Algebraic data types. TURTLE supports the definition of user-defined algebraic data types. These data types are declared in **datatype** declarations, like the declaration of the type *tree* in the following example:

```
datatype tree = leaf(value: int) or
                node(left: tree, right: tree, key: int);
```

```
// Constructors
fun leaf (value: int): tree
fun node (left: tree, right: tree, key: int): tree
// Discriminators
fun leaf? (t: tree): bool
fun node? (t: tree): bool
// Selectors
fun value (t: tree): int
fun left (t: tree): tree
fun right (t: tree): tree
fun key (t: tree): int
// Mutators
fun value! (t: tree, value: int): ()
fun left! (t: tree, left: tree): ()
fun right! (t: tree, right: tree): ()
fun key! (t: tree, key: int): ()
```

Fig. 4. Induced signature for the *tree* data type

Using this data type declaration, the TURTLE compiler automatically generates a set of functions for creating instances of the type, for accessing the fields and for examining the variant of a given value of the type. Figure 4 shows the names and types of these generated functions. The constructor functions receive the values which will be stored into the fields of the value as parameters and create either a leaf or a node value. The discriminator functions are used to determine the variant of a given tree value, and the selectors return the values stored in the corresponding fields. The mutators modify the fields of a structured value by storing new values into the appropriate storage locations and return the unit type (). Mutator functions have been added to allow imperative programming with data structures constructed from user-defined data types.

Tail-recursion elimination. Supporting functional programming requires a proper implementation of functional programming concepts. One of these concepts is to use recursive function calls instead of loops for expressing iteration. TURTLE supports proper tail-recursion, that means that an iterative algorithm expressed as a tail-recursive sequence of function calls uses constant space, even when more than one function is involved in the recursive call chain. This is implemented by compiling each TURTLE module into one (possibly large) C function. Calls between functions in one module can then be implemented by simple C **goto** instructions instead of C function calls. This compilation scheme solves the problem of mutually recursive function calls in one module (so-called *intra-module calls*), but not between different modules (*inter-module calls*), because TURTLE also supports separate compilation.

In order to achieve proper tail-recursive function calls, even across module boundaries, the TURTLE compiler uses a compilation technique similar to the

one used in the Gambit Scheme compiler for its C back-end. Feeley et al. [4] describe how to compile languages with higher-order functions to portable C. The solution is to add a wrapper function to the run-time system whose purpose is simply to repeatedly call the C functions into which the TURTLE modules have been compiled. The C functions take a function descriptor as an argument which tells which of the TURTLE functions represented by the C function is to be called. Whenever an inter-module call is made, the C function passes a descriptor of the function to be called back to the wrapper function.

Higher-order functions. Functions are represented as *closure objects* at runtime. A closure object contains a pointer to the code of the function and the environment in effect when the closure was created. An environment holds the values of all free variables of a function. Since the free variables of top-level functions are the global variables whose addresses are fixed, there is no need to create closures for these functions. The TURTLE implementation uses a technique also from [4] for reducing the overhead of calling functions: top-level functions are not represented directly by their machine code addresses, but by statically allocated descriptors which have the same memory layout as closure objects. This makes it possible to call all functions in the same way, while avoiding to create closure objects (which have to be copied on garbage collection, see Sect. 3.2) for top-level functions.

3.2 Run-time System

TURTLE is a high-level language, supporting constraints as well as functional programming with higher-order functions. A language implementation for such a language requires a large run-time system to handle all the low-level functionality, for example constraint solving and memory management. The TURTLE run-time system is implemented as a shared library, written in C.

A TURTLE program at runtime consists of a *code section*, a *data section*, a *stack section*, a *run-time library*, *constraint solvers*, *constraint stores* and a *heap*. The code section contains the machine code of the program and the data section holds both the global variables of the TURTLE program and of the run-time system. The stack is provided by the operating system and is used by the run-time system, by the constraint solvers and for interfacing with the operating system. The TURTLE run-time contains the code of the run-time system. The constraint solvers and the constraint stores manage the active constraints and determine the values of constrainable variables. The heap stores all dynamically allocated memory and is organized in two semi-spaces for garbage collection.

Constraint solvers. Two constraint solvers are currently implemented for TURTLE. The first is a finite-domain solver over the integers. This solver is a simple backtracking implementation without any consistency checks or other optimizations and mainly serves as a proof-of-concept for the easy integration of constraint solvers into the system. The second is a solver over cycle-free linear

equalities and inequalities over the reals. It is based on the Indigo algorithm, an interval based local propagation solver [5]. Both solvers implement the run-time/constraint solver interface described in Sect. 3.1. They receive the symbolic representations of constraints on the run-time stack and create data structures for handling them on the heap. For each constrainable variable and each constraint added through this interface, the solvers create data structures for maintaining the lower/upper bounds of the variables. The finite-domain solver uses this information for simply reducing the number of instantiations it must perform, whereas the Indigo solver uses propagation for reducing the domains of the variables.

Garbage collection. The dynamic memory of the TURTLE system is maintained automatically by the garbage collector included in the run-time system. The collector employs a simple *stop&copy* algorithm, as described by Cheney [6]. When the garbage collector is invoked, all memory cells reachable from the machine registers and the run-time stack are copied into the second half of the heap and then all cells which were not copied are reclaimed. Since these cells are not reachable anymore, they cannot affect any future computation and may therefore be recycled.

The garbage collector is also responsible for determining whether any constrainable variables have left their scopes, and to notify the constraint solvers of that fact. This is necessary, because all constraints placed on such variables need to be removed from the stores, so that they cannot influence the program execution any more.

The Trampoline. The compilation scheme described in Sect. 3.1 for functions and user-defined constraints, where each module is compiled into a single C function requires some support in the run-time system. The run-time system contains a short function (called *trampoline*) which calls the TURTLE functions requiring inter-module calls. The function is simply a loop which repeatedly calls the function contained in the global program counter. This function is also responsible for repeatedly checking whether an interrupt (user interrupt or operating system signal) has occured, and for calling the TURTLE interrupt handler.

3.3 Library Modules

A library containing often-used data structures and functions is very important if a language is intended to be used in practice. Therefore, a standard library for TURTLE has been designed and implemented in the reference implementation.

The library provides a range of useful data types, such as trees and hash tables, support modules for the built-in data types for list, array, string and number manipulation and of course functions for imperative in- and output. Additionally, some low-level library modules have been included for interfacing with the operating system, such as for process management and network programming. The library makes intensive use of TURTLE features such as the module system,

parametrized modules and higher-order functions. Thus the library implementation was very useful in debugging the compiler and testing the language design. The structuring of the library was inspired by the design of the "Bibliotheca Opalica", the standard library of the functional language Opal [7].

4 Related Work and Conclusion

This paper describes the implementation of the constraint imperative programming language Turtle. It combines well-known techniques for implementing imperative and functional languages with new compilation schemes for the constraint extensions supported by Turtle: constrainable variables, constraint statements, user-defined constraints and the interface between the user program and the constraint solvers. We will now relate our implementation to other work and draw a conclusion.

The constraint imperative programming language Kaleidoscope [8] combines object-oriented and constraint programming. The implementation of its compiler and run-time system [9,10] is based on a translation of all imperative source language constructs (assignments etc.) into primitive constraints, which are in turn handled by constraint solvers. Kaleidoscope is thus based on a constraint solving virtual machine on which imperative programming is modelled. The approach we have taken in designing and implementing Turtle is the opposite: we have started with an imperative language and added constraints on top of it. Apt et al. [11] have designed an extension of Modula-2 with non-determinism, based on backtracking. This language, called Alma-0, has been implemented, but the proposed extensions to a constraint imperative language have not [12], making it difficult to compare it to our approach. Other work comparable to ours are constraint libraries for imperative languages, such as ILOG [13] for C++ or JACK [14] for Java. Their advantage is the easy integration into existing programs written in imperative languages, but their disadvantage is the semantic gap between their constraint solving capabilities and the imperative execution model of their underlying languages.

The implementation presented in this paper has been used to implement various example programs, ranging from simple constraint imperative programs solving crypto-arithmetic puzzles to a working web server and a front-end (scanner and parser) for the Turtle language. The performance of the functional and imperative part of the language is quite satisfactory, for some simple test programs the Turtle implementation yields programs which are by a factor of 10 slower than comparable programs written in C. This overhead is partly due to the automatic memory management and to the trampoline technique necessary for tail-recursive inter-module calls, which are not available in C. The constraint part of the language was not measured against other systems, because the solvers are very weak and cannot compete with any reasonable constraint programming system.

As we have shown, many techniques for implementing imperative and functional languages can be transferred to the implementation of integrated pro-

gramming languages and seamlessly combined. The combination of imperative and functional language constructs in Turtle is already efficiently usable, and with the integration of more powerful constraint solvers, constraint imperative programming with higher-order functions will also be usable in practice.

References

1. Freeman-Benson, B.N.: Constraint Imperative Programming. PhD thesis, University of Washington, Dept. of Computer Science and Engineering (1991)
2. Grabmüller, M.: Constraint Imperative Programming. Diploma Thesis, Technische Universität Berlin (2003)
3. Borning, A., Freeman-Benson, B., Wilson, M.: Constraint hierarchies. Lisp and Symbolic Computation **5** (1992) 223–270
4. Feeley, M., Miller, J.S., Rozas, G.J., Wilson, J.A.: Compiling higher-order languages into fully tail-recursive portable C. Technical Report 1078, Département d'informatique et de recherche opérationelle, Université de Montréal (1997)
5. Borning, A., Anderson, R., Freeman-Benson, B.: The Indigo algorithm. Technical Report 96-05-01, Dept. of Computer Science and Engineering, University of Washington (1996)
6. Cheney, C.J.: A non-recursive list compaction algorithm. Communications of the ACM **13** (1970) 677–678
7. Pepper, P.: Funktionale Programmierung in OPAL, ML, HASKELL und GOFER. 2nd edn. Springer (2003)
8. Lopez, G., Freeman-Benson, B., Borning, A.: Kaleidoscope: A constraint imperative programming language. In Mayoh, B., Tyugu, E., Penjaam, J., eds.: Constraint Programming: Proc. 1993 NATO ASI Parnu, Estonia, Springer (1994) 305–321
9. Lopez, G., Freeman-Benson, B.N., Borning, A.: Implementing constraint imperative programming languages: the Kaleidoscope'93 virtual machine. In: Proceedings of the 1994 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. (1994) 259–271
10. Lopez, G.: The Design and Implementation of Kaleidoscope, A Constraint Imperative Programming Language. PhD thesis, University of Washington, Department of Computer Science and Engineering (1997)
11. Apt, K.R., Brunekreef, J., Partington, V., Schaerf, A.: Alma-0: An imperative language that supports declarative programming. ACM Toplas **20** (1998) 1014–1066
12. Apt, K.R., Schaerf, A.: The Alma project, or how first order logic can help us in imperative programming. In: Correct System Design. Number 1710 in LNCS, Springer (1999) 89–113
13. Puget, J.F.: A C++ Implementation of CLP. In: Proceedings of the Second Singapore International Conference on Intelligent Systems, Singapore (1994)
14. Abdennadher, S., Krämer, E., Saft, M., Schmauss, M.: JACK: A Java constraint kit. In: WFLP 2001, University of Kiel; Technical Report No. 2017 (2001)

Constraint Imperative Programming with C++

Olaf Krzikalla
Reico GmbH
krzikalla@gmx.de

Abstract. Constraint-based programming is of declarative nature. Problem solutions are obtained by specifying their desired properties, whereas in imperative programs the steps that lead to a solution must be defined explicitly. This paper introduces the Turtle Library, which combines constraint-based and imperative paradigms. The Turtle Library is based on the language Turtle[1] and enables constraint imperative programming with C++.

1 Constraint Imperative Programming at a Glance

In an imperative programming language the programmer describes how a solution for a given problem has to be computed. In contrast to that, in a declarative language the programmer specifies what has to be evaluated. Constraint-based programming is a rather new member of the declarative paradigm that was first developed from logic programming languages. In constraint-based programming the programmer describes the solution only by specifying the variables, their properties and the constraints over the set of variables. Actually, no algorithms have to be written. The compiler and run-time environment are responsible for providing appropriate algorithms and eventually obtaining a solution.

Meanwhile, constraint-based programming has been extended by concepts of other - mostly declarative - programming languages. However, the combination of imperative and constraint-based languages is far less explored. Borning and Freeman-Benson[2] introduced the term 'constraint-imperative programming' and developed the language Kaleidoscope[3], combining constraint and object-oriented programming. But object-orientation is no precondition for constraint-imperative programming. This paper deals with more fundamental problems of the integration of constraints and constraint solvers in imperative language concepts. This integration promises some advantages. Imperative programming is a well known paradigm, which is intuitively understood by most programmers. A lot of efficient and industrial-strength imperative languages exist. However, an imperative program for a difficult algorithm is sometimes very cumbersome. Especially for this sort of problems declarative languages have proven their power. Constraint programming enables the programmer to specify required relations between objects directly rather than to ensure these relations by algorithms only. So constraint programs not only often become more compact and readable, but also less erroneous than their imperative counterparts.

Constraint imperative programming tries to combine the advantages of constraint-based and traditional imperative programming. A recent development in

this field is the language Turtle, a constraint imperative programming language developed at the Technische Universität Berlin. Based on the ideas presented in [1] I developed the Turtle Library, a constraint imperative programming approach in C++.

2 The Basic Concept of Turtle

The fundamental difference between imperative and declarative languages is the model of time. In pure declarative languages a timing model simply does not exist - computations are specified independent of time. On the other hand, an imperative language always describes transformations of a given state at one point in time to another state at the next point in time. Computations are specified by sequences of statements.

Whenever declarative and imperative languages are combined, one of the main issues is the interaction of the integrated declarative concepts with the imperative timing model. In Turtle this is solved by introducing a lifetime for constraints and the statement *require*, which defines a constraint:

```
require constraint;
```

When a *require* is reached during the execution of the program, the given constraint is added to a global constraint store and taken into account during further computations - its lifetime starts. A constraint doesn't exist (and the system doesn't know anything about it) until the corresponding *require*-statement is executed. Eventually a sequence of *require*-statements form a conjunction of the appropriate constraints in the constraint store. Constraints in the constraint store are considered active.

Of course, if a constraint starts to exist at a certain time, it also can be removed at a certain time:

```
require constraint in
  statement;
  ...
end;
```

The given constraint exists only between the *in* and *end*. When the program reaches the *end* statement (or otherwise leaves the block), the constraint is removed from the constraint store - its lifetime ends. After this the constraint isn't active any longer.

In order to deal with over- and underconstrained problems constraints need to be labelled with strengths to form a constraint hierarchy. Although a constraint imperative system without constraint hierarchies could be designed, its usefulness would be drastically reduced, because it would be difficult to constrain variables while the program dynamically adds or removes constraints. In Turtle each constraint can have a strength annotation in its definition:

```
require constraint1 : strong;
require constraint2 : mandatory;
```

When a constraint is annotated with a strength, it is added to the store with the given strength, otherwise with the strongest strength *mandatory*. This strength was specified in the previous example for clarity only.

Constraints are defined on constrainable variables. Most of the time a constrainable variable acts like a normal variable: it can be used in expressions and as a function argument. Only in a constraint statement they differ from their normal counterparts. A normal variable is treated like a constant, but a constrainable variable acts like a variable in the mathematical sense, and the constraint solver may change its value in order to satisfy all constraints existing at this point in time.

```
var x : int; // a normal variable
var y : ! int; // the exclamation defines a constrained variable
x := 0;
require y <= x in
  ... // during the execution of this block Turtle ensures y <= 0
end;
```

Constraints in Turtle are boolean expressions. During the execution of a *require* statement the constraint solver computes a certain value for each constrained variable, such that all active constraints evaluate to true. Constraints are handled strictly *eager*, i.e. changing a non-constrained variable after it was used in a constraint doesn't affect the constraint store. Whenever the program reads a constrained variable, the value last computed by the solver for this variable is supplied. An exception is raised, if it isn't possible to satisfy all mandatory constraints during the execution of a *require* statement.

In Turtle constraints can be used for computing solutions to a certain problem like other constraint programming approaches. But they are not limited to this usage. *require* statements introduce conditions *a priori*, which are maintained automatically by the constraint solver. Hence backtracking like in approaches with *a posteriori* tests (e.g. Alma-0[6]) is not neccessary. Due to the *a priori* nature of constraints in Turtle they can be used to describe and preserve program invariants or - more generally - to express in declarative manner the meaning of an otherwise imperative program without disrupting the familiar execution flow.

3 A Turtle in C++

The concepts of Turtle were first implemented in a language developed from scratch. This approach was chosen because some other features like higher-order functions should also be integrated. And a new language seemed to be the best choice for the seamless combination of imperative, functional and constraint programming. However, a new language is always in a difficult position. The

knowledge base is small, tools don't exist, and further development is sometimes driven by academic interests only.

All concepts of Turtle related to constraint programming are also implementable in C++. That is why I think a Turtle Library written in pure C++ serves both the widespreading and further development of Turtle better. In addition it allows an application programmer to use the benefits of constraint programming in his professional work. In the recent years a lot of developments—especially on the field of generic programming in C++ - made it possible to move almost all concepts from the Turtle language to the C++ Turtle Library without any losses. Furthermore, the generic approach of the Turtle Library enables every user to add, change or optimize constraint solvers at will. This is especially important for user-defined domains and offers a wide application field for the Turtle Library. The Turtle Library might be used to solve operational research problems or to program a graphical user interface. Both problems are typical constraint problems. In the first problem constraint programming is used only to obtain a solution, which often can be done in a constraint logic language too or by using a rather imperative approach[5]. But for the second problem constraint imperative programming really shines. The 'canonical' example is a graphical element, which can be dragged by the mouse inside certain borders[4]. The imperative approach looks like this:

```
void drag ()
{
  while (mouse.pressed) {  //message processing is left out
    int y = mouse.y;
    if (y > border.max)
      y = border.max;
    if (y < border.min)
      y = border.min;
    draw_element (fix_x, y, graphic);
  }
}
```

Using the Turtle Library the example would look as follows:

```
void drag ()
{
  while (mouse.pressed) {
    constrained<int> y = mouse.y;
    require (y >= border.min && y <= border.max);
    draw_element (fix_x, y(), graphic);
  }
}
```

The above is not only shorter, but expresses the relation between the border-object and the y-coordinate in exactly the way a programmer would think about it.

3.1 Constrained Variables

A constrained variable is of the generic type `constrained`. A constrained variable has identity semantics, the copy constructor and standard assignment operator aren't implemented. If they are needed, an appropriate wrapper (e.g., a reference counted pointer) has to be defined. The public interface given here is described in detail in the following sections.

```
template<class T>
class constrained
{
  public:
    constrained (const T& prefer = T());
    T operator ()() const throw (overconstrained_error, ...);
    void unfix() const;
};
```

The template parameter specifies the value type of the variable. It might be a fundamental type like `int` or `double` or an user-defined class. Domains are formed by non-intersecting sets of value types and for each domain an appropriate constraint solver has to be provided. Thus each value type is unambiguously bound to a constraint solver. However Turtle can be used for hybrid domains, because the interface enables the implementation of a constraint solver responsible for more than one value type.

3.2 Declaring Constraints

Constraints can be declared as straightforward as presented in the Section 2:

```
constrained<double> a, b;
double c = 2.0;
require (a >= 0.0);
require (a <= b && a + b <= c);
```

The composition of the boolean expression inside a `require` is done using operator overloading and expression template techniques. Which operators are supported for a certain value type is defined by the domain and the available constraint solver. E.g. it is rather pointless to support `>`, `<` or `!=` for floating point values[1]. In domains other than the algebraic ones it's often better to avoid otherwise meaningless operator overloading. For this purpose named predicates can be defined and used instead:

```
edge e = /*...*/;  //compute an edge
constrained<vertex> p;
require (point_on_edge (e, p));
```

[1] Due to the same reasons even the support of == could be argued.

The operator `&&` forms a conjunction of two expressions just like two subsequent requires, hence

```
constrained<double> a;
require (a >= 0 && a <= 2);
```

is equivalent to

```
constrained<double> a;
require (a >= 0);
require (a <= 2);
```

The operator `||` defines a disjunction. A disjunction can be seen as a branch in a tree of solutions. Subsequent requires add their constraints to all leafs of the tree.

```
constrained<double> a, b;
require (a == 0 || a == 1);
require (b == a + 1);
// the store now contains :
// (b == a + 1 && a == 0) || (b == a + 1 && a == 1)
```

The Turtle Library provides a simple generic algorithm for handling disjunctions. A certain constraint solver may implement a more sophisticated approach to compute and maintain solution trees efficiently.

Constraint strengths can be given as a second argument to require like in the Turtle language:

```
require (a == b, weak);
```

Of course these values are only of interest if the underlying constraint solver supports hierarchical constraints.

The Turtle Library internally stores the constraints in several constraint substores. A constraint sub-store is defined as the set of all constraints over a set of constrained variables, where each variable of the set is linked to each other variable of the set. Two variables x and y are linked, if they either both appear in one constraint or if x appears in a constraint containing a variable linked to y.

```
constrained<double> a, b;
require (a >= 0.0);  // generate constraint sub-store 1
require (b >= 0.0);  // generate constraint sub-store 2
require (a <= b);    // sub-store 1 and 2 are merged together
```

The function template `require` returns a handle to manage the lifetime of the constraint. If the return value is ignored, the imposed constraint exists as long as all constrained variables in this constraint:

```
constrained<int> a;
{
  constrained<int> b;
  require (a == b);
  //...
//leaving the scope of b, hence a == b
//is removed from the constraint store:
}
```

Otherwise, the lifetime of the constraint is also bound to the lifetime of the returned constraint handle. Constraint handles are useful especially when imperative execution flow elements (e.g. loops) and constraints are used together:

```
constrained<int> a;
//setup some inital constraints over a
while (not_done()) {
  int b = compute_something();
  constraint_handle<int> z = require (a >= b);
  //...
//leaving the scope of z, hence a >= b
//is removed from the constraint store:
}
```

Still, a constraint exists no longer than all constrained variables in it. When the handle ceases to exist after the appropriate constraint did, it is ignored.

3.3 Obtaining Values from Constrained Variables

The function call operator `operator()() const` is overloaded to obtain a value from a constrained variable in a convenient way:

```
std::cout << a();  //prints a value matching all constraints to a
```

Whenever this operator is invoked, the constraint solver is started to determine the value of the appropriate variable. How the value is determined depends mainly on the solver. When the store is overconstrained and no value can be determined, an exception of type `overconstrained_error` (derived from `std::logic_error`) is raised.

But more often underconstrained situations occur. For this purpose the Turtle Library supports a preferred value. A value of type T can be assigned to a `constrained<T>` or used to construct such a variable. This value then becomes the preferred value of the constrained variable. Now, if it turns out that more than one solution exists for a certain variable, the solution closest to the preferred value is taken:

```
constrained<double> a (3);
require (a <= 2.5);
std::cout << a();  // prints 2.5
```

To a certain degree the preferred value acts like a weak constraint. This is especially useful, if the constraint solver itself doesn't support constraint hierarchies. Thus a hierarchical constraint solver isn't as necessary as in the original Turtle language.

The evaluation of the preferred value is done by the solver implementation. It can be used to define a *threshhold* or *destination* value enabling the solver to terminate the search through the solution tree as soon as possible.

Some domains consist of incompareable values making it impossible to define a closest solution. In this case no general behaviour can be defined. Instead the solver implementation has to define the use of the preferred value.

3.4 Implicit Fixing

Once a value is determined for a constrained variable, this value has to be taken into account for further calculations. The constrained variable itself gets implicitly fixed to the determined value:

```
constrained<int> a (2), b (0);
require (a == b);
std::cout << a();  // prints 2
std::cout << b();  // also prints 2
```

Without implicit fixing the value of b would be evaluated to 0 and hence violate the required constraint a == b. Implicit fixing is done by generating a new constraint of the form variable == value. Due to this important side effect the evaluation order of constrained variables must be carefully considered. If the output lines of the above example were exchanged, both lines would print 0. And the following leads to unspecified behavior:

```
std::cout << a() << b();  // which variable is evaluated first?
```

The implicit fix is not immediatly added to the constraint sub-store but kept in a delay store inside the sub-store. If only one implicit fix exists in a constraint sub-store, and the same variable shall be evaluated again, the fix is erased before the evaluation (later in the process a new fix will be added). If more implicit fixes exist, always all are taken into account.

```
constrained<int> a (2), b (0);
require (a == b);
for (int i = 0; i < 3; ++i) {
  int j;
  std::cin >> j;
  a = j;
  // prints j, because the only implicit fixed variable is a:
  std::cout << a();
}
```

```
constrained<int> a (2), b (0);
require (a == b);
std::cout << b();  // prints 0, fixes b
for (int i = 0; i < 3; ++i) {
  int j;
  std::cin >> j;
  a = j;
  // always prints 0, because b is fixed, but a is evaluated:
  std::cout << a();
}
```

As shown in the last example, sometimes implicit fixes are harmful, especially if more than one variable is evaluated inside a loop. Thats why a constrained variable can be unfixed explicitly via the member function `unfix()`:

```
constrained<int> a (2), b (0);
require (a == b);
for (int i = 0; i < 3; ++i) {
  int j;
  std::cin >> j;
  a = j;
  std::cout << a();  // prints j and get fixed
  std::cout << b();  // prints also j and get fixed
  //now more than one fix exist, so all fixes would be considered
  //during further evaluations unless we explicitly
  //unfix the variables:
  a.unfix();
  b.unfix();
}
```

The computation of a value for a constrained variable differs a lot from the original Turtle language. While in the Turtle language the values of constrained variables are already determined during a require statement, the Turtle Library delays the computation until a read-action to a constrained variable occurs. The disadvantage of this approach seems the need of implicit fixing, which isn't part of the Turtle language. But actually the problem exists also there:

var x, y : ! int;
require x == 0 : weak;
require y == 2 : medium;
writeln (x); // prints 0
require x == y;
writeln (y); // should print 0

On the other hand the delay of the computation offers some advantages. First, only when the computation is delayed until a read-action, the preferred value can be evaluated correctly. Otherwise a change of the preferred value after some

requires could be ignored. Second, a solver knows which constrained variable actually is being read, can consider this fact during the computation and hence doesn't have to evaluate all variables in every case. And third, lazy evaluation becomes possible. Although also the Turtle Library handle constraints *eager* mostly, it is not limited to this.

4 Programming with the Turtle Library

The Turtle Library can be downloaded from
`http://home.t-online.de/home/krize6/turtle.htm`.
At this page also some technical issues are discussed in more detail. Especially the steps needed to integrate a new constraint solver in the Turtle Library are described. Some more sophisticated examples of constraint imperative programming are already provided. They demonstrate the use of some techniques and little patterns to make constraint imperative programming more convenient and flexible.

4.1 User-defined Constraints and Dynamic Expressions

Often the declarative power of expression templates is sufficient to express the constraints in a compact and readable manner. But some constraints are so common that they deserve an own name. Such user-defined constraints can be generated using the function template `build_constraint`, which takes an constraint just like require, but only builds the internal representation of the given expression without adding it to the constraint store.

```
typedef constrained<int> int_c;

constraint_solver<int>::expr domain (const int_c& x, int min, int max)
{
  return build_constraint (x >= min && x <= max);
}

int_c a, b, c;
require (domain (a, 0, 9));
require (domain (b, 0, 99));
require (domain (c, -1, 1));
```

The naming of complex static expressions further enhances the readability of a program. But besides this constraint imperative programming also needs a way to create constraints dynamically. For this the Turtle Library provides a generic class `dynamic_expr`, which holds an (sub)expression and can be used like that, but has value semantics. A rather complex example is the function `example_dynamic_puzzle`, which is part of the sample file provided on the internet page of the Turtle Library.

4.2 Optimization

Constraint programming supplies a lot of tools to optimize a given function for a given set of constraints. Optimization is one the main usages of constraint programming. Hence, optimization should be possible with the Turtle Library, too. By using a preferred value for a given expression, optimization can be done without the needs of special library functions. Consider the following example:

```
typedef constrained<double> double_c;

double_c x, y;
require (y >= 0);
require (y >= 3 - 2 * x);
```

Given these constraints the sum of x and y shall be minimized. These can be done by a little pattern of the following three lines:

```
double_c min (- 1000.0);
require (min == x + y);
std::cout << min(); // prints 1.5
```

First a constrained variable has to be declared and the preferred value have to be set to an absolute minimal or maximal border. [2] Second, this variable has to be set equal to the expression to be optimized. And third, by reading the variable the value closest to the given preferred value gets calculated and stored in the variable. Furthermore the implicit fixing also immediately limits other constrained variables to values at the searched optimum.

5 Conclusion and Future Works

The Turtle Library defines an interface for the integration of constraint programming concepts in an imperative language and provides an implementation of this interface for a popular language. Hopes are, that this opens a wider application field for constraint imperative programming. Only the practical use will show further needs. E.g. if an implicit fix of a constrained variable has to be considered is defined by a rather complex rule. It's unclear if this rule is of any practical value. Also, for the moment there is no way to unfix a bunch of variables at once (e.g. all variables of a sub-store).

The modelling of algebraic problems using the Turtle Library is already very convenient. But the generic approach offers a lot more. A lot of publications in the recent decade has shown, that constraint programming is well-suited for several problem domains. But unfortunately a lot of these publications either introduced a whole new language or at least extended an existing language by

[2] This example is rather abstract and hence knows no 'absolute' minimum. In practical applications it should be always possible to find a reasonable value (see also `example_knapsack`).

adding new language constructs (and thus became incompatible to the parent language). But an application programmer can't just move from one language to the next at will. Due to business, management and also educational issues he has to stick to one - often for years. With the Turtle Library now even the application programmer gets a tool to use constraints in C++ in the convenient declarative manner as it is already used for years in other languages.

References

1. Grabmüller, M. and Hofstedt, P.: Turtle: A Constraint Imperative Programming Language. In *Proceedings of the Twenty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*. Research and Development in Intelligent Systems XX, 2003. To appear.
2. Freeman-Benson, B.N.: Constraint Imperative Programming. PhD Thesis, University of Washington, 1991. Published as Department of Computer Science and Engenieering Technical Report 91-07-02
3. Borning, A. and Freeman-Benson, B.N.: The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In *Proceedings of the IEEE Computer Society 1992 International Conference on Computer Languages*, pages 174-180, 1992
4. Lopez, G.: The design and implementation of Kaleidoscope, a constraint imperative programming language. PhD Thesis, University of Washington, 1997.
5. ILOG. ILog Web Site.
   `http://www.ilog.com`, last visited 2003-06-23
6. Apt, K.R., Brunekreef, J., Partington, V. and Schaerf, A.: Alma-0: An imperative language that supports declarative programming. ACM Toplas, 20(5):1014-1066, 1998.

`firstcs` — A Pure Java Constraint Programming Engine

Matthias Hoche, Henry Müller, Hans Schlenker, Armin Wolf

Fraunhofer FIRST
Kekuléstraße 7, D-12489 Berlin, Germany
{Matthias.Hoche|Henry.Mueller|Hans.Schlenker|Armin.Wolf}@first.fraunhofer.de

Abstract. This work presents the object-oriented Java constraint programming engine `firstcs` for finite domain constraint solving. Beyond the architecture of the system and the supported constraints, this presentation focuses on the available constraint processing and search strategies. The presentation is completed by some applications realised with this engine.

1 Introduction

Java is an object-oriented, state-of-the-art programming language that is well-suited for the rapid development of remarkable large and complex interactive and/or distributed computational applications [3, 8]. Thus, Java is more and more used in commercial applications. One of the demanding areas in these commercial applications is optimisation, as planning and scheduling in supply chain management or enterprice resource planning (ERP).

There are some constraint solving approaches for (distributed) Java-based applications: JSolver [6] is a Finite Domain solver, written purely in Java. It was one of the first such attempts in Java. JSolver is now owned by ILOG, whose *Solver* (written in C++) is the most famous and most successful current constraint solver. ILOG's Irvin Lustig stated in 2001, that *Java-constraint programming tools are still in a state of development. The next year should see the introduction of pure Java constraint programming engines* [5, 14]. JSolver is ILOG's basis for that. The Java Constraint Library JCL [20] implements binary constraints with explicit constraint representation (enumeration of admissible value combinations) and some well-known propagation and search algorithms. There are two approaches dealing with Constraint Handling Rules (CHR) and Java: [19] and [22]. POOC [18] is a Platform for Object-Oriented Constraint programming and provides a generic Java-Interface to some C(L)P systems. Finally, Koalog [12] is – like JSolver and `firstcs` – a pure-Java FD-solver and probably the one that currently has the largest set of functionality.

This work presents a pure and extendible Java constraint programming engine for finite domain constraint solving widely used for planning, scheduling, and configuration problems. The engine supports the integration of new constraints as well as new search strategies.

2 The Constraint Programming Engine's Architecture

The kernel of our Java constraint programming engine called `firstcs` is formed by a Java class called `CS` which is an acronym for the term *Constraint System*. Each object of this class is indeed a constraint system managing finite domain variables and constraints over these variables. Due to the object-oriented design, it is possible to generate and manipulate several constraint systems in a single application. In the current version these systems are independent: they neither share variables nor constraints. This restriction will be overcome in future versions of the engine.

There are the Java classes `Domain`, `Variable`, `Constraint` and the subclasses of `Constraint` around the kernel providing the tool box to model and solve constraint problems.

The class `Domain` implements the finite domains (fd) of the variables, i.e. finite integer sets represented by lists of integer intervals. There are several methods to manipulate these sets, e.g. the usual set operations. All these methods return a boolean value which is false if and only if the manipulated set becomes the empty set. This information is used to detect inconsistencies during constraint propagation.

The class `Variable` implements the fd-variables representing the unkowns of a constraint problem. Their admissible values are restricted by their finite domains and their constraints. Thus, they are implemented as *attributed variables* [10]: Together with the domains, the constraints are attached to their variables. This construct is commonly used in constraint logic programming systems, e.g. like SICStus Prolog[1].

The abstract class `Constraint` samples all the concrete constraint classes. Beyond other application-specialised constraints, these are:

- `Abs` constrains a variable to be the absolute of another, i.e. $x = |y|$.
- `Before` constrains an activity to be finished before another one starts, i.e. $a.start + a.duration \leq b.start$.
- `Equal`, `Greater`, `GreaterEqual`, `Less`, `LessEqual`, and `NotEqual` relate two variables with respect to the corresponding arithmetical relation, i.e. $s < t$.
- `Kronecker` states that a variable has a given integer value if another boolean value is true (1) or false (0), i.e. $\delta_{v,i} = 1$ if $v = i$ and 0 otherwise.
- `Resource` and `MultiResource` state that some activities are processed either on an exclusively available resource or on alternatively available resources.
- `SetUpTime` and `SetUpCost` state sequence-dependent setup times or cost for activities that are processed successively on some resources.
- `Product`, `Sum` and `WeightedSum` establish arithmetic relations between variables, especially cost functions for optimisation problems.

In these concrete constraint classes there are constraint-specific implementations of the method `activate()` defined abstractly. This method performs local constraint propagation, i.e. the domains of the constraints' variables are

[1] See `http://www.sics.se/sicstus.html`.

pruned and either other constraints are (re)activated via common variables or an `InconsistencyException` is thrown if an inconsistency is detected. Expections are thrown instead of returning a value, e.g. false, to force an explicit handling of inconsistencies. Thus, it is impossible to ignore inconsistencies avoiding senseless deductions, i.e. when *ex falso quod libet* holds.

The `Resource` constraint is implemented as a *dynamic global constraint* as proposed in [4] and uses state-of-the-art pruning algorithms [23]. Thus, after the creation of a new resource constraint, i.e. `Resource resource = new Resource()`, it is possible to add a new task with variable `start` time and `duration` incrementally, i.e. `resource.addTask(start, duration)`. The addition of any constaint, i.e. of an object of any subclass of the abstract class `Constraint`, is realised by the method `void add(Constraint c)`. It is possible to undo both kinds of additions via a backtracking mechanism, further explained in Section 4.

3 Customizing Constraint Processing

In contrast to other constraint programming systems, especially to constraint logic programming systems, with the addition of a constraint its activation, i.e. the propagation of its consequences, is not automatically performed: Any possible pruning of the variables' domains with respect to this constraint is delayed by default. It must be performed either via constraint propagation or explicitly by a method call. Thus, it is possible to build-up a constraint system representing a problem to be solved completely before propagating the constraints' consequences. This may improve the overall performance of the constraint processing as the following example shows:

Example 1. Considering the constraint problem $var_{i-1} < var_i$ for $i = 1, \ldots, n-1$ any incremental constraint propagation is $O(n^2)$: Any extension of $var_0 < \cdots < var_i$ to $var_0 < \cdots < var_i < var_{i+1}$ will adopt the domains of all considered variables. However, a delayed propagation triggered after adding the constraints might be $O(n)$. The corresponding Java program realises both: incremental propagations immediately after adding constraint by constraint and a delayed propagation after adding all constraints – the flag `INCREMENTAL` triggers both cases.

```
CS cs = new CS();
Variable[] var = new Variable[n];
for (int i=0; i<n; i++) {
    var[i] = new Variable(i, n);
    if (i>0) {
        cs.add(new Less(var[i-1], var[i]));
        if (INCREMENTAL) cs.activate();
    }
}
if (!INCREMENTAL) cs.activate();
```

Here, `cs.activate()` activates all constraints that were added to the constraint system `cs` but are not yet activated. Thus, in the incremental case only the most recently added constraint is activated while in the non-incremental case all added constraints are activated. Runtime experiments have shown that the delayed

non-incemental propagation is in fact linear if the constraints are activated in a last-in-first-out ordering. □

For further performance improvements we implemented several strategies to schedule the constraints' activations while calculating the global fix-point of the entire constraint propagation process (cf. [2]). Therefore, the delayed propagation can be used to control the processing of the constraints. Information of changes in variable domains or special constraint-properties are used to check the necessity of an activation or to influence the order of activations.

The goal of controlling constraint-processing is to find a heuristic that allows a complete propagation with a minimum of time-consuming constraint activations. Our approach is based on two different abstraction levels. On the one level we distinguish between general and specific triggers for the re-activation of constraints. On the other level we have realised different sequenzing strategies for these activations.

The general trigger for constraint re-activation is called *non-directed propagation*: Whenever the domain of a variable changes, the constraints on this variable are scheduled for re-activation.

More specific information for re-activation is used by *directed-propagation*: The kind how the variables' domains are changed is used to decide which constraint really has to be activated. Therefore for each variable in a constraint there are additional properties: `val`, `min`, `max`, `mixed`, and `dom`. These are used to decide whether the constraint has to be activated if the variable has a determined value or the domain's minimum, the maximum, some of them, or anything in the domain has changed (cf. [7]).

In general, we either re-activate the constraints in last-in-first-out (LIFO) or in first-in-first-out (FIFO) manner. LIFO means that the constraints on the variable with the most recently changed domain will be re-activated before any other constraints. If the variable's domain changes again the activation is repeated until its domain stays unchanged. Then the propagation continues with the next re-activated constraints. FIFO instead propagates the constraints on the variable having a changed domain once. If the variable's domain changes again by its constraints, it is queued for another activation. However, the re-activation is done after the re-activation of previously queued constraints.

We also tried to avoid useless propagation by using the complexity of the constraints to further influence the order of re-activations.

The *weighted* approach is a heuristic to reduce the activation of constraints of high complexity, i.e. like the `Resource` constraints. Our anticipation is that if constraints having constant complexity are processed before constraints having linear or even quadratic or cubic complexity, all the possible domain reductions are performed with fewer activations of more complex constraints. Therefore, we implemented a new constraint processing that is based on several sets of constraints corresponding to the different complexity classes, instead of only one. This strategy does a good job, as we see in Table 1.

In total, we implemented 3 different scheduling strategies and for each of it a LIFO and FIFO variant.

ordering	propagation	constraint-activations %						runtime %	
		total			av			total	av
		before	resource	all	before	resource	all		
LIFO	*NON-DIRECTED*	*100*	*100*	*100*	*100*	*100*	*100*	*100*	*100*
	DIRECTED	95	98	96	87	98	89	99	102
	WEIGHTED	162	46	134	137	51	118	49	58
FIFO	NON-DIRECTED	105	48	91	98	48	87	58	59
	DIRECTED	93	52	83	86	51	79	63	64
	WEIGHTED	118	**24**	96	102	**29**	86	**33**	**40**

Table 1. Comparison of different activation strategies.

Table 1 shows benchmark results of implementations of the 3 different strategies and its LIFO and FIFO variants. Therefore the constraint activations and runtime data of every strategy were logged for more than 20 job-shop scheduling problems. The problems were taken from [1, 9, 13]. These job-shop scheduling problems are modelled by the use of `Before` constraints having constant complexity and some `Resource` constraints having quadratic complexity. This mixture of complexities allows a good testing of the *weighted* strategy.

The benefits of the delayed propagation are obvious but in contrast to Example 1 the results show a great advantage of the FIFO implementations. All FIFO strategies are better than the respective LIFO strategies. The propagation of some domain changes over the complete constraint network at first has a great impact on the calculation of the global fix-point. In combination with the *weighted* approach we achieve best results. Especially larger problems perform well with the *weighted* propagation. As you can see, we reached 33% as a total value and 40% in average, where the relations *total* and *average* are calculated as follows.

total: sum up all benchmark results of a considered strategy, then relate it to the sum of the base implementation,

average: relate each benchmark result of the considered strategy to the corresponding result of the base implementation, then average all these ratios.

This means that in *average* calculations each benchmark-problem has the same impact. In *total* calculations the impact depends on the problem complexity, i.e. the runtime or number of activations. That is the reason for the 40% in average, because larger problems benefit more of weighted propagation. Problems with less complexity have a smaller improvement because of the generated overhead. Anyway, this strategy reaches best runtime results for all benchmark problems and is permanently almost three times faster than the *non-directed* strategy that serves as the reference base.

4 Realizing Search

After the addition of all constraints defining a problem to be solved and their propagation a search process is used in general to find a solution of the considered problem. The constraint engine `firstcs` supports search based on different

pre-defined strategies but also the necessary basics for any user-defined search based on backtracking. These basics are choicepoints, i.e. objects of the class `ChoicePoint`. For a given constraint system, we are able to generate several choicepoints to store the system's state at a specific program state. Therefore, a choicepoint is set with the method `set()`, signalling the engine that the current state of the constraint system has to be stored for any future backtracking. The call of the method `backtrack()` for a previously set choicepoint will restore the stored state at the program state where the choicepoint was set. For any re-use of a choicepoint at another program state it might be reset with the method `reset()`.

The usage of choicepoints is illustrated for simple depth-first search: Here, variables in a given array are labelled iteratively choosing a value from the minimum up to the maximum of their domains. For each variable a choicepoint is set before labelling it. If the labelling results in an inconsistency, a corresponding exception is thrown and catched and furthermore search backtracks to the set choicepoint undoing the labelling and all of its consequences. This allows the selection of the next value, if there is any. If there is no inconsistency, labelling is applied recursively to the not yet labelled variables (see Figure 1).

```
static boolean label(CS cs, int i, Variable[] var) {
    if (i == var.length) // base case:
        return true;
    ChoicePoint cp = new ChoicePoint(cs);
    cp.set();
    for (int val = var[i].min(); val <= var[i].max(); val++)
        try {
            var[i].equal(val);
            cs.activate();
            if (label(cs, i+1, var)) {
                cp.reset();
                return true;
            } // else:
            cp.backtrack();
        } catch (InconsistencyException e) {
            cp.backtrack();
        }
    cp.reset();
    return false;
}
```

Fig. 1. The labelling algorithm

For users that are not familiar with the implementation of search strategies the engine offers some pre-defined search strategies, i.e. subclasses of the class `Label`. All these classes offer a method `nextSolution()` that allows an iteration over all solutions of a given constraint problem. Some of these classes offer the methods `nextMinimalSolution(Variable objective)` and `nextMaximalSolution(Variable objective)` additionally, allowing an iteration over all minimal and maximal solutions with respect to a given objective

function. Therefore, the variable `objective` has to be constrained to the function's result. Beyond others the engine offers the following subclasses of `Label`:

- `StdLabel` implements the previously presented depth-first search. Incremental search as proposed in [21] is used to find optimal solutions.
- `ResourceLabel` implements a specialised search for job-shop scheduling problems. Before any labelling of the tasks' start times search looks for a linear ordering of the tasks on the considered resource.
- `OrderLabel` implements the Reduce-To-The-Max search algorithm presented in [17, 24] for contiguous task scheduling and optimisation problems. A dichotomizing algorithm is used to determine the optimal of the objective function.

The usage of any of these classes is quite simple. Let a constraint system `cs` with constraints be given that restrict the values of the variables in an array `var`. Then the following piece of code will find and print all solutions of the constraint problem. Finally, it will reset the constraint system to the state before the search:

```
StdLabel label = new StdLabel(cs, var);
label.set();
while (label.nextSolution())
    for (int i=0; i <= var.length; i++)
        System.out.println(var[i]);
label.reset();
```

Furthermore, we extended constraint processing by justifying all prunings that result from constraint propagation. Especially, dead ends detected during search, i.e. inconsistencies resulting from the decisions made at some choice-points are justified. Thus, intelligent search strategies are realisable. In fact, we implemented a variant of conflict-directed backjumping (CBJ) [16] and applied this search strategy to random 3-SAT-problems.

The main intelligence of CBJ lies in its ability to know the cause of an inconsistency, if it occurs. This enables a direct jump to the cause, which saves all the useless work that a standard backtracking algorithm (BT) would do to get back to it. BT can only try to find a feasible value for the current variable, and if it finds none, it has to go back step by step. Thus, it will try all other values for the variables between the current and the causal one.

To accomplish CBJ, firstly we need an additional data structure for every variable to hold the causes for domain modification. We call this a *justification* of a variable. It contains references to all variables, which are responsible for the domain state of its variable. Secondly, we need a data structure to manage the backjumping. It holds references of variables, which forestalled the assignment of its variable with a value. It is called the *conflict set* of a variable. In the worst case, CBJ jumps through the search space in steps of 1 level, which is equivalent to standard backtracking. Mostly, this is not the case, as we will see later, but we assert that in the worst case, CBJ behaves like BT.

Let us have a look at the "cbj-enhanced" labelling algorithm. Basically, it is the same as in Figure 1, but extended with CBJ "intelligence":

```
int cause = 0; // initialise cause value
static boolean label(CS cs, int i, Variable[] var) {
    if ( i == var.length ) { return true; } // base case
    ChoicePoint cp = new ChoicePoint(cs);
    cp.set();
    for ( int val = var[i].min(); val <= var[i].max(); val++ )
        try {
(1)         var[i].equal(val, new Justification(i));
            cs.activate();
            if ( label(cs, i+1, var) ) {
(*)             var[i].conf.reset();
                cp.reset();
                return true;
            } // else:
(2)         if ( cause > i ){
(*)             var[i].conf.reset();
                cp.reset();
                return false;
            } else if ( cause==i ) { cause = 0; }
            cp.backtrack();
        } catch (InconsistencyException e) {
(3)         if ( !e.var.just.isEmpty() ){
                var[i].conf.addSet( e.var.just );
            }
            cp.backtrack();
        }
(4) if ( !var[i].conf.isEmpty() ){
        cause = var[i].conf.getReason(i);
        var[cause].conf.addSet(var[i].conf);
    }
(*) var[i].conf.reset();
    cp.reset();
    return false;
}
```

First of all, in `(1)` a variable assignment during labelling is now justified with its level, which acts as a reference to the variable. If an assignment leads to an inconsistency exception, the justification of the variable where the inconsistency occured, extends the conflict set of the current variable in `(3)`. If no assignment can be found for the current variable, one has to jump back. In `(4)` the greatest value of the conflict set, which is smaller than the current level i, determines the target of the jump with `getReason(i)`. The backjumping itself is performed in `(2)`: As long as the level `i` of the current variable is greater than the causal variable's level `cause`, the algorithm steps back. The handling of the conflict sets cannot be handled in `cp.reset()`, as we want to collect justifications across several assignments attempts of a variable, so there are handled manually in `(*)`.

Search is a demanding job. For this reason, we should have an eye on the performance of our labelling algorithm. In our case, it is important to use an efficient implementation for the justifications and the conflict sets, because they are used intensely. At the moment, we prefer a fast set implementation which is more efficient and faster than the usual appropiate Java structures.

Let us take a look at a few results in Table 2. Based on a CHR approach [22] we developed some boolean constraints for the solution of random 3-SAT-problems, i.e. the so called AIM-instances [11]. Among other things these instances are classified into solvable yes-instances and unsolvable no-instances, so there is either exactly one solution or no solution at all. We concentrated on the no-instances in the benchmark set because the whole search space has to

file	alg	-	static1	static2	dyn	static1 & dyn	static2 & dyn
$1\_6 - no - 1.cnf$	bt	1008585	17705	1686	1008585	17705	1686
	cbj	877	58	151	216	36	104
	ratio[%]	0.09	0.33	8.96	0.02	0.20	6.17
$1\_6 - no - 2.cnf$	bt	220520	159187	6782	220520	159187	6782
	cbj	82	615	259	55	87	198
	ratio[%]	0.04	0.39	3.82	0.02	0.05	2.92
$1\_6 - no - 3.cnf$	bt	5751947	16385611	302	5751947	16385611	302
	cbj	1554	21375	67	587	1271	49
	ratio[%]	0.03	0.13	22.19	0.01	0.01	16.23
$1\_6 - no - 4.cnf$	bt	967818	67404	280062	967818	67404	280062
	cbj	62	42	21	23	32	21
	ratio[%]	0.01	0.06	0.01	0.00	0.05	0.01
$2\_0 - no - 1.cnf$	bt	453303	337314	182	453303	337314	182
	cbj	21977	652	40	309	47	46
	ratio[%]	4.85	0.19	21.98	0.07	0.01	25.27
$2\_0 - no - 2.cnf$	bt	35157	64974	361	35157	64974	361
	cbj	769	8556	49	148	1007	51
	ratio[%]	2.19	13.17	13.57	0.42	1.55	14.13
$2\_0 - no - 3.cnf$	bt	86099	9839	58	86099	9839	58
	cbj	19729	660	47	1505	798	47
	ratio[%]	21.91	6.71	81.03	1.75	8.11	81.03
$2\_0 - no - 4.cnf$	bt	30257	65102	4259	30257	65102	4259
	cbj	190	6318	48	93	304	31
	ratio[%]	0.63	9.70	1.13	0.31	0.47	0.73
sum	bt	8553686	17107136	293692	8553686	17107136	293692
	cbj	45240	38276	682	2936	3582	547
	ratio[%]	0.53	0.22	0.23	0.03	0.02	0.19
avg	bt	1069210.75	2138392.00	36711.50	1069210.75	2138392.00	36711.50
	cbj	5655.00	4784.50	85.25	367.00	447.75	68.38
	ratio[%]	3.84	3.84	19.09	0.33	1.31	18.31

Table 2. A benchmark set shows the progress from simple BT to heuristic assisted CBJ. *Ratio* shows the relative improvement of CBJ to BT depending on the configuration in percent. *Sum* shows the overall improvement, *avg* shows the average of the results.

be scoured completely, for a non-existing solution. The yes-instances were neglected, because the labelling algorithm aborts processing, as soon as it finds the solution. In Table 2 one can see the comparison between BT and CBJ for all no-instances with 50 variables. The comparison concerns the number of jumps needed by an algorithm to process a problem.

Although we reached satisfying results - as one can see - with pure CBJ, we drove it even further and developed heuristics to reduce the number of jumps a lot. Two static sort methods (static1, static2) were created, which can be applied to define an initial ordering of the variables. Futhermore, a method was developed for dynamic variable sorting (dyn) during the labelling process. The functionality of those heuristics will be presented aside more detailed information on CBJ and intelligent search in [15].

All together, we managed to solve the instance-collection with CBJ on average in 3.84% of jumps needed for BT, in combination with heuristics the ratio is even better. With the best combination of static and dynamic ordering we even came down to satisfying 0.0064% = 547 [cbj+static2&dyn] / 8553686 [bt] * 100%. For the future, we are looking forward to implement CBJ for more complex constraints, especially the (multi-)resource constraints.

5 A Practical Application

There are several applications realised with our constraint programming engine `firstcs`. One of the practical applications is a participant booking and planning system, sucessfully applied to organise workshops with parallel sessions.

Participant	Marketing	Personnel Management	Human Resources Development	Project Management	Risk Management
Ms. Breitschopf	X	X	X		
Ms. Dunkel	X		X		X
Ms. Friedrich	X	X		X	
Ms. Helmig		X	X	X	
Ms. Karlson			X	X	X
Ms. Mayerhofer	X	X		X	
Ms. Oppermann			X	X	X
Ms. Quendlin		X		X	X
Ms. Tengelmeier		X	X		
Ms. Ulmer				X	X
Mr. Anderl	X			X	X
Mr. Cornelsen	X	X	X		
Mr. Emmerich	X		X		X
Mr. Guenther	X		X	X	
Mr. Jensen	X		X		X
Mr. Ludewig	X	X			
Mr. Nickel		X	X		
Mr. Paulsen			X	X	
Mr. Ritsch				X	X
Mr. Strohmer	X		X		

Table 3. The participants' choices for their management seminar.

The participant booking planning system called DOTPlan was developed in a few days. It is mainly based on the `Kronecker` constraint that was designed and implemented while realizing the whole system and on `WeightedSum` constraints, especially used to represent the objectives to be optimised. Given the participants' choices of their selected sessions as in Table 3 the system satisfies all these choices optimally, i.e for a minimal number of repetitions and an equally distributed turnout.

The `Kronecker` constraint is used to represent the fact that the selected course for a person p at day d is c, i.e. $\delta_{\mathsf{course}[p][d],c} = 1$ if $\mathsf{course}[p][d] = c$ and 0 otherwise. These boolean variables are further used in `WeightedSum` constraints representing the sums of entries per persons, sessions and days.

An optimal plan of the sample problem is shown in Table 4: It is impossible to distribute the seminars over two days because there is at least one participant who has chosen three different courses. The optimality of the turnout will become clear considering the summary in Table 4: It is neither possible to increase the minimal number of participants per day nor to decrease their maximal number. A minimal number of 4 requires at least $3 \times 4 = 12$ participants in each course and a maximal number of 4 requires at most $3 \times 4 = 12$ participants in each course; both requirements are not satisfiable in the considered example.

Participant	1st Seminar Day	2nd Seminar Day	3rd Seminar Day
Ms. Breitschopf	Human Res. Develop.	Personnel Management	Marketing
Ms. Dunkel	Marketing	Risk Management	Human Res. Develop.
Ms. Friedrich	Project Management	Marketing	Personnel Management
Ms. Helmig	Project Management	Human Res. Develop.	Personnel Management
Ms. Karlson	Risk Management	Project Management	Human Res. Develop.
Ms. Mayerhofer	Personnel Management	Marketing	Project Management
Ms. Oppermann	Human Res. Develop.	Project Management	Risk Management
Ms. Quendlin	Personnel Management	Risk Management	Project Management
Ms. Tengelmeier	Personnel Management	–	Human Res. Develop.
Ms. Ulmer	Risk Management	Project Management	–
Mr. Anderl	Marketing	Risk Management	Project Management
Mr. Cornelsen	Human Res. Develop.	Personnel Management	Marketing
Mr. Emmerich	Risk Management	Human Res. Develop.	Marketing
Mr. Guenther	Project Management	Human Res. Develop.	Marketing
Mr. Jensen	Marketing	Human Res. Develop.	Risk Management
Mr. Ludewig	–	Personnel Management	Marketing
Mr. Nickel	Human Res. Develop.	–	Personnel Management
Mr. Paulsen	Project Management	–	Human Res. Develop.
Mr. Ritsch	Project Management	–	Risk Management
Mr. Strohmer	–	Marketing	Human Res. Develop.

Course	Marketing	Personnel Management	Human Resources Development	Project Management	Risk Management	Sum
1st Seminar Day	3	3	4	5	3	18
2nd Seminar Day	3	3	4	3	3	16
3rd Seminar Day	5	3	5	3	3	19
Sum	11	9	13	11	9	–

Table 4. An optimal plan for the management seminar.

References

1. David Applegate and William Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 27(3):149–156, 1991.
2. Krzysztof R. Apt. From chaotic iteration to constraint propagation. In *Proceedings of 24th International Colloquium on Automata, Languages and Programming (ICALP '97)*, number 1256 in Lecture Notes in Computer Science, pages 36–55. Springer-Verlag, 1997.
3. Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, June 2000.
4. Roman Barták. Dynamic global constaints: A first view. In *Proceedings of ERCIM Workshop on Constraints*, Pague, June 2001.
5. Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, pages 87–96, October 1997.
6. Andy Hon Wai Chun. Constraint programming in Java with JSolver. In *Proceedings of PACLP99, The Practical Application of Constraint Technologies and Logic Programming*, London, April 1999.
7. Philippe Codognet and Daniel Diaz. Compiling constraints in `clp(fd)`. *The Journal of Logic Programming*, pages 185–226, 1996.
8. David Flanagan. *Java in a Nutshell*. O'Reilly, 3rd edition, November 1999.
9. G. L. Thompson H. Fisher. Probabilistic learning combinations of local job-shop scheduling rules. In G. L. Thompson J. F. Muth, editor, *Industrial Scheduling*, pages 225–251. Prentice Hall, Englewood Cliffs, New Jersey, 1963.

10. Christian Holzbaur. *Specification of Constraint Based Inference Mechanism through Extended Unification*. PhD thesis, Dept. of Medical Cybernetics & AI, University of Vienna, 1990.
11. K. Iwama, E. Miyano, and Y. Asahiro. Random generation of test instances with controlled attributes. In *Cliques, Coloring, and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 377–394. American Mathematical Society, 1996.
12. Koalog. *Koalog Constraint Solver*. `http://www.koalog.com/php/jcs.php`.
13. S. Lawrence. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement). Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1984.
14. Irvin J. Lustig. Optimization and Java. *Java Developer's Journal*, 6(6):110–116, 2001.
15. Henry Müller. Analyse und Entwicklung von intelligenten abhängigkeitsgesteuerten Suchverfahren für einen Java-basierten Constraintlöser. Master's thesis, Technische Universität Berlin, 2003. Work in Progress, will be published between Nov. 2003 and Jan. 2004.
16. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993. (Also available as Technical Report AISL-46-91, Stratchclyde, 1991).
17. Hans Schlenker. Reduce-To-The-Max: ein schneller Algorithmus für Multi-Ressourcen-Probleme. In Francois Bry, Ulrich Geske, and Dietmar Seipel, editors, *14. Workshop Logische Programmierung*, number 90 in GMD Report, pages 55–64, 26th–28th January 2000.
18. Hans Schlenker and Georg Ringwelski. POOC - a platform for object-oriented constraint programming. In *ERCIM/CologNet Workshop on Constraint Solving and Constraint Logic Programming*. Springer LNCS 2627, 2002.
19. Matthias Schmauss. An implementation of CHR in Java. Master's thesis, Ludwig Maximilians Universität München, Institut für Informatik, May 1999.
20. Marc Torrens, Rainer Weigel, and Boi Faltings. Java constraint library: Bringing constraint technology on the Internet using Java. In *Proceedings of the CP-97 Workshop on Constraint Reasoning on the Internet*, November 1997.
21. Pascal van Hentenryck and Thierry le Provost. Incremental search in constraint logic programming. *New Generation Computing*, 9(3 & 4):257–275, 1991.
22. Armin Wolf. Adaptive constraint handling with CHR in Java. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science. Springer Verlag, 2001.
23. Armin Wolf. Pruning while sweeping over task intervals. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)*, Lecture Notes in Computer Science, Kinsale, County Cork, Ireland, 29 September – 3 October 2003. Springer Verlag. (to appear).
24. Armin Wolf. A specialized search algorithm for contiguous task scheduling problems. In *Proceedings of the Joint Annual Workshop of the ERCIM Working Group on Constraints and the CoLogNET area on Constraint and Logic Programming*, MTA SZTAKI, Budapest, Hungary, 30 June – 2 July 2003.

RCoRP'03: Fifth International Workshop on Rule-Based Constraint Reasoning and Programming

September 29, 2003
Kinsale, County Cork, Ireland
at the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03)

Rule-based formalisms are ubiquitous in computer science, and even more so in constraint reasoning and programming. In constraint reasoning, algorithms are often specified using inference rules, rewrite rules, sequents, proof rules or first-order axioms written as implications. Advanced programming languages like CHR, CLAIRE and ELAN allow to implement both constraint solvers and programs using constraints in a rule-based formalism.

This workshop invites papers describing ongoing work in using rule-based formalisms in constraint reasoning and programming including

- specification of algorithms for solving constraints by rules,
- implementations of constraint solvers and programs solving problems in a novel way using rule-based programming languages that go beyond constraint logic programming,
- automatic generation of rule-based constraint solvers,
- analysis of rule-based programs, and
- other issues related to rule-based language design and implementation.

Organization

Workshop organizers:

Slim Abdennadher (University of Munich)
Thom Frühwirth (University of Ulm)
Arnaud Lallouet (University of Orléans)

Program Committee:

Slim Abdennadher (University of Munich)
Thi Bich Hanh Dao (University of Orléans)
Abdelali Ed-dbali (University of Orléans)
Thom Frühwirth (University of Ulm)
Arnaud Lallouet (University of Orléans)
Eric Monfroy (University of Nantes)

Delaying "big" operators in order to construct some new consistencies

Andreï Legtchenko

Université d'Orléans – LIFO
BP 6759 – F-45067 Orléans – France

Abstract. What makes a good consistency ? Depending on the constraint, it may be a good pruning power or a low computational cost. By "weakening" arc-consistency, we propose to define new automatically generated solvers which form a sequence of consistencies weaker then arc-consistency. The method exploits on some form of regularity in the cloud of constraint solutions. This approach is illustrated on the constraints $word_n(X_1, X_2, .., X_n)$ and crossword CSP, where interesting speed-up are achieved.

1 Introduction

Since their introduction [7], CSP consistencies have been recognized as one of the most powerful tool to strengthen search mechanisms. Since then, their considerable pruning power has motivated a lot of efforts to find new consistencies and to improve the algorithms to compute them.

Consistencies can be partially ordered according to their pruning power. However, this pruning power should be put into balance with the complexity of enforcing them. For example, path-consistency is often not worth it: its pruning power is great, but the price to pay is high. Maintaining path-consistency during search is thus often beaten in practice by weaker consistencies. Similarly, on many useful CSPs, bound-consistency is faster than arc-consistency even if it does not dig holes in the variables domains: this is left to the search mechanism ensuring the completeness of constraint solving.

Recently, it has been shown that consistencies can be built automatically using machine learning techniques. In [2], a consistency weaker than bound-consistency but as close to it as possible was constructed. The method described in this paper allows to build a full range of comparable consistencies for a given constraint. These consistencies are weaker and sometimes quicker then arc-consistency.
The paper is structured as follows:

– *A framework to express consistencies.* Consistencies are usually built as global CSP properties. But it is now rather common to express them modularly by the greatest fixpoint of a set of operators associated to the constraints, which is computed using a chaotic iteration [1]. We present a framework which allows, starting from arbitrary operators, to progressively add properties in order to build a consistency.

– *A consistency construction method.* For a given constraint, we express the arc-consistency as a set of particular reduction operators. These operators do not have the same computational cost. Expensive ones are delayed until instantiation of all variables. The closure of all the operators defines a new consistency for a given constraint, weaker but quicker than arc-consistency.
– *An example.* The interest of these consistencies is shown by the constraints "words" and the crossword CPSs.

2 Consistency as an operator

Let V be a set of variables and $D = (D_X)_{X \in V}$ their (finite) domains. For $W \subseteq V$, we denote by D^W the set of tuples on W, namely $\Pi_{X \in W} D_X$. Projection of a tuple or a set of tuples on a variable or a set of variables is denoted by $|$, natural join of two sets of tuples is denoted by $\bowtie$. If A is a set, then $\mathcal{P}(A)$ denotes its powerset and $|A|$ its cardinal.

Definition 1 (Constraint). *A* constraint *c is a pair (W, T) where:*

– *$W \subseteq V$ is the* arity *of the constraint c and is denoted by $var(c)$.*
– *$T \subseteq D^W$ is the set of* solutions *of c and is denoted by $sol(c)$.*

The *join* of two constraints is defined as a natural extension of the join of tuples: $c \bowtie c' = (var(c) \cup var(c'), sol(c) \bowtie sol(c'))$.

A *CSP* is a set of constraints. Join is naturally extended to CSPs and the *solutions* of a CSP C are $sol(\bowtie C)$. A direct computation of this join is too expensive to be tractable, especially when considering that it needs to represent tuples of the CSP's arity. This is why a framework based on approximations is preferred, the most successful of them being the domain reduction scheme where variable domains are the only reduced constraints (see [1, 4] for a more general framework). So, for $W \subseteq V$, a search state consists in a set of yet possible values for each variable: $s_W = (s_X)_{X \in W}$ such that s_X is a subset of D_X. The search space is $S_W = \Pi_{X \in W} \mathcal{P}(D_X)$. The set S_W, ordered by pointwise inclusion $\subseteq$, is a complete lattice. Likewise, union and intersection on search states are defined pointwise. The whole search space S_V is simply denoted by S.

Some search states we call *singletonic* play a special role in our framework. A singletonic search state comprises a single value for each variable, and hence represents a single tuple. A tuple is promoted to a singletonic search state by the operator $\lceil\ \rceil$: for $t \in D^W$, let $\lceil t \rceil = (\{t_X\})_{X \in W} \in S_W$. This notation is extended to a set of tuples: for $E \subseteq D^W$, let $\lceil E \rceil = \{\lceil t \rceil \mid t \in E\} \subseteq S_W$. Conversely, a search state is converted into the set of tuples it represents by taking its cartesian product Π : for $s \in S_W$, $\Pi s = \Pi_{X \in W} s_X \subseteq D^W$. We denote by $Sing_W$ the set $\lceil D^W \rceil$ of singletonic search states. By definition, $\lceil D^W \rceil \subseteq S_W$.

A consistency is generally described by a property $Cons \subseteq S$ which holds for certain search states and is classically modeled by the common greatest fixpoint of a set of operators associated to the constraints. By extension, in this paper, we call *consistency* for a constraint c an operator on S_W having some properties

which are introduced in the rest of this section. Let f be an operator on S_W. We denote by $Fix(f)$ the set of fixpoints of f which define the set of *consistent states* according to f.

For $W \subseteq W' \subseteq V$, an operator f on S_W can be *extended* to f' on $S_{W'}$ by taking: $\forall s \in S_{W'}, f'(s) = s'$ with $\forall X \in W' \backslash W, s'_X = s_X$ and $\forall X \in W, s'_X = f(s|_W)_X$. Then $s \in Fix(f') \Leftrightarrow s|_W \in Fix(f)$. This extension is useful for the operator to be combined with others at the level of a CSP.

In order for an operator to be related to a constraint, we need to ensure that it is contracting and that no solution tuple could be rejected anywhere in the search space. An operator having such property is called a *preconsistency*:

Definition 2 (Preconsistency). *An operator $f : S_W \rightarrow S_W$ is a* preconsistency *for $c = (W, T)$ if:*

- *f is monotonic, i.e. $\forall s, s' \in S_W, s \subseteq s' \Rightarrow f(s) \subseteq f(s')$.*
- *f is contracting, i.e. $\forall s \in S_W, f(s) \subseteq s$.*
- *f is correct, i.e. $\forall s \in S_W, \Pi s \cap sol(c) \subseteq \Pi f(s) \cap sol(c)$.*

In the last property, the second inclusion is also called *correctness* of the operator with respect to the constraint; it means that if a state contains a solution tuple, this one will not be eliminated by consistency. Since a preconsistency is also contracting, this inclusion is actually also an equality. This notion of preconsistency is interesting in the context of CSP resolution. These operators can be included in chaotic iteration, because of their properties.

An operator on S_W is *associated* to a constraint $c = (W, T)$ if its singletonic fixpoints represent the constraint's solution tuples T:

Definition 3 (Associated Operator). *An operator $f : S_W \rightarrow S_W$ is* associated *to a constraint c if $Fix(f) \cap Sing_W = \lceil sol(c) \rceil$*

However, nothing is said about its behavior on non-singletonic states. This property is also called *singleton completeness*. Note that a preconsistency is not automatically associated to its constraint since the set of its singletonic fixpoints may be larger. When it coincides, we call such an operator a *consistency*:

Definition 4 (Consistency). *An operator f is a* consistency *for c if it is associated to c and it is a preconsistency for c.*

Note that a consistency can be viewed as an extension to S_W of the satisfiability test made on singletonic states. Consistency operators can be easily scheduled by a chaotic iteration algorithm [1]. By the singleton completeness property, the consistency check for a candidate tuple can be done by the propagation mechanism itself. Let $C = \{c_1, \ldots, c_n\}$ be a CSP and $F = \{f_1, \ldots, f_n\}$ be a set of consistencies on S associated respectively to $\{c_1, \ldots, c_n\}$. If all constraints are not defined on the same set of variables, it is always possible to use the extension of the operators on the union of all variables which appear in the constraints. The common closure of the operators of F can be computed by a chaotic iteration [1]. It follows from the main confluence theorem of chaotic iterations that a consistency can be constituted by combining the mutual strengths of several

operators. Since we have $Fix(F) = \bigcap_{f \in F} Fix(f)$ and since each consistency does preserve the tuples of its associated constraint, the computed closure of all operators associated to the CSP C does not reject a tuple of $c_1 \bowtie \ldots \bowtie c_n$ for any search state $s \in S$ because of an operator of F.

Proposition 5 (Composition). *The composition of two preconsistencies for a constraint c via chaotic iteration is still a preconsistency for c. We call* $\circ$ *this composition.*

The proof is straightforward by [1]. The same property holds for consistencies instead of preconsistencies. Let us now define some consistencies associated to a constraint c:

- ID_c is a family of contracting operators such that any $id_c \in ID_c$ verify: $\forall s \in S_W \backslash Sing_W$, $id_c(s) = s$ and $\forall s \in Sing_W, s \in \lceil sol(c) \rceil \Leftrightarrow id_c(s) = s$. In particular, on non-solution singletonic states, id_c reduces at least one variable's domain to $\emptyset$. The non-uniqueness of id_c comes from the fact that all search states s such that $\Pi s = \emptyset$ represent the empty set of solution for a constraint. In the following, we denote by id_c any member of ID_c.
- ac_c is the well-known arc-consistency operator defined by $\forall s \in S_W$, $ac_c(s) = ((sol(c) \cap \Pi s)_X)_{X \in W}$.

We now suppose that each variable domain D_X is equipped with a total ordering $\leq$. The notation $[a..b]$ is used for the classical notion of interval $\{e \in D_X \mid a \leq e \leq b\}$. We call Int_X the interval lattice built on D_X and for $W \subseteq V$, $Int_W = \Pi_{X \in W} Int_X$. For $A \subseteq D_X$, we denote by $[A]$ the set $[\min(A)..\max(A)]$. We extend this notation for any search state: $\forall s \in S_W, [s] = \Pi_{X \in W}[s_X]$. Bound-consistency consists in contracting only the bounds of a variable's domain, represented as an interval:

- bc_c : is the bound-consistency operator defined by $bc_c(s) = [((sol(c) \cap \Pi s)_X)_{X \in W}]$.

3 Delaying "big" operators

Powerful consistencies are not always the best choice. It is sometimes more interesting to find the optimal ratio between the advantage of pruning and the computational cost needed to enforce it. We consider a constraint, defined in extension by the set of its solutions. By using machine learning techniques, we construct a set of operators. The closure of these operators by chaotic iteration defines a consistency for given constraint. The used techniques may be clustering [6], genetic algorithms [5] or other optimisation algorithms. In [5], new consistencies for a given constraint are obtained by approximation (in the sense of function approximation) of the bound-consistency. At first, a fixed form of expression for operator is chosen, and some coefficients of this expression are computed by a genetic algorithm, in order to mimic the behaviour of the bound-consistency. In [6], solutions of the given constraint are mapped in a multi-dimensional numerical space. Then, by using a clustering algorithm we split this space in some blocks. Finally, we compute a consistency by considering these blocks of solutions instead of each solution separately. In this paper we use an another approach,

inspired by the pruning of decision trees. In any case, resulting consistencies are weaker than arc-consistency, and often well adapted for numerous constraints and CSPs.

In the beginning, we express the computation of arc-consistency with a set of special *elementary reduction functions*. Then we weaken the arc-consistency. It is done by delaying too expensive functions until instantiation of all variables. So the reduction power decreases, but the computation becomes quicker. By the choice of the cost threshold, we define a sequence of comparable consistencies.

Preliminaries. Let us recall the definition of arc-consistency from section 2: $\forall s \in S_W,\ ac_c(s) = ((sol(c) \cap \Pi s)_X)_{X \in W}$.

Definition 6 (Support). *Let $c = (W, T)$ be a constraint. Let $X \in W$ and $a \in D_X$. We call support of "$X = a$" a tuple $t \in sol(c)$ such that $t_X = a$.*

We call $T_{X=a} \subseteq sol(c)$ the set of all supports of $X = a$. A value a has to be maintained in the current domain of X only if we have at least one $t \in T_{X=a}$ that all projections on $Y \in W \backslash \{X\}$ are included in s_Y.

Definition 7 (Supported value). *Let $c = (W, T)$ be a constraint, $X \in W$ and $a \in D_X$. We call $\mathtt{Supp}_{X=a}(s)$ the following property: $\bigvee_{t \in T_{X=a}} (\bigwedge_{Y \in W \backslash \{X\}} t_Y \in s_Y)$. $X = a$ is supported if $\mathtt{Supp}_{X=a}(s) = true$.*

If $X = a$ is not supported, a does not participate to any solution of the CSP and therefore can be eliminated. With this notion, we define a set of functions which return the values which have to be eliminated. Each value in the initial domain of each variable has its own elementary reduction function. If a value must be eliminated, its function returns this value as a singleton, and the empty set otherwise.

Definition 8 (Elementary reduction function). *For all $X \in W$ and for all $a \in D_X$, we define a function $r_{X=a} : S_W \longrightarrow \mathcal{P}(D_X)$ by*

$$\forall s \in S_W,\ r_{X=a}(s) = \begin{cases} \{a\},\ if\ \neg \mathtt{Supp_{X=a}(s)} \\ \emptyset,\ else. \end{cases}$$

Now arc-consistency can be defined using elementary reduction functions as follows: $\forall s \in S_W,\ ac_c(s) = (s_X \backslash \bigcup_{a \in D_X} r_{X=a}(s))_{X \in W}$.

Weakening arc-consistency. We want a consistency quicker then arc-consistency, even if it is less powerful. Now we split arc-consistency into two operators:

Definition 9 (Operator Small). *Let c be a constraint, and n an integer. The operator $\mathtt{Small}_c(n)$: $S_W \longrightarrow S_W$ is:*

$$\forall s \in S_W,\ \mathtt{Small}_c(n)(s) = (s_X \backslash \bigcup_{a \in D_X,\ |T_{X=a}| \leq n} r_{X=a}(s))_{X \in W}.$$

Definition 10 (Operator Big). *Let c be a constraint and n an integer. The operator $\texttt{Big}_c(n) : S_W \longrightarrow S_W$ is:*

$$\forall s \in S_W,\ \texttt{Big}_c(n)(s) = \begin{cases} (s_X \setminus \bigcup_{a \in D_X,\ |T_{X=a}|>n} r_{X=a}(s))_{X \in W},\ if\ s \in Sing_W \\ s,\ else. \end{cases}$$

Note that $\texttt{Big}_c(n)$ does not reduce non-singletonic states. This operator is useful only to reject non-solution tuples.

Proposition 11. *For all integer n, $\texttt{Small}_c(n)$ is a preconsistency for c.*

Proof. First, we show that $\texttt{Small}_c(n)$ is monotonic. Let s and s' in S_W such that $s \subseteq s'$. Then, $\forall X \in W$, $\forall a \in D_X$, $\texttt{Supp}_{X=a}(s) \Rightarrow \texttt{Supp}_{X=a}(s')$. From which it follows that $\forall X \in W$, $\forall a \in D_X$, $\neg\texttt{Supp}_{X=a}(s') \Rightarrow \neg\texttt{Supp}_{X=a}(s)$. In the case of s', there are less values to eliminate than in the case of s. So $\texttt{Small}_c(n)(s) \subseteq \texttt{Small}_c(n)(s')$. The operator $\texttt{Small}_c(n)$ is monotonic.

$\texttt{Small}_c(n)$ is contracting by construction. It it also correct by construction: it eliminates less values than arc-consistency. Hence $\texttt{Small}_c(n)$ is a preconsistency.

The operator $\texttt{Big}_c(n)$ is also a preconsistency, the proof is similar to $\texttt{Small}_c(n)$.

Proposition 12. *Let $c = (W, T)$ a constraint and n an integer. The composition $\texttt{Small}_c(n) \circ \texttt{Big}_c(n)$ is a consistency for c .*

Proof. According to the proposition 5, $\texttt{Small}_c(n) \circ \texttt{Big}_c(n)$ is a preconsistency. But $\forall s \in Sing_W$, $\texttt{Small}_c(n) \circ \texttt{Big}_c(n)(s) = ac_c(s)$, so $\texttt{Small}_c(n) \circ \texttt{Big}_c(n)$ is associated to c. Therefore $\texttt{Small}_c(n) \circ \texttt{Big}_c(n)$ is a consistency for c.

Elementary reduction functions of $\texttt{Big}_c(n)$ are not fired on the non-singletonic states. If the threshold n is not too big, the set of elementary reduction functions of $\texttt{Big}_c(n)$ is not empty. In that case, consistency $\texttt{Small}_c(n) \circ \texttt{Big}_c(n)$ is weaker than arc-consistency, but it can be computed quickly, since that we have less functions to evaluate. If n is big enough, we have the same reduction power and the computational cost than arc-consistency.

Implementation. A system generating the operators $\texttt{Small}_c(n)$ and $\texttt{Big}_c(n)$ has been implemented. The language used to express the operators is the indexical language of GNU-Prolog [3]. An indexical operator is written "`X in r`" where `X` is the name of a variable, and `r` is the range of possible values for `X` and which may depend on other variables' current domains. If we call x the current domain of X, the indexical `X in r` can be read declaratively as the second-order constraint $x \subseteq r$ or operationally as the operator $x \mapsto x \cap r$. For a given constraint and a threshold, our system returns two sets of $|W|$ indexical operators, i.e. one for each variable. The first set defines the $\texttt{Big}_c(n)$ operator, and the second $\texttt{Small}_c(n)$. In total, we have $2 * |W|$ indexical operators for a constraint $c = (W, T)$. The closure by chaotic iteration of all $2*|W|$ operators is equivalent to the consistency $\texttt{Small}_c(n) \circ \texttt{Big}_c(n)$. The indexical operators for $\texttt{Big}_c(n)$ are delayed with the special trigger `val` which delays the operator until the variable is instantiated. This is why these operators are not iterated on non-singletonic search states.

Example 13. Let `c(X,Y,Z)` be a constraint. $D_X = D_Y = \{0, 1\}$ and $D_Z = \{0, 1, 2\}$.

c :

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1
1	1	2
0	1	2

The sets of supports for Z are:
$T_{Z=0} = \{(0,0,0),(0,1,0),(1,0,0)\}$
$T_{Z=1} = \{(1,1,1)\}$
$T_{Z=2} = \{(1,1,2),(0,1,2)\}$

So, for all $s \in S_{\{X,Y,Z\}}$, we have:

$$\texttt{Supp}_{Z=0}(s) = (0 \in s_X \wedge 0 \in s_Y) \vee (0 \in s_X \wedge 1 \in s_Y) \vee (1 \in s_X \wedge 0 \in s_Y)$$
$$\texttt{Supp}_{Z=1}(s) = 1 \in s_X \wedge 1 \in s_Y$$
$$\texttt{Supp}_{Z=2}(s) = (1 \in s_X \wedge 1 \in s_Y) \vee (0 \in s_X \wedge 1 \in s_Y).$$

For the variable Z, we can make two indexical operators following the definition of $\texttt{Small}_c$ and $\texttt{Big}_c$. Let set the threshold to 2. The operators for Z are:

$$\texttt{Small}_c(2)_Z\colon s_Z \longrightarrow s_Z \backslash (r_{Z=1}(s) \cup r_{Z=2}(s))$$
$$\texttt{Big}_c(2)_Z\colon s_Z \longrightarrow s_Z \backslash r_{Z=0}(s)$$

The operator $\texttt{Big}_c(2)_Z$ is delayed because its reduction power is small (only one value can be eliminated), and the effort to compute $r_{Z=0}(s)$ is contingently high. This operator is fired only if $s \in Sing_{\{X,Y,Z\}}$.

4 Example

Our method exploits some form of regularity in the cloud of constraint solutions. We suppose that all domains are provided with a total ordering. Hence we can map all the solution tuples in a $|W|$-dimensioned numerical space. Let $c = (W, T)$ be a constraint, let $X \in W$ and $a \in D_X$. The size of the body of the elementary reduction function $r_{X=a}$ is proportional to the number of solutions in the hyperplane orthogonal to the axis of X and which pass by $X = a$. If there are many solutions of c in this hyperplane, it is probable that a will not be eliminated from the domain of X. Moreover, the test is expensive to compute. So, the computation is delayed.

This distribution occurs, for example, in the case of the constraints $word_n(X_1, X_2, .., X_n)$. For $n = 3$, the constraint $word_3(X, Y, Z)$ means that XYZ is a 3-letters English word. We consider that domains are ordered by lexicographic ordering. Figure 1 illustrates projections of $word_3(X, Y, Z)$ on different planes. When using UNIX dictionary /usr/dict/word, this constraint has 576 solutions. The constraints $word_4(X, Y, Z, U)$ (with 2236 solutions) and $word_5(X, Y, Z, U, V)$ (with 4176 solutions) have the same regularity.

The CSP we use consists in finding a solution for crossword grids of different sizes. The CSP is composed only by a set of constraints $word_n$. The domain of all variables is $\{a, .., z\}$. An example of a 7x7 grid and its model is presented in figure 2 and table 1. Only the first solution is computed. Some benchmarks are presented in the tables 2, 3, 4. The full reduction power of arc-consistency is obtained from the following thresholds: 8 for $word_2$, 130 for $word_3$, 500 for $word_4$, 1200 for $word_5$. It means that with these thresholds (and higher), the $\texttt{Big}_c(n)$

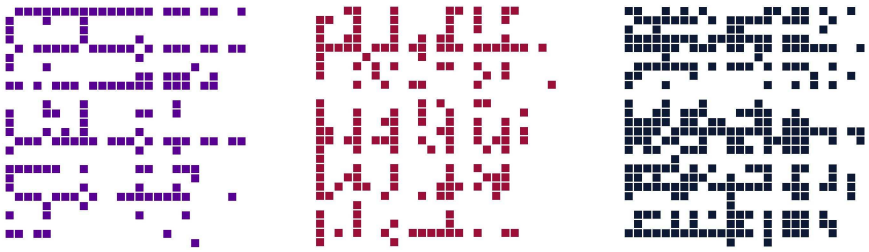


Fig. 1. Projections of $word_3(X, Y, Z)$ on the planes XY, YZ, XZ

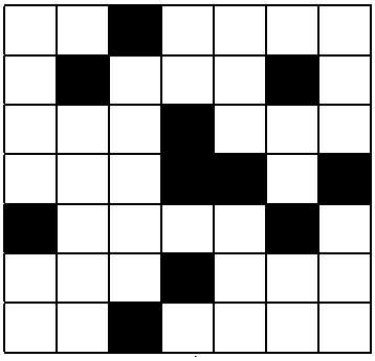


Fig. 2. A 7x7 grid

operator is empty. The `fd_relation` time is the computation time with the built-in GNU-Prolog predicate implementing arc-consistency for a constraint given by the table of its solutions. These new consistencies show an interesting speed-ups, from 1.68 to 4.8. But optimal thresholds are found empirically, and they are different from one grid to another. Thus, this method can be considered only as an additional tool in the resolving environnement. The main contribution of this approach is to show the interest of using constraint regularities to construct efficient operators.

5 Conclusion

In this paper, we propose a new method which allows to build a full range of consistencies weaker but quicker than arc-consistency. In this approach, we exploit a form of regularity in the constraint solutions to construct a set of operators. The operators which are too expensive to compute are delayed, so their closure

$X_{1,1}$	$X_{1,2}$	□	$X_{1,4}$	$X_{1,5}$	$X_{1,6}$	$X_{1,7}$
$X_{2,1}$	□	$X_{2,3}$	$X_{2,4}$	$X_{2,5}$	□	$X_{2,7}$
$X_{3,1}$	$X_{3,2}$	$X_{3,3}$	□	$X_{3,5}$	$X_{3,6}$	$X_{3,7}$
$X_{4,1}$	$X_{4,2}$	$X_{4,3}$	□	□	$X_{4,6}$	□
□	$X_{5,2}$	$X_{5,3}$	$X_{5,4}$	$X_{5,5}$	□	$X_{5,7}$
$X_{6,1}$	$X_{6,2}$	$X_{6,3}$	□	$X_{6,5}$	$X_{6,6}$	$X_{6,7}$
$X_{7,1}$	$X_{7,2}$	□	$X_{7,4}$	$X_{7,5}$	$X_{7,6}$	$X_{7,7}$

Table 1. A model for 7x7 grid

Threshold for				**Time**
$word_2$	$word_3$	$word_4$	$word_5$	
3	70	240	510	45ms
8	30	50	10	280ms
1	5	300	10	11ms
3	6	400	10	16ms
1	5	200	10	120ms
`fd_relation`				53ms

Table 2. Some results for 7x7 grid

Threshold for				**Time**
$word_2$	$word_3$	$word_4$	$word_5$	
3	70	240	310	380ms
3	30	80	200	>15min
3	70	240	410	58ms
3	70	260	410	76ms
3	65	240	410	78ms
`fd_relation`				102ms

Table 3. Some results for 10x10 grid

Threshold for				**Time**
$word_2$	$word_3$	$word_4$	$word_5$	
1	5	50	310	1h
3	130	500	600	193ms
3	50	240	510	6490ms
3	5	240	310	5490ms
3	70	260	510	160ms
3	100	240	510	160ms
3	130	300	510	170ms
1	70	240	510	150ms
`fd_relation`				252ms

Table 4. Some results for 15x15 grid

of by a chaotic iteration defines a new consistency, weaker but quicker than arc-consistency. The interest of this method is illustrated on crossword CSPs.

Acknowledgements. I would like to thank Abdel Ali Ed-Dbali for his help in testing crossword CSP, and also the rest of the Solar Team.

References

1. K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.
2. Thi-Bich-Hanh Dao, Arnaud Lallouet, Andrei Legtchenko, and Lionel Martin. Indexical-based solver learning. In Pascal van Hentenryck, editor, *International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *LNCS*, pages 541–555, Ithaca, NY, USA, Sept. 7 - 13 2002. Springer.
3. Daniel Diaz and Philippe Codognet. Design and implementation of the Gnu-Prolog system. *Journal of Functional and Logic Programming*, 2001(6), 2001.
4. Arnaud Lallouet, Thi-Bich-Hanh Dao, and AbdelAli Ed-Dbali. Language, definition and optimal computation of CSP approximations. In Susan Haller and Ingrid Russell, editors, *Flairs'03, International Florida Artificial Intelligence Conference*, St Augustine, FL, USA, 2003. AAAI Press.
5. Arnaud Lallouet, Thi-Bich-Hanh Dao, Andreï Legtchenko, and AbdelAli Ed-Dbali. Finite domain constraint solver learning. In Georg Gottlob, editor, *International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, 2003. AAAI Press. poster.
6. Arnaud Lallouet, Andreï Legtchenko, Thi-Bich-Hanh Dao, and AbdelAli Ed-Dbali. Intermediate (learned) consistencies. In Francesca Rossi, editor, *International Conference on Principles and Practice of Constraint Programming*, LNCS, Kinsale, County Cork, Ireland, 2003. Springer. Poster.
7. Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

Pure Prolog Execution in 21 Rules

M. Kulaš

FernUniversität Hagen, FB Informatik, D-58084 Hagen, Germany
marija.kulas@fernuni-hagen.de

Abstract. A simple mathematical definition of the 4-port model for pure Prolog is given. The model combines the intuition of ports with a compact representation of execution state. Forward and backward derivation steps are possible. The model satisfies a modularity claim, making it suitable for formal reasoning.

1 Introduction

In order to formally handle (specify and prove) some properties of Prolog execution, we needed above all a definition of a port. A port is perhaps the single most popular notion in Prolog debugging, but theoretically it appears still rather elusive. The notion stems from the seminal article of L. Byrd [Byr80] which identifies four different types of control flow in a Prolog execution, as movements in and out of procedure *boxes* via the four *ports* of these boxes:

- *call*, entering the procedure in order to solve a goal,
- *exit*, leaving the procedure after a success, i. e. a solution for the goal is found,
- *fail*, leaving the procedure after the failure, i. e. there are no (more) solutions,
- *redo*, re-entering the procedure, i. e. another solution is sought for.

In this work, we present a formal definition of ports, which is a calculus of execution states, and hence provide a formal model of pure Prolog execution, S:PP. Our approach is to define ports by virtue of their effect, as *port transitions*. A port transition relates two *events*. An event is a state in the execution of a given query Q with respect to a given Prolog program Π. There are two restrictions we make:

1. the program Π has to be pure
2. the program Π shall first be transformed into a canonical form.

The first restriction concerns only the presentation in this paper, since our model has been prototypically extended to cover the control flow of full Standard Prolog, as given in [DEDC96]. The canonical form we use is the common single-clause representation. This representation is arguably 'near enough' to the original program, the only differences concern the head-unification (which is now delegated to the body) and the choices (which are now uniformly expressed as disjunction).

2 Preliminaries and the main idea

First we define the canonical form, into which the original program has to be transformed. Such a syntactic form appears as an intermediate stage in defining the Clark's completion of a logic program, and is used in logic program analysis. However, we are not aware of any consensus upon the name for this form. Some of the names in the literature are *single-clausal form* [Lin95] and *normalisation of a logic program* [KL02]. Here we use the name *canonical form*, partly on the grounds of our imposing a transformation on if-then as well (this additional transformation is of no interest in the present paper, which has to do only with pure Prolog, but we state it for completeness).

Definition 1 (canonical form of a predicate) We say that a predicate P/n is in the canonical form, if its definition consists of a single clause $P(X_1, ..., X_n) :- B; Bs$. Here B is a "canonical body", of the form $X_1{=}T_1, \ldots, X_n{=}T_n$, G, Gs, and $P(X_1, ..., X_n)$ is a "canonical head", i. e. $X_1, ..., X_n$ are distinct variables not appearing in G, Gs, $T_1, ..., T_n$. Further, Bs is a disjunction of canonical bodies (possibly empty), Gs is a conjunction of goals (possibly empty), and G is a goal (for facts: true). Additionally, each if-then goal $A \to B$ must be part of an if-then-else (like $A \to B; \mathit{fail}$). ■

Example 1 (canonical form) For the following program

```
q(a,b).
q(Z,c) :- r(Z).
r(c).
```

we obtain as canonical form

```
q(X,Y) :- X=a, Y=b, true; X=Z, Y=c, r(Z).
r(X) :- X=c, true.
```

❑

Having each predicate represented as one clause, and bearing in mind the box metaphor above, we identified some elementary execution steps. For simplicity we first disregard variables.

The following table should give some intuition about the idea. The symbols α, β in this table serve to identify the appropriate redo-transition, depending on the exit-transition. Transitions are deterministic, since the rules do not overlap.

Term	Port transitions in the context of Term			
$H{:-}B$	$\mathit{call}\ H \rightarrowtriangle \mathit{call}\ B$	$\mathit{exit}\ B \rightarrowtriangle \mathit{exit}\ H$	$\mathit{fail}\ B \rightarrowtriangle \mathit{fail}\ H$	$\mathit{redo}\ H \rightarrowtriangle \mathit{redo}\ B$
A,B	$\mathit{call}\ A,B \rightarrowtriangle \mathit{call}\ A$	$\mathit{exit}\ A \rightarrowtriangle \mathit{call}\ B$	$\mathit{fail}\ A \rightarrowtriangle \mathit{fail}\ A,B$	$\mathit{redo}\ A,B \rightarrowtriangle \mathit{redo}\ B$
		$\mathit{exit}\ B \rightarrowtriangle \mathit{exit}\ A,B$	$\mathit{fail}\ B \rightarrowtriangle \mathit{redo}\ A$	
$A;B$	$\mathit{call}\ A;B \rightarrowtriangle \mathit{call}\ A$	$\mathit{exit}\ A \rightarrowtriangle {}^{\alpha}\mathit{exit}\ A;B$	$\mathit{fail}\ A \rightarrowtriangle \mathit{call}\ B$	${}^{\alpha}\mathit{redo}\ A;B \rightarrowtriangle \mathit{redo}\ A$
		$\mathit{exit}\ B \rightarrowtriangle {}^{\beta}\mathit{exit}\ A;B$	$\mathit{fail}\ B \rightarrowtriangle \mathit{fail}\ A;B$	${}^{\beta}\mathit{redo}\ A;B \rightarrowtriangle \mathit{redo}\ B$
true	$\mathit{call}\ \mathsf{true} \rightarrowtriangle \mathit{exit}\ \mathsf{true}$			$\mathit{redo}\ \mathsf{true} \rightarrowtriangle \mathit{fail}\ \mathsf{true}$
fail	$\mathit{call}\ \mathsf{fail} \rightarrowtriangle \mathit{fail}\ \mathsf{fail}$			

Table 1. The idea of port transitions

Remark 1 (general goals) Observe that we extend the notion of a port, initially conceived for predicates, to *general goals*. The shifting of attention from predicates to goals is the key idea of this approach. □

Notation 1 (distinguishing meta-level from object-level) In the following we show object-level terms (i. e. actual Prolog terms) in sans serif, like true. Meta-level terms (i. e. anything else in the calculus) will be shown in italics, like call,Σ, or in blackboard font, like , . □

Each transition pertains to a certain context, as indicated in Table 1. In the next step towards the new definition of ports we shall make this dependency explicit, by adding a parameter to each event.

Example 2 (good, bad and main) Relative to the program

```
main :- good, bad.
good.
```

there are the following execution steps for main:

```
call main  ⇾
  call (good, bad)  ⇾
    call good  ⇾
      call true  ⇾
      exit true  ⇾
    exit good ⇾
    call bad  ⇾
    fail bad  ⇾
    redo good  ⇾
      redo true  ⇾
      fail true  ⇾
    fail good  ⇾
  fail (good, bad)  ⇾
fail main
```

The indentations should suggest the context of the transitions, which is not very satisfying, since we want our representation to be entirely symbolic, and therefore visual aspects may not be part of the definition. So we provide the context information within the calculus, by means of a *stack of ancestors*, or A-stack. Hereby we define the immediate ancestor (the parent) of a goal to be the context of the transition. On some reflection, this is not enough. In case of a redo of an atomary goal, like *redo* true above, we need to know how the goal was resolved, in order to see the remaining alternatives. Since it is possible, in full Prolog, that a predicate definition changes between an exit and a redo, simply accessing the program would not guarantee the retrieval of the definition effective at the time of call. For this reason we memorize, at an exit of an atomary goal, the effectively used definition (more about this on page 6). Also, on exit from a disjunction, some kind of memoing of the used disjunct is necessary. So we tried combining the memoing (both kinds of memos: used definitions and used disjuncts) with the administration of variable bindings, into one *stack of bets*, or B-stack. One claim of this paper is that an A-stack and a B-stack are sufficient to represent the execution of pure Prolog. As an illustration of the two-stack idea, let us show the above derivation in complete detail. In Appendix B an example with variables is given. Each stack is enclosed in parentheses, • separates the elements, and *nil* marks the bottom of a stack.

$$
\begin{aligned}
&\mathit{call}\ \mathsf{main}\,,\ \{\mathsf{top}\},\ \{\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{call}\ (\mathsf{good},\mathsf{bad})\,,\ \{\mathsf{main}\bullet\mathit{nil}\},\ \{\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{call}\ \mathsf{good}\,,\ \{1/\mathsf{good},\mathsf{bad}\bullet\mathsf{main}\bullet\mathit{nil}\},\ \{\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{call}\ \mathsf{true}\,,\ \{\mathsf{good}\bullet 1/\mathsf{good},\mathsf{bad}\bullet\mathsf{main}\bullet\mathit{nil}\},\ \{\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{exit}\ \mathsf{true}\,,\ \{\mathsf{good}\bullet 1/\mathsf{good},\mathsf{bad}\bullet\mathsf{main}\bullet\mathit{nil}\},\ \{\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{exit}\ \mathsf{good}\,,\ \{1/\mathsf{good},\mathsf{bad}\bullet\mathsf{main}\bullet\mathit{nil}\},\ \{BY(\mathsf{true},\mathsf{good})\bullet\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{call}\ \mathsf{bad}\,,\ \{2/\mathsf{good},\mathsf{bad}\bullet\mathsf{main}\bullet\mathit{nil}\},\ \{BY(\mathsf{true},\mathsf{good})\bullet\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{fail}\ \mathsf{bad}\,,\ \{2/\mathsf{good},\mathsf{bad}\bullet\mathsf{main}\bullet\mathit{nil}\},\ \{BY(\mathsf{true},\mathsf{good})\bullet\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{redo}\ \mathsf{good}\,,\ \{1/\mathsf{good},\mathsf{bad}\bullet\mathsf{main}\bullet\mathit{nil}\},\ \{BY(\mathsf{true},\mathsf{good})\bullet\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{redo}\ \mathsf{true}\,,\ \{\mathsf{good}\bullet 1/\mathsf{good},\mathsf{bad}\bullet\mathsf{main}\bullet\mathit{nil}\},\ \{\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{fail}\ \mathsf{true}\,,\ \{\mathsf{good}\bullet 1/\mathsf{good},\mathsf{bad}\bullet\mathsf{main}\bullet\mathit{nil}\},\ \{\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{fail}\ \mathsf{good}\,,\ \{1/\mathsf{good},\mathsf{bad}\bullet\mathsf{main}\bullet\mathit{nil}\},\ \{\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{fail}\ (\mathsf{good},\mathsf{bad})\,,\ \{\mathsf{main}\bullet\mathit{nil}\},\ \{\mathit{nil}\}\\
&\ \rightarrowtriangle\ \mathit{fail}\ \mathsf{main}\,,\ \{\mathit{nil}\},\ \{\mathit{nil}\}
\end{aligned}
$$

❑

3 The calculus S:PP

We consider pure Prolog programs as given in Fig. 1, syntax domain "program", under restriction that every "definition" has to be in the canonical form.

Definition 2 (event) An *event* is a quadruple $(Port, Goal, A\text{-}stack, B\text{-}stack)$, as given by the grammar in Fig. 1, syntax domain "event". ■

Intuitively, an event is a state of Prolog execution, determined by four parameters:

- port
- current goal
- history of current goal (stack of generalized ancestors, for short: *A-stack)*
- current environment (stack of generalized bindings, *bets*, for short: *B-stack)*

Definition 3 (transition rule) Let Π be a program. Port transition rules wrt Π are listed in Fig. 2. ■

event ::= **port goal** $\langle \frac{\textbf{stack of bets}}{\textbf{stack of ancestors}} \rangle$

event ::= **port goal , {stack of ancestors}, {stack of bets}** % inline

definition ::= atom :− goal
program ::= {definition.}$^+$
port ::= *call* | *exit* | *fail* | *redo*
goal ::= true | fail | atom | term=term | goal;goal | goal,goal
ancestor ::= true | fail | atom | term=term | tag/goal;goal | tag/goal,goal
tag ::= 1 | 2
memo ::= $BY(\text{goal}, \text{atom})$ | $OR(\text{goal}, (\text{tag/goal;goal}))$
bet ::= mgu | memo
stack of Xs ::= *nil* | X • stack of Xs

Variables

,	: stack of ancestors,	U	: ancestor
,	: stack of bets,	Σ	: bet
σ	: substitution		
A, B, C, G, H	: goal		
G_A	: atom		
T	: term		

Semantic functions

$T_1 = T_2$:= T_1 and T_2 are identical
$\sigma(T)$ = application of σ upon T
mgu(T_1, T_2) = mgu of T_1 and T_2
substOf( ) = current substitution = composition of all mgus from

substOf(*nil*)(T) := T

$$\text{substOf}(\Sigma \bullet\ )(T) := \begin{cases} \Sigma(\text{substOf}(\ )(T)), & \text{if } \Sigma \text{ is an mgu} \\ \text{substOf}(\ )(T), & \text{if } \Sigma \text{ is a memo} \end{cases}$$

Syntactic domains that we do not redefine, but take in their usual sense:
term (taken in the Prolog sense, as a superset of goal);
atom (atomary goal in logic programming);
substitution, mgu.

Fig. 1. Language of events

Conjunction

$$call\ A, B\ \langle - \rangle \ \rightarrow\!\!\!\!\triangleright\ call\ A\ \langle \overline{1/A,B\bullet} \rangle \qquad \text{(S:conj:1)}$$

$$exit\ A'\ \langle \overline{1/A,B\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ call\ B''\ \langle \overline{2/A,B\bullet} \rangle,\ \text{with } B'' := \text{substOf}(\ \ )(B) \qquad \text{(S:conj:2)}$$

$$fail\ A'\ \langle \overline{1/A,B\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ fail\ A, B\ \langle - \rangle \qquad \text{(S:conj:3)}$$

$$exit\ B'\ \langle \overline{2/A,B\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ exit\ A, B\ \langle - \rangle \qquad \text{(S:conj:4)}$$

$$fail\ B'\ \langle \overline{2/A,B\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ redo\ A\ \langle \overline{1/A,B\bullet} \rangle \qquad \text{(S:conj:5)}$$

$$redo\ A, B\ \langle - \rangle \ \rightarrow\!\!\!\!\triangleright\ redo\ B\ \langle \overline{2/A,B\bullet} \rangle \qquad \text{(S:conj:6)}$$

Disjunction

$$call\ A; B\ \langle - \rangle \ \rightarrow\!\!\!\!\triangleright\ call\ A\ \langle \overline{1/A;B\bullet} \rangle \qquad \text{(S:disj:1)}$$

$$fail\ A\ \langle \overline{1/A;B\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ call\ B\ \langle \overline{2/A;B\bullet} \rangle \qquad \text{(S:disj:2)}$$

$$fail\ B\ \langle \overline{2/A;B\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ fail\ A; B\ \langle - \rangle \qquad \text{(S:disj:3)}$$

$$exit\ A\ \langle \overline{1/A;B\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ exit\ A; B\ \langle \overline{OR(A,(1/A;B))\bullet} \rangle \qquad \text{(S:disj:4)}$$

$$exit\ B\ \langle \overline{2/A;B\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ exit\ A; B\ \langle \overline{OR(B,(2/A;B))\bullet} \rangle \qquad \text{(S:disj:5)}$$

$$redo\ A; B\ \langle \overline{OR(C,(N/A;B))\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ redo\ C\ \langle \overline{N/A;B\bullet} \rangle \qquad \text{(S:disj:6)}$$

True

$$call\ \mathsf{true}\ \langle - \rangle \ \rightarrow\!\!\!\!\triangleright\ exit\ \mathsf{true}\ \langle - \rangle \qquad \text{(S:true:1)}$$

$$redo\ \mathsf{true}\ \langle - \rangle \ \rightarrow\!\!\!\!\triangleright\ fail\ \mathsf{true}\ \langle - \rangle \qquad \text{(S:true:2)}$$

Fail

$$call\ \mathsf{fail}\ \langle - \rangle \ \rightarrow\!\!\!\!\triangleright\ fail\ \mathsf{fail}\ \langle - \rangle \qquad \text{(S:fail)}$$

Explicit unification

$$call\ T_1{=}T_2\ \langle - \rangle \ \rightarrow\!\!\!\!\triangleright\ \begin{cases} exit\ T_1{=}T_2\ \langle \overline{\sigma\bullet} \rangle, & \text{if } \mathrm{mgu}(T_1, T_2) = \sigma \\ fail\ T_1{=}T_2\ \langle - \rangle, & \text{otherwise} \end{cases} \qquad \text{(S:unif:1)}$$

$$redo\ T_1{=}T_2\ \langle \overline{\sigma\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ fail\ T_1{=}T_2\ \langle - \rangle \qquad \text{(S:unif:2)}$$

User-defined atomary goal G_A

$$call\ G_A\ \langle - \rangle \ \rightarrow\!\!\!\!\triangleright\ \begin{cases} call\ \sigma(B)\ \langle \overline{G_A\bullet} \rangle, & \text{if } H :- B \text{ is a fresh renaming of a} \\ & \quad \text{clause in } \Pi, \text{ and } \mathrm{mgu}(G_A, H) = \sigma, \text{ and } \sigma(G_A) = G_A \\ fail\ G_A\ \langle - \rangle, & \text{otherwise} \end{cases} \qquad \text{(S:atom:1)}$$

$$exit\ B\ \langle \overline{G_A\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ exit\ G_A\ \langle \overline{BY(B,G_A)\bullet} \rangle \qquad \text{(S:atom:2)}$$

$$fail\ B\ \langle \overline{G_A\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ fail\ G_A\ \langle - \rangle \qquad \text{(S:atom:3)}$$

$$redo\ G_A\ \langle \overline{BY(B,G'_A)\bullet} \rangle \ \rightarrow\!\!\!\!\triangleright\ redo\ B\ \langle \overline{G'_A\bullet} \rangle \qquad \text{(S:atom:4)}$$

Fig. 2. Operational semantics S:PP of pure Prolog

3.1 Remarks on the calculus

About event:

- *Current goal* is a generalization of *selected literal*: rather than focusing upon single literals, we focus upon goals.
- *Ancestor* of a goal is defined in a disambiguating manner, via *tags*.
- The notion of *environment* is generalized, to contain following *bets*:
  1. variable bindings,
  2. choices taken (OR-branches),
  3. used predicate definitions.

  Environment is represented by one stack, storing each bet as soon as it is computed. For an event to represent the state of pure Prolog execution, suffices here one environment and one ancestor stack.

About transitions:

- Port transition relation is functional. The same holds for its converse, if restricted on *legal events*, i. e. events that can be reached from an *initial event* of the form $\mathit{call}\ G\ \langle \frac{\mathit{nil}}{\mathit{nil}} \rangle$.
- This uniqueness of legal derivations enables *forward and backward* derivation steps, in the spirit of the Byrd's article.
- *Modularity* of derivation: The execution of a goal can be abstracted like for example $\mathit{call}\ G\ \langle - \rangle \overset{*}{\rightarrowtriangle} \mathit{exit}\ G\ \langle \overset{+}{\text{——}} \rangle$. Notice the same A-stack.

REMARK 2 (ATOMARY GOAL) By *atom* or *atomary goal* we denote only user-defined predications. So true, fail or $T_1 = T_2$ shall not be considered atoms. □

REMARK 3 (MGU) The most general unifiers σ shall be chosen to be idempotent, i. e. $\sigma(\sigma(T)) = \sigma(T)$. □

REMARK 4 (TAGS) The names A' or B' of (S:conj:2)–(S:conj:5) should only suggest that the argument is related to A or B, but the actual retrieval is determined by the tags $\mathsf{1}$ and $\mathsf{2}$, saying that respectively the first or the second conjunct are currently being tried. For example, the rule (S:conj:1) states that the call of A, B leads to the call of A with immediate ancestor $\mathsf{1}/A, B$. This kind of add-on mechanism is necessary to be able to correctly handle a query like A, A where retrieval by unification would get stuck on the first conjunct. □

REMARK 5 (CANONICAL FORM) Note the requirement $\sigma(G_A) = G_A$ in (S:atom:1). Since the clauses are in canonical form, unifying the head of a clause with a goal could do no more than rename the goal. Since we do not need a renaming of the goal, we may fix the mgu to just operate on the clause. □

REMARK 6 (LOGICAL UPDATE VIEW) Observe how (S:atom:2) and (S:atom:4) serve to implement the *logical update view* of Lindholm and O'Keefe [LO87], saying that the definition of a predicate shall be fixed at the time of its call. This is further explained in the following remark. □

REMARK 7 ("LAZY" BINDING) Although we memorize the used predicate definition *on exit*, the definition will be unaffected by exit bindings, because *bindings are applied lazily*: Instead of "eagerly" applying any bindings as they occur (e. g. in $T_1 = T_2$, in resolution or in read), we chose to do this only in conjunction (in rule

(S:conj:2)) and nowhere else. Due to the rules (S:conj:1) and (S:conj:4), the exit bindings shall not affect the predicate definition like e. g. $\mathsf{p(X)} :- \mathsf{q(X)}, \mathsf{r(X)}$.

Also, lazy bindings enable less 'jumpy' trace. A jumpy trace can be illustrated by the following exit event (assuming we applied bindings eagerly):

$$\mathit{exit}\ \mathsf{append([O],B,[O|B])}\,,\ \{2/(\mathsf{[I|B]} = \mathsf{[I|B]}), \mathsf{append([],B,B)} \bullet\ \ \},$$

The problem consists in exiting the goal $\mathsf{append([],B,B)}$ via $\mathsf{append([O],B,[O|B])}$, the latter of course being no instance of the former. By means of lazy binding, we avoid the jumpiness, and at the same time make memoing definitions on exit possible. To ensure that the trace of a query execution shows the correct bindings, an event shall be printed only after the current substitution has been applied to it.

A perhaps more important collateral advantage of lazy binding is that a successful derivation (see Definition 11) can always be abstracted as follows:

$$\mathit{call}\ \mathit{Goal} \overset{*}{\rightarrowtriangle} \mathit{exit}\ \mathit{Goal}$$

even if *Goal* happened to get further instantiated in the course of this derivation. The instantiation will be reflected in the B-stack but not in the goal itself. □

4 Modelling Prolog execution

Definition 4 (port transition relation, converse) Let Π be a program. *Port transition relation* $\rightarrowtriangle$ wrt Π is defined in Fig. 2. The converse relation shall be denoted by $\trianglelefteq$. If $E_1 \rightarrowtriangle E$, we say that E_1 *leads to* E. An event E can be *entered*, if some event leads to it. An event E can be *left*, if it leads to some event. ■

Lemma 1 The relation $\rightarrowtriangle$ is functional, i. e. for each event E there can be at most one event E_1 such that $E \rightarrowtriangle E_1$. ■

Proof: The premisses of the transition rules are mutually disjunct, i. e. there are no critical pairs. □

Example 3 (converse relation) The converse of the port transition relation is not functional, since there may be more than one event leading to the same event:

$$\begin{aligned} \mathit{call}\ T_1{=}T_2\ \langle \tfrac{\mathit{nil}}{\mathit{nil}} \rangle &\rightarrowtriangle \mathit{fail}\ T_1{=}T_2\ \langle \tfrac{\mathit{nil}}{\mathit{nil}} \rangle \\ \mathit{redo}\ T_1{=}T_2\ \langle \tfrac{\sigma \bullet \mathit{nil}}{\mathit{nil}} \rangle &\rightarrowtriangle \mathit{fail}\ T_1{=}T_2\ \langle \tfrac{\mathit{nil}}{\mathit{nil}} \rangle \end{aligned}$$

We could have prevented the ambiguous situation above and made converse relation functional as well, by giving natural conditions on redo-transitions for atomary goal and unification. However, further down it will be shown that, for events that are *legal*, the converse relation is functional anyway. ❑

Definition 5 (derivation) Let Π be a program. Let E_0, E be events. A *Π-derivation of E from E_0*, written as $E_0 \overset{*}{\rightarrowtriangle} E$, is a path from E_0 to E in the port transition relation wrt Π. We say that E can be *reached* from E_0. ■

Definition 6 (initial event, top-level goal) An *initial event* is any event of the form $\mathit{call}\ Q\ \langle \frac{\mathit{nil}}{\mathit{nil}} \rangle$, where Q is a goal. The goal Q of an initial event is called a *top-level goal*, or a *query*. ■

Definition 7 (legal derivation, legal event, execution) Let Π be a program. If there is a goal Q such that

$$call\ Q\ \langle \tfrac{nil}{nil} \rangle \overset{*}{\rightarrowtriangle} E_0 \overset{*}{\rightarrowtriangle} E$$

is a Π-derivation, then we say that $E_0 \overset{*}{\rightarrowtriangle} E$ is a *legal Π-derivation*, E is a *legal Π-event*, and $call\ Q\ \langle \frac{nil}{nil} \rangle \overset{*}{\rightarrowtriangle} E_0$ is a *Π-execution* of the query Q. ■

Definition 8 (final event) A legal event E is a *final* event wrt program Π, if there is no transition $E \rightarrowtriangle E_1$ wrt Π. ■

Definition 9 (parent of goal) If $E = Port\ G\ \langle - \rangle$ is an event, and $\quad = P \bullet$, then we say that P is the *parent* of G. ■

NOTATION 2 (SELECTOR TAGS) Function $\mathrm{Sel}(U)$ is defined as follows:

$$\mathrm{Sel}((1/A, B)) := A,\ \mathrm{Sel}((2/A, B)) := B$$

and analogously for disjunction. □

Definition 10 (push/pop event) Let E be an event with the port $Port$. If $Port$ is one of $call, redo$, then E is a *push* event. If $Port$ is one of $exit, fail$, then E is a *pop* event. ■

Lemma 2 (final event) If E is a legal pop event, and its A-stack is not empty, then $\exists E_1 : E \rightarrowtriangle E_1$ ■

PROOF (SKETCH): According to the rules (see also Appendix A), the possibilities to leave an exit event are:

$$\begin{aligned}
exit\ A'\ \langle \tfrac{}{1/A,B\bullet} \rangle &\rightarrowtriangle call\ B''\ \langle \tfrac{}{2/A,B\bullet} \rangle, \text{ with } B'' := \mathrm{substOf}(\ \ )(B) \\
exit\ B'\ \langle \tfrac{}{2/A,B\bullet} \rangle &\rightarrowtriangle exit\ A, B\ \langle - \rangle \\
exit\ A\ \langle \tfrac{}{1/A;B\bullet} \rangle &\rightarrowtriangle exit\ A; B\ \langle \tfrac{OR(A,(1/A;B))\bullet}{} \rangle \\
exit\ B\ \langle \tfrac{}{2/A;B\bullet} \rangle &\rightarrowtriangle exit\ A; B\ \langle \tfrac{OR(B,(2/A;B))\bullet}{} \rangle \\
exit\ B\ \langle \tfrac{}{G_A\bullet} \rangle &\rightarrowtriangle exit\ G_A\ \langle \tfrac{BY(B,G_A)\bullet}{} \rangle
\end{aligned}$$

These rules state that it is always possible to leave an exit event $exit\ G\ \langle - \rangle$, save for the following two restrictions: The parent goal may not be true, fail or a unification; and if the parent goal P is a disjunction, then there has to hold

$$G = \mathrm{Sel}(P) \tag{1}$$

i. e. it is not possible to leave an event $exit\ A'\ \langle \tfrac{}{1/A;B\bullet} \rangle$ if $A' \neq A$ (and similarly for the second disjunct). The first restriction is void, since a parent cannot be true, fail or a unification anyway, according to the rules. It remains to show that the second restriction is also void, i. e. a legal exit event has necessarily the property (1). Looking at the rules for entering an exit event, we note that the goal part of an exit event either comes from the A-stack, or is true or $T_1 = T_2$. The latter two possibilities we may exclude, because $exit\ \mathsf{true}\ \langle \tfrac{}{1/A;B\bullet} \rangle$ can only be derived from $call\ \mathsf{true}\ \langle \tfrac{}{1/A;B\bullet} \rangle$, which cannot be reached if $\mathsf{true} \neq A$. Similarly for unification.

So the goal part of a legal exit event must come from the A-stack. The elements of the A-stack originate from call/redo events, and they have the property (1). In conclusion, we can always leave a legal exit event with a nonempty A-stack. Similarly for a fail event. □

Proposition 1 (uniqueness) If E is a legal event, then E can have only one legal predecessor, and only one successor. In case E is non-initial, there is exactly one legal predecessor. In case E is non-final, there is exactly one successor. ■

PROOF: The successor part follows from the functionality of $\rightarrowtriangle$. Looking at the rules, we note that only two kinds of events may have more than one predecessor: *fail* $G_A\,\langle - \rangle$ and *fail* $T_1 = T_2\,\langle - \rangle$. Let *fail* $T_1 = T_2\,\langle - \rangle$ be a legal event. Its predecessor may have been *call* $T_1 = T_2\,\langle - \rangle$, on the condition that T_1 and T_2 have no mgu (rule (S:unif:1)), or it could have been *redo* $T_1 = T_2\,\langle \frac{\sigma\,\bullet}{\quad} \rangle$ (rule (S:unif:2)). In the latter case, *redo* $T_1 = T_2\,\langle \frac{\sigma\,\bullet}{\quad} \rangle$ must be a legal event, so the B-stack $\sigma\,\bullet$ had to be derived. The only rule able to derive such a B-stack is (S:unif:1), on the condition that the previous event was *call* $T_1 = T_2\,\langle - \rangle$ and $\mathrm{mgu}(T_1, T_2) = \sigma$. Hence, there can be only one legal predecessor of *fail* $T_1 = T_2\,\langle - \rangle$, depending solely on T_1 and T_2. By a similar argument we can prove that *fail* $G_A\,\langle - \rangle$ can have only one legal predecessor. This concludes the proof of functionality of the converse relation, if restricted to the set of legal events. □

NOTATION 3 (IMPOSSIBLE EVENT) As a notational convenience, all the events which are not final and do not lead to any further events by means of transitions with respect to the given program, are said to lead to the *impossible event*, written as $\bot$. Analogously for events that are not initial events and cannot be entered. In particular, *redo* $\mathsf{fail} \rightarrowtriangle \bot$ and *exit* $\mathsf{fail} \lhd\!\!- \bot$ with respect to any program. Some impossible events are: *call* $G\,\langle \frac{\sigma\,\bullet\,nil}{nil} \rangle$, *redo* $G\,\langle \frac{}{nil} \rangle$ (cannot be entered, non-initial), and *redo* $\mathsf{p}\,\langle \frac{nil}{} \rangle$ (cannot be left, non-final). □

Lemma 3 (non-legal event) If $E \overset{*}{\rightarrowtriangle} \bot$, then E is not legal. If $E \overset{*}{\lhd\!\!-} \bot$, then E is not legal. ■

PROOF: Let $E \lhd\!\!- E_1$. If E is legal, then, because of the uniqueness of the transition, E_1 has to be legal as well. □

Lemma 4 (call is up-to-date) For a legal call event *call* $G\,\langle - \rangle$ holds that $G = \mathrm{substOf}(\ )(G)$, meaning that the substitutions from the B-stack are *already applied* upon the goal to be called. In other words, the goal of any legal call event is up-to-date relative to the current substitution. ■

Notice that this property holds only for call events.

NOTATION 4 (STACK CONCATENATION) Concatenation of stacks we denote by $+$. Concatenating to both stacks of an event we denote by $\ddagger$: If $E = \mathit{Port}\ G\,\langle - \rangle$, then $E \ddagger \langle - \rangle := \mathit{Port}\ G\,\langle \frac{+}{+} \rangle$. □

Proposition 2 (modularity of derivation) Let Π be a program. Let *Pop* be one of *exit*, *fail*. If

$$\mathit{call}\ G\,\langle \tfrac{nil}{nil} \rangle \rightarrowtriangle E_1 \rightarrowtriangle \ldots \rightarrowtriangle E_n \rightarrowtriangle \mathit{Pop}\ G\,\langle \tfrac{}{nil} \rangle$$

is a legal Π-derivation, then for every A-stack and for every B-stack such that $call\ G\ \langle - \rangle$ is a legal event, holds:

$$call\ G\ \langle - \rangle \twoheadrightarrow E_1 \ddagger \langle - \rangle \twoheadrightarrow \ldots \twoheadrightarrow E_n \ddagger \langle - \rangle \twoheadrightarrow Pop\ G\ \langle \frac{+}{\quad} \rangle$$

is also a legal Π-derivation. ■

PROOF: Observe that our rules (with the exception of (S:conj:2)) refer only to the existence of the top element of some stack, never to the emptiness of a stack. Since the top element of a stack S cannot change after appending another stack to S, it is possible to emulate each of the original derivation steps using the 'new' stacks.

It remains to consider the rule (S:conj:2), which applies the whole current substitution upon the second conjunct. First note that any variables in a legal derivation stem either from the top-level goal or are fresh. According to the Lemma 4, a call event is always up-to-date, i. e. the current substitution has already been applied to the goal. The most general unifiers may be chosen to be idempotent, so a multiple application of a substitution amounts to a single application. Hence, if $call\ G\ \langle - \rangle$ is a legal event, the substitution of cannot affect any variables of the original derivation. □

5 Applications

5.1 Specifying program properties

Uniqueness and modularity of legal port derivations allow us to succinctly define some traditional notions.

Definition 11 (termination, success, failure) A goal G is said to terminate wrt program Π, if there is a Π-derivation

$$call\ G\ \langle \frac{nil}{nil} \rangle \overset{*}{\twoheadrightarrow} Pop\ G\ \langle \frac{}{nil} \rangle$$

where Pop is one of $exit, fail$. In case of $exit$, the derivation is *successful*, otherwise it is *failed*. ■

In a failed derivation, $= nil$.

Definition 12 (computed answer) In a successful derivation

$$call\ G\ \langle \frac{nil}{nil} \rangle \overset{*}{\twoheadrightarrow} exit\ G\ \langle \frac{}{nil} \rangle$$

is substOf( ), restricted upon the variables of G, called the *computed answer substitution* for G. ■

5.2 Proving program properties

Uniqueness of legal derivation steps enables *forward and backward* derivation steps, in the spirit of the Byrd's article. Push events (call, redo) are more amenable to forward steps, and pop events (exit, fail) are more amenable to backward steps. We illustrate this by a small example.

Lemma 5 If the events on the left-hand sides are legal, the following are legal derivations (for appropriate ,):

$$exit\ A; B, \mathsf{fail}\ \langle - \rangle \twoheadleftarrow exit\ A\ \langle \frac{}{1/A;B,\mathsf{fail}\bullet} \rangle \tag{2}$$

$$redo\ A; B, \mathsf{fail}\ \langle - \rangle \twoheadrightarrow redo\ A\ \langle \frac{}{1/A;B,\mathsf{fail}\bullet} \rangle \tag{3}$$

■

PROOF: The first statement claims: If $exit\ A; B, \mathsf{fail}\,\langle - \rangle$ is legal, then it was reached via $exit\ A$. Without inspecting , in general it is not known whether a disjunction succeeded via its first, or via its second member. But in this particular disjunction, the second member cannot succeed: Assume there are some , with $exit\ A; B, \mathsf{fail}\,\langle - \rangle \leftarrow exit\ B, \mathsf{fail}\,\langle — \rangle$. According to the rules:

$$exit\ B, \mathsf{fail}\,\langle — \rangle \leftarrow exit\ \mathsf{fail}\,\langle \tfrac{}{2/B, \mathsf{fail}\,\bullet} \rangle \leftarrow \bot$$

So according to Lemma 3, $exit\ B, \mathsf{fail}\,\langle — \rangle$ is not a legal event, which proves (2). Similarly, the non-legal derivation $redo\ B, \mathsf{fail} \rightarrow redo\ \mathsf{fail} \rightarrow \bot$ proves (3). □

Modularity of legal derivations enables *abstracting the execution* of a goal, like in the following example.

Example 4 (modularity) Assume that a goal A succeeds, i. e. $call\ A\,\langle \frac{nil}{nil} \rangle \overset{*}{\rightarrow} exit\ A\,\langle \frac{}{nil} \rangle$. Then we have the following legal derivation:

$$\begin{aligned} call\ A, B\,\langle \tfrac{nil}{nil} \rangle &\rightarrow call\ A\,\langle \tfrac{nil}{1/A, B \bullet nil} \rangle, \text{ by (S:conj:1)} \\ &\overset{*}{\rightarrow} exit\ A\,\langle \tfrac{\bullet\, nil}{1/A, B \bullet nil} \rangle, \text{ by modularity and success of } A \\ &\rightarrow call\ B'\,\langle \tfrac{\bullet\, nil}{2/A, B \bullet nil} \rangle, \text{ by (S:conj:2), where } B' = \mathrm{substOf}(\ \ )(B) \end{aligned}$$

If A fails, then we have:

$$\begin{aligned} call\ A, B\,\langle \tfrac{nil}{nil} \rangle &\rightarrow call\ A\,\langle \tfrac{nil}{1/A, B \bullet nil} \rangle, \text{ by (S:conj:1)} \\ &\overset{*}{\rightarrow} fail\ A\,\langle \tfrac{nil}{1/A, B \bullet nil} \rangle, \text{ by modularity and failure of } A \\ &\rightarrow fail\ A, B\,\langle \tfrac{nil}{nil} \rangle, \text{ by (S:conj:3)} \end{aligned}$$

❑

6 Conclusions and outlook

In this paper we give a simple mathematical definition S:PP of the 4-port model of pure Prolog. Some potential for formal verification of pure Prolog has been outlined. There are two interesting directions for future work in this area:

(1) formal specification of the control flow of *full Standard Prolog* (currently we have a prototype for this, within the 4-port model)

(2) formal specification and proof of some non-trivial program properties, like adequacy and non-interference of a practical program transformation.

7 Related work

Concerning attempts to formally define the 4-port model, we are aware of only few previous works. One is a graph-based model of Tobermann and Beckstein [TB93], who formalize the graph traversal idea of Byrd, defining the notion of a *trace* (of a given query with respect to a given program), as a path in a trace graph. The ports are quite lucidly defined as hierarchical nodes of such a graph. However, even for a simple recursive program and a ground query, with a finite SLD-tree, the corresponding trace graph is infinite, which limits its applicability. Another model of Byrd box is a continuation-based approach of Jahier, Ducassé and Ridoux [JDR00]. There is also a stack-based attempt in [Kul00], but although it provides

for some parametrizing, it suffers essentially the same problem as the continuation-based approach, and also the prototypical implementation of the tracer given in [Byr80], taken as a specification of Prolog execution: In these three attempts, a port is represented by some semantic action (e. g. writing of a message), instead of a formal method. Therefore it is not clear how to use any of these models to prove some port-related assertions.

In contrast to the few specifications of the Byrd box, there are many more general models of pure (or even full) Prolog execution. Due to space limitations we mention here only some models, directly relevant to S:PP, and for a more comprehensive discussion see e. g. [KB01]. Comparable to our work are the stack-based approaches. Stärk gives in [Stä98], as a side issue, a simple operational semantics of pure logic programming. A state of execution is a stack of frame stacks, where each frame consists of a goal (ancestor) and an environment. In comparison, our state of execution consists of exactly one environment and one ancestor stack. The seminal paper of Jones and Mycroft [JM84] was the first to present a stack-based model of execution, applicable to pure Prolog with cut added. It uses a sequence of frames. In these stack-based approaches (including our previous attempt [KB01]), there is no *modularity*, i. e. it is not possible to abstract the execution of a subgoal.

Acknowledgments

Many thanks for helpful comments are due to anonymous referees.

References

[Byr80] Lawrence Byrd. Understanding the control flow of Prolog programs. In S. A. Tärnlund, editor, *Proc. of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, 1980. Also as D. A. I. Research Paper No. 151.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard (Reference Manual)*. Springer-Verlag, 1996.

[JDR00] E. Jahier, M. Ducassé, and O. Ridoux. Specifying Byrd's box model with a continuation semantics. In *Proc. of the WLPE'99, Las Cruces, NM*, volume 30 of *ENTCS*. Elsevier, 2000. http://www.elsevier.nl/locate/entcs/volume30.html.

[JM84] N. D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *Proc. of the 1st Int. Symposium on Logic Programming (SLP'84)*, pages 281–288, Atlantic City, 1984.

[KB01] M. Kulaš and C. Beierle. Defining Standard Prolog in rewriting logic. In K. Futatsugi, editor, *Proc. of the 3rd Int. Workshop on Rewriting Logic and its Applications (WRLA 2000), Kanazawa*, volume 36 of *ENTCS*. Elsevier, 2001. http://www.elsevier.nl/locate/entcs/volume36.html.

[KL02] A. King and L. Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming*, 2(4):517–547, 2002.

[Kul00] M. Kulaš. A rewriting Prolog semantics. In M. Leuschel, A. Podelski, R. Ramakrishnan C. and U. Ultes-Nitsche, editors, *Proc. of the CL 2000 Workshop on Verification and Computational Logic (VCL 2000), London*, 2000.

[Lin95] T. Lindgren. Control flow analysis of Prolog (extended remix). Technical Report 112, Uppsala University, 1995. http://www.csd.uu.se/papers/reports.html.

[LO87] T. Lindholm and R. A. O'Keefe. Efficient implementation of a defensible semantics for dynamic Prolog code. In *Proc. of the 4th Int. Conference on Logic Programming (ICLP'87)*, pages 21–39, Melbourne, 1987.

[Stä98] Robert F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *J. of Logic Programming*, 36(3):241–269, 1998. Source distribution http://www.inf.ethz.ch/~staerk/lptp.html.

[TB93] G. Tobermann and C. Beckstein. What's in a trace: The box model revisited. In *Proc. of the 1st Int. Workshop on Automated and Algorithmic Debugging (AADEBUG'93), Linköping*, volume 749 of *LNCS*. Springer-Verlag, 1993.

A Leaving events

Leaving a call event

$$call\ A, B\ \langle - \rangle \twoheadrightarrow call\ A\ \langle \overline{1/A,B\bullet} \rangle \qquad \text{(S:conj:1)}$$

$$call\ A; B\ \langle - \rangle \twoheadrightarrow call\ A\ \langle \overline{1/A;B\bullet} \rangle \qquad \text{(S:disj:1)}$$

$$call\ \mathsf{true}\ \langle - \rangle \twoheadrightarrow exit\ \mathsf{true}\ \langle - \rangle \qquad \text{(S:true:1)}$$

$$call\ \mathsf{fail}\ \langle - \rangle \twoheadrightarrow fail\ \mathsf{fail}\ \langle - \rangle \qquad \text{(S:fail)}$$

$$call\ T_1 = T_2\ \langle - \rangle \twoheadrightarrow \begin{cases} exit\ T_1 = T_2\ \langle \overline{\sigma\bullet} \rangle, & \text{if } \mathrm{mgu}(T_1, T_2) = \sigma \\ fail\ T_1 = T_2\ \langle - \rangle, & \text{otherwise} \end{cases} \qquad \text{(S:unif:1)}$$

$$call\ G_A\ \langle - \rangle \twoheadrightarrow \begin{cases} call\ \sigma(B)\ \langle \overline{G_A\bullet} \rangle, & \text{if } H :- B \text{ is a fresh renaming of a} \\ & \quad \text{clause in } \Pi, \text{ and } \mathrm{mgu}(G_A, H) = \sigma, \text{ and } \sigma(G_A) = G_A \\ fail\ G_A\ \langle - \rangle, & \text{otherwise} \end{cases} \qquad \text{(S:atom:1)}$$

Leaving a redo event

$$redo\ A, B\ \langle - \rangle \twoheadrightarrow redo\ B\ \langle \overline{2/A,B\bullet} \rangle \qquad \text{(S:conj:6)}$$

$$redo\ A; B\ \langle \overline{OR(C,(N/A;B))\bullet} \rangle \twoheadrightarrow redo\ C\ \langle \overline{N/A;B\bullet} \rangle \qquad \text{(S:disj:6)}$$

$$redo\ \mathsf{true}\ \langle - \rangle \twoheadrightarrow fail\ \mathsf{true}\ \langle - \rangle \qquad \text{(S:true:2)}$$

$$redo\ T_1 = T_2\ \langle \overline{\sigma\bullet} \rangle \twoheadrightarrow fail\ T_1 = T_2\ \langle - \rangle \qquad \text{(S:unif:2)}$$

$$redo\ G_A\ \langle \overline{BY(B,G'_A)\bullet} \rangle \twoheadrightarrow redo\ B\ \langle \overline{G'_A\bullet} \rangle \qquad \text{(S:atom:4)}$$

Leaving an exit event

$$exit\ A'\ \langle \overline{1/A,B\bullet} \rangle \twoheadrightarrow call\ B''\ \langle \overline{2/A,B\bullet} \rangle, \text{ with } B'' := \mathrm{substOf}(\ \ )(B) \qquad \text{(S:conj:2)}$$

$$exit\ B'\ \langle \overline{2/A,B\bullet} \rangle \twoheadrightarrow exit\ A, B\ \langle - \rangle \qquad \text{(S:conj:4)}$$

$$exit\ A\ \langle \overline{1/A;B\bullet} \rangle \twoheadrightarrow exit\ A; B\ \langle \overline{OR(A,(1/A;B))\bullet} \rangle \qquad \text{(S:disj:4)}$$

$$exit\ B\ \langle \overline{2/A;B\bullet} \rangle \twoheadrightarrow exit\ A; B\ \langle \overline{OR(B,(2/A;B))\bullet} \rangle \qquad \text{(S:disj:5)}$$

$$exit\ B\ \langle \overline{G_A\bullet} \rangle \twoheadrightarrow exit\ G_A\ \langle \overline{BY(B,G_A)\bullet} \rangle \qquad \text{(S:atom:2)}$$

Leaving a fail event

$$fail\ A'\ \langle \overline{1/A,B\bullet} \rangle \twoheadrightarrow fail\ A, B\ \langle - \rangle \qquad \text{(S:conj:3)}$$

$$fail\ B'\ \langle \overline{2/A,B\bullet} \rangle \twoheadrightarrow redo\ A\ \langle \overline{1/A,B\bullet} \rangle \qquad \text{(S:conj:5)}$$

$$fail\ A\ \langle \overline{1/A;B\bullet} \rangle \twoheadrightarrow call\ B\ \langle \overline{2/A;B\bullet} \rangle \qquad \text{(S:disj:2)}$$

$$fail\ B\ \langle \overline{2/A;B\bullet} \rangle \twoheadrightarrow fail\ A; B\ \langle - \rangle \qquad \text{(S:disj:3)}$$

$$fail\ B\ \langle \overline{G_A\bullet} \rangle \twoheadrightarrow fail\ G_A\ \langle - \rangle \qquad \text{(S:atom:3)}$$

B An example with variables

Assume the following program Π:

```
post(X,Y) :- one(X,Y), two(X,Y).
one(X,_) :- X=1.
two(_,Y) :- Y=a; Y=b.
```

Table 2 below shows the complete Π-execution of the goal post(X, Y), fail in the model S:PP. Highlighted are the A-stacks and the mgus. Notice the "lazy" binding of variables in the current goal.

call (post(X, Y), fail) , $\{nil\}$, $\{nil\}$
→ *call* post(X, Y) , $\{1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{nil\}$
→ *call* (one(X, Y), two(X, Y)) , $\{\text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{nil\}$
→ *call* one(X, Y) , $\{1/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{nil\}$
→ *call* X=1 , $\{\text{one(X, Y)} \bullet 1/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{nil\}$
→ *exit* X=1 , $\{\text{one(X, Y)} \bullet 1/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{[X/1] \bullet nil\}$
→ *exit* one(X, Y) , $\{1/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *call* two(1, Y) , $\{2/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *call* (Y=a; Y=b) , $\{\text{two(1, Y)} \bullet 2/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *call* Y=a , $\{(1/(\text{Y=a}); \text{Y=b}) \bullet \text{two(1, Y)} \bullet 2/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *exit* Y=a , $\{(1/(\text{Y=a}); \text{Y=b}) \bullet \text{two(1, Y)} \bullet 2/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{[Y/a] \bullet BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *exit* (Y=a; Y=b) , $\{\text{two(1, Y)} \bullet 2/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{OR(\text{Y=a}, (1/(\text{Y=a}); \text{Y=b})) \bullet [Y/a] \bullet BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *exit* two(1, Y) , $\{2/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY((\text{Y=a}; \text{Y=b}), \text{two(1, Y)}) \bullet OR(\text{Y=a}, (1/(\text{Y=a}); \text{Y=b})) \bullet [Y/a] \bullet BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *exit* (one(X, Y), two(X, Y)) , $\{\text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY((\text{Y=a}; \text{Y=b}), \text{two(1, Y)}) \bullet OR(\text{Y=a}, (1/(\text{Y=a}); \text{Y=b})) \bullet [Y/a] \bullet BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *exit* post(X, Y) , $\{1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY((\text{one(X, Y)}, \text{two(X, Y)}), \text{post(X, Y)}) \bullet BY((\text{Y=a}; \text{Y=b}), \text{two(1, Y)}) \bullet OR(\text{Y=a}, (1/(\text{Y=a}); \text{Y=b})) \bullet [Y/a] \bullet BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *call* fail , $\{2/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY((\text{one(X, Y)}, \text{two(X, Y)}), \text{post(X, Y)}) \bullet BY((\text{Y=a}; \text{Y=b}), \text{two(1, Y)}) \bullet OR(\text{Y=a}, (1/(\text{Y=a}); \text{Y=b})) \bullet [Y/a] \bullet BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *fail* fail , $\{2/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY((\text{one(X, Y)}, \text{two(X, Y)}), \text{post(X, Y)}) \bullet BY((\text{Y=a}; \text{Y=b}), \text{two(1, Y)}) \bullet OR(\text{Y=a}, (1/(\text{Y=a}); \text{Y=b})) \bullet [Y/a] \bullet BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *redo* post(X, Y) , $\{1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY((\text{one(X, Y)}, \text{two(X, Y)}), \text{post(X, Y)}) \bullet BY((\text{Y=a}; \text{Y=b}), \text{two(1, Y)}) \bullet OR(\text{Y=a}, (1/(\text{Y=a}); \text{Y=b})) \bullet [Y/a] \bullet BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *redo* (one(X, Y), two(X, Y)) , $\{\text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY((\text{Y=a}; \text{Y=b}), \text{two(1, Y)}) \bullet OR(\text{Y=a}, (1/(\text{Y=a}); \text{Y=b})) \bullet [Y/a] \bullet BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$
→ *redo* two(X, Y) , $\{2/\text{one(X, Y)}, \text{two(X, Y)} \bullet \text{post(X, Y)} \bullet 1/\text{post(X, Y)}, \text{fail} \bullet nil\}$, $\{BY((\text{Y=a}; \text{Y=b}), \text{two(1, Y)}) \bullet OR(\text{Y=a}, (1/(\text{Y=a}); \text{Y=b})) \bullet [Y/a] \bullet BY(\text{X=1}, \text{one(X, Y)}) \bullet [X/1] \bullet nil\}$

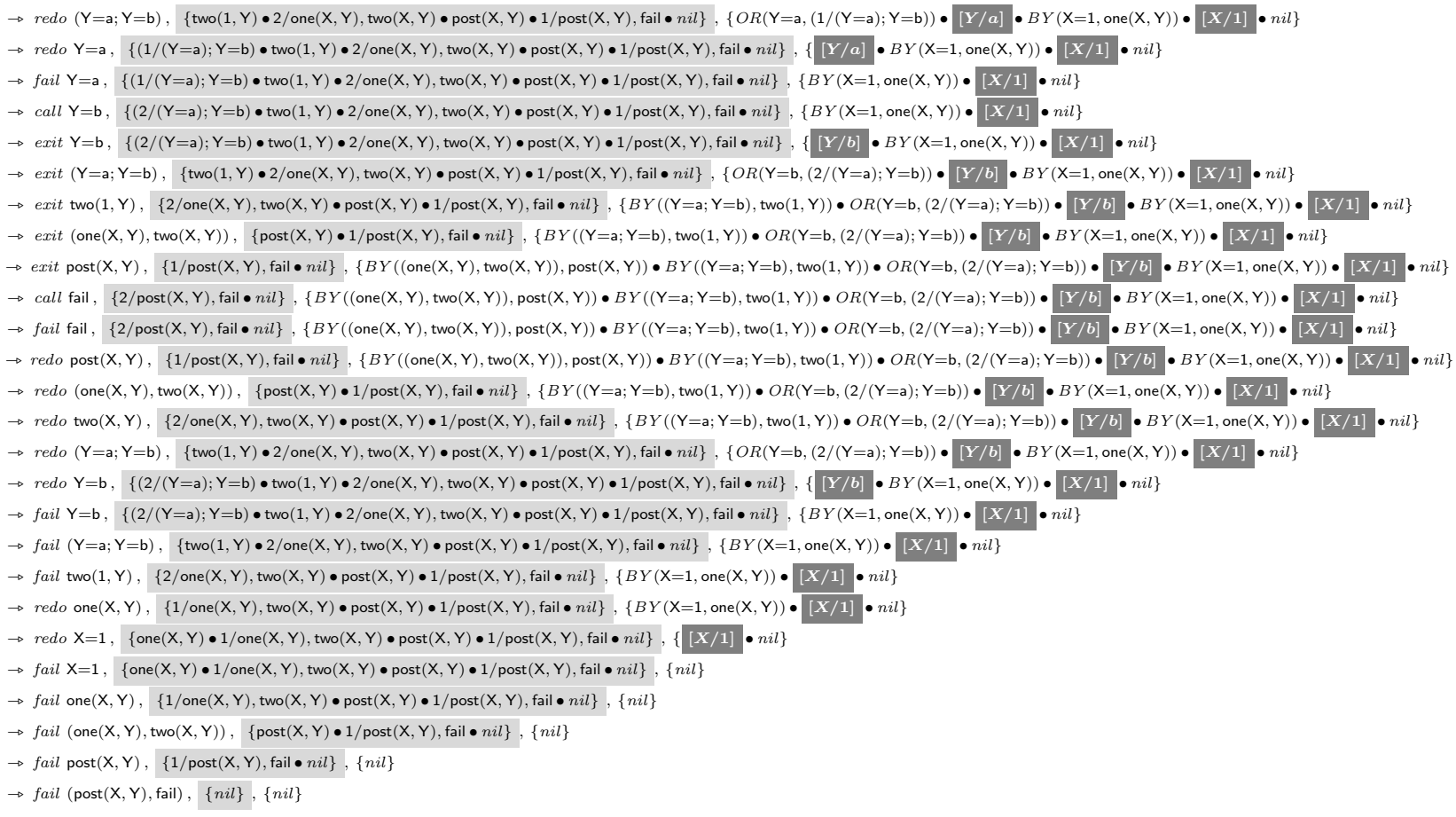


Table 2. Execution of a query in S:PP